S

Schedulers for Optimistic Rate Based Flow Control

Panagiota Fatourou
Department of Computer Science, University of Ioannina, Ioannina, Greece

Keywords

Bandwidth allocation; Rate adjustment; Rate allocation

Years and Authors of Summarized Original Work

2005; Fatourou, Mavronicolas, Spirakis

Problem Definition

The problem concerns the design of efficient rate-based flow control algorithms for virtual-circuit communication networks where a connection is established by allocating a fixed path, called *session*, between the source and the destination. Rate-based flow-control algorithms repeatedly adjust the transmission rates of different sessions in an end-to-end manner with primary objectives to optimize the network utilization and achieve some kind of fairness in sharing bandwidth between different sessions.

A widely-accepted fairness criterion for flow-control is *max-min fairness* which requires that the rate of a session can be increased only if this increase does not cause a decrease to any other session with smaller or equal rate. Once max-min fairness has been achieved, no session rate can be increased any further without violating the above condition or exceeding the bandwidth *capacity* of some link. Call *max-min* rates the session rates when max-min fairness has been reached.

Rate-based flow control algorithms perform rate adjustments through a sequence of *operations* in a way that the capacities of network links are never exceeded. Some of these algorithms, called *conservative* [3, 6, 10, 11, 12], employ operations that gradually increase session rates until they converge to the max-min rates without ever performing any rate decreases. On the other hand, *optimistic* algorithms, introduced more recently by Afek, Mansour, and Ostfeld [1], allow for decreases, so that a session's rate may be intermediately be larger than its final max-min rate.

Optimistic algorithms [1, 7] employ a specific rate adjustment operation, called *update operation* (introduced in [1]). The goal of an `update` operation is to achieve fairness among a set of neighboring sessions and optimize the network utilization in a local basis. More specifically, an `update` operation calculates an increase for the rate of a particular session (the *updated* session) for each link the session traverses. The calculated increase on a particular link is the maximum possible that respects the max-min fairness condition between the sessions traversing the link; that

M.-Y. Kao (ed.), *Encyclopedia of Algorithms*,
DOI 10.1007/978-1-4939-2864-4

is, this increase should not cause a decrease to the rate of any other session traversing the link with smaller rate than the rate of the updated session after the increase. Once the maximum increase on each link has been calculated the minimum among them is applied to the session's rate (let e be the link for which the minimum increase has been calculated). This causes the decrease of the rates of those sessions traversing e which had larger rates than the increased rate of the updated session to the new rate. Moreover, the `update` operation guarantees that all the capacity of link e is allocated to the sessions traversing it (so the bandwidth of this link is fully utilized).

One important performance parameter of a rate-based flow control algorithm is its *locality* which is characterized by the amount of knowledge the algorithm requires to decide which session's rate to update next. *Oblivious* algorithms do not assume any knowledge of the network topology or the current session rates. *Partially oblivious* algorithms have access to session rates but they are unaware of the network topology, while *non-oblivious* algorithms require full knowledge of both the network topology and the session rates. Another crucial performance parameter of rate-based flow control algorithms is the *convergence complexity* measured as the maximum number of rate-adjustment operations performed in any execution until max-min fairness is achieved.

Key Results

Fatourou, Mavronicolas and Spirakis [7] have studied the convergence complexity of optimistic rate-based flow control algorithms under varying degrees of locality. More specifically, they have proved lower and upper bounds on the convergence complexity of oblivious, partially-oblivious and non-oblivious algorithms. These bounds are expressed in terms of n the number of sessions laid out on the network.

Theorem 1 (Lower Bound for Oblivious Algorithms, Fatourou, Mavronicolas and Spirakis [7]) *Any optimistic, oblivious, rate-based flow control algorithm requires $\Omega(n^2)$ update operations to compute the max-min rates.*

Fatourou, Mavronicolas and Spirakis [7] have presented algorithm `RoundRobin`, which applies `update` operations to sessions in a round robin order. Obviously, `RoundRobin` is oblivious. It has been proved [7] that the convergence complexity of `RoundRobin` is $O(n^2)$. This shows that the lower bound for oblivious algorithms is tight.

Theorem 2 (Upper Bound for Oblivious Algorithms, Fatourou, Mavronicolas and Spirakis [7]) `RoundRobin` *computes the max-min rates after performing $O(n^2)$ update operations.*

`RoundRobin` belongs to a class of oblivious algorithms, called `Epoch` [7]. Each algorithm of this class repeatedly chooses some permutation of all session indices and applies `update` operations on the sessions in the order determined by this permutation. This is performed n times. Clearly, `Epoch` is a class of oblivious algorithms. It has been proved [7] that each of the algorithms in this class has convergence complexity $O(n^2)$.

Another oblivious algorithm, called `Arbitrary`, has been presented in [1]. The algorithm works in a very simple way by choosing the next session to be updated in an arbitrary way, but it requires an exponential number of `update` operations to compute the max-min rates.

Fatourou, Mavronicolas and Spirakis [7] have proved that partially-oblivious algorithms do not achieve better convergence complexity than oblivious algorithms despite the knowledge they employ.

Theorem 3 (Lower Bound for Partially Oblivious Algorithms, Fatourou, Mavronicolas and Spirakis [7]) *Any optimistic, partially oblivious, rate-based flow control algorithm*

requires $\Omega(n^2)$ update operations to compute the max-min rates.

Afek, Mansour and Ostfeld [1] have presented a partially oblivious algorithm, called `GlobalMin`. The algorithm chooses as the session to update next the one with the minimum rate among all sessions. The convergence complexity of `GlobalMin` is $O(n^2)$ [1]. This shows that the lower bound for partially-oblivious algorithms is tight.

Theorem 4 (Upper Bound for Partially Oblivious algorithms, Afek, Mansour and Ostfeld [1]) `GlobalMin` *computes the max-min rates after performing $O(n^2)$ update operations.*

Another partially-oblivious algorithm, called `LocalMin`, is also presented in [1]. The algorithm chooses to schedule next a session which has a minimum rate among all the sessions that share a link with it. `LocalMin` has time complexity $O(n^2)$.

Fatourou, Mavronicolas and Spirakis [7] have presented a non-oblivious algorithm, called `Linear`, that exhibits linear convergence complexity. `Linear` follows the classical idea [3, 12] of selecting as the next updated session one of the sessions that traverse the most congested link in the network. To discover such a session, `Linear` requires knowledge of the network topology and the session rates.

Theorem 5 (Upper Bound for Non-Oblivious Algorithms, Fatourou, Mavronicolas and Spirakis [7]) *Linear computes the max-min rates after performing O(n) update operations.*

The convergence complexity of `Linear` is optimal, since n rate adjustments must be performed in any execution of an optimistic rate-based flow control algorithm (assuming that the initial session rates are zero). However, this comes at a remarkable cost in locality which makes `Linear` impractical.

Applications

Flow control is the dominant technique used in most communication networks for preventing data traffic congestion when the externally injected transmission load is larger than what can be handled even with optimal routing. Flow control is also used to ensure high network utilization and fairness among the different connections. Examples of networking technologies where flow control techniques have been extensively employed to achieve these goals are TCP streams [5] and ATM networks [4]. An overview of flow control in practice is provided in [3].

The idea of controlling the rate of a traffic source originates back to the data networking protocols of the ANSI Frame Relay Standard. Rate-based flow control is considered attractive due to its simplicity and its low hardware requirements. It has been chosen by the ATM Forum on Traffic Management as the best suited technique for the goals of ABR service [4].

A substantial amount of research work has been devoted in past to conservative flow control algorithms [3, 6, 10, 11, 12]. The optimistic framework has been introduced much later by Afek et al. [1] as a more suitable approach for real dynamic networks where decreases of session rates may be necessary (e.g., for accommodating the arrival of new sessions). The algorithms presented in [7] improve upon the original algorithms proposed in [1] in terms of either convergence complexity, or locality, or both. Moreover, they identify that certain classical scheduling techniques, such as round-robin [11], or adjusting the rates of sessions traversing one of the most congested links [3, 12] can be efficient under the optimistic framework. The first general lower bounds on the convergence complexity of rate-based flow control algorithms are also presented in [7].

The performance of optimistic algorithms has been theoretically analyzed in terms of an abstraction, namely the `update` operation, which has been designed to address most of the intricacies encountered by rate-based flow control algorithms. However, the `update` operation

masks low-level implementation details, while it may incur non-trivial, local computations on the switches of the network. Fatourou, Mavronicolas and Spirakis [9] have studied the impact on the efficiency of optimistic algorithms of local computations required at network switches in order to implement the `update` operation, and proposed a distributed scheme that implements a broad class of such algorithms. On a different avenue, Afek, Mansour and Ostfeld [2] have proposed a simple flow control scheme, called `Phantom`, which employs the idea of considering an imaginary session on each link [10, 12], and they have discussed how `Phantom` can be applied to ATM networks and networks of TCP routers.

A broad class of modern distributed applications (e.g., remote video, multimedia conferencing, data visualization, virtual reality, etc.) exhibit highly differing bandwidth requirements and need some kind of quality of service guarantees. To efficiently support a wide diversity of applications sharing available bandwidth, a lot of research work has been devoted on incorporating priority schemes on current networking technologies. Priorities offer a basis for modeling the diverse resource requirements of modern distributed applications, and they have been used to accommodate the needs of network management policies, traffic levels, or pricing. The first efforts for embedding priority issues into max-min fair, rate-based flow control were performed in [10, 12]. An extension of the classical theory of max-min fair, rate-based flow control to accommodate priorities of different sessions has been presented in [8]. (A number of other papers addressing similar generalizations of max-min fairness to account for priorities and utility have been presented after the original publication of [8].)

Many modern applications are not based solely on point-to-point communication but they rather require multipoint-to-multipoint transmissions. A max-min fair rate-based flow control algorithm for multicast networks is presented in [14]. Max-min fair allocation of bandwidth in wireless adhoc networks is studied in [15].

Open Problems

The research work on optimistic, rate-based flow control algorithms leaves open several interesting questions. The convergence complexity of the proposed optimistic algorithms has been analyzed only for a static set of sessions laid out on the network. It would be interesting to evaluate these algorithms under a dynamic network setting, and possibly extend the techniques they employ to efficiently accommodate arriving and departing sessions.

Although max-min fairness has emerged as the most frequently praised fairness criterion for flow control algorithms, achieving it might be expensive in highly dynamic situations. Afek et al. [1] have proposed a modified version of the `update` operation, called *approximate* `update`, which applies an increase to some session only if it is larger than some quantity $\delta > 0$. An *approximate* optimistic algorithm uses the approximate `update` operation and terminates if no session rate can be increased by more than δ. Obviously such an algorithm does not necessarily reach max-min fairness. It has been proved [1] that for some network topologies every approximate optimistic algorithm may converge to session rates that are away from their max-min counterparts by an exponential factor. The consideration of other versions of `update` operation or different termination conditions might lead to better max-min fairness approximations and deserves more study; different choices may also significantly impact the convergence complexity of approximate optimistic algorithms. It would be also interesting to derive trade-off results between the convergence complexity of such algorithms and the distance of the terminating rates they achieve to the max-min rates.

Fairness formulations that naturally approximate the max-min condition have been proposed by Kleinberg et al. [13] as suitable fairness criteria for certain routing and load balancing applications. Studying these formulations under the rate-based flow control setting is an interesting open problem.

Cross-References

▸ Multicommodity Flow, Well-Linked Terminals and Routing Problems

Recommended Reading

1. Afek Y, Mansour Y, Ostfeld Z (1999) Convergence complexity of optimistic rate based flow control algorithms. J Algorithms 30(1):106–143
2. Afek Y, Mansour Y, Ostfeld Z (2000) Phantom: a simple and effective flow control scheme. Comput Netw 32(3):277–305
3. Bertsekas DP, Gallager RG (1992) Data networks, 2nd edn. Prentice Hall, Englewood Cliffs
4. Bonomi F, Fendick K (1995) The rate-based flow control for available bit rate ATM service. IEEE/ACM Trans Netw 9(2):25–39
5. Brakmo LS, Peterson L (1995) TCP vegas: end-to-end congestion avoidance on a global internet. IEEE J Sel Areas Commun 13(8):1465–1480
6. Charny A (1994) An algorithm for rate-allocation in a packet switching network with feedback. Technical report MIT/LCS/TR-601, Massachusetts Institute of Technology, Apr 1994
7. Fatourou P, Mavronicolas M, Spirakis P (2005) Efficiency of oblivious versus non-oblivious schedulers for optimistic, rate-based flow control. SIAM J Comput 34(5):1216–1252
8. Fatourou P, Mavronicolas M, Spirakis P (2005) Max-min fair flow control sensitive to priorities. J Interconnect Netw 6(2):85–114, Also in Proceedings of the 2nd international conference on principles of distributed computing, pp 45–59 (1998)
9. Fatourou P, Mavronicolas M, Spirakis P (1998) The global efficiency of distributed, rate-based flow control algorithms. In: Proceedings of the 5th colloquium on structural information and communication complexity, June 1998, pp 244–258
10. Gafni E, Bertsekas D (1984) Dynamic control of session input rates in communication networks. IEEE Trans Autom Control 29(11):1009–1016
11. Hahne E (1991) Round Robin scheduling for max-min fairness in data networks. IEEE J Sel Areas Commun 9(7):1024–1039
12. Jaffe J (1981) Bottleneck flow control. IEEE Trans Commun 29(7):954–962
13. Kleinberg J, Rabani Y, Tardos É (1999) Fairness in routing and load balancing. In: Proceedings of the 40th annual IEEE symposium on foundations of computer science, Oct 1999, pp 568–578
14. Sarkar S, Tassiulas L (2005) Fair distributed congestion control in multirate multicast networks. IEEE/ACM Trans Netw 13(1):121–133
15. Tassiulas L, Sarkar S (2005) Maxmin fair scheduling in wireless adhoc networks. IEEE J Sel Areas Commun 23(1):163–173

Scheduling in Data Broadcasting

Xiaofeng Gao
Department of Computer Science, Shanghai Jiao Tong University, Shanghai, China

Keywords

Algebraic algorithm; Data retrieval; Scheduling; Wireless data broadcast

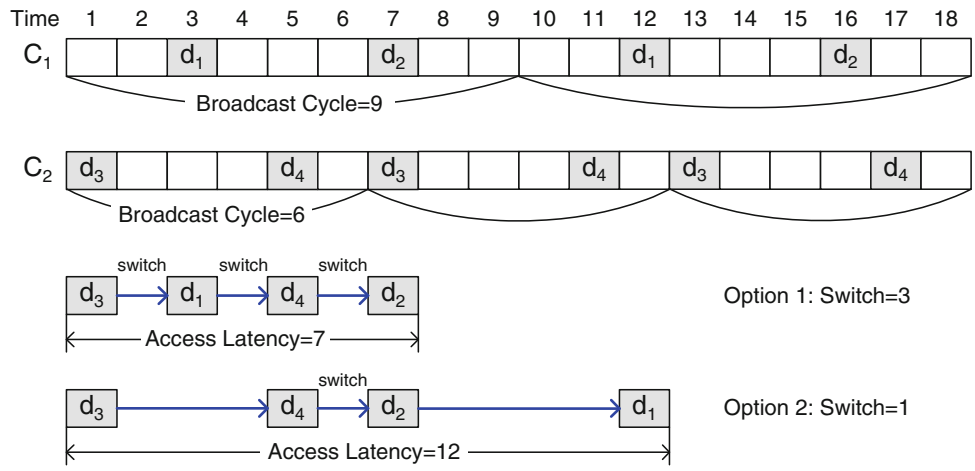
Years and Authors of Summarized Original Work

2011; Gao, Lu, Wu, Fu
2013; Gao, Lu, Wu, Fu

Problem Definition

Wireless data broadcasting means a set of data are repeatedly broadcast from a base station to a mass number of wireless and mobile clients. If a client wants a specific datum, it will access onto the broadcasting channel, get the location (appearance time) of the datum with the help indices, and wait until the datum has been broadcast. The scheduling problem in data broadcasting deals with the design of an efficient permutation strategy for a client to download a required subset of data from an multichannel broadcasting system, with both time and energy constraints. Here time constraint means the client wants the minimum downloading time from when it starts the query until the moment it has successfully download each piece of datum, while the energy constraint means the client wants the minimum switching numbers among channels to reduce extra battery consumption. Correspondingly, we can define the scheduling problem formally as follows:

A client wants to download a group of k data items $D = \{d_1, d_2, \ldots, d_k\}$, each with different sizes. Those data items are broadcasted on n different channels $C = \{c_1, c_2, \ldots, c_n\}$ repeatedly together with many other data items. Each channel may have different bandwidth and



Scheduling in Data Broadcasting, Fig. 1 Example of possible objective contradiction

broadcast cycle length. Let the time to download the smallest transmission packet be a unit time, and the length of d_i can be represented as l_i (also referring as downloading time).

Assume the client knows the locations (channel id and time offsets) of the required data set beforehand at the starting time $t = 0$ (with the help of indices, which is beyond the scope of our problem), and the target is how to download k known data from n channels efficiently with minimum downloading time (we also refer it as *access latency*) and minimum switching numbers.

Unfortunately, the two objectives in this problem are conflicting to each other. Figure 1 is an example to illustrate this phenomenon. In Fig. 1, there are two channels broadcasting 15 data items repeatedly. Suppose the gray data items {1,2,3,4} are of the request. The starting point of the client retrieving process is at $t = 0$. If we want to minimize the access latency, the request should be retrieved in the order of "$3 \rightarrow 1 \rightarrow 4 \rightarrow 2$" which takes only 7 time units but needs 3 switches (shown as Option 1 in Fig. 1). However, if we want to minimize the switches, the best retrieving order should be "$3 \rightarrow 4 \rightarrow 2 \rightarrow 1$" which needs only 1 switch but takes 12 time units (shown as Option 2 in Fig. 1). This example exhibits that access latency and number of switches cannot be minimized at the same time. They are contradictory factors.

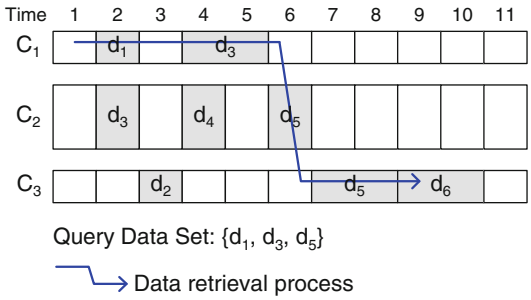
As a consequence, we want to fix one factor and minimize another objective, and thus have the following objective:

Objective

We hope to design a data downloading order for a client to download k data items from n broadcasting channels, such that the access latency t is minimized if we restrict the switch number among channels (denoted as h); otherwise, we will minimize the number of switches h once the access latency t is bounded.

Constraints

1. **Switch Constraint**: Note that if a client is downloading a data from channel c_i at time t_0, then it cannot switch to channel c_j, where $j \neq i$, to download another data at time $t_0 + 1$ due to connection protocols. Thus, we assume if a client wants to download data from another channel, it needs at least one time unit for channel switching. Figure 2 gives a typical process of data retrieval in multichannel broadcast environments. The query data set is $\{d_1, d_3, d_5\}$, and a user can download data object d_1 and d_3 from channel c_1 and then switch to channel c_3 at time $t = 6$ to download data object d_5 at time $t = 7$. However, after time $t = 5$, the user cannot switch from channel c_1 to c_2 to download data



Scheduling in Data Broadcasting, Fig. 2 Switch constraint

d_5 at time $t = 6$. From Fig. 2, we also get that the bandwidths of different channels are not necessarily the same. Actually, the bandwidth of channel c_2 is twice as that of c_1 or c_3, thus d_3 or d_5, which take two time slots on c_1 or c_3, can be broadcasted in one time unit by c_2.

2. **Objective Constraint**: We have to setup a reasonable threshold for latency constraint t and switch constraint h, such that we would achieve a feasible solution for the corresponding minimized switches and shortest access latency.

Problem 1 (Scheduling in Data Broadcasting)

INPUT: *The required data subset* $D = \{d_1, d_2, \ldots, d_k\}$ *broadcast on* n *different channels with their locations and downloading time* l_i, *a switch constraint* h *or latency constraint* t.

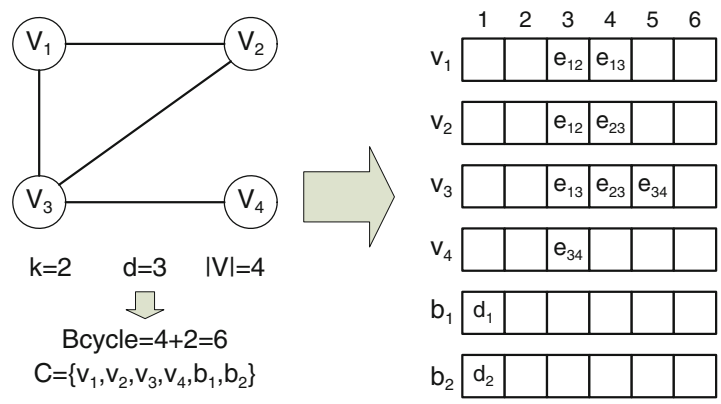
OUTPUT: *A permutation of* D *such that if starting from time slot zero, a client would achieve the shortest access latency (with switch threshold* h*) or the minimum switch numbers (with latency threshold* t*) if it follows this permutation to download each data item sequentially with switch constraint.*

Key Results

Scheduling is an important part in the wireless data broadcast system. Researchers tend to divide the scheduling problems into two subproblems. The first one is the data allocation problem in the server side, while the other one is the data retrieval problem in the client size.

With respect to server side scheduling, several works have been proposed to improve the system performance [1–5]. Acharya et al. [1] first dealt with the data allocation problem for single-channel environment. He proposed a scheduling algorithm considering data access frequencies and allowed frequent accessed data to be broadcasted more often. Most works concerned multi-channel environment. For data set with uniform length, Yee et al. [2] proposed an $O(t^2m)$ time-complexity dynamic programming algorithm to find the optimal schedule and also a near optimal greedy algorithm to reduce the time complexity. For nonuniform lengths case, Ardizzoni et al. [3] proved that this problem is strong NP-hard. Ardizzoni et al. [3], Anticaglia et al. [4], and Kenyon et al. [5] designed algorithms based on greedy and heuristic strategy.

Also most of the literature discussed the data allocation problem from server's point of view; several works [6–10] considered the data retrieval scheduling problem from the client's point of view. Shi et al. [6] defined the data retrieval problem in MIMO environment as parallel data retrieval scheduling with MIMO Antennae (PADRS-MIMO) and proposed two greedy heuristics to guarantee minimum switchings among channels or reduce the downloading time when the number of antennae in the mobile devices are limited. Lu et al. [7, 8] defined the largest number data retrieval (LNDR) and maximum cost data retrieval (MCDR) problems and considered the hopping cost. He also proved that when the hopping cost cannot be ignored, LNDR is NP-hard and designed a 1/2-approximation algorithm. Gao et al. [9, 10] designed a randomized algebraic algorithm that takes both energy cost and access time into consideration to schedule the data retrieval process in multichannel environments. The algorithm proposed can detect whether a given data retrieval problem has a solution with access time t and number of switchings h in $O\left(2^k(hnt)^{O(1)}\right)$ time, where n is the number of channels and k is the number of requested data items.



Scheduling in Data Broadcasting, Fig. 3 Example of VC reduction

Hardness Analysis

Define a tuple $s = \{i_s, j_s, t_s, t'_s\}$ to denote the datum d_{i_s}, which can be downloaded from channel c_{j_s} during the time span $[t_s, t'_s]$; then it is clear that a valid data retrieval schedule is a sequence of k intervals $s_1, s_2, \ldots, s_k$, each tuple corresponds to a distinct data item in D, and there are no conflicts between any two of the k tuples. To analyze the NP-hardness, we then define the decision problem of MCDR.

Definition 1 (Decision MCDR) Given a data set D, a channel set C, a time threshold t, and a switching threshold h, find a valid data retrieval schedule to download all the data in D from C before time t with at most h switchings.

Theorem 1 *MCDR problem is NP-hard.*

Proof We use $VC \leq_p MCDR$ to prove this theorem. Here VC is the decision problem of vertex cover, say, given a graph $G = (V, E)$, we want to find a minimum size vertex subset $VC \subseteq V$ such that for any edge $(v_i, v_j) \in E$, either $v_i \in VC$ or $v_j \in VC$. An instance of vector cover is: given a graph $G = (V, E)$ and integer k, does it have a vertex cover VC with size k? Then we construct an instance of MCDR from G and k as follows:

1. For each vertex $v_i \in V$, define a channel v_i. Define another k channels $b_1, \ldots, b_k$. Then the channel set is $C = \{v_1, \ldots, v_{|V|}, b_1, \ldots, b_k\}$. Totally $|V| + k$ channels. Let δ be the maximum vertex degree in G, and then each channel has broadcast cycle length $\delta + 3$.
2. For each edge $(v_i, v_j) \in E$, define a unit length data item e_{ij} in data set D_e and append it on channel c_i and c_j (the order can be arbitrary and starting from the third time unit).
3. For each channel b_i, define a unit length data item d_i in data set D_d and allocate it on the first time unit of channel b_i.
4. The data set $D = D_e \cup D_b$.

Figure 3 is an example to show how to construct the broadcast system. In this figure, $\delta = 3$, $k = 2$, $|V| = 4$; thus, the channel set should be $\{v_1, v_2, v_3, v_4, b_1, b_2\}$, each having broadcast length $\delta + 3 = 6$. Each e_{ij} represents an edge (v_i, v_j), and it is clear that if we download all data items from channel v_i, then it means we cover the edges connecting node v_i.

Next, we prove that G has a vertex cover with size k if and only if there is a valid data retrieval schedule S such that $t = k(\delta+3)$ and $h = 2k-1$.

$\Longrightarrow$: If G has a vertex cover VC with size k, then we can select the corresponding k channels in $\{v_i | v_i \in VC\}$ to receive all the data in k cycles. At the beginning of ith cycle (iteration), the client will visit b_i at $t = 1$, and hop to some $v_i \in VC$ channel, stay on this channel till the last time unit of the broadcast cycle, and then hop to b_{i+1}. There are k b_is, so each iteration client will download one of them. VC is a vertex cover, so following all

$v_i \in VC$ we must download every e_{ij}. The length of each broadcast cycle is $\delta + 3$, so the total access latency is $k(\delta + 3)$. In each broadcast cycle, the client will switch twice (except the last cycle), so $h = 2k - 1$.

$\Longleftarrow$: Assume MCDR has a valid schedule S with $t = k(\delta + 3)$ and $h = 2k - 1$. Let us consider D_b first. There are k b_i's located at the first time unit on k different channels. It means we have to switch at least $k - 1$ hops to download D_b, and then we only have another k hops for D_e, which means we can visit at most k channels in $\{v_i\}$. At the beginning of each broadcast cycle, we always stay at some channel b_i to download d_i, and then we switch to some v_i, and at the end of this cycle, we have to switch to channel b_{i+1} for d_{i+1}. This means we cannot switch to two vertex channels within one broadcast cycle, otherwise we cannot download $D = D_e \cup D_b$ in k iterations. Since S is valid, we visit k vertex channels and download all D_e data items, it means these k vertices form a vertex cover with size k.

This reduction can be done in polynomial time, and we can conclude that MCDR is NP-hard.

Randomized Algebraic Algorithm

To solve the above decision problem, we developed a randomized algebraic algorithm. It can detect if a given problem has a schedule to download all the requested data before time t and with at most h channel switchings in $O\left(2^k(nht)^{O(1)}\right)$ time, where n is the number of channels and k is the number of required data items. We also provide a fixed parameter tractable (FPT) algorithm with computational time $O\left(2^l(nht)^{O(1)}\right)$. It can determine whether there is a scheduling to download l data items from D in at most n time slots and at most h channel switches. Service provider can adjust n and h freely to fit their own requirement. We firstly give some preliminaries and then present our algorithms in detail.

Preliminaries

Here we introduce some notions about group algebra which are not often used in algorithm design.

Definition 2 Assume that $x_1, \ldots, x_k$ are variables in group algebra. Then,

1. A *monomial* has format $x_1^{a_1} x_2^{a_2} \ldots x_k^{a_k}$.
2. A *multilinear monomial* is a monomial such that each variable has degree exactly one. For example, $x_3x_5x_6$ is a multilinear monomial, but $x_3x_5^2x_6^3$ is not.
3. For a *polynomial* $p(x_1, \ldots, x_k)$, its *sum of product expansion* is $\sum_j p_j(x_1, \ldots, x_k)$, where each p_j is a monomial, which has a format $c_j x_1^{a_{j_1}} \ldots x_k^{a_{j_k}}$ with c_j respect to its coefficient.
4. $G_2 = (\{0, 1\}, +, \cdot)$ is a field with two elements $\{0, 1\}$ and two operations $+$ and $\cdot$. The addition operation is under the modular of 2 (mod 2).
5. Z_2^k is the group of binary k-vectors. Let w_0 denote the all-zero vector, which is the identity of Z_2^k, and then for every $v \in Z_2^k$, $v^2 = w_0$, $v \cdot w_0 = v$.

The operations between elements in the group algebra are standard.

Algorithm Description

The basic idea of our algebraic algorithm is that for each item $d_i \in D$, where D is the query data set, we create a variable x_i to represent it. Therefore, given $D = \{d_1, d_2, \ldots, d_k\}$, we construct a variable set $X = \{x_1, x_2, \ldots, x_k\}$. We then design a circuit $H_{t,h,n}$ such that a schedule without conflict will be generated by a multilinear monomial in the sum of product expansion of the circuit. The existence of schedules to download all the data items in D from the multiple channel set C is converted into the existence of multilinear monomials of $H_{t,h,n}$. Replace each variable by a specified binary vector which can remove all of the non-multilinear monomials by converting them to zero. Thus, the data retrieval problem is transformed into testing if a multivariate polynomial is zero. It is well known that randomized

algorithms can be used to check if a circuit is identical to zero in polynomial time. Thus, we have the following statements.

Lemma 1 *There is a polynomial time algorithm such that given a channel c_i, a time interval $[t_1, t_2]$, and an integer m, it constructs a circuit of polynomial $P_{i,t_1,t_2,m}$ such that for any subset $D' = \{d_{i_1}, \ldots, d_{i_m}\} \subseteq D$ which has a size of m and is downloadable in the time interval $[t_1, t_2]$ from channel c_i, the product expansion of $P_{i,t_1,t_2,m}$ contains a multilinear monomial $x_{i_1} x_{i_2} \ldots x_{i_m}$.*

Proof We can use a recursive way to compute the circuit $P_{i,t_1,t_2,m}$ in polynomial time.

1. $P_{i,t_1,t_2,0} = 0$.
2. $P_{i,t_1,t_2,1} = \sum_j x_j$, $x_j \subseteq X$, and the corresponding datad_j is entirely in the time interval $[t_1, t_2]$ of channel c_i.
3. $P_{i,t_1,t_2,l+1} = \sum_j x_j \cdot P_{i,t_1,t'_2,l} + P_{i,t_1,t'_2,l+1}$, d_j starts at time $t'_2 + 1$ and ends before time t_2 on channel c_i.

When computing $P_{i,t_1,t_2,l+1}$, x_j multiplies $P_{i,t_1,t'_2,l}$ is based on the case that d_j is downloadable from time $t'_2 + 1$ to t_2 in the final phase, and the other l data items are downloadable before time t'_2. The term $P_{i,t_1,t'_2,l+1}$ is the case that $l + 1$ items are downloaded before time t'_2. Note that the parameter m in $P_{i,t1,t2,m}$ controls the total number of data to be downloaded.

Definition 3 A subset data items $D' = \{d_{i_1}, \ldots, d_{i_m}\} \subseteq D$ is (i, t, h)-downloadable if we can download all data items in D' before time t, the total number of channel switches is at most h, and the last downloaded item is from channel c_i.

Lemma 2 *Given two integers t and h, there is a polynomial time algorithm to construct a circuit of polynomial $F_{i,t,h,m}$ such that for any (i, t, h)-downloadable subset $D' = \{d_{i_1}, \ldots, d_{i_m}\} \subseteq D$, the product expansion of $F_{i,t,h,m}$ contains a multilinear monomial $\left(x_{i_1}, \ldots, x_{i_m}\right) Y$, where Y is a multilinear monomial which does not include any variable in X.*

Proof We still use a recursive way to construct the circuit. Some additional variables are used as needed. Without loss of generality, we assume the data retrieval process start at time 0.

1. $F_{i,t,0,0} = 0$.
2. $F_{i,t,0,1} = P_{i,1,t,1} \cdot y_{i,t,0,1}$.
3. $F_{i,t,h'+1,m'+1} = y_{i,t,h'+1,m'+1,0} \left(\sum_{t'<t} F_{i,t',h'+1,m'} \cdot P_{i,t'+1,t,1}\right) + y_{i,t,h'+1,m'+1,1} \left(\sum_{j \neq i} \sum_{t'<t} F_{i,t'-1,h',m'} \cdot P_{i,t'+1,t,1}\right)$.

Then we can get Lemma 2 immediately.

The computation of $F_{i,t,h'+1,m'+1}$ is based on two cases, and we use two variables, $y_{i,t,h'+1,m'+1,0}$ and $y_{i,t,h'+1,m'+1,1}$, to mark them respectively. We now present an algorithm that involves one layer randomization to determine if there is a schedule to download all the data items in D before time t and with at most h channel switchings.

Theorem 2 *There is an $O\left(2^k (hnt)^{O(1)}\right)$ time randomized algorithm to determine if there is a scheduling to download $|D| = k$ data items before time t and the number of channel switches is at most h, where n is the total number of channels.*

Proof By Lemma 2, we can construct a circuit $H_{t,h,n} = \sum_{i=1}^{n} F_{i,t,h,k}$ in polynomial time. It is easy to see there is a scheduling for downloading the k data items before time t and with h channel switches, if and only if the sum product expansion of $H_{t,h,n}$ has a multilinear monomial $(x_1, \ldots, x_k)Y$.

We can replace each s_i by a vector $w_i = w_0^T + v_i^T$, where w_0 is the all-zero vector of dimension k and v_i is a binary vector of dimension k with its ith element being 1 and all other elements being 0. Assume $k = 3$, we define the following operations:

$$v_a \cdot v_b = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} (a_1 + b_1)(\text{mod}2) \\ (a_1 + b_2)(\text{mod}2) \\ (a_1 + b_3)(\text{mod}2) \end{pmatrix} \quad (1)$$

$$(v_a + v_b) \cdot v_c = v_a \cdot v_c + v_b \cdot v_c \quad (2)$$

By Eqs. 1 and 2, for any k-dimensional binary vector $w' = w_0 + v$, we have $w'^2 = w_0^2 + 2w_0 \cdot v + v^2 = w_0 + 2(w_0 \cdot v) + w_0 = 2(w_0 \cdot v) + 2w_0 = 0$, because of the coefficients are in the field of G_2. The replacement $x_i = w_i (i = 1, \ldots, m)$ makes all the non-multilinear monomials become zero. Meanwhile, all the multilinear monomials remain nonzero. Hence, it is clear that there is a scheduling to download all the data items in D before time t and with at most h channel switchings if and only if $H_{t,h,n|x_i=w_i(i=1,\ldots,k)}$ is a nonzero polynomial. The variables in Y makes it impossible to have cancelation when adding two identical multilinear monomials, which can be generated from different paths with variables in $\{x_1, \ldots, x_k\}$. It is well known that randomized algorithms can be used to check if a circuit is identical to zero in polynomial time [11, 12].

The algorithm generates less than 2^k terms during the computing process since there are at most 2^k distinct binary vectors. Therefore, the computational time is $O\left(2^k (nht)^{O(1)}\right)$.

Example Let $H_1 = x_1 x_2 y_1 + x_2^2 y_2$ and $H_2 = x_1^2 y_1 + x_2^2 y_2$. Consider the replacement $x_1 = w_1 = \binom{0}{0} + \binom{1}{0}$ and $x_2 = w_2 = \binom{0}{0} + \binom{0}{1}$. We have the following steps of operations.

$$\begin{aligned} &H_1|x_1 = w_1, x_2 = w_2 \\ &= \left(\binom{0}{0} + \binom{1}{0}\right)\left(\binom{0}{0} + \binom{0}{1}\right) y_1 \\ &\quad + \left(\binom{0}{0} + \binom{0}{1}\right)^2 y_2 \\ &= \left(\binom{0}{0} + \binom{0}{1} + \binom{1}{0} + \binom{1}{1}\right) y_1 \\ &\quad + \left(\binom{0}{0} + \binom{0}{1}\right) y_2 \\ &= \left(\binom{0}{0} + \binom{0}{1} + \binom{1}{0} + \binom{1}{1}\right) y_1 \\ &\quad + \left(2\binom{0}{0} + 2\binom{0}{1}\right) y_2 \\ &= \left(\binom{0}{0} + \binom{0}{1} + \binom{1}{0} + \binom{1}{1}\right) y_1 + (0+0) y_2 \\ &= \left(\binom{0}{0} + \binom{0}{1} + \binom{1}{0} + \binom{1}{1}\right) y_1 + 0 \\ &= \left(\binom{0}{0} + \binom{0}{1} + \binom{1}{0} + \binom{1}{1}\right) y_1 \\ &\neq 0 \end{aligned}$$

$$\begin{aligned} &H_2|x_1 = w_1, x_2 = w_2 \\ &= \left(\binom{0}{0} + \binom{1}{0}\right)^2 y_1 + \left(\binom{0}{0} + \binom{0}{1}\right)^2 y_2 \\ &= \left(\binom{0}{0} + \binom{1}{0} + \binom{1}{0} + \binom{0}{0}\right) y_1 \\ &\quad + \left(\binom{0}{0} + \binom{0}{1} + \binom{0}{1} + \binom{0}{0}\right) y_2 \\ &= \left(2\binom{0}{0} + 2\binom{1}{0}\right) y_1 + \left(2\binom{0}{0} + 2\binom{0}{1}\right) y_2 \\ &= (0+0) y_1 + (0+0) y_2 \\ &= 0 \end{aligned}$$

H_1 is a polynomial that contains a multilinear monomial. It becomes nonzero after replacement. H_2 is a polynomial that is without multilinear monomials. It becomes zero after the replacement. If we just down a subset of l data items in set D, we have the following theorem that involves two layers of randomization.

Theorem 3 *There is an $O\left(2^l (hnt)^{O(1)}\right)$ time randomized algorithm to determine if there is a scheduling to download l data items from D in at most t time units and at most h channel switches.*

Proof By Lemma 2, we can construct a polynomial $H_{t,h,l} = \sum_{i=1}^{l} F_{i,t,h,l}$ in polynomial time. Replace each x_i with a vector $w_i = w_0^T + v_i^T$, where w_0 is the all-zero vector of dimension l and v_i is a random distinct vector of dimension l. The replacement $x_i = w_i$ $(i = 1, \ldots, k)$ makes all monomials which has non-multilinear monomial at x part become zero.

Therefore, there is a scheduling to download l data items of D before time t and with the number of switches no more than h if and only if $H_{t,h,k|x_i=w_i(i=1,\ldots,k)}$ is not a zero polynomial in the field of G_2. Assume that the product expansion of $H_{t.h,l}$ has a multilinear monomial $(x_{i_1}, \ldots, x_{i_l})Y$, where Y is a multilinear monomial with variables not in $x_1, \ldots, x_k$. For a series of randomly assigned vectors with dimension l: $v_{j_1}, \ldots, v_{j_l}$, the probability that v_{j_i} is a linear combination of $v_{j_1}, \ldots, v_{j_{i-1}}$ is at most $\frac{2^{i-1}}{2^l} = \frac{1}{2^{l-i+1}}$. Therefore, with probability at most $\sum_{i=1}^{l} \frac{1}{2^{l-i+1}}$, v_{j_i} is a linear combination of $v_{j_1}, \ldots, v_{j_{i-1}}$ for some $i \leq l$. When $v_{j_i}, \ldots, v_{j_l}$ are linearly independent, the product of $v_{j_1}, \ldots, v_{j_l}$ is nonzero. Every multilinear monomial in the product expansion has different variables to form Y since it is determined by a unique path to generate the polynomial. Therefore, for those random vectors v_i, every multilinear monomial has a chance at least $1 - \frac{3}{4} = \frac{1}{4}$ to be nonzero. Therefore, if there is a solution, $H_{t,h,k|x_i=w_i(i,1,\ldots,l)}$ with random assignment Y is not zero in the field of G_2 with probability at least $\frac{1}{4}$.

After the replacements, it generates less than $2l$ terms since there are at most $2k$ different vectors for a group of Z_2^l. The coefficient of each vector is kept as a polynomial size circuit. Therefore, the computational time of our algorithm is $O\left(2^l (hnt)^{O(1)}\right)$, and if we run it 30 times, the error rate is $\left(\frac{3}{4}\right)^{30} < 0.0002$.

Applications

Scheduling problem is one of the most fundamental problems in combinatorial optimization, which could model various real-world practical applications. Especially, scheduling problem at client hand side would be very useful for data retrieval problem in wireless data broadcasting or data streaming environment to reduce energy consumption and improve query efficiency. Such problem would also be helpful for parallel query applications in distributed storage systems.

Open Problems

How to download data items efficiently in wireless data broadcast environment can usually be formulized as NP-hard problems with different constraints, and can be categorized into two kinds: single channel process and multiple channel process. The best known result for the former problem is constant-factor approximations, while currently there is no polynomial time approximation scheme (PTAS) for both of them. The results for this problem is also helpful for parallel data retrieval problem in distributed data storage system and cloud system.

Experimental Results

Many literature proposed experimental results for scheduling problem in data broadcasting. Shi et al. [6] simulated a base station with n broadcast channels and 10,000 items, each of size 1KB, and multiple clients with various requests of data. The access probability of the database follows Zipf distribution, n varies from 5 to 30, the number of antennae varies from 1 to 10, and the size of a request varies from 10 to 1,000. For each experiment, they generated 100 requests to get the average access latency and number of switchings during data retrieval. Lv et al. [7, 8] constructed two types of broadcast programs: special data broadcast without channel switching time (SDB) and general data broadcast with channel switching time (GDB). In both types of programs, they simulated a base station with n broadcast channels; the bandwidth of each channel is 1Mbit/sec. The database to be broadcasted has N data items, each of size 512 bytes. The time duration is denoted by t. The data items of query data set D

is generated with access probabilities following the Zipf distribution.

URLs to Code and Data Sets

Shi et al. [6] provided the program for users to test parameter setting for their own data sets and available channels (http://theory.utdallas.edu/dataengineering).

Cross-References

- Efficient Polynomial Time Approximation Scheme for Scheduling Jobs on Uniform Processors

Recommended Reading

1. Acharya S, Alonso R, Franklin M, Zdonik S (1995) Broadcast disks: data management for asymmetric communication environments. In: The ACM special interest group on management of data conference (SIGMOD), San Jose, 22–25 May 1995, pp 199–210
2. Yee W, Navathe S, Omiecinski E, Jermaine C (2002) Efficient data allocation over multiple channels at broadcast servers. IEEE Trans Comput 51(10):1231–1236
3. Ardizzoni E, Bertossi A, Pinotti M, Ramaprasad S, Rizzi R, Shashanka M (2005) Optimal skewed data allocation on multiple channels with flat broadcast per channel. IEEE Trans Comput 54(5):558–572
4. Anticaglia S, Barsi F, Bertossi A, Iamele L, Pinotti M (2008) Efficient heuristics for data broadcasting on multiple channels. Wirel Netw 14(2):219–231
5. Kenyon C, Schabanel N (1999) The data broadcast problem with non-uniform transmission times. In: Proceedings of the tenth annual ACM-SIAM symposium on discrete algorithms (SODA), Baltimore, 17–19 Jan 1999, pp 547–556
6. Shi Y, Gao X, Zhong J, Wu W (2010) Efficient parallel data retrieval protocols with MIMO antennae for data broadcast in 4G wireless communications. In: The 21st international conference on database and expert systems applications (DEXA), Bilbao, 30 Aug–3 Sept 2010, pp 80–95
7. Lu Z, Shi Y, Wu W, Fu B (2012) Efficient data retrieval scheduling for multi-channel wireless data broadcast. In: International conference on computer communications (INFOCOM), Orlando, 25–30 Mar 2012, pp 891–899
8. Lu Z, Shi Y, Wu W, Fu B (2014) Data retrieval scheduling for multi-Item requests in multi-channel wirelessBroadcast environments. IEEE Trans Mobile Comput 13(4):752–765
9. Gao X, Lu Z, Wu W, Fu B (2011) Algebraic algorithm for scheduling data retrieval in multi-channel wireless data broadcast environments. In: The 6th annual international conference on combinatorial optimization and applications (COCOA), Zhangjiajie, 4–6 Aug 2011, pp 74–81
10. Gao X, Lu Z, Wu W, Fu B (2013) Algebraic data retrieval algorithms for multi-channel wireless data broadcast. Theor Comput Sci 497:123–130
11. Williams R (2009) Finding paths of length k in $O*(2^k)time$. Inf Process Lett 109(6):315–318
12. Koutis I (2008) Faster algebraic algorithms for path and packing problems. In: The 2008 international colloquium on automata, languages and programming (ICALP), Reykjavik,6– 13 July 2008, pp 575–586

Scheduling with a Reordering Buffer

Matthias Englert[1] and Matthias Westermann[2]
[1]Department of Computer Science, University of Warwick, Coventry, UK
[2]Department of Computer Science, TU Dortmund University, Dortmund, Germany

Keywords

Machine scheduling; Minimum makespan scheduling; Online algorithms; Reordering buffer; Sorting buffer

Years and Authors of Summarized Original Work

2002; Räcke, Sohler, Westermann
2009; Gamzu, Segev
2010; Englert, Räcke, Westermann
2011; Adamaszek, Czumaj, Englert, Räcke
2011; Dósa, Epstein
2013; Avigdor-Elgrabli, Rabani
2014; Englert, Özmen, Westermann

Problem Definition

The problem known as the reordering buffer problem or as the sorting buffer problem is concerned with sorting a sequence of colored

items according to their color using a limited size buffer. More precisely the items are to be processed and arrive one by one. Arriving items must first be placed into a buffer which can hold up to k items. Once the buffer is completely filled, an algorithm has to free space by selecting one of the items in the buffer for processing and removing that item from the buffer. After items stop arriving, the remaining items in the buffer may be processed in any order. Whenever an item is processed that has a different color than the item processed in the step before, this generates a cost of 1. The objective is to minimize the total cost.

Metric Space Generalization

This problem can be further generalized. Items, instead of having a color, correspond to points in a metric space. A single server must process all items. In order to process an item, the server has to move to the corresponding point in the metric space. At every point, the server has to chose one of the first k as of yet unprocessed items for processing and move the server accordingly. The goal is to minimize the total distance the server travels.

The uniform metric in which any two points either have distance 0 or distance 1 from one another corresponds to the original "color sorting" setting. Other metrics studied include line metrics and "star" metrics which are the distance metrics over weighted undirected trees of diameter 2.

Block Operation Setting

Another variant is the so-called block operation setting. Once again, the input consists of a sequence of colored items. The first k items are placed in a buffer. In each step, an algorithm selects one of the colors and processes all items of that color currently stored in the buffer, incurring a cost of 1. This is called a block operation. The processed items are removed from the buffer and replaced with the next items from the input sequence (if there are any). The goal is once again to minimize the total cost.

The difference between this block operation setting and the original setting is most pronounced for an input sequence consisting of ℓ items of a single color. While in the original setting such a sequence would not produce any cost, the cost in the block device setting would be ℓ/k since only k items can be processed per block operation.

Minor Variants Found in the Literature

In some cases, there are slight differences in which these problems are defined in the literature. Does a cost incur for the first ever processed item or, similarly, is the first position of the server in the metric space part of the input or does the algorithm get to chose that position (without incurring any cost)? Does an item first have to be placed in the buffer or can the algorithm process an arriving item directly, thereby bypassing the buffer? Do we need to remove the remaining items in the buffer once new items stopped arriving? It turns out however that these details are inconsequential for most of the results we are interested in.

Key Results

The main focus of study in the area of scheduling with a reordering buffer has been on online algorithms. In the online setting, the algorithm's decisions have to be based solely on the items that arrived in the past and must not depend on items arriving in the future. An online algorithm is called c-competitive if the cost of the algorithm is at most c times that of an optimal off-line solution.

The Online Problem

Deterministic Algorithms

Räcke, Sohler, and Westermann [29] first introduced the problem for the uniform metric

and gave a $O(\log^2 k)$-competitive online algorithm. After further improvements to $O(\log k)$ [18] and $O(\log k / \log\log k)$ [6] eventually, an $O(\sqrt{\log k})$-competitive online algorithms was designed [2]. This is almost optimal since a lower bound of $\Omega(\sqrt{\log k / \log\log k})$ is known [2]. Many of these upper bounds also generalize from the uniform metric to star metrics.

While the proof techniques for some of these results differ significantly, the basic idea behind all the algorithms is the same. As long as the buffer contains an item of the same color as the previously processed item, such an item is processed next. Otherwise, the algorithm has to pick a different color and performs a color switch which incurs a cost of 1. In order to decide which color to switch to, each color is assigned a "penalty" counter which is initially set to 0 and is reset to 0 whenever the color is selected for processing. If there is a color with penalty at least k, then an item with that color is selected next. Otherwise, an arbitrary color is selected and the penalty counters for each color are increased proportional to the number of items of that color that are stored in the buffer. For the $O(\sqrt{\log k})$-competitive algorithm, instead of picking an arbitrary color, a more sophisticated rule is used.

Randomized Algorithms

Randomized algorithms can achieve much smaller competitive ratios. The first randomized algorithm with a competitive ratio of $O(\log\log k)$ was given for the block operation model [3]. Shortly afterward, a randomized algorithm with the same competitive ratio was presented for the original model [8]. This is best possible since a matching lower bound is known [2]. These randomized algorithms are based on online primal-dual LP schemes [11].

Other Metric Spaces

Apart from the uniform metric, line metrics have received some attention. After a randomized $O(\log^2 n)$-competitive online algorithm for n equally spaced points on a line [27], an improved deterministic $O(\log n)$-competitive algorithm was given [24]. A deterministic $O(\log N \log\log N)$-competitive algorithm for a line metric with not necessarily equally spaced points was also given, but here N refers to the number of items in the input sequence [24]. An easy observation, however, shaves off the $\log\log N$ factor and improves the analysis to show $O(\log N)$ competitiveness (Cygan, Mucha, Private communication, 2011). There is still a significant gap between this upper bound and the best known lower bound of about 2.154 [24].

For general metric spaces, a randomized $O(\log^2 k \log n)$-competitive online algorithm is known, where n is the number of points in the metric space [19]. This result is based on a deterministic algorithm for trees that is turned into an algorithm for general metrics by using a metric embedding [23].

Stochastic Inputs

In a setting where the input is not adversarial constructed but where the colors of the items are drawn i.i.d. from an unknown distribution, a constant competitive ratio is achievable [22]. This result also holds when the colors of the items are fixed by an adversary but the order in which the items arrive is random. The proof is based on the fact that a constant competitive online algorithm is known for adversarial inputs, if the online algorithm can use a buffer that is four times as large as the one used by the optimal off-line algorithm. In the stochastic input setting, this difference in buffer size does not lead to significantly different cost, i.e., the cost of an optimal algorithm with buffer size k is only by a constant factor larger than the cost of an optimal algorithm with buffer size $4k$. This is not true for adversarial inputs [1].

The Off-Line Problem

The reordering buffer problem is NP-hard [5, 12] for the uniform metric, and the complexity for line metrics is unknown. Therefore, several papers focus on approximating the off-line scenario.

A constant factor approximation is known for the uniform metric [7]. For star metrics, the best known approximation factor of $O(\log\log k\gamma)$ is

achieved by a randomized algorithm, where γ denotes the ratio of the maximum to the minimum weight [25]. Both results are based on the intricate rounding of the solution to an LP relaxation of the corresponding problem.

Bicriteria Approximations

For more general metric spaces, the best approximation ratios are achieved by bicriteria approximations, i.e., the approximation algorithm can make use of more buffer capacity than an optimal algorithm. For metric spaces given by the distance metric over a weighted undirected tree, a bicriteria approximation with approximation factor 9 to cost and $4 + 1/k$ to buffer size is known [10]. Using metric embeddings [23], this implies a randomized bicriteria approximation with approximation factor $O(\log n)$ to cost and $O(1)$ to buffer size, where n denotes the number of points in the metric space.

The Maximization Problem

In the maximization version of the problem, the goal is to maximize the total cost savings that result from reordering the input sequence. In terms of an optimal solution, the minimization and maximization scenario are identical. However, in terms of approximation, they behave quite differently in the sense that a c-approximate solution for the maximization problem usually has very different cost from a c-approximate solution for the minimization problem. For the uniform metric, the first result was an approximation algorithm with an approximation factor of 20 [28]. This was later improved to a factor of 9 [9].

Online Minimum Makespan Scheduling

Reordering buffers have also been studied in connection with other scheduling problems, in particular online minimum makespan scheduling. As in the classic problem without reordering, the input consists of a sequence of jobs with processing times, and a scheduling algorithm has to assign the jobs to m parallel machines, with the objective to minimize the makespan, which is the time it takes until all jobs are processed. However, it is not required that each arriving job has to be assigned immediately to one of the machines. A reordering buffer can be used to reorder the input sequence of jobs. At each point in time, the reordering buffer contains the first k jobs of the input sequence that have not been assigned so far. An online scheduling algorithm has to decide which job to assign to which machine next. Upon its decision, the corresponding job is removed from the buffer and assigned to the corresponding machine, and thereafter the next job in the input sequence takes its place.

Non-preemptive Scheduling

For non-preemptive scheduling, Englert, Özmen, and Westermann [20] give, for m identical machines, a tight bound on the competitive ratio. Depending on m, the achieved competitive ratio lies between 4/3 and 1.4659. This optimal ratio is achieved with a buffer of size of at most $\lceil 2.5 \cdot m \rceil + 2$. They show that larger buffer sizes do not result in an additional advantage and that a buffer of size $\Omega(m)$ is necessary to achieve this competitive ratio. This improves upon an optimal algorithm for two identical machines [26].

Further, they present several algorithms for different buffer sizes. In addition, for m uniformly related machines, they give a scheduling algorithm that achieves a competitive ratio of 2 with a reordering buffer of size m.

Subsequently to [20], a variety of related papers appeared (compare, e.g., [4, 14–16, 21]). For 2 uniformly related machines with speed ratio $s \geq 1$, it is shown that, for any $s > 1$, a buffer of size 3 is sufficient to achieve an optimal competitive ratio, and in the case $s \geq 2$, a buffer of size 2 already allows to achieve an optimal ratio [15].

Job Migrations

The results of [20] can be generalized to the problem of online minimum makespan scheduling with job migrations, i.e., where no reordering buffer is available, but a limited number of job reassignments may be performed. For m identical

machines, the same competitive ratio as in [20] can be achieved [4]. The algorithm uses, for $m \geq 11$, at most $7m$ migration operations and, for smaller m, $8m$ to $10m$ migration operations. A number of papers consider similar models (compare, e.g., [13, 17, 30, 31]).

Preemptive Scheduling

For preemptive scheduling on m identical machines, tight bounds on the competitive ratio can be achieved for any m. This bound is $4/3$ for even values of m and slightly lower for odd values of m [16]. A buffer of size $\Theta(m)$ is sufficient to achieve this bound, but a buffer of size $o(m)$ does not reduce the best overall competitive ratio $e/(e-1)$ that is known for the case without reordering [16].

Cross-References

- Approximation Schemes for Makespan Minimization
- Efficient Polynomial Time Approximation Scheme for Scheduling Jobs on Uniform Processors
- Online Preemptive Scheduling on Parallel Machines

Recommended Reading

1. Aboud A (2008) Correlation clustering with penalties and approximating the reordering buffer management problem. Master's thesis, Computer Science Department, The Technion—Israel Institute of Technology
2. Adamaszek A, Czumaj A, Englert M, Räcke H (2011) Almost tight bounds for reordering buffer management. In: Proceedings of the 43rd ACM symposium on theory of computing (STOC), San Jose, pp 607–616
3. Adamaszek A, Czumaj A, Englert M, Räcke H (2012) Optimal online buffer scheduling for block devices. In: Proceedings of the 44th ACM symposium on theory of computing (STOC), New York, pp 589–598
4. Albers S, Hellwig M (2012) On the value of job migration in online makespan minimization. In: Proceedings of the 20th European symposium on algorithms (ESA), Ljubljana, pp 84–95
5. Asahiro Y, Kawahara K, Miyano E (2012) NP-hardness of the sorting buffer problem on the uniform metric. Discret Appl Math 160(10–11):1453–1464
6. Avigdor-Elgrabli N, Rabani Y (2010) An improved competitive algorithm for reordering buffer management. In: Proceedings of the 21st ACM-SIAM symposium on discrete algorithms (SODA), Austin, pp 13–21
7. Avigdor-Elgrabli N, Rabani Y (2013) A constant factor approximation algorithm for reordering buffer management. In: Proceedings of the 24th ACM-SIAM symposium on discrete algorithms (SODA), New Orleans, pp 973–984
8. Avigdor-Elgrabli N, Rabani Y (2013) An optimal randomized online algorithm for reordering buffer management. In: Proceedings of the 54th IEEE symposium on foundations of computer science (FOCS), Berkeley, pp 1–10
9. Bar-Yehuda R, Laserson J (2007) Exploiting locality: approximating sorting buffers. J Discret Algorithms 5(4):729–738
10. Barman S, Chawla S, Umboh S (2012) A bicriteria approximation for the reordering buffer problem. In: Proceedings of the 20th European symposium on algorithms (ESA), Ljubljana, pp 157–168
11. Buchbinder N, Naor J (2009) The design of competitive online algorithms via a primal-dual approach. Found Trends Theor Comput Sci 3(2–3):93–263
12. Chan H, Megow N, Sitters R, van Stee R (2012) A note on sorting buffers offline. Theor Comput Sci 423:11–18
13. Chen X, Lan Y, Benko A, Dósa G, Han X (2011) Optimal algorithms for online scheduling with bounded rearrangement at the end. Theor Comput Sci 412(45):6269–6278
14. Ding N, Lan Y, Chen X, Dósa G, Guo H, Han X (2014) Online minimum makespan scheduling with a buffer. Int J Found Comput Sci 25(5):525–536
15. Dósa G, Epstein L (2010) Online scheduling with a buffer on related machines. J Comb Optim 20(2):161–179
16. Dósa G, Epstein L (2011) Preemptive online scheduling with reordering. SIAM J Discret Math 25(1):21–49
17. Dósa G, Wang Y, Han X, Guo H (2011) Online scheduling with rearrangement on two related machines. Theor Comput Sci 412(8–10):642–653
18. Englert M, Westermann M (2005) Reordering buffer management for non-uniform cost models. In: Proceedings of the 32nd international colloquium on automata, languages and programming (ICALP), Lisbon, pp 627–638
19. Englert M, Räcke H, Westermann M (2010) Reordering buffers for general metric spaces. Theory Comput 6(1):27–46
20. Englert M, Özmen D, Westermann M (2014) The power of reordering for online minimum makespan scheduling. SIAM J Comput 43(3):1220–1237
21. Epstein L, Levin A, van Stee R (2011) Max-min online allocations with a reordering buffer. SIAM J Discret Math 25(3):1230–1250

22. Esfandiari H, Hajiaghayi M, Khani MR, Liaghat V, Mahini H, Räcke H (2014) Online stochastic reordering buffer scheduling. In: Proceedings of the 41st international colloquium on automata, languages and programming (ICALP), Copenhagen, pp 465–476
23. Fakcharoenphol J, Rao SB, Talwar K (2004) A tight bound on approximating arbitrary metrics by tree metrics. J Comput Syst Sci 69(3):485–497
24. Gamzu I, Segev D (2009) Improved online algorithms for the sorting buffer problem on line metrics. ACM Trans Algorithms 6(1):15:1–15:14
25. Im S, Moseley B (2014) New approximations for reordering buffer management. In: Proceedings of the 25th ACM-SIAM symposium on discrete algorithms (SODA), Portland, pp 1093–1111
26. Kellerer H, Kotov V, Speranza MG, Tuza Z (1997) Semi on-line algorithms for the partition problem. Oper Res Lett 21(5):235–242
27. Khandekar R, Pandit V (2010) Online and offline algorithms for the sorting buffers problem on the line metric. J Discret Algorithms 8(1):24–35
28. Kohrt JS, Pruhs K (2004) A constant factor approximation algorithm for sorting buffers. In: Proceedings of the 6th Latin American symposium on theoretical informatics (LATIN), Buenos Aires, pp 193–202
29. Räcke H, Sohler C, Westermann M (2002) Online scheduling for sorting buffers. In: Proceedings of the 10th European symposium on algorithms (ESA), Rome, pp 820–832
30. Tan Z, Yu S (2008) Online scheduling with reassignment. Oper Res Lett 36(2):250–254
31. Wang Y, Benko A, Chen X, Dósa G, Guo H, Han X, Sik-Lányi C (2012) Online scheduling with one rearrangement at the end: revisited. Inform Process Lett 112(16):641–645

Secretary Problems and Online Auctions

MohammadHossein Bateni
Google Inc., New York, NY, USA

Keywords

Competitive analysis; Knapsack constraint; Matroid constraint; Mechanism design; Online algorithm; Secretary problem; Strategyproof; Submodularity; Truthfulness

Years and Authors of Summarized Original Work

2004; Hajiaghayi, Kleinberg, Parkes
2008; Babaioff, Immorlica, Kempe, Kleinberg
2013; Bateni, Hajiaghayi, Zadimoghaddam

Problem Definition

The classic secretary problem, a prime example of stopping theory, has been studied extensively in the computer science literature. Consider the scenario where an employer is interested in hiring one secretary out of a pool of candidates. The difficulty is that, although the employer does not know the utility of a candidate before she is interviewed, the irrevocable hiring decision for each candidate has to be made right after the interview and prior to interviewing the subsequent candidates. The goal is nonetheless to pick the best candidate or maximize the probability of achieving this.

Optimization Angle

The above scenario is hopeless from an algorithmic point of view since an adversarial input makes it impossible to hire the best candidate. We can take either of two paths to make the problem tractable: restrict the set of utilities or the arrival order of candidates. The former path yields, for instance, the stochastic variant of the problem. However, we follow the second idea here that leads to the classic secretary problem. The extra assumption, then, is that the candidates arrive in a random error; i.e., although each candidate may have an arbitrary adversarial utility, every permutation of the candidates is equally likely to be the arrival order.

A folklore solution to the problem, often attributed to [3], is to look into the first $\frac{1}{e}$ fraction of the candidates (called the "tuning set"), without giving them any offers, and then hire the first candidate with utility more than every one in the tuning set. It is not difficult to show that this approach hires the best candidate with probability

at least $\frac{1}{e}$. Indeed, it is known that this is the best possible performance.

There are two questions to be answered, once we extend the problem to multiple secretaries.

1. *What subsets of secretaries can be hired together?* The simplest answer is to allow at most k secretaries to be hired. Alternately, we can place (several) knapsack and/or matroid constraints on the feasible set. The former assigns a cost to each hire – say, the requested salary – that is to be paid out of a given budget. The latter permits only those combinations that form an independent set according to a given matroid. It is easy to see that both generalize the cardinality constraint.
2. *How do we compute the utility of a set?* The utility of a set can be defined as the sum of the utilities of individual secretaries in the set. More generally, a submodular or subadditive function may be employed to describe the utility of a set.

We then attempt to hire a feasible set of secretaries of maximum expected utility.

Mechanism Design Angle

Mechanism design literature has looked at this problem from a slightly different angle. In this setting, the players that arrive in a random order declare a *bid* – i.e., how much they value the item being sold – and then the seller decides who should get the item (or items) and how much they should be charged. Such decisions are to be taken irrevocably as in the optimization problem discussed above.

The players can play strategically, though, by declaring higher or lower bids in order to increase their chances of winning the item or to reduce the price they pay. In addition, they may declare their arrival/departure time untruthfully to achieve a better result. We want to design a "truthful" auction that precludes such undesirable outcomes. Although we allow the player to declare any nonnegative bid (if it is in her favor), we do not let them state an arrival time that is earlier than their actual one. (Presence intervals may be overlapping and/or nested.) We say that a mechanism is value-strategyproof if no player can benefit from declaring a bid different from her real value. Similarly the mechanism is called time-strategyproof if there is no benefit in stating the arrival/departure times untruthfully. We look for mechanisms that are both time- and value-strategyproof.

Key Results

Optimization

Kleinberg [6] studies the multiple-choice generalization where the goal is to hire k candidates, whose total utility (defined as the sum of the individual utilities) is maximized. He presents a tight performance guarantee of $1 + \Theta\left(\frac{1}{\sqrt{k}}\right)$ for the problem. In the case of $k = 1$, this is equivalent to the classic secretary problem. (The nontrivial direction follows from a construction where the utilities are hugely different.) Kleinberg's algorithm partitions the set of candidates into two (almost) equal pieces, recursively hires $\frac{k}{2}$ secretaries in the first, sets the threshold for the second piece by looking at the solution to the first piece, and picks as many as $\frac{k}{2}$ secretaries in the second piece who are better than threshold.

Babaioff et al. [1] look at the generalization where there is a restriction on the set of candidates that can be hired together; the restriction is in the form of a matroid. They present an $O(\log n)$ competitive ratio in this case along with improved bounds when the matroid has a special form. Their general matroid algorithm partitions the items into logarithmically many sets of almost equal utility and focuses (randomly) on one such set, which reduces the problem into that of maximizing the cardinality of the solution (solved via the greedy method).

The case of submodular utilities is discussed in Bateni et al. [2]: several matroid or knapsack constraints can be placed on the set of feasible candidates, and the total utility of a set is computed by a submodular function of the participating candidates. They provide constant competitive ratios as long as a fixed number of knapsack

constraints are present. When (a constant number of) matroid constraints are involved, too, their performance guarantees grow to $O(\log^2 k)$ where k is the rank of the matroid. They divide the input into different pieces where at most one secretary should be picked from each, not losing too much utility in the process. As a result, the submodular function collapses to an additive one within each piece (by taking the marginal values of secretaries with respect to the current solution). The classic algorithm is then used inside each piece. The main idea behind the matroid algorithm is that we only need to show that, whatever choices we have already committed to, there are enough options left that can appropriately augment the current solution. The argument goes by proving the existence of a magical solution with k' secretaries any of whose $\frac{k'}{2}$-size subsets has significant contribution (say, at least a $\frac{1}{\log k}$ fraction of the optimum) in the submodular function. Had we known k', a simple greedy algorithm would have sufficed to find a solution similar to the magical set. At the cost of another factor $O(\log k)$, we can guess k'.

Furthermore, Bateni et al. show that subadditive utility functions make the problem much more difficult. In particular, they provide matching $\Theta(\sqrt{k})$ competitive ratios.

Mechanism Design

The Dynkin's algorithm for the classic secretary problem can be readily turned into an auction: set the price after observing the tuning set, and then sell to anyone with a higher bid. This mechanism is not truthful, though, since high-bid players spanning across the time threshold have an incentive to declare later arrival time (i.e., after the threshold); this way, they will win the item but do not set the price.

Nevertheless, Hajiaghayi et al. [5] show how one can modify the mechanism slightly to make it truthful: after the threshold, consider the option of selling the item to the agent with the highest bid so far – if she is still present – and charge her the second-highest bid so far. Their method achieves constant competitiveness for both efficiency and revenue. Their $1/e$ competitiveness for efficiency is best possible since it generalizes the optimization problem; however, when comparing the revenue to that achieved by the Vickrey auction, their upper bound of $1/e^2$ for competitiveness fares against a lower bound of $1/e$. (It is possible, they show, to modify the mechanism slightly to trade efficiency loss for revenue gain; for instance, simultaneous 4 competitiveness for both objectives is possible.)

The general idea for the transformation is to define a "tuning period" where the price is set for everyone. Then, not only a simple auction-like mechanism is employed in the "hiring phase" to obtain a strategyproof mechanism, but also extra care should be given to the "transition phase" (from tuning to hiring) so as not to incentivize untruthful declaration of arrival time for those whose presence spans the transition. The same approach can be applied to the multiple-choice secretary problem to obtain constant-factor competitive mechanisms (for efficiency and revenue), but this bound is far from the one achieved in the optimization setting by Kleinberg [6].

Open Problems

Though there has been some improvements on the matroid case, we still do not know which cases are hard and admit no constant-factor competitive ratio. For submodular utilities (and simple cardinality constraints), in particular, there is a gap between $\left(1-\frac{1}{e}\right)/(e+1)$ algorithmic result [4] and the $1-\frac{1}{e}$ $\left(\text{or } 1-\frac{1}{\sqrt{k}}\right)$ target known for linear utilities.

Cross-References

▸ Algorithmic Mechanism Design

Recommended Reading

1. Babaioff M, Immorlica N, Kempe D, Kleinberg R (2008) Online auctions and generalized secretary problems. SIGecom Exch 7(2):1–11. doi:http://doi.acm.org/10.1145/1399589.1399596

2. Bateni M, Hajiaghayi MT, Zadimoghaddam M (2013) Submodular secretary problem and extensions. ACM Trans Algorithms 9(4):32
3. Dynkin EB (1963) The optimum choice of the instant for stopping a markov process. Sov Math Dokl 4:627–629
4. Feldman M, Naor J, Schwartz R (2011) Improved competitive ratios for submodular secretary problems (extended abstract). In: APPROX, Princeton, pp 218–229
5. Hajiaghayi MT, Kleinberg R, Parkes DC (2004) Adaptive limited-supply online auctions. In: EC, New York, pp 71–80
6. Kleinberg R (2005) A multiple-choice secretary algorithm with applications to online auctions. In: SODA, Vancouver, pp 630–631

Self-Assembly at Temperature 1

Pierre-Étienne Meunier
Le Laboratoire d'Informatique Fondamentale de Marseille (LIF), Aix-Marseille Université, Marseille, France

Keywords

Computational geometry; Concurrency; Self-assembly

Years and Authors of Summarized Original Work

2000; Rothemund, Winfree
2009; Doty, Patitz, Summers
2011; Cook, Fu, Schweller
2014; Meunier, Patitz, Summers, Theyssier, Winslow, Woods

Problem Definition

Temperature 1 (also called *noncooperative*) self-assembly is a model of the formation of structures by growing and branching tips. Despite its ubiquity in nature (in systems such as plants and mycelium or percolation processes) and apparent dynamic simplicity, it is one of the least understood models of self-assembly.

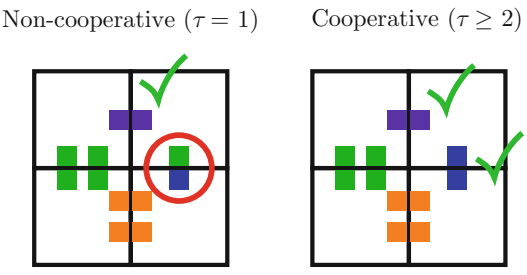
This model was introduced in a broader framework called the *abstract Tile Assembly Model* (aTAM) [10]. In the aTAM, we consider *tile assembly systems*, which are defined by a finite set T of square or cubic *tile types*, an initial *seed assembly* σ (one or more tiles stuck together), and an integer temperature $\tau = 1, 2, 3, \ldots$. All tiles, on each of their sides, have *glues* with an integer *color* and an integer *strength*.

The dynamics of tile self-assembly starts from the seed assembly and proceeds one tile at a time, asynchronously and nondeterministically. A tile can stick to an existing assembly if it can be placed so that the sum of the strengths on its sides matching the existing assembly is at least the temperature. In the case of temperature 1, this means that tiles can be placed as soon as one of their sides matches the existing assembly. At higher temperatures, we can require that newly placed tiles match *several* of their neighbors to attach.

Ultimately, after a countable (potentially infinite) number of steps, no tile can be added to the assembly, in which case we call it *terminal*. Like in Wang tilings, tiles cannot overlap, be rotated, or be flipped. However, tiles can have *mismatches* with their neighbors (Fig. 1).

Key Results

The first comparison between temperatures 1 and 2 was shown by Rothemund and Winfree [9], with the motivation of computing and efficiently



Self-Assembly at Temperature 1, Fig. 1 In the non-cooperative model, tiles can attach as soon as one side matches the neighborhood

building arbitrary shapes at the nanoscale. In this context, the generally accepted definition of "efficient" is *with significantly less tile types than the size of the output*.

Assembling Simple Shapes Efficiently

The first step toward these goals is the programming of simple shapes like squares or trees. At temperature ≥ 2, constructions with Turing machines can be used to show the following bound:

Theorem 1 (from [9]) *The smallest two-dimensional tileset T_n producing only squares of size $n \times n$ from a single-tile seed is of size $\theta\left(\frac{\log n}{\log \log n}\right)$.*

The smallest number of tile types that can assemble exactly a set of shapes is called the *tile complexity* of that set. In the noncooperative model, the following upper bound is known:

Theorem 2 (from [9]) *For all integer n, there is a tileset T_n, of size $2n - 1$, that produces only squares of size $n \times n$ from a single-tile seed.*

Whether this upper bound is optimal is still one of the major open problems of the model, and little progress has been made since its identification. The real motivation behind this question is whether we (or natural systems) can perform useful computations with this model.

Finding the smallest tileset for assembling an input shape can also be treated as an optimization problem: see Adleman et al. [1] for the case of tree shapes.

The Role of Geometry

A partial answer to this question was found by Cook, Fu, and Schweller [2], who tried to "fake" cooperation by blocking the growth of some parts of the assembly. They introduced two different ways to do this: removing the planarity constraint and allowing errors.

In both cases, "faking" cooperation means producing the same assemblies as a temperature 2 tile assembly systems up to rescaling by a constant factor.

Three-Dimensional Noncooperative Self-Assembly

In three dimensions, temperature 1 self-assembly is able to simulate Turing computations:

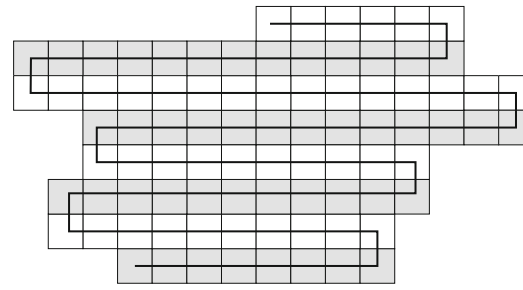
Theorem 3 (from [2]) *There is a three-dimensional tileset T such that for all Turing machine $\mathcal{M}$ and input $x \in \mathbb{N}$, there is a computable seed assembly $\sigma_{\mathcal{M},x}$ and a tile $t \in T$, such that all terminal assemblies of $(T, \sigma_{\mathcal{M},x}, 1)$ contain t if and only if $\mathcal{M}$ accepts input x.*

The construction simulates a Turing-universal cooperative tile assembly system called a *zigzag system*, in which rows grow on top of each other, alternatively to the left and to the right, using cooperation to copy and update the previous row (Fig. 2).

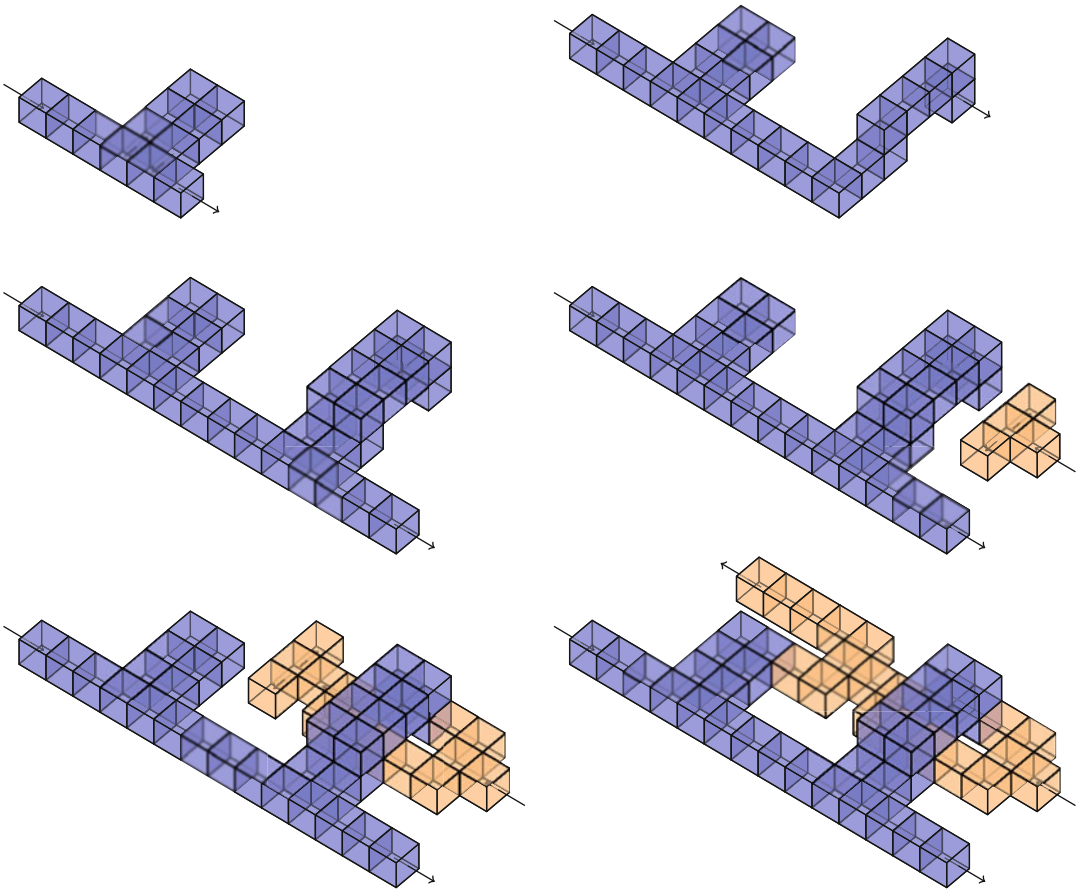
The idea is pictured on Fig. 3. A "main" path grows on each row, building "bridges" and "blockers" (in blue on Fig. 3) that encode bits. These bits can be read by the next row: before reading a bit, the main path (in orange on Fig. 3) of the row forks into two branches, respectively probing for a bridge (encoding a 1) and a blocker (encoding a 0). Exactly one branch passes through and can accumulate successive bits in its state, until a full tile has been read. Then, it rewrites bits encoding the next tile for the row above.

Allowing Erroneous Blocking

Adapting the mechanism used in the 3D construction to the planar case is widely conjectured impossible [9], because allowing the "wrong" branch to grow and collide against a previous



Self-Assembly at Temperature 1, Fig. 2 An example zigzag system (Figure from [2])



Self-Assembly at Temperature 1, Fig. 3 Bit selection in 3d

part of the assembly, in Fig. 3, *encloses* the other "correct" branch inside a finite portion of the plane.

However, it becomes possible if we consider a stochastic assembly schedule, where at each time step, exactly one tile attaches, and all tiles that can attach do so with equal probability. If we repeat the above construction k times consecutively, only one needs to succeed. We can therefore lower the probability of failure of each bit selection to 2^{-k}:

Theorem 4 (from [2]) *For all $\varepsilon > 0$ and all zigzag tile systems $\mathcal{T} = (T, s, 2)$, whose producible assemblies have size at most some constant r, there is a planar temperature 1 probabilistic tile assembly system $\mathcal{S}$ that simulates $\mathcal{T}$ without error with probability at least $1 - \varepsilon$.*

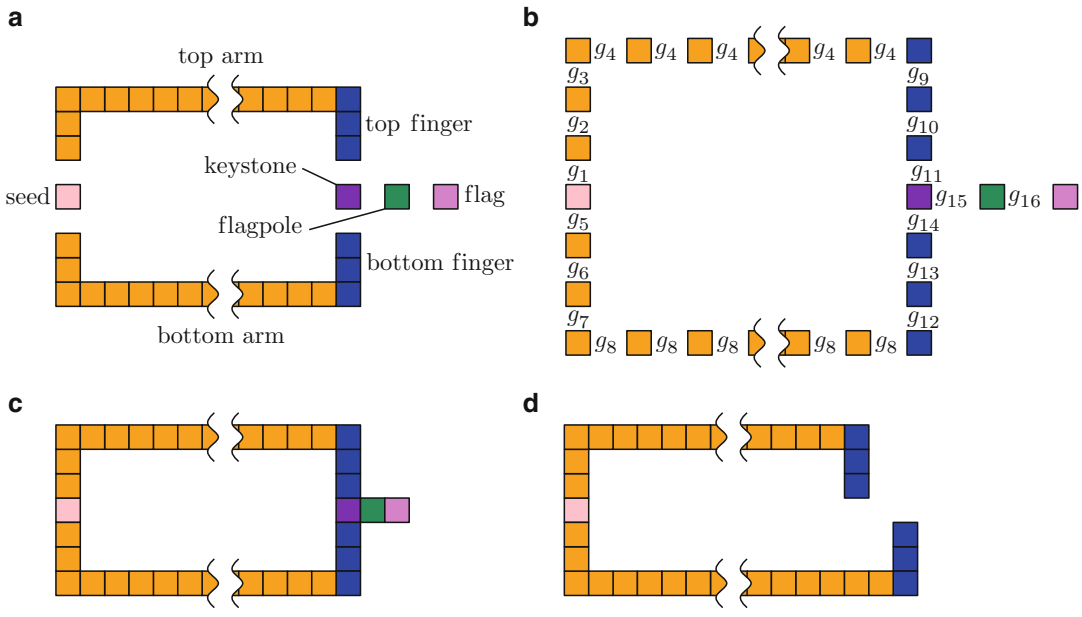
Of course, this construction means that the number of tile types and scaling factor will increase by a factor depending on ε and r.

Simulation up to Rescaling

One of the latest developments of tile assembly is the notion of *intrinsic universality* [3, 4, 11], a notion of simulation by rescaling only between tile assembly systems.

This idea is useful in particular to compare different models, because it provides qualitative properties to check, as opposed to quantitative properties such as tile complexity. The general argument is:

- At temperature 2 in two dimensions, there is a tileset known from [4] to be able to simu-



Self-Assembly at Temperature 1, Fig. 4 Tile assembly system $\mathcal{T}$ (Figure from [7])

late any other tile assembly system, modulo rescaling.

- However, there is a tile assembly system $\mathcal{T}$ that no tileset in model X (in our case, temperature 1) can simulate without errors.
- Therefore, model X is not as powerful as planar temperature 2.

This argument was used, for instance, to prove the first separation result between temperature 2 and the fully general model of temperature 1 [7]:

Theorem 5 (from [7]) *There is a planar (temperature 2) tile assembly system $\mathcal{T}$ (whose productions are pictured on Fig. 4) that no (two- or three- dimensional) tile assembly system $(A, \alpha, 1)$ can simulate up to rescaling.*

The proof uses a combinatorial argument (called the *window movie lemma*) to show that if there were a tile assembly system simulating all productions of $\mathcal{T}$, then it would also be able to produce other "illegal" assemblies (see Fig. 5) that do not represent any of $\mathcal{T}$'s producible assemblies.

Important Particular Cases

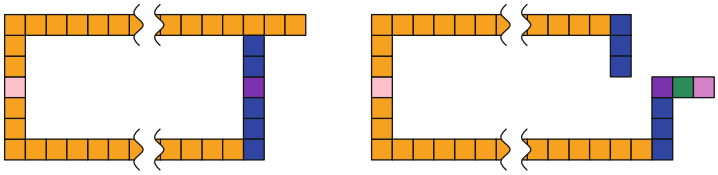
Noncooperative self-assembly, when restricted to dimension one, is similar to nondeterministic finite automata. It is therefore natural to look for a *pumping lemma*.

The first result in this direction was proven by Doty, Patitz, and Summers [5], who introduced the notion of *pumpable paths*: a path P is *pumpable* if it contains a subsegment $P_{i,i+1,\ldots,j}$ that can be repeated arbitrarily many (consecutive) times along $\overrightarrow{P_i P_j}$ while remaining self-avoiding.

Theorem 6 (from [5]) *Let $\mathcal{T}$ be a tile assembly system that assembles exactly one (potentially infinite) terminal assembly α. If any path in α, longer than a constant c, is pumpable, then there are finite families of vectors $\mathbf{b_1}, \ldots, \mathbf{b_n}$, $\mathbf{u_1}, \ldots, \mathbf{u_n}$, and $\mathbf{v_1}, \ldots, \mathbf{v_n} \in \mathbb{Z}^2$, such that:*

$$\mathrm{dom}(\alpha) = \bigcup_{1 \le i \le n} \{\mathbf{b_i} + j\,\mathbf{u_i} + k\,\mathbf{v_i} | j, k \in \mathbb{N}\}$$

In [5], examples were identified, of paths with segments that could be repeated, but only finitely many times due to collisions. The formalization of these examples was later done by Manuch,



Self-Assembly at Temperature 1, Fig. 5 Illegal productions, that a temperature 1 system that can simulate all productions of $\mathcal{T}$ must also be able to simulate (Figure modified from [7])

Stacho, and Stoll [6], proving lower bounds under the hypothesis that no mismatches can occur.

This approach was then extended by Reif and Song [8], to show that tile assembly systems without mismatches have a recursive set of productions. However, the decidability of the "no mismatches" hypothesis is still an open problem.

Applications

Given the successful experimental applications of tile self-assembly, particularly in the field of DNA nanotechnologies, it seems natural to try to implement them: indeed, intuition suggests that they would make no errors in cooperation tiles. However, no successful construction of noncooperative experiments has been reported; the reason might be that ensuring uniqueness of the seed is impossible, as any two tiles in solution together might bind, without any of them being bound to the seed.

Open Problems

Aside from understanding the exact geometric requirements for Turing universality, a number of open problems have been identified in this model:

1. From [9]: What is the tile complexity of squares of size $n \times n$ in the planar, temperature 1 model?

   A related problem, which has been in the folklore for some time, is the existence of a shape of tile complexity arbitrarily smaller than its Manhattan diameter.
2. From [5]: If $\mathcal{T}$ is a tile assembly system with exactly one terminal assembly, is there a constant c such that any path longer than c is pumpable?
3. From [7]: Is there a temperature 1 tile assembly system with a non-recursive set of productions?
4. From [7]: Is there a single tileset able to simulate any temperature 1 tile assembly system up to rescaling, using only noncooperative bindings?

Cross-References

- Experimental Implementation of Tile Assembly
- Intrinsic Universality in Self-Assembly

Recommended Reading

1. Adleman LM, Cheng Q, Goel A, Huang MDA, Kempe D, de Espanés PM, Rothemund PWK (2002) Combinatorial optimization problems in self-assembly. In: Proceedings of the thirty-fourth annual ACM symposium on theory of computing (STOC), Montréal, pp 23–32
2. Cook M, Fu Y, Schweller RT (2011) Temperature 1 self-assembly: deterministic assembly in 3D and probabilistic assembly in 2D. In: Proceedings of the 22nd annual ACM-SIAM symposium on discrete algorithms (SODA), San Francisco, pp 570–589, arxiv preprint: arXiv:0912.0027
3. Doty D, Lutz JH, Patitz MJ, Summers SM, Woods D (2009) Intrinsic universality in self-assembly. In: Proceedings of the 27th international symposium on theoretical aspects of computer science (STACS), Nancy, pp 275–286. arxiv preprint: arXiv:1001.0208
4. Doty D, Lutz JH, Patitz MJ, Schweller RT, Summers SM, Woods D (2012) The tile assembly model is intrinsically universal. In: Proceedings of the 53rd

annual IEEE symposium on foundations of computer science (FOCS), New Brunswick, pp 439–446. arxiv preprint: arXiv:1111.3097
5. Doty D, Patitz MJ, Summers SM (2009) Limitations of self-assembly at temperature 1. In: Proceedings of the fifteenth international meeting on DNA computing and molecular programming, Fayetteville, 8–11 June 2009, pp 283–294. arxiv preprint: arXiv:0906.3251
6. Maňuch J, Stacho L, Stoll C (2010) Two lower bounds for self-assemblies at temperature 1. J Comput Biol 17(6):841–852
7. Meunier PE, Patitz MJ, Summers SM, Theyssier G, Winslow A, Woods D (2014) Intrinsic universality in tile self-assembly requires cooperation. In: Proceedings of the 25th annual ACM-SIAM symposium on discrete algorithms (SODA), Portland, pp 752–771. arxiv preprint: arXiv:1304.1679
8. Reif JH, Song T (2013) Complexity and computability of temperature-1 tilings. In: FNANO 2013, poster abstract
9. Rothemund PWK, Winfree E (2000) The program-size complexity of self-assembled squares (extended abstract). In: Proceedings of the thirty-second annual ACM symposium on theory of computing (STOC), Portland. ACM, pp 459–468. doi:http://doi.acm.org/10.1145/335305.335358
10. Winfree E (1998) Algorithmic self-assembly of DNA. PhD thesis, California Institute of Technology
11. Woods D (2013) Intrinsic universality and the computational power of self-assembly. In: Neary T, Cook M (eds) MCU, Zürich. EPTCS, vol 128, pp 16–22

Self-Assembly of Fractals

Matthew J. Patitz
Department of Computer Science and Computer Engineering, University of Arkansas, Fayetteville, AR, USA

Keywords

Discrete self-similar fractals; Fractal dimension; Self-assembly; Tile Assembly Model

Years and Authors of Summarized Original Work

2009; Kautz, Lathrop
2009; Lathrop, Lutz, Summers
2010; Patitz, Summers
2012; Lutz, Shutters
2013; Kautz, Shutters
2014; Barth, Furcy, Summers, Totzke

Problem Definition

This problem is concerned with the self-assembly fractal patterns and structures. More specifically, it deals with discrete self-similar fractals and different notions of them self-assembling from tiles in the abstract Tile Assembly Model (aTAM) and derivative models. The self-assembly of fractals and fractal-like structures is particularly interesting due to their pervasiveness in nature, as well their complex aperiodic structures which result in them occupying less dimensional space than the space they are embedded within.

Using the terminology from [1], we define $\mathbb{N}_g$ as the subset $\{0, 1, \ldots, g-1\}$ of $\mathbb{N}$, and if $A, B \subseteq \mathbb{N}^2$ and $k \in \mathbb{N}$, then $A + kB = \{\mathbf{m} + k\mathbf{n} | \mathbf{m} \in A$ and $\mathbf{n} \in B\}$. We then define discrete self-similar fractals as follows.

We say that $\mathbf{X} \subset \mathbb{N}^2$ is a *discrete self-similar fractal* (or *dssf* for short) if there exist $1 < g \in \mathbb{N}$ and a set $\{(0,0)\} \subset G \subset \mathbb{N}_g^2$ with at least one point in every row and column, such that $\mathbf{X} = \bigcup_{i=1}^{\infty} X_i$, where X_i, the ith stage of $\mathbf{X}$, is defined by $X_1 = G$ and $X_{i+1} = X_i + g^i G$. We say that G is the generator of $\mathbf{X}$.

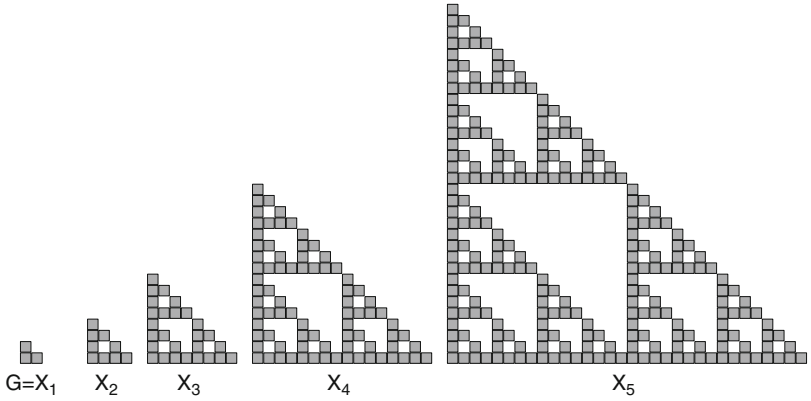
Figure 1 shows, as an example, the first 5 stages of the discrete self-similar fractal known as the Sierpinski triangle. In this example, $G = \{(0,0), (1,0), (0,1)\}$.

In general, we ask whether or not a given dssf $\mathbf{X}$ can self-assemble within a given model.

Variants

The general problem of determining whether or not a discrete self-similar fractal self-assembles within a given model has several variants, which determine the way in which the fractal shape is represented within a resulting assembly.

1. **Weak self-assembly.** If a dssf $\mathbf{X}$ weakly self-assembles using a tile set T, then there exists a subset of tile types $B \subseteq T$ such that, in



Self-Assembly of Fractals, Fig. 1 Example discrete self-similar fractal: the first 5 stages of the Sierpinski triangle

the terminal assembly α, for every point $\mathbf{p} \in \text{dom}\,\alpha$ such that $\mathbf{p} \in \mathbf{X}$, the tile type at location $\mathbf{p}$ in α is a type within B, and for every point $\mathbf{p} \in \text{dom}\,\alpha$ such that $\mathbf{p} \notin \mathbf{X}$, the tile type at location $\mathbf{p}$ in α is not within B. That is, the tile types in the subset B precisely "paint a picture" of $\mathbf{X}$, while tiles of types not in B may appear in locations outside of $\mathbf{X}$.

2. **Strict self-assembly.** If a dssf $\mathbf{X}$ strictly self-assembles, then it weakly self-assembles with $B = T$, i.e., the locations that exist in the domain of the terminal assembly are exactly those of $\mathbf{X}$.
3. **Approximate self-assembly.** A dssf $\mathbf{X}$, and thus a strictly self-assembled version of $\mathbf{X}$, has fractal dimension (i.e., zeta-dimension [3]) <2, and a weakly self-assembled version has dimension 2. Since it appears to be difficult if not impossible to strictly self-assemble many (or all) dssf's, it is interesting to consider if an approximation of a dssf $\mathbf{X}$ which retains the same fractal dimension as $\mathbf{X}$ can strictly self-assemble.

Key Results

Self-assembly of dssf's has been studied in all of the above variants and within the aTAM, 2HAM, and STAM [9]. As previously mentioned, the complexity of dssf's makes them interesting to study since they are infinite, aperiodic structures. This requires any system in which they self-assemble to rely on algorithmic self-assembly (rather than unique tile types hard coded to each position of the shape), and for this reason early experimental results even included the weak self-assembly of the initial few stages of the Sierpinski triangle [11] as a proof of concept that DNA-based tile implementations of the aTAM are capable of algorithmic self-assembly. Nonetheless, as infinite structures, dssf's are more often the focus of theoretical studies.

Weak Self-Assembly

As seen in [11], it is possible for a very simple tile set of only 7 tile types to weakly self-assemble the Sierpinski triangle. This tile set can essentially be thought of as computing the `xor` function on two inputs (i.e., $00 \to 0$, $01 \to 1$, $10 \to 1$, and $11 \to 0$), with the glues with which a tile initially binds to an assembly encoding the input bits and those to which tiles later attach encoding the output bits.

In [4] it was noted that another characterization of the Sierpinski triangle is as the nonzero residues modulo 2 of Pascal's triangle. They then provided a characterization of an infinite class of dssf's, known as *generalized Sierpinski carpets*, which can be defined as the residues, modulo a prime number, of the entries in a two-dimensional matrix generated by a simple recursive equation. (A well-known example among this class of

dssf's is the Sierpinski carpet.) They then proved that all generalized Sierpinski carpets weakly self-assemble in the aTAM.

Strict Self-Assembly

Although weak self-assembly of many dssf's can be achieved with very simple tile sets, it turns out that strict self-assembly is an entirely different, and much more difficult, problem. In fact, in [6] they proved that it is impossible for the Sierpinski triangle to strictly self-assemble in the aTAM. Furthermore, their proof showed that to be the case regardless of the temperature parameter. This result was extended in [10] to a proof that an infinite class of "pinch-point" dssf's, which includes the Sierpinski triangle, cannot strictly self-assemble in the aTAM at any temperature. Pinch-point fractals are those whose generators have exactly one point in their topmost row, the leftmost, and one in their eastmost column, the bottommost. Yet another extension was provided in [1], where the authors defined "tree" fractals, which again include the Sierpinski triangle, as those with generators which are trees and which have a single point in their topmost row and a single point in their rightmost column. They then proved that, regardless of the temperature or even of the scale factor, no tree fractal strictly self-assembles in the aTAM.

Additional results related to strict self-assembly of dssf's include the proof in [10] that in the aTAM at temperature 1 (i.e., systems with $\tau = 1$), it is impossible for any dssf to self-assemble within a locally deterministic system (see [12] for a definition of local determinism), and in [2] it was proven that the Sierpinski triangle also cannot self-assemble in the 2-Handed Assembly Model, at any temperature.

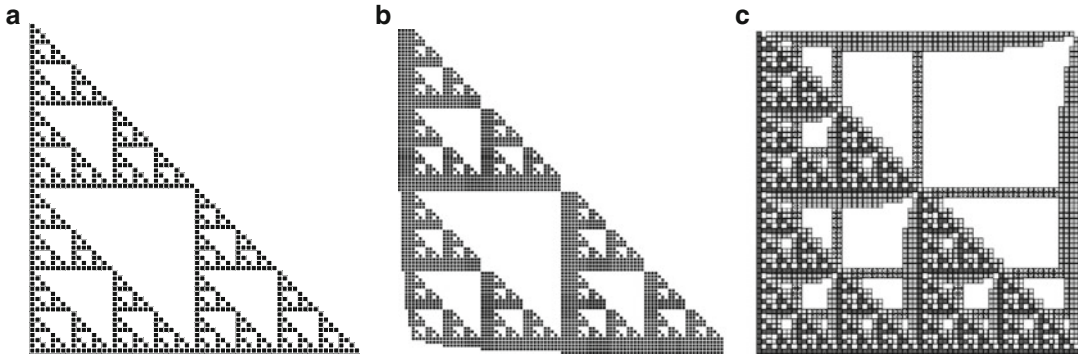
To date, the single positive result related to the strict self-assembly of a dssf is for an "active" model of self-assembly, where tiles are allowed to change the states of their glues during assembly, called the Signal-passing Tile Assembly Model (STAM). In [9] they gave a construction proving that the Sierpinski triangle can self-assemble within the STAM at temperature 1 and scale factor 2. By the result of [1], this is impossible in the aTAM and demonstrates the power of the active nature of the STAM, as that construction essentially builds stages of the Sierpinski triangle in a manner analogous to weak self-assembly, but then causes the unwanted interior portions to dissociate and then break apart.

Approximate Self-Assembly

It has been shown that an infinite subset of dssf's can weakly self-assemble in the aTAM, while another infinite subset cannot strictly self-assemble. Recall also that dssf's have fractal dimension <2, and since their strictly self-assembled versions retain their original fractal dimensions, so do they. However, their weakly self-assembled versions have dimension 2. Therefore, the question arises about whether or not some transformation of a dssf (especially, a dssf which cannot strictly self-assemble), which visually approximates the original dssf while retaining its fractal dimension, can strictly self-assemble in the aTAM.

This question was first answered positively in [6], where they defined a transformation for the Sierpinski triangle which they called "fibering," and they then gave a construction proving that the so-called fibered Sierpinski triangle does strictly self-assemble in the aTAM while maintaining the Sierpinski triangle's fractal dimension of ≈ 1.585. An example can be seen in Fig. 2b, showing how the fibering consists an additional row of tiles added to the south and west borders of each copy of each subsequent stage of the fractal. In [10] they extended the technique of fibering to include an infinite subclass of dssf's (which again includes the Sierpinski triangle) which they called "nice" dssf's. Nice dssf's are those whose generators are connected and contain all points on the west and south boundaries.

While the fibering technique creates visual approximations of fractals, it results in subsequent stages being further and further separated from each other. To counter this drawback, in [8] they introduced a technique for fibering the Sierpinski triangle "in place." An example can be seen in Fig. 2c, showing how this version of fibering only uses space on the interior of each stage of the fractal, thus allowing the stages to remain in the same positions relative to each other.



Self-Assembly of Fractals, Fig. 2 Various patterns corresponding to the Sierpinski triangle. (**a**) A portion of the discrete Sierpinski triangle. (**b**) A portion of the fibered Sierpinski triangle of [7] (Figure from [7]). (**c**) A portion of the in-place fibered Sierpinski triangle of [8] (Figure from [8])

Furthermore, this technique retains the same fractal dimension as the Sierpinski triangle, and they showed that it is impossible to use asymptotically less space than their construction while strictly self-assembling a shape which contains the Sierpinski triangle as a subset. In [5] this technique was extended to strictly self-assemble approximations for every generalized Sierpinski carpet.

Open Problems

1. Does there exist a discrete self-similar fractal which can strictly self-assemble in the aTAM, or conversely, can it be shown that none does?
2. What is the class of discrete self-similar fractals for which an approximation, such as fibering or in-place fibering, which maintains the original fractal dimension, strictly self-assembles in the aTAM?

URLs to Code and Data Sets

ISU TAS simulation software for the aTAM, kTAM, and 2HAM (http://self-assembly.net/wiki/index.php?title=ISU_TAS) and the Fibered Fractal Tiler for defining discrete self-similar fractals which can be fibered and generating the corresponding aTAM tile sets (http://self-assembly.net/wiki/index.php?title=Fibered_Fractal_Tiler).

Cross-References

- ▸ Experimental Implementation of Tile Assembly
- ▸ Self-Assembly at Temperature 1
- ▸ Self-Assembly of Squares and Scaled Shapes

Recommended Reading

1. Barth K, Furcy D, Summers SM, Totzke P (2014) Scaled tree fractals do not strictly self-assemble. In: Unconventional computation & natural computation (UCNC) 2014, University of Western Ontario, London, 14–18 July 2014 (to appear)
2. Cannon S, Demaine ED, Demaine ML, Eisenstat S, Patitz MJ, Schweller R, Summers SM, Winslow A (2012) Two hands are better than one (up to constant factors). Technical report 1201.1650, Computing Research Repository. http://arxiv.org/abs/1201.1650
3. Doty D, Gu X, Lutz JH, Mayordomo E, Moser P (2005) Zeta-dimension. In: Proceedings of the thirtieth international symposium on mathematical foundations of computer science, Gdansk. Springer, pp 283–294
4. Kautz SM, Lathrop JI (2009) Self-assembly of the Sierpinski carpet and related fractals. In: Proceedings of the fifteenth international meeting on DNA computing and molecular programming, Fayetteville, 8–11 June 2009, pp 78–87
5. Kautz S, Shutters B (2013) Self-assembling rulers for approximating generalized Sierpinski

carpets. Algorithmica 67(2):207–233. doi:10.1007/s00453-012-9691-x, http://dx.doi.org/10.1007/s00453-012-9691-x
6. Lathrop JI, Lutz JH, Summers SM (2007) Strict self-assembly of discrete Sierpinski triangles. In: Proceedings of the third conference on computability in Europe, Siena, 18–23 June 2007
7. Lathrop JI, Lutz JH, Summers SM (2009) Strict self-assembly of discrete Sierpinski triangles. Theor Comput Sci 410:384–405
8. Lutz JH, Shutters B (2012) Approximate self-assembly of the Sierpinski triangle. Theory Comput Syst 51(3):372–400
9. Padilla JE, Patitz MJ, Pena R, Schweller RT, Seeman NC, Sheline R, Summers SM, Zhong X (2013) Asynchronous signal passing for tile self-assembly: fuel efficient computation and efficient assembly of shapes. In: UCNC, Milan, pp 174–185
10. Patitz MJ, Summers SM (2008) Self-assembly of discrete self-similar fractals (extended abstract). In: Proceedings of the fourteenth international meeting on DNA computing, Prague, 2–6 June 2008 (to appear)
11. Rothemund PWK, Papadakis N, Winfree E (2004) Algorithmic self-assembly of DNA Sierpinski triangles. PLoS Biol 2(12):e424. doi:10.1371/journal.pbio.0020424, http://dx.doi.org/10.1371
12. Soloveichik D, Winfree E (2007) Complexity of self-assembled shapes. SIAM J Comput 36(6):1544–1569

Self-Assembly of Squares and Scaled Shapes

Robert Schweller
Department of Computer Science, University of Texas Rio Grande Valley, Edinburg, TX, USA

Keywords

Algorithmic self-assembly; Kolmogorov complexity; Scaled shapes; Self-assembly; Tile assembly model; Tile complexity

Years and Authors of Summarized Original Work

2000; Rothemund, Winfree
2001; Adleman, Cheng, Goel, Huang
2005; Cheng, Aggarwal, Goldwasser, Kao, Schweller, Espanes
2007; Soloveichik, Winfree

Problem Definition

Abstract Tile Assembly Model

The abstract Tile Assembly Model (aTAM) [3] is a mathematical model of self-assembly in which system components are four-sided Wang tiles with glue types assigned to each tile edge. Any pair of glue types are assigned some nonnegative interaction strength denoting how strongly the pair of glues bind. An aTAM system is an ordered triplet (T, τ, σ) consisting of a set of tiles T, a positive integer threshold parameter τ called the system's *temperature*, and a special tile $\sigma \in T$ denoted as the *seed* tile. Assembly proceeds by attaching copies of tiles from T to a growing seed assembly whenever the placement of a tile on the 2D grid achieves a total strength of attachment from abutting edges, determined by the sum of pairwise glue interactions, that meets or exceeds the temperature parameter τ. The pairwise strength assignment between glues on tile edges is often restricted to be "linear" in that identical glue pairs may be assigned arbitrary positive values, while non-equal pairs are required to have interaction strengths of 0. We denote this restricted version of the model as the *standard* aTAM. When this restriction is not applied, i.e., any pair of glues may be assigned any positive integer strength, we call the model the *flexible glue* aTAM.

Given the aTAM's model of growth, we may consider the problem of designing an aTAM system which is guaranteed to grow into a target shape S, given by a set of 2D integer coordinates, and stop growing. Such systems are guaranteed to exist for any finite shape S, but solutions will typically vary in the number of tiles $|T|$ used. For a given shape S, an interesting problem is to design a system that assembles S while using the fewest, or close to the fewest, number of tiles $|T|$ possible. This fewest possible number of tiles required for the assembly of a given shape S is termed the *program-size* complexity of S.

Problem 1 Let $K_{SA}(n)$ and $K_{SA}^{\sim}(n)$ denote the program-size complexity of an $n \times n$ square for the standard aTAM and the flexible glue aTAM, respectively. What are $K_{SA}(n)$ and $K_{SA}^{\sim}(n)$?

Problem 2 Let $K_{SA}(n,k)$ and $K_{SA}^{\sim}(n,k)$ denote the program-size complexity of a $k \times n$ rectangle for the standard aTAM and the flexible glue aTAM, respectively. What are $K_{SA}(n,k)$ and $K_{SA}^{\sim}(n,k)$?

Problem 3 For an arbitrary given shape S, what is the program-size complexity of S? Let the *scale-free* program size of S be the smallest tile set system that uniquely builds some scaled-up version S. Let $K_{SA}(S)$ and $K_{SA}^{\sim}(S)$ denote the *scale-free* program size of S for the standard aTAM and the flexible glue aTAM, respectively. What are $K_{SA}(S)$ and $K_{SA}^{\sim}(S)$?

Key Results

The best known bounds for program-size complexity for squares, rectangles, and general scaled shapes are presented in this section.

$n \times n$ Squares

The efficient self-assembly of $n \times n$ squares has served as a benchmark for self-assembly algorithms within the aTAM and more general tile assembly models. Within the aTAM, the problem is well understood up to constant factors. The first result states a general upper bound for the program size of self-assembled squares for general n, which is matched by an information-theoretic lower bound that holds for almost all integers n. The precise bounds differ between the standard and flexible glue models but are tight in both cases. The lower bound of inequality (1) is proven in [3] and is based on the Kolmogorov complexity of the integer n. The lower bound of (2) is proven in [2] by the same approach. The upper bound of (1) is proven in [1] and offers an improvement over the initial upper bound of $O(\log n)$ from [3]. The $O(\log n)$ result of [3] is achieved by implementing a key primitive in tile self-assembly: a binary counter of $\log n$ tiles that grows to length n. The improvement of [1] is achieved by modifying the counter concept to work with an optimal, variable base. The upper bound of (2) is proven in [2] and is obtained by combining the aTAM counter primitives with a scheme for efficiently seeding the counter by extracting bits from the values of the flexible glue interactions.

Theorem 1 *There exist positive constants c_1 and c_2 such that for almost all integers $n \in \mathbb{N}$, the following inequalities hold. Moreover, the upper bounds hold for all $n \in \mathbb{N}$.*

$$c_1 \frac{\log n}{\log \log n} \leq K_{SA}(n) \leq c_2 \frac{\log n}{\log \log n}. \quad (1)$$

$$c_1 \sqrt{\log n} \leq K_{SA}^{\sim}(n) \leq c_2 \sqrt{\log n}. \quad (2)$$

While the above theorem presents a tight understanding of the program-size complexity for most self-assembled squares, the information-theoretic lower bound allows for special values of n to be assembled with a much smaller program size. The program size is in fact as small as one could reasonably hope for. In [3], a tile system is presented that simulates a Busy Beaver Turing Machine and assembles correspondingly large squares for each tile set size. This construction yields the following theorem implying that the largest self-assembled square for a given number of tiles grows faster than any computable function!

Theorem 2 *There exists a positive constant c such that for infinitely many n, $K_{SA}(n) \leq c f(n)$ for $f(n)$ any nondecreasing unbounded computable function.*

Thin Rectangles

The program size of self-assembled squares and other thick rectangles is dictated by information-theoretic bounds which stem from the aTAM's ability to simulate arbitrary Turing machines given enough geometric space to work within. When this space is cut down, such as in the case of building a thin $k \times n$ rectangle, the program size is limited by geometric factors. The following upper and lower bounds are shown in [2] and represent the best known bounds for *thin* $k \times n$ rectangles in which $k = O(\log / \log \log n)$. The lower bound is achieved by a pigeon-hole pumping argument on the types of tiles placed, along with their order of placement, along a

width k column of the target rectangle. The upper bound is based on the construction of a general-base, general-width counter, which generalizes the binary counter concept of [3].

Theorem 3 *There exist positive constants c_1 and c_2 such that for any $n, k \in \mathbb{N}$, the following inequalities hold.*

$$c_1 \frac{n^{1/k}}{k} \leq K^{\sim}_{SA}(n,k) \leq K_{SA}(n,k) \leq c_2(n^{1/k}+k).$$

Scaled Shapes

The program size of general shapes is difficult to analyze as it is highly dependent on geometric features of the target shape. However, if we consider the assembly of an arbitrarily scaled-up version of a target shape, these geometric difficulties can be eliminated and a very general result can be achieved. The next result from [4] shows that the scale-free program size of S is closely related to the Kolmogorov complexity of S. In particular, the scale-free program-size complexity of S is a log factor less than the Kolmogorov complexity of S for the standard model, and the scale-free program size complexity of S is the square root of the Kolmogorov complexity of S for the flexible glue model. The standard model result is shown in [4] and is achieved by encoding a compressed description of S in a small tile set which is extracted by a set of tiles simulating a Turing machine that extracts the pixels of S from this compressed representation. The need for the scale factor increase of S is to allow room for the Turing machine simulation. In fact, the required scale factor is the run time of the Turing machine that decompresses the optimal encoding of S. The flexible glue result is achieved by combining portions of the flexible glue construction for squares [2] with the construction of [4]. In the following theorem, $K(S)$ denotes the Kolmogorov complexity of S with respect to some fixed universal Turing machine.

Theorem 4 *For any shape S, there exist positive constants c_1 and c_2 such that*

$$c_1 \frac{K(S)}{\log K(S)} \leq K_{SA}(S) \leq c_2 \frac{K(S)}{\log K(S)}. \quad (3)$$

$$c_1 \sqrt{K(S)} \leq K^{\sim}_{SA}(S) \leq c_2 \sqrt{K(S)}. \quad (4)$$

Open Problems

A few important open problems in this area are as follows. In the case of squares, the program size is well understood as long as the temperature of the system is at least two. A long-standing open problem has been to determine the program-size complexity of $n \times n$ squares for temperature-1 self-assembly in which each positive glue force alone is sufficient to cause a tile attachment. To date, no known method is able to achieve $o(n)$ tile complexity at temperature-1 for an $n \times n$ square, but no proof exists that this cannot be done. With respect to thin $k \times n$ rectangles, the best upper and lower bound have a gap with respect to variable k. Does there exist a more efficient rectangle construction, or can a higher lower bound be derived? Finally, while the scale-free program-size complexity of general shapes is well understood, little is known about the (unscaled) program size of general shapes. What new tools and geometric classifications can be developed to analyze and bound this complexity for general shapes?

Cross-References

- Active Self-Assembly and Molecular Robotics with Nubots
- Combinatorial Optimization and Verification in Self-Assembly
- Intrinsic Universality in Self-Assembly
- Patterned Self-Assembly Tile Set Synthesis
- Randomized Self-Assembly
- Robustness in Self-Assembly
- Self-Assembly at Temperature 1
- Self-Assembly of Fractals
- Self-Assembly of Squares and Scaled Shapes
- Self-Assembly with General Shaped Tiles
- Temperature Programming in Self-Assembly

Recommended Reading

1. Adleman L, Cheng Q, Goel A, Huang, M-D (2001) Running time and program size for self-assembled squares. In Proceedings of the thirty-third annual ACM symposium on theory of computing, New York. ACM, pp 740–748
2. Cheng Q, Aggarwal G, Goldwasser MH, Kao M-Y, Schweller RT, de Espanés PM (2005) Complexities for generalized models of self-assembly. SIAM J Comput 34:1493–1515
3. Rothemund PWK, Winfree E (2000) The program-size complexity of self-assembled squares (extended abstract). In Proceedings of the 32nd ACM symposium on theory of computing, STOC'00, Portland, pp 459–468
4. Soloveichik D, Winfree E (2007) Complexity of self-assembled shapes. SIAM J Comput 36(6):1544–1569

Self-Assembly with General Shaped Tiles

Andrew Winslow
Department of Computer Science, Tufts University, Medford, MA, USA

Keywords

Cellular automata; DNA computing; Geometric computing; Natural computing; Turing universality

Years and Authors of Summarized Original Work

2012; Fu, Patitz, Schweller, Sheline
2014; Demaine, Demaine, Fekete, Patitz, Schweller, Winslow, Woods

Problem Definition

Self-assembly is an asynchronous, decentralized process in which particles aggregate to form superstructures according to localized interactions. The most well-studied models of these particle systems, e.g., the abstract Tile Assembly Model of Winfree [11], utilize square-shaped particles arranged on a lattice by attaching edgewise. Particles attach to form larger *assemblies*, and a pair of assemblies or tiles can attach if they can translate to a nonoverlapping configuration with a set of k coincident edges, where $k \geq \tau$, a parameter of the system called the *temperature*.

In *seeded* assembly, individual particles attach to a growing *seed assembly*. This assembly may begin as a single-tile or a multi-tile assembly. In *unseeded* assembly (also called *hierarchical* [3], *two-handed* [2], or *polyomino* [7] assembly), there is no such restriction. The set of assemblies to which a single tile cannot attach (in seeded assembly) or that cannot attach to any other assembly (in unseeded assembly) are the *terminal assemblies* of the system.

Objectives In general, the goal is to design a system of minimal complexity that assembles into a unique terminal assembly with a desired shape or property. In models using square tiles, this is equivalent to designing a system using the fewest tile types. When tiles are allowed to be more general shapes, then the option of trading tile types for tile complexity becomes available. The motivation for this work is to understand how more complex tile shapes can be used to reduce the number of tile types in a system, and two benchmark problems regarding the computational power and efficiency of tile systems are considered in the context of systems of non-square tiles:

Problem 1 (Square Assembly)

INPUT*: A natural number n.*
OUTPUT*: A self-assembly system with a unique terminal assembly consisting of n^2 tiles in a $n \times n$ square shape.*

Problem 2 (Computational Power) *What systems of non-square tiles are capable of simulating computation, and to what extent?*

Key Results

In general, it is the case that allowing non-square tiles permits an asymptotic reduction in the number of tile types, and systems of very

few non-square tiles are capable of universal computation. At a high-level, such reductions are achieved by simulating many tiles via translations and rotations of a single tile type.

Models

Fu, Patitz, Schweller, and Sheline [6] introduce two models of general shaped types. The first, called the *geometric Tile Assembly Model (gTAM)*, is a model of seeded, translation-only assembly where tiles are polyomino-shaped – equivalent to prebuilt assemblies of square tiles. The second is an unseeded version they call the *Two-Handed Planar Geometric Tile Assembly Model (2GAM)*, which has the added restriction that assemblies can only attach if there exists a continuous motion bringing the two assemblies together during which they remain disjoint. This can be thought of as a restriction that the assemblies live in the plane and do not make use of the third dimension to maneuver into place.

Demaine et al. [4] introduce the *polygonal free-body Tile Assembly Model (pfbTAM)* in which tiles may have arbitrary simple polygonal shapes, attaching edgewise along equal-length edges. Systems without and without rotation are both permitted – we note that rotation is forbidden in the gTAM and 2GAM (as well as the aTAM).

Efficient Construction

Fu, Patitz, Schweller, and Sheline [6] prove that both the gTAM and 2GAM allow an asymptotic reduction in the number of tiles needed to assemble an $n \times n$ square of tiles. For the gTAM, they prove that such a square can be assembled using a temperature-1 system of $O(\sqrt{\log n})$ tile types, beating the optimal (temperature-2) system of $\Omega(\log n / \log \log n)$ square tiles by Adleman et al. [1]. This is a reduction in both the number of tile types (by a quadratic factor) and temperature. The temperature reduction is especially significant, as a lower bound of $\Omega(n)$ for temperature-1 aTAM systems is a widely believed conjecture [8–10].

For the 2GAM, they reduce the number of tile types even further, using a temperature-2 system $O(\log \log n)$ tile types to assemble an $n \times n$ square. However, this system comes with the caveat that system makes use of either a disconnected tile shape or a slightly three-dimensional shape.

Computational Power

Positive results on the computational power of general shaped tile systems fall into two categories: Turing universality and bounded-time computation. Fu, Patitz, Schweller, and Sheline [6] prove that any Turing machine computation can be carried out by a temperature-1 gTAM system. As with the temperature-1 construction of squares, this result is surprising due to the open conjecture regarding the computational power of square tile systems at temperature 1.

Demaine et al. [4] prove that any Turing machine computation can be carried out by a temperature-2 pfbTAM system (with rotation) consisting of a single tile. Their result actually proves that any aTAM system can be simulated by such a system, and thus Turing universality is achieved by simulating aTAM systems carrying out computation. Combined with the intrinsic universality result of Doty et al. [5], this result can be extended to prove that a *single* temperature-2 pfbTAM system (with rotation) consisting of a single tile can carry out any Turing machine computation, given an appropriate seed assembly consisting of copies of this tile.

Finally, Demaine et al. also prove that temperature-3 pfbTAM systems (*without* rotation) consisting of a single tile can carry out simulation of computationally universal cellular automata for a number of steps limited by the size of the seed assembly. Specifically, they prove that n steps can be carried out using a seed assembly of $O(n)$ tiles. A loose lower bound is also proved, namely, that more than three tiles are needed to carry out any computation.

Applications

The generic ability to reduce the number of tile types in a system by increasing the geometric complexity of these tiles extends many other

constructions in theoretical tile assembly. Additionally, there may be practical barriers to systems of many tile types, e.g., additional cost of manufacturing or longer assembly time due to heterogenous combinations of many particle types, that can be reduced or eliminated by replacing these systems with systems of fewer, more complex tile.

Open Problems

Obtaining an upper bound on the number of steps of a cellular automaton simulable by single-tile translation-only systems remains open. It is conjectured that a seed assembly of size n can only carry out $O(n^2)$ steps.

Cross-References

- Combinatorial Optimization and Verification in Self-Assembly
- Experimental Implementation of Tile Assembly
- Self-Assembly at Temperature 1

Recommended Reading

1. Adleman L, Cheng Q, Goel A, Huang MD (2001) Running time and program size for self-assembled squares. In: Proceedings of symposium on theory of computing (STOC), Heraklion
2. Cannon S, Demaine ED, Demaine ML, Eisenstat S, Patitz MJ, Schweller RT, Summers SM, Winslow A (2013) Two hands are better than one (up to constant factors): self-assembly in the 2HAM vs. aTAM. In: STACS 2013, Kiel, LIPIcs, vol 20. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp 172–184
3. Chen H, Doty D (2012) Parallelism and time in hierarchical self-assembly. In: Proceedings of the 23rd annual ACM-SIAM symposium on discrete algorithms (SODA), Kyoto, pp 1163–1182
4. Demaine ED, Demaine ML, Fekete SP, Patitz MJ, Schweller RT, Winslow A, Woods D (2014) One tile to rule them all: simulating any tile assembly system with a single universal tile. In: Esparza J, Fraigniaud P, Husfeldt T, Koutsoupias E (eds) Automata, languages and programming (ICALP), Copenhagen. LNCS, vol 8572. Springer, Berlin/Heidelberg, pp 368–379
5. Doty D, Lutz JH, Patitz MJ, Schweller RT, Summers SM, Woods D (2012) The tile assembly model is intrinsically universal. In: Proceedings of the 53rd annual symposium on foundations of computer science (FOCS), New Brunswick, pp 302–310
6. Fu B, Patitz MJ, Schweller RT, Sheline B (2012) Self-assembly with geometric tiles. In: Czumaj A, Mehlhorn K, Pitts A, Wattenhofer R (eds) Automata, languages and programming (ICALP), Warwick. LNCS, vol 7391. Springer, Berlin/New York, pp 714–725
7. Luhrs C (2010) Polyomino-safe DNA self-assembly via block replacement. Nat Comput 9(1):97–109
8. Meunier PE, Patitz MJ, Summers SM, Theyssier G, Winslow A, Woods D (2014) Intrinsic universality in tile self-assembly requires cooperation. In: Proceedings of the 25th annual ACM-SIAM symposium on discrete algorithms (SODA), Portland, pp 752–771
9. Rothemund PWK, Winfree E (2000) The program-size complexity of self-assembled squares (extended abstract). In: Proceedings of ACM symposium on theory of computing (STOC), Portland, pp 459–468
10. Summers SM (2010) Universality in algorithmic self-assembly. PhD thesis, Iowa State University
11. Winfree E (1998) Algorithmic self-assembly of DNA. PhD thesis, Caltech

Selfish Bin Packing Problems

Leah Epstein
Department of Mathematics, University of Haifa, Haifa, Israel

Keywords

Bin packing; Price of anarchy; Selfish agents

Years and Authors of Summarized Original Work

2006; Bilò
2008, 2011; Epstein, Kleiman

Problem Definition

In bin packing games with selfish items, n items are to be packed into (at most) n bins, where each item chooses a bin that it wishes to be packed

into. The cost of an item i of size $0 < s_i \leq 1$ is defined based on its size and the contents of its bin. Nash equilibria (NE) are defined as solutions where there is no item that can change its choice unilaterally and gain from this change. Bin packing games were inspired by the well-known bin packing problem [2]. In this problem, a set of items, each of size in $(0, 1]$, is given. The goal is to partition (or pack) the items into a minimum number of subsets that are called *bins*. Each bin has unit capacity, and the load of a bin is defined to be the total size of items packed into it (where the load cannot exceed 1). The problem is NP-hard in the strong sense, and thus theoretical research has focused on studying and developing approximation algorithms, which allow to design nearly optimal solutions, and on online algorithms, which receive the items one by one and must assign each item to a bin immediately and irrevocably (without any information on further items).

In a bin packing game, every item is operated by a selfish player. There are n bins, and the strategy of a player is the bin that it selects. If the resulting packing is valid (i.e., the load of no bin exceeds 1), then the set of items sharing a bin share its cost proportionally, i.e., let B be a bin (a subset of items). The cost of $i \in B$ is $s_i / \left(\sum_{j \in B} s_j \right)$. If the resulting packing is invalid, any item packed into an invalid bin has infinite cost. We are interested in pure Nash equilibria, and by the term NE, we refer to such an equilibrium. The problem was presented by Bilò [1].

There are several directions which can be explored. First, one would like to find out if any bin packing game has an NE. If this is the case, other kinds of equilibria might be of interest as well. For a class of games (such that each of them has an NE), a process of convergence is defined as follows. The process starts with an arbitrary configuration, and at each time, an item that can reduce its cost is selected and moved to another bin (where the cost of this item will be smaller than its cost before it is moved). Such a process can also be seen as local search. Items are moved one at a time; a single move (for one item) is called a step. Note that one item can participate in multiple steps. The questions which can be asked are whether the process converges for any initial packing (i.e., reaches a state that no further step can be applied) and how large can the number of steps be. As it turns out, any bin packing game has at least one NE, and the processes described here always converge [1, 8]. Since it is possible that the process converges in exponential time, it is of interest to develop a polynomial time algorithm that computes NE packings. Such an algorithm for this problem defined above was designed by Yu and Zhang [9]. Finally, once the existence of NE packing has been established, the primary goal becomes the study of the quality of worst-case equilibria. This concept is called *price of anarchy*. For a given game G (i.e., a set of items which is an input for bin packing), the price of anarchy of this game, denoted by $POA(G)$, is the ratio between the maximum number of nonempty bins in any NE packing and the minimum number of bins in any packing (the number of bins in an optimal packing, also called the social optimum, denoted by $OPT(G)$). The price of stability is similar, but best-case equilibria are studied, and as Bilò [1] proved that any game has a social optimum that is an NE, the price of stability is 1 for any game.

The price of anarchy (POA) of a class of games (here, the class of all bin packing games) is defined to be the supremum POA over all games in the class. However, as bin packing is typically studied with respect to the asymptotic approximation ratio, the POA for the bin packing class of games is defined, similarly to the asymptotic approximation ratio, as $\lim_{M \to \infty} \sup_{\{G : OPT(G) \geq M\}} POA(G)$.

Key Results

The POA was studied already in [1], where Bilò provided the first bounds on it, a lower bound of $\frac{8}{5}$ and an upper bound of $\frac{5}{3}$. The quality of NE solutions was further investigated in [4], where nearly tight bounds for the PoA were given, an

upper bound of 1.6428 and a lower bound of 1.6416 (see also [9]). The parametric POA, which is the POA for subclasses of games where the size of no item exceeds a given value, was considered as well [5].

NE packings are related to outputs of the algorithm First Fit (FF) for bin packing [7]. FF is in fact an online algorithm that packs each item, in turn, into a minimum index bin where it fits (using an empty bin if there is no other option). It is not difficult to see that every NE is an output of FF; sort the bins of the NE by non-increasing loads, and create a list of items according to the ordering of bins. FF will create exactly the bins of the original packing. Interestingly, the POA is significantly smaller than the asymptotic approximation ratio of FF (which is equal to 1.7 [7]). Note that the PoA is not equal to the approximation ratio of any natural algorithm for bin packing.

Some intuition regarding the difference between the asymptotic approximation ratio of First Fit and the POA of this class of games can be shown using a small example. Consider items of the following sizes (for a sufficiently small $\varepsilon > 0$): $\frac{1}{6} - 2\varepsilon$ (small items), $\frac{1}{3} + \varepsilon$ (medium items), and $\frac{1}{2} + \varepsilon$ (large items). The worst-case examples for FF are similar to this example, though the items of the first two types have a number of different sizes; small items can be slightly smaller or slightly larger than $\frac{1}{6}$, and medium items can be slightly smaller or slightly larger than $\frac{1}{3}$. Given the item types defined above, assume that there are $6N$ items of each type (for some positive integer N), when FF receives this input (sorted by non-decreasing size), it creates N bins with six small items packed into each bin, $3N$ bins with two medium items packed into each bin, and the remaining items are packed into dedicated bins. This packing is not an NE, as a medium item reduces its cost from $\frac{1}{2}$ to approximately $\frac{2}{5}$ if it joins a large item. Indeed, roughly speaking, if an NE packing consists of a large number of bins (compared to an optimal solution), a bin of this NE packing either has an item whose size exceeds $\frac{1}{2}$ or its load cannot be as small as approximately $\frac{2}{3}$. This allows a tighter analysis. Interestingly, in worst-case examples for the POA, medium items have sizes that are close to $\frac{1}{4}$ instead of $\frac{1}{3}$.

Related Results

Bin packing games, where the cost of an item is defined differently, were studied. One option is to assign equal costs to all players (which are packed together into a valid bin) [3, 6]. A generalized version where each item has a positive weight, and costs are based on cost sharing proportional to the weights of items that share a bin [3] was studied as well. The weights of items in the games described above (those of [1, 4]) are equal to their sizes. These are two classes of games, for which the POA turns out to be of interest. The POA for the class of games with equal weights is slightly (strictly) below 1.7, and in the case of general weights, the POA is equal to 1.7 [3]. Another topic of interest is the quality of other kinds of equilibria. Those are strong equilibria, which are solutions that are also resilient to deviations of subsets of items reducing their costs, and Pareto optimal equilibria, where the solution is required to be weakly (or strictly) Pareto optimal, that is, there is no alternative packing where all items reduce their costs (or a packing where no item increases its cost and at least one item reduces it) [3]. For these last kinds of equilibria, the POA is still above 1.6 (but at most 1.7).

Cross-References

▸ Subset Sum Algorithm for Bin Packing

Recommended Reading

1. Bilò V (2006) On the packing of selfish items. In: Proceedings of the 20th international parallel and distributed processing symposium (IPDPS2006). IEEE, Rhodes, Greece, 9pp
2. Coffman E Jr, Csirik J (2007) Performance guarantees for one-dimensional bin packing. In: Gonzalez TF (ed) Handbook of approximation algorithms and metaheuristics, chap 32. Chapman & Hall/CRC, Boca Raton, pp (32–1)–(32–18)

3. Dósa Gy, Epstein L (2012) Generalized selfish bin packing. CoRR, abs/1202.4080
4. Epstein L, Kleiman E (2011) Selfish bin packing. Algorithmica 60(2):368–394
5. Epstein L, Kleiman E, Mestre J (2009) Parametric packing of selfish items and the subset sum algorithm. In: Proceedings of the 5th international workshop on internet and network economics (WINE2009), Rome, Italy, pp 67–78
6. Han X, Dósa Gy, Ting HF, Ye D, Zhang Y (2013) A note on a selfish bin packing problem. J Glob Optim 56(4):1457–1462
7. Johnson DS, Demers AJ, Ullman JD, Garey MR, Graham RL (1974) Worst-case performance bounds for simple one-dimensional packing algorithms. SIAM J Comput 3(4):299–325
8. Miyazawa FK, Vignatti AL (2009) Convergence time to Nash equilibrium in selfish bin packing. Electron Notes Discret Math 35:151–156
9. Yu G, Zhang G (2008) Bin packing of selfish items. In: The 4th international workshop on internet and network economics (WINE2008), Shanghai, China, pp 446–453

Selfish Unsplittable Flows: Algorithms for Pure Equilibria

Paul (Pavlos) Spirakis
Computer Engineering and Informatics, Research and Academic Computer Technology Institute, Patras University, Patras, Greece
Computer Science, University of Liverpool, Liverpool, UK
Computer Technology Institute (CTI), Patras, Greece

Keywords

Atomic network congestion games; Cost of anarchy

Years and Authors of Summarized Original Work

2005; Fotakis, Kontogiannis, Spirakis

Problem Definition

Consider having a set of resources E in a system. For each $e \in E$, let $d_e(\cdot)$ be the delay per user that requests its service, as a function of the total usage of this resource by all the users. Each such function is considered to be non−decreasing in the total usage of the corresponding resource. Each resource may be represented by a pair of points: an entry point to the resource and an exit point from it. So, each resource is represented by an arc from its entry point to its exit point and the model associates with this arc the cost (e.g., the delay as a function of the load of this resource) that each user has to pay if she is served by this resource. The entry/exit points of the resources need not be unique; they may coincide in order to express the possibility of offering joint service to users, that consists of a sequence of resources. Here, denote by V the set of all entry/exit points of the resources in the system. Any nonempty collection of resources corresponding to a directed path in $G \equiv (V, E)$ comprises an *action* in the system.

Let $N \equiv [n]$ be a set of users, each willing to adopt some action in the system. $\forall i \in N$, let w_i denote user i's *demand* (e.g., the flow rate from a source node to a destination node), while $\Pi_i \subseteq 2^E \setminus \emptyset$ is the collection of actions, any of which would satisfy user i (e.g., alternative routes from a source to a destination node, if G represents a communication network). The collection Π_i is called the *action set* of user i and each of its elements contains at least one resource. Any vector $\boldsymbol{r} = (r_1, \ldots, r_n) \in \Pi \equiv \times_{i=1}^{n} \Pi_i$ is a *pure strategies profile*, or a *configuration* of the users. Any vector of real functions $\boldsymbol{p} = (p_1, p_2, \ldots, p_n)$ s.t. $\forall i \in [n]$, $p_i : \Pi_i \to [0, 1]$ is a probability distribution over the set of allowable actions for user i (i.e., $\sum_{r_i \in \Pi_i} p_i(r_i) = 1$), and is called a *mixed strategies profile* for the n users.

A congestion model typically deals with users of identical demands, and thus, user cost function depending on the *number* of users adopting each action [1, 4, 6]. In this work the more general case is considered, where a *weighted congestion model* is the tuple $((w_i)_{i \in N}, (\Pi_i)_{i \in N}, (d_e)_{e \in E})$. That is, the users are allowed to have different demands for service from the whole system, and thus affect the resource delay functions in a different way, depending on their own weights.

A *weighted congestion game* associated with this model, is a game in strategic form with the set of users N and user demands $(w_i)_{i\in N}$, the action sets $(\Pi_i)_{i\in N}$ and cost functions $(\lambda^i_{r_i})_{i\in N, r_i\in\Pi_i}$ defined as follows: For any configuration $\boldsymbol{r}\in\Pi$ and $\forall e\in E$, let $\Lambda_e(\boldsymbol{r})=\{i\in N : e\in r_i\}$ be the set of users exploiting resource e according to $\boldsymbol{r}$ (called the *view* of resource e wrt configuration $\boldsymbol{r}$). The *cost* $\lambda^i(\boldsymbol{r})$ *of user* i *for adopting strategy* $r_i\in\Pi_i$ in a given configuration $\boldsymbol{r}$ is equal to the cumulative *delay* $\lambda_{r_i}(\boldsymbol{r})$ along this path:

$$\lambda^i(\boldsymbol{r})=\lambda_{r_i}(\boldsymbol{r})=\sum_{e\in r_i} d_e(\theta_e(\boldsymbol{r})) \qquad (1)$$

where, $\forall e\in E, \theta_e(\boldsymbol{r})\equiv\sum_{i\in\Lambda_e(\boldsymbol{r})} w_i$ is the load on resource e wrt the configuration $\boldsymbol{r}$.

On the other hand, for a mixed strategies profile $\boldsymbol{p}$, the *expected cost of user* i *for adopting strategy* $r_i\in\Pi_i$ is

$$\lambda^i_{r_i}(\boldsymbol{p})=\sum_{\boldsymbol{r^{-i}}\in\Pi^{-i}} P(\boldsymbol{p^{-i}},\boldsymbol{r^{-i}})\cdot \sum_{e\in r_i} d_e\left(\theta_e(\boldsymbol{r^{-i}}\oplus r_i)\right) \qquad (2)$$

where, $\boldsymbol{r^{-i}}$ is a configuration of all the users except for user i, $\boldsymbol{p^{-i}}$ is the mixed strategies profile of all users except for i, $\boldsymbol{r^{-i}}\oplus r_i$ is the new configuration with user i choosing strategy r_i, and $P(\boldsymbol{p^{-i}},\boldsymbol{r^{-i}})\equiv\prod_{j\in N\setminus\{i\}} p_j(r_j)$ is the occurrence probability of $\boldsymbol{r^{-i}}$.

Remark 1 Here notation is abused a little bit and the model considers the user costs $\lambda^i_{r_i}$ as functions whose exact definition depends on the other users' strategies: In the general case of a mixed strategies profile $\boldsymbol{p}$, (2) is valid and expresses the expected cost of user i wrt $\boldsymbol{p}$, conditioned on the event that i chooses path r_i. If the other users adopt a pure strategies profile $\boldsymbol{r^{-i}}$, we get the special form of (1) that expresses the exact cost of user i choosing action r_i.

A congestion game in which all users are indistinguishable (i.e., they have the same user cost functions) and have the same action set, is called *symmetric*. When each user's action set Π_i consists of sets of resources that comprise (simple) paths between a unique origin-destination pair of nodes (s_i,t_i) in a network $G=(V,E)$, the model refers to a *network congestion game*. If additionally all origin-destination pairs of the users coincide with a unique pair (s,t) one gets a *single commodity network congestion game* and then all users share exactly the same action set. Observe that a single-commodity network congestion game is not necessarily symmetric because the users may have different demands and thus their cost functions will also differ.

Selfish Behavior

Fix an arbitrary (mixed in general) strategies profile $\boldsymbol{p}$ for a congestion game $\left((w_i)_{i\in N},(\Pi_i)_{i\in N},(d_e)_{e\in E}\right)$. We say that $\boldsymbol{p}$ is a *Nash Equilibrium (NE)* if and only if $\forall i\in N, \forall r_i,\pi_i\in\Pi_i, p_i(r_i)>0\Rightarrow\lambda^i_{r_i}(\boldsymbol{p})\le\lambda^i_{\pi_i}(\boldsymbol{p})$. A configuration $\boldsymbol{r}\in\Pi$ is a *Pure Nash Equilibrium (PNE)* if and only if $(\forall i\in N,\forall\pi_i\in\Pi_i,\lambda_{r_i}(\boldsymbol{r})\le\lambda_{\pi_i}(\boldsymbol{r^{-i}}\oplus\pi_i)$ where, $\boldsymbol{r}^{-i}\oplus\pi_i$ is the same configuration with $\boldsymbol{r}$ except for user i that now chooses action π_i.

Key Results

In this section the article deals with the existence and tractability of PNE in weighted network congestion games. First, it is shown that it is not always the case that a PNE exists, even for a weighted single-commodity network congestion game with only linear and 2-wise linear (e.g., the maximum of two linear functions) resource delays. In contrast, it is well known [1, 6] that any unweighted (not necessarily single-commodity, or even network) congestion game has a PNE, for any kind of nondecreasing delays. It should be mentioned that the same result has been independently proved also by [3].

Lemma 1 *There exist instances of weighted single–commodity network congestion games with resource delays being either linear or 2–wise linear functions of the loads, for which there is no PNE.*

Theorem 2 *For any weighted multi–commodity network congestion game with linear resource delays, at least one PNE exists and can be computed in pseudo-polynomial time.*

Proof Fix an arbitrary network $G = (V, E)$ with linear resource/edge delays $d_e(x) = a_e x + b_e$, $e \in E$, $a_e, b_e \geq 0$. Let $\boldsymbol{r} \in \Pi$ be an arbitrary configuration for the corresponding weighted multi–commodity congestion game on G. For the configuration r consider the potential $\Phi(\boldsymbol{r}) = C(\boldsymbol{r}) + W(\boldsymbol{r})$, where

$$C(\boldsymbol{r}) = \sum_{e \in E} d_e(\theta_e(\boldsymbol{r}))\theta_e(\boldsymbol{r}) = \sum_{e \in E} [a_e \theta_e^2(\boldsymbol{r}) + b_e \theta_e(\boldsymbol{r})],$$

and

$$W(\boldsymbol{r}) = \sum_{i=1}^{n} \sum_{e \in r_i} d_e(w_i) w_i = \sum_{e \in E} \sum_{i \in \widetilde{e}(\boldsymbol{r})} d_e(w_i) w_i = \sum_{e \in E} \sum_{i \in \widetilde{e}(\boldsymbol{r})} (a_e w_i^2 + b_e w_i)$$

one concludes that

$$\Phi(\boldsymbol{r}') - \Phi(\boldsymbol{r}) = 2w_i [\lambda^i(\boldsymbol{r}') - \lambda^i(\boldsymbol{r})] ,$$

Note that the potential is a global system function whose changes are proportional to selfish cost improvements of any user. The global minima of the potential then correspond to configurations in which no user can improve her cost acting unilaterally. Therefore, any weighted multi–commodity network congestion game with linear resource delays admits a PNE. □

Applications

In [5] many experiments have been conducted for several classes of pragmatic networks. The experiments show even faster convergence to pure Nash Equilibria.

Open Problems

The Potential function reported here is polynomial on the loads of the users. It is open whether one can find a purely combinatorial potential, which will allow strong polynomial time for finding Pure Nash equilibria.

Cross-References

- Best Response Algorithms for Selfish Routing
- Computing Pure Equilibria in the Game of Parallel Links
- General Equilibrium

Recommended Reading

1. Fabrikant A, Papadimitriou C, Talwar K (2004) The complexity of pure nash equilibria. In: Proceedings of the 36th ACM symposium on theory of computing (STOC'04). ACM, Chicago
2. Fotakis D, Kontogiannis S, Spirakis P (2005) Selfish unsplittable flows. J Theory Comput Sci 348:226–239
3. Libman L, Orda A (2001) Atomic resource sharing in noncooperative networks. Telecommun Syst 17(4):385–409
4. Monderer D, Shapley L (1996) Potential games. Game Econ Behav 14:124–143
5. Panagopoulou P, Spirakis P (2006) Algorithms for pure nash equilibrium in weighted congestion games. ACM J Exp Algorithms 11:2.7
6. Rosenthal RW (1973) A class of games possessing pure-strategy nash equilibria. Int J Game Theory 2:65–67

Self-Stabilization

Ted Herman
Department of Computer Science, University of Iowa, Iowa City, IA, USA

Keywords

Autonomic system control; Autopoesis; Homeostasis

Years and Authors of Summarized Original Work

1974; Dijkstra

Problem Definition

An algorithm is self-stabilizing if it eventually manifests correct behavior regardless of initial state. The general problem is to devise self-stabilizing solutions for a specified task. The property of self-stabilization is now known to be feasible for a variety of tasks in distributed computing. Self-stabilization is important for distributed systems and network protocols subject to transient faults. Self-stabilizing systems automatically recover from faults that corrupt state.

The operational interpretation of self-stabilization is depicted in Fig. 1. Part (a) of the figure is an informal presentation of the behavior of a self-stabilizing system, with time on the x-axis and some informal measure of correctness on the y-axis. The curve illustrates a system trajectory, through a sequence of states, during execution. At the initial state, the system state is incorrect; later, the system enters a correct state, then returns to an incorrect state, and subsequently stabilizes to an indefinite period where all states are correct. This period of stability is disrupted by a transient fault that moves the system to an incorrect state, after which the scenario above repeats. Part (b) of the figure illustrates the scenario in terms of state predicates. The box represents the predicate *true*, which characterizes all possible states. Predicate $\mathcal{C}$ characterizes the correct states of the system, and $\mathcal{L} \subset \mathcal{C}$ depicts the closed *legitimacy* predicate. Reaching a state in $\mathcal{L}$ corresponds to entering a period of stability in part (a). Given an algorithm A with this type of behavior, it is said that A self-stabilizes to $\mathcal{L}$; when $\mathcal{L}$ is implicitly understood, the statement is simplified to: A is self-stabilizing.

Problem [3]. The first setting for self-stabilization posed by Dijkstra is a ring of n processes numbered 0 through $n-1$. Let the state of process i be denoted by $x[i]$. Communication is unidirectional in the ring using a *shared state* model. An atomic step of process i can be expressed by a guarded assignment of the form $g(x[i \ominus 1], x[i]) \rightarrow x[i] := f(x[i \ominus 1], x[i])$. Here, $\ominus$ is subtraction modulo n, so that $x[i \ominus 1]$ is the state of the previous process in the ring with respect to process i. The guard g is a boolean expression; if $g(x[i \ominus 1], x[i])$ is *true*, then process i is said to be *privileged* (or enabled). Thus in one atomic step, privileged process i reads the state of the previous process and computes a new state. Execution scheduling is controlled by a *central daemon*, which fairly chooses one among all enabled processes to take the next step. The problem is to devise g and f so that, regardless of initial states of $x[i]$, $0 \leq i < n$, eventually there is one privilege and every process enjoys a privilege infinitely often.

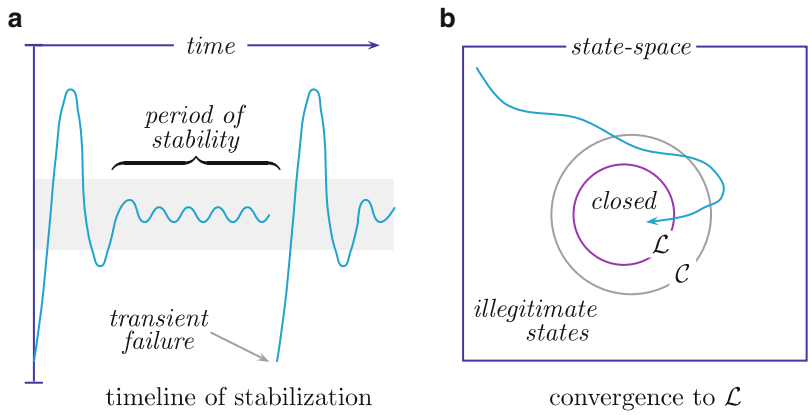
Complexity Metrics

The complexity of self-stabilization is evaluated by measuring the resource needed for convergence from an arbitrary initial state. Most prominent in the literature of self-stabilization are metrics for worst-case time of convergence and space required by an algorithm solving the given task. Additionally, for reactive self-stabilizing algorithms, metrics are evaluated for the stable behavior of the algorithm, that is, starting from a legitimate state, and compared to non-stabilizing algorithms, to measure costs of self-stabilization.

Key Results

Composition

Many self-stabilizing protocols have a layered construction. Let $\{ A_i \}_{i=0}^{m-1}$ be a set programs with the property that for every state variable x, if program A_i writes x, then no program A_j, for $j > i$, writes x. Programs in $\{ A_j \}_{j=i+1}^{m-1}$ may read variables written by A_i, that is, they use the output of A_i as input. Fair composition of programs B and C, written $B \,[]\, C$, assumes fair scheduling of steps of B and C. Let X_j be the set



Self-Stabilization, Fig. 1 Self-stabilization trajectories

of variables read by A_j and possibly written by $\{ A_i \}_{i=0}^{j-1}$.

Theorem 1 (Fair Composition [4]) *Suppose A_i is self-stabilizing to $\mathcal{L}_i$ under the assumption that all variables in X_i remain constant throughout any execution; then $A_0 [] A_1 [] \cdots [] A_{m-1}$ self-stabilizes to $\{ \mathcal{L}_i \}_{i=0}^{m-1}$.*

Fair composition with a layered set $\{ A_i \}_{i=0}^{m-1}$ corresponds to sequential composition of phases in a distributed algorithm. For instance, let B be a self-stabilizing algorithm for mutual exclusion in a network that assumes the existence of a rooted, spanning tree and let algorithm C be a self-stabilizing algorithm to construct a rooted spanning tree in a connected network; then $B [] C$ is a self-stabilizing mutual exclusion algorithm for a connected network.

Synchronization Tasks

One question related to the problem posed in section "Problem Definition" is whether or not there can be a *uniform* solution, where all processes have identical algorithms. Dijkstra's result for the unidirectional ring is a semi-uniform solution (all but one process have the same algorithm), using n states per process. The state of each process is a counter: process 0 increments the counter modulo k, where $k \geq n$ suffices for convergence; the other processes copy the counter of the preceding process in the ring. At a legitimate state, each time process 0 increments the counter, the resulting value is different from all other counters in the ring. This ring algorithm turns out to be self-stabilizing for the *distributed daemon* (any subset of privileged processes may execute in parallel) when $k > n$. Subsequent results have established that mutual exclusion on a unidirection ring is $\Theta(1)$ space per process with a non-uniform solution. Deterministic uniform solutions to this task are generally impossible, with the exceptional case where n is and prime. Randomized uniform solutions are known for arbitrary n, using $O(\lg \alpha)$ space where α is the smallest number that does not divide n. Some lower bounds on space for uniform solutions are derived in [7]. Time complexity of Dijkstra's algorithm is $O(n^2)$ rounds, and some randomized solutions have been shown to have expected $O(n^2)$ convergence time.

Dijkstra also presented a solution to mutual exclusion for a linear array of processes, using $O(1)$ space per process [3]. This result was later generalized to a rooted tree of processes, but with mutual exclusion relaxed to having one privilege along any path from root to leaf. Subsequent research built on this theme, showing how tasks for distributed wave computations have self-stabilizing solutions. Tasks of phase synchronization and clock synchronization have also been solved. See reference [9] for an example of self-stabilizing mutual exclusion in a multiprocessor shared memory model.

Graph Algorithms

Communication networks are commonly represented with graph models and the need for distributed graph algorithms that tolerate

transient faults motivates study of such tasks. Specific results in this area include self-stabilizing algorithms for spanning trees, center-finding, matching, planarity testing, coloring, finding independent sets, and so forth. Generally, all graph tasks can be solved by self-stabilizing algorithms: tasks that have network topology and possibly related factors, such as edge weights, for input, and define outputs to be a function of the inputs, can be solved by general methods for self-stabilization. These general methods require considerable space and time resource, and may also use stronger model assumptions than needed for specific tasks, for instance unique process identifiers and an assumed bound on network diameter. Therefore research continues on graph algorithms.

One discovery emerging from research on self-stabilizing graph algorithms is the difference between algorithms that terminate and those that continuously change state, even after outputs are stable. Consider the task of constructing a spanning tree rooted at process r. Some algorithms self-stabilize to the property that, for every $p \neq r$, the variable u_p refers to p's parent in the spanning tree and the state remains unchanged. Other algorithms are self-stabilizing protocols for token circulation with the side-effect that the circulation route of the token establishes a spanning tree. The former type of algorithm has $O(\lg n)$ space per process, whereas the latter has $O(\lg \delta)$ where δ is the degree (number of neighbors) of a process. This difference was formalized in the notion of *silent* algorithms, which eventually stop changing any communication value; it was shown in [5] for the link register model that silent algorithms for many graph tasks have $\Omega(\lg n)$ space.

Transformation

The simple presentation of [3] is enabled by the abstract computation model, which hides details of communication, program control, and atomicity. Self-stabilization becomes more complicated when considering conventional architectures that have messages, buffers, and program counters. A natural question is how to transform or refine self-stabilizing algorithms expressed in abstract models to concrete models closer to practice. As an example, consider the problem of transforming algorithms written for the central daemon to the distributed daemon model. This transformation can be reduced to finding a self-stabilizing token-passing algorithm for the distributed daemon model such that, eventually, no two neighboring processes concurrently have a token; multiple tokens can increase the efficiency of the transformation.

General Methods

The general problem of constructing a self-stabilizing algorithm for an input nonreactive task can be solved using standard tools of distributed computing: snapshot, broadcast, system reset, and synchronization tasks are building blocks so that the global state can be continuously validated (in some fortunate cases $\mathcal{L}$ can be locally checked and corrected). These building blocks have self-stabilizing solutions, enabling the general approach.

Fault Tolerance

The connection between self-stabilization and transient faults is implicit in the definition. Self-stabilization is also applicable in executions that asynchronously change inputs, silently crash and restart, and perturb communication [10]. One objection to the mechanism of self-stabilization, particularly when general methods are applied, is that a small transient fault can lead to a system-wide correction. This problem has been investigated, for example in [8], where it is shown how convergence can be optimized for a limited number of faults. Self-stabilization has also been combined with other types of failure tolerance, though this is not always possible: the task of counting the number of processes in a ring has no self-stabilizing solution in the shared state model if a process may crash [1], unless a failure detector is provided.

Applications

Many network protocols are self-stabilizing by the following simple strategy: periodically,

they discard current data and regenerate it from trusted information sources. This idea does not work in purely asynchronous systems; the availability of real-time clocks enables the simple strategy. Similarly, watchdogs with hardware clocks can provide an effective basis for self-stabilization [6].

Cross-References

▸ Concurrent Programming, Mutual Exclusion

Recommended Reading

1. Anagnostou E, Hadzilacos V (1993) Tolerating transient and permanent failures. In: Distributed algorithms 7th international workshop. LNCS, vol 725. Springer, Heidelberg, pp 174–188
2. Cournier A, Datta AK, Petit F, Villain V (2002) Snap-stabilizing PIF algorithm in arbitrary networks. In: Proceedings of the 22nd international conference distributed computing systems, Vienna, July 2002, pp 199–206
3. Dijkstra EW (1974) Self stabilizing systems in spite of distributed control. Commun ACM 17(11):643–644. See also EWD391 (1973) In: Selected writings on computing: a personal perspective. Springer, New York, pp 41–46 (1982)
4. Dolev S (2000) Self-stabilization. MIT, Cambridge
5. Dolev S, Gouda MG, Schneider M (1996) Memory requirements for silent stabilization. In: Proceedings of the 15th annual ACM symposium on principles of distributed computing, Philadelphia, May 1996, pp 27–34
6. Dolev S, Yagel R (2004) Toward self-stabilizing operating systems. In: 2nd international workshop on self-adaptive and autonomic computing systems, Zaragoza, Aug 2004, pp 684–688
7. Israeli A, Jalfon M (1990) Token management schemes and random walks yield self-stabilizing mutual exclusion. In: proceedings of the 9th annual ACM symposium on principles of distributed computing, Quebec City, Aug 1990, pp 119–131
8. Kutten S, Patt-Shamir B (1997) Time-adaptive self stabilization. In: Proceedings of the 16th annual ACM symposium on principles of distributed computing, Santa Barbara, Aug 1997, pp 149–158
9. Lamport L (1986) The mutual exclusion problem: part II-statement and solutions. J ACM 33(2):327–348
10. Varghese G, Jayaram M (2000) The fault span of crash failures. J ACM 47(2):244–293

Semi-supervised Learning

Avrim Blum
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

Keywords

Co-training; Learning from labeled and unlabeled data; Semi-supervised SVM

Years and Authors of Summarized Original Work

1998; Blum, Mitchell
1999; Joachims
2010; Balcan, Blum

Problem Definition

Semi-supervised learning [1, 4, 5, 8, 12] refers to the problem of using a large unlabeled data set U together with a given labeled data set L in order to generate prediction rules that are more accurate on new data than would have been achieved using just L alone. Semi-supervised learning is motivated by the fact that in many settings (e.g., document classification, image classification, speech recognition), unlabeled data is plentiful but labeled data is more limited or expensive, e.g., due to the need for human labelers. Therefore, one would like to make use of the unlabeled data if possible.

The general idea behind semi-supervised learning is that unlabeled data, while missing the labels, nonetheless often contains useful information. As an example, suppose one believes the correct decision boundary for some classification problem should be a linear separator that separates most of the data by a large margin. By observing enough unlabeled data to estimate the probability mass near to any given linear separator, one could in principle then discard separators in advance that slice through dense regions and instead focus attention on just

those that indeed separate most of the distribution by a large margin. This is the high-level idea behind semi-supervised SVMs. Alternatively, suppose data objects can be described by two different "kinds" of features, and one believes that each kind should be sufficient to produce an accurate classifier. Then one might want to train a *pair* of classifiers and use unlabeled data for which one classifier is confident but the other is not to bootstrap, labeling such examples with the confident classifier and then feeding them as training data to the less-confident classifier. This is the high-level idea behind Co-Training. Or, if one believes "similar examples should generally have the same label," one might construct a graph with an edge between examples that are sufficiently similar and aim for a classifier that is correct on the labeled data and has a small cut value on the unlabeled data; this is the high-level idea behind graph-based methods. (These will all be discussed in more detail later.) General surveys of semi-supervised learning appear in [5, 12].

A Formal Framework

We now present a formal model for analyzing semi-supervised learning due to Balcan and Blum [1]. This model was developed to provide a unified explanation for a wide range of semi-supervised learning algorithms including the semi-supervised SVMs, Co-Training, and graph-based methods mentioned above. Before describing it, however, we first describe the classic PAC and agnostic learning models for *supervised* learning that this model builds on.

In the PAC and agnostic learning models, data is assumed to be drawn iid from some fixed but initially unknown distribution D over an instance space $\mathcal{X}$ and labeled by some unknown target function $c^* : \mathcal{X} \to \{0, 1\}$. The *error* of some hypothesis function h is defined as $\text{err}(h) = \Pr_{x\sim D}[h(x) \neq c^*(x)]$. In the PAC model (also known as the *realizable case*), we assume that c^* is a member of some known class of functions $\mathcal{C}$, and we say that an algorithm PAC-learns $\mathcal{C}$ if for any given $\epsilon, \delta > 0$, with probability $\geq 1 - \delta$, it produces a hypothesis h such that $\text{err}(h) \leq \epsilon$. In the *agnostic case*, we do not assume that $c^* \in \mathcal{C}$ and instead aim to achieve error close to $\inf_{f\in\mathcal{C}}[\text{err}(f)]$.

The PAC and agnostic learning models in essence assume that one's prior beliefs about the target be described in terms of a class of functions $\mathcal{C}$. In order to capture the reasoning used in semi-supervised learning, however, we need to also describe beliefs about the *relation* between the target function and the data distribution. This is done in the model of Balcan and Blum [1] via a *notion of compatibility* χ between a hypothesis h and a distribution D. Formally, χ maps pairs (h, D) to $[0, 1]$ with $\chi(h, D) = 1$ meaning that h is highly compatible with D and $\chi(h, D) = 0$ meaning that h is very *in*compatible with D. The quantity $1 - \chi(h, D)$ is called the *unlabeled error rate* of h and denoted $\text{err}_{\text{unl}}(h)$. Note that for χ to be useful, it must be estimatable from a finite sample; to this end, χ is further required to be an expectation over individual examples. That is, overloading notation for convenience, we require $\chi(h, D) = \mathbf{E}_{x\sim D}[\chi(h, x)]$, where $\chi : \mathcal{C} \times \mathcal{X} \to [0, 1]$. As with the class $\mathcal{C}$, one can either assume that the target is fully compatible ($\text{err}_{\text{unl}}(c^*) = 0$) or instead aim to do well as a function of how compatible the target is. The case that we assume $c^* \in \mathcal{C}$ and $\text{err}_{\text{unl}}(c^*) = 0$ is termed the "doubly realizable case." The concept class $\mathcal{C}$ and compatibility notion χ are both viewed as *known*.

Examples

Suppose we believe the target should separate most data by a large margin γ. We can represent this belief by defining $\chi(h, x) = 0$ if x is within distance γ of the decision boundary of h and $\chi(h, x) = 1$ otherwise. In this case, $\text{err}_{\text{unl}}(h)$ will denote the probability mass of D within distance γ of h's decision boundary. Alternatively, if we do not wish to commit to a specific value of γ, we could define $\chi(h, x)$ to be a smooth function of the distance of x to the separator defined by h. As a very different example, in co-training (described in more detail below), we assume each example can be described using two "views" that each are sufficient for classification, that is, there exist c_1^*, c_2^* such that for each example $x = \langle x_1, x_2\rangle$, we have $c_1^*(x_1) = c_2^*(x_2)$. We can represent this belief by defining a hypothesis

$h = \langle h_1, h_2 \rangle$ to be compatible with an example $\langle x_1, x_2 \rangle$ if $h_1(x_1) = h_2(x_2)$ and incompatible otherwise; $\text{err}_{\text{unl}}(h)$ is then the probability mass of examples, under D, where the two halves of h disagree.

Intuition

In this framework, the way that unlabeled data helps in learning can be intuitively described as follows. Suppose one is given a concept class $\mathcal{C}$ (such as linear separators) and a compatibility notion χ (such as penalizing h for points within distance γ of the decision boundary). Suppose also that one believes $c^* \in \mathcal{C}$ (or at least is close) and that $\text{err}_{\text{unl}}(c^*) = 0$ (or at least is small). Then, unlabeled data can help by allowing one to estimate the *unlabeled error rate* of all $h \in \mathcal{C}$, thereby in principle reducing the search space from $\mathcal{C}$ (all linear separators) down to just the subset of $\mathcal{C}$ that is highly compatible with D. The key challenge is how this can be done efficiently (in theory, in practice, or both) for natural notions of compatibility, as well as identifying types of compatibility that data in important problems can be expected to satisfy.

Key Results

The following, from [1], illustrate formally how unlabeled data can help in this model. Fix some concept class $\mathcal{C}$ and compatibility notion χ. Given a labeled sample L, define $\widehat{\text{err}}(h)$ to be the fraction of mistakes of h on L. Given an unlabeled sample U, define $\chi(h, U) = \mathbf{E}_{x \sim U}[\chi(h, x)]$ and define $\widehat{\text{err}}_{\text{unl}}(h) = 1 - \chi(h, U)$. That is, $\widehat{\text{err}}(h)$ and $\widehat{\text{err}}_{\text{unl}}(h)$ are the empirical error rate and unlabeled error rate of h, respectively. Finally, given $\alpha > 0$, define $\mathcal{C}_{D,\chi}(\alpha)$ to be the set of functions $f \in \mathcal{C}$ such that $\text{err}_{\text{unl}}(f) \leq \alpha$.

Theorem 1 ([1]) *If $c^* \in \mathcal{C}$ then with probability at least $1 - \delta$, for a random labeled set L and unlabeled set U, the $h \in \mathcal{C}$ that optimizes $\widehat{\text{err}}_{\text{unl}}(h)$ subject to $\widehat{\text{err}}(h) = 0$ will have $\text{err}(h) \leq \epsilon$ for*

$$|U| \geq \frac{2}{\epsilon^2}\left[\ln|\mathcal{C}| + \ln\frac{4}{\delta}\right],$$

$$|L| \geq \frac{1}{\epsilon}\left[\ln|\mathcal{C}_{D,\chi}(\text{err}_{\text{unl}}(c^*) + 2\epsilon)| + \ln\frac{2}{\delta}\right].$$

Equivalently, for $|U|$ satisfying the above bound, for any $|L|$, with probability at least $1 - \delta$, the $h \in \mathcal{C}$ that optimizes $\widehat{\text{err}}_{\text{unl}}(h)$ subject to $\widehat{\text{err}}(h) = 0$ has

$$\text{err}(h) \leq \frac{1}{|L|}\left[\ln|\mathcal{C}_{D,\chi}(\text{err}_{\text{unl}}(c^*) + 2\epsilon)| + \ln\frac{2}{\delta}\right].$$

One can view Theorem 1 as bounding the number of labeled examples needed to learn well as a function of the "helpfulness" of the distribution D with respect to χ, for sufficiently large U. Namely, a helpful distribution is one in which $\mathcal{C}_{D,\chi}(\alpha)$ is small for α slightly larger than the compatibility of the true target function, so we do not need much labeled data to identify a good function among those in $\mathcal{C}_{D,\chi}(\alpha)$.

For infinite hypothesis classes, one needs to consider both the complexity of the class $\mathcal{C}$ and the complexity of the compatibility notion χ. Specifically, given $h \in \mathcal{C}$, define $\chi_h(x) = \chi(h, x)$ and let $\text{VCdim}(\chi(\mathcal{C}))$ denote the VC-dimension of the set $\{\chi_h | h \in \mathcal{C}\}$. A sample complexity bound from [1] based on ϵ-cover size is the following.

Theorem 2 ([1]) *Assume $c^* \in \mathcal{C}$ and let p be the size of the smallest set of functions H such that every function in $\mathcal{C}_{D,\chi}(\text{err}_{\text{unl}}(c^*) + \epsilon/3)$ is $\epsilon/6$-close to some function in H. Then $|U| = O\left(\frac{\max[\text{VCdim}(\mathcal{C}), \text{VCdim}(\chi(\mathcal{C}))]}{\epsilon^2}\ln\frac{1}{\epsilon} + \frac{1}{\epsilon^2}\ln\frac{2}{\delta}\right)$ and $|L| = O\left(\frac{1}{\epsilon}\ln\frac{p}{\delta}\right)$ is sufficient to identify a function $f \in \mathcal{C}$ of error at most ϵ with probability at least $1 - \delta$.*

Finally, for the general (agnostic) case that $c^* \notin \mathcal{C}$, we can define a regularizer based on empirical unlabeled error rates, and then get good bounds for optimizing a combination of the empirical labeled error and the regularization term. Specifically, for a hypothesis h, define $\hat{N}(h)$ to be the number of ways of partitioning the first $|L|$ points in U using $\{f \in \mathcal{C} : \widehat{\text{err}}_{\text{unl}}(f) \leq \widehat{\text{err}}_{\text{unl}}(h)\}$. Then we have

Theorem 3 ([1]) *With probability at least* $1-\delta$, *the hypothesis*

$$h = \arg\min_{h' \in \mathcal{C}} [\widehat{\text{err}}(h') + R(h')], \quad \textit{where}$$

$$R(h') = \sqrt{\frac{24 \ln(\hat{N}(h'))}{|L|}},$$

satisfies

$$\text{err}(h) \leq \min_{h' \in \mathcal{C}} \left[\text{err}(h') + R(h')\right] + 5\sqrt{\frac{\ln(8/\delta)}{|L|}}.$$

Co-Training

Co-Training is a semi-supervised learning method due to [4] for settings in which examples can be thought of as having two "views," that is, two distinct types of information. For example, in classifying webpages (e.g., into student home page, faculty member home page, course home page, etc.), one could use the words on the page itself, but one could also use information from links pointing *to* that page [4]. Or, in classifying visual images, one might have two cameras or even two different filters or preprocessing steps on images from the same camera [9]. Or, in understanding video, one can use visual images and spoken dialogue [7]. In such settings, one can think of an example x as a pair $x = \langle x_1, x_2 \rangle$. The idea of Co-Training is that if each view is in principle enough to achieve a good classification by itself, but each provides somewhat different information, then one can hope to improve performance using unlabeled data. Specifically, in Co-Training, one maintains two hypotheses, one for each view (e.g., a hypothesis that classifies webpages based on the text on the page itself and one that classifies webpages based on information from links pointing to the page). A hypothesis pair $h = \langle h_1, h_2 \rangle$ is compatible with an example $\langle x_1, x_2 \rangle$ if $h_1(x_1) = h_2(x_2)$ and is incompatible otherwise. So the unlabeled error rate of a hypothesis (pair) $h = \langle h_1, h_2 \rangle$ is the probability mass of examples $\langle x_1, x_2 \rangle$ on which the two parts of h disagree.

In practice, there are two primary ways that this notion of compatibility is used to learn from a small amount of labeled data and a large amount of unlabeled data. The first is *iterative co-training*, introduced in [4]. In iterative co-training, a small labeled sample L is used to produce predictors for each view that are confident in some part of their respective input spaces and not confident in other parts. Then, the algorithm searches through the (large) unlabeled set U to find examples $x = \langle x_1, x_2 \rangle$ for which one classifier is confident and the other is not. These examples are labeled by the confident classifier and handed to the less-confident classifier to improve its predictor. The other primary method is to optimize a global objective that combines accuracy over the labeled sample L with agreement over the unlabeled sample U. That is, one searches for the hypothesis pair h that minimizes $\widehat{\text{err}}(h) + \lambda \widehat{\text{err}}_{\text{unl}}(h)$ for some regularization parameter λ [6, 10]. This is generally a non-convex optimization problem, and so various heuristics are typically applied to perform the optimization.

Theoretically, the guarantees for Co-Training are strongest when the data satisfies independence given the label (with some probability p, a random positive example $\langle x_1, x_2 \rangle$ is drawn from $D_1^+ \times D_2^+$, and with probability $1-p$, a random negative example is drawn from $D_1^- \times D_2^-$) and in the realizable case (there exist targets $c_1^*, c_2^* \in \mathcal{C}$ such that all examples $\langle x_1, x_2 \rangle$ in the support of the distribution satisfy $c_1^*(x_1) = c_2^*(x_2)$). Specifically, two key results are

Theorem 4 ([4]) *Any class* $\mathcal{C}$ *that is efficiently PAC-learnable from random classification noise is efficiently learnable from unlabeled data alone in the realizable Co-Training setting, if data satisfies independence given the label and one is given an initial weakly useful predictor* $h_1(x_1)$.

Here, h is a *weakly useful predictor* of a function f if for some $\epsilon > 1/poly(n)$ we have both (a) $\Pr_{x \sim D}[h(x) = 1] \geq \epsilon$ and (b) $\Pr_{x \sim D}[f(x) = 1 | h(x) = 1] \geq \Pr_{x \sim D}[f(x) = 1] + \epsilon$. Theorem 4 implies that if one is able to use a small labeled sample to produce an initial hypothesis that gives a slight "edge" in predicting the target beyond just the overall class probabilities, then under independence given the label one can boost that to a high-accuracy predictor from just unlabeled data.

Furthermore, ignoring computation time, under independence given the label, any class of finite VC-dimension is learnable from a single labeled example. In the case of linear separators, this can be done computationally efficiently.

Theorem 5 ([1]) *Any class $\mathcal{C}$ of finite VC-dimension is learnable from polynomially many unlabeled examples and a single labeled example if D satisfies independence given the label. Furthermore, for linear separators this can be done in polynomial time.*

Semi-supervised SVMs

Semi-supervised SVMs (also called transductive SVMs) [8, 11] aim to find a linear separator that separates both the labeled sample L and the unlabeled sample U by the largest possible margin. That is, one wants to find a separator such that for γ as large as possible, all labeled examples are on the correct side of the separator by distance at least γ and all unlabeled examples are on *some* side of the separator by distance at least γ. In practice, one combines a large-margin objective with a hinge-loss penalty for labeled examples that fail to satisfy the condition, and a "hat-loss" penalty for unlabeled examples that fail to satisfy the condition. Formally, the goal is to minimize $c_1 w^T w + c_2 \sum_{(x_i,y_i)\in L} \alpha_i + c_3 \sum_{x_j \in U} \beta_j$ subject to $(w^T x_i) y_i \geq 1 - \alpha_i$ for all $(x_i, y_i) \in L$ and $(w^T x_j)\tilde{y}_j \geq 1 - \beta_j$ for all $x_j \in U$ (and $\alpha_i, \beta_j \geq 0$), where $y_i \in \{-1, 1\}$ is the (known) label of $x_i \in L$ and $\tilde{y}_j \in \{-1, 1\}$ is a variable representing the algorithm's guess of the label of $x_j \in U$. While the optimization problem is NP-hard, a number of heuristics have been developed. For example, Joachims [8] uses an iterative labeling heuristic to approximately optimize the objective. Semi-supervised SVMs have been shown to achieve high accuracy in a number of text classification domains where unlabeled data is plentiful [8].

Graph-Based Methods

Graph-based methods [3, 13] can be viewed as a (transductive) semi-supervised version of nearest-neighbor learning. In these methods, one creates a graph with a vertex for each example in $L \cup U$ and an edge between two examples x, x' if they are deemed to be sufficiently "similar" (or with edge weights based on *how* similar they are deemed to be). Similarity can be directly based on distance between the examples in the input space or given by some provided kernel function $k(x, x')$. Given the labels for the examples in L, one then finds a "most compatible" labeling for the examples in U, based on the belief that similar examples will typically have the same label. Specifically, in the mincut approach of [3], the labeling h produced is the cut of least total weight subject to agreeing with the known labels on examples in L or equivalently the cut that agrees with L minimizing $\sum_{e=(x,x')} w_e |h(x) - h(x')|$. In the algorithm of [13], in order to produce a smoother solution, the algorithm instead views the graph as an electrical network, finding the cut agreeing with L that minimizes $\sum_{e=(x,x')} w_e (h(x) - h(x'))^2$.

Open Problems

There are a number of open problems in developing computationally efficient semi-supervised learning algorithms. For example, can one extend the algorithm of Theorem 5 for Co-Training with linear separators to weaker conditions than independence given the label, while maintaining computational efficiency? (Note: A number of weaker conditions are known to produce good sample bounds if computational considerations are ignored [2].) More broadly, can one develop efficient algorithms for other classes or notions of compatibility that meet the cover-based sample complexity bounds of Theorem 2? Additional open problems are given in [1].

Recommended Reading

1. Balcan MF, Blum A (2010) A discriminative model for semi-supervised learning. J ACM

57(3):19:1–19:46. doi:10.1145/1706591.1706599. http://doi.acm.org/10.1145/1706591.1706599
2. Balcan MF, Blum A, Yang K (2004) Co-training and expansion: towards bridging theory and practice. In: Proceedings of 18th conference on neural information processing systems, Vancouver
3. Blum A, Chawla S (2001) Learning from labeled and unlabeled data using graph mincuts. In: Proceedings of 18th international conference on machine learning, Williams College
4. Blum A, Mitchell TM (1998) Combining labeled and unlabeled data with co-training. In: Proceedings of the 11th annual conference on computational learning theory, Madison, pp 92–100
5. Chapelle O, Schölkopf B, Zien A (eds) (2006) Semi-supervised learning. MIT, Cambridge. http://www.kyb.tuebingen.mpg.de/ssl-book
6. Collins M, Singer Y (1999) Unsupervised models for named entity classification. In: Proceedings of the joint SIGDAT conference on empirical methods in natural language processing and very large corpora, College Park, pp 189–196
7. Gupta S, Kim J, Grauman K, Mooney R (2008) Watch, listen & learn: co-training on captioned images and videos. In: Machine learning and knowledge discovery in databases (ECML PKDD). Lecture notes in computer science, vol 5211. Springer, Berlin/Heidelberg, pp 457–472. 10.1007/978-3-540-87479-9_48. http://dx.doi.org/10.1007/978-3-540-87479-9_48
8. Joachims T (1999) Transductive inference for text classification using support vector machines. In: Proceedings of 16th international conference on machine learning, Bled, pp 200–209
9. Levin A, Viola P, Freund Y (2003) Unsupervised improvement of visual detectors using co-training. In: Proceedings of the ninth IEEE international conference on computer vision, ICCV '03, vol 2, Nice. IEEE Computer Society, Washington, DC, pp 626–633. http://dl.acm.org/citation.cfm?id=946247.946615
10. Nigam K, Ghani R (2000) Analyzing the effectiveness and applicability of co-training. In: Proceedings of ACM CIKM international conference on information and knowledge management, McLean, pp 86–93
11. Vapnik V (1998) Statistical learning theory, vol 2. Wiley, New York
12. Zhu X (2006) Semi-supervised learning literature survey Computer sciences TR 1530 University of Wisconsin, Madison
13. Zhu X, Ghahramani Z, Lafferty J (2003) Semi-supervised learning using Gaussian fields and harmonic functions. In: Proceedings of 20th international conference on machine learning, Washington, DC, pp 912–919

Separators in Graphs

Goran Konjevod
Department of Computer Science and Engineering, Arizona State University, Tempe, AZ, USA

Keywords

Balanced cuts

Years and Authors of Summarized Original Work

1998; Leighton, Rao
1999; Leighton, Rao

Problem Definition

The (balanced) separator problem asks for a cut of minimum (edge)-weight in a graph, such that the two shores of the cut have approximately equal (node)-weight.

Formally, given an undirected graph $G = (V, E)$, with a nonnegative edge-weight function $c : E \to \mathbb{R}_+$, a nonnegative node-weight function $\pi : V \to \mathbb{R}_+$, and a constant $b \leq 1/2$, a cut $(S : V \setminus S)$ is said to be *b-balanced*, or a *$(b, 1-b)$-separator*, if $b\pi(V) \leq \pi(S) \leq (1-b)\pi(V)$ (where $\pi(S)$ stands for $\sum_{v \in S} \pi(v)$).

Problem 1 (b-balanced separator)
INPUT: Edge- and node-weighted graph $G = (V, E, c, \pi)$, constant $b \leq 1/2$.
OUTPUT: A b-balanced cut $(S : V \setminus S)$. Goal: minimize the edge weight $c(\delta(S))$.

Closely related is the *product sparsest cut problem*.

Problem 2 ((Product) Sparsest cut)
INPUT: Edge- and node-weighted graph $G = (V, E, c, \pi)$.
OUTPUT: A cut $(S : V \setminus S)$ minimizing the ratio-cost $\frac{c}{c}(\delta(S)))/(\pi(S)\pi(V \setminus S))$.

Problem 2 is the most general version of sparsest cut solved by Leighton and Rao. Setting all

S

node weights are equal to 1 leads to the uniform version, Problem 3.

Problem 3 ((Uniform) Sparsest cut)
INPUT: Edge-weighted graph $G = (V, E, c)$.
OUTPUT: A cut $(S : V \setminus S)$ minimizing the ratio-cost $(c(\delta(S)))/(|S||V \setminus S|)$.

Sparsest cut arises as the (integral version of the) linear programming dual of *concurrent multicommodity flow* (Problem 4). An instance of a multicommodity flow problem is defined on an edge-weighted graph by specifying for each of k *commodities* a *source* $s_i \in V$, a *sink* $t_i \in V$, and a *demand* D_i. A feasible solution to the multicommodity flow problem defines for each commodity a flow function on E, thus routing a certain amount of flow from s_i to t_i. The edge weights represent capacities, and for each edge e, a capacity constraint is enforced: the sum of all commodities' flows through e is at most the capacity $c(e)$.

Problem 4 (Concurrent multicommodity flow)
INPUT: Edge-weighted graph $G = (V, E, c)$, commodities $(s_1, t_1, D_1), \ldots (s_k, t_k, D_k)$.
OUTPUT: A multicommodity flow that routes fD_i units of commodity i from s_i to t_i for each i simultaneously, without violating the capacity of any edge. Goal: maximize f.

Problem 4 can be solved in polynomial time by linear programming, and approximated arbitrarily well by several more efficient combinatorial algorithms (section "Implementation"). The maximum value f for which there exists a multicommodity flow is called the *max-flow* of the instance. The *min-cut* is the minimum ratio $(c(\delta(S)))/(D(S, V \setminus S))$, where $D(S, V \setminus S) = \sum_{i:|\{s_i,t_i\}\cap S|=1} D_i$. This dual interpretation motivates the most general version of the problem, the *nonuniform sparsest cut* (Problem 5).

Problem 5 ((Nonuniform) Sparsest cut)
INPUT: Edge-weighted graph $G = (V, E, c)$, commodities $(s_1, t_1, D_1), \ldots (s_k, t_k, D_k)$.
OUTPUT: A min-cut $(S : V \setminus S)$, that is, a cut of minimum ratio-cost $(c(\delta(S)))/(D(S, V \setminus S))$.

(Most literature focuses on either the uniform or the general nonuniform version, and both of these two versions are sometimes referred to as just the "sparsest cut" problem.)

Key Results

Even when all (edge- and node-) weights are equal to 1, finding a minimum-weight b-balanced cut is NP-hard (for $b = 1/2$, the problem becomes *graph bisection*). Leighton and Rao [23, 24] give a pseudo-approximation algorithm for the general problem.

Theorem 1 *There is a polynomial-time algorithm that, given a weighted graph $G = (V, E, c, \pi)$, $b \leq 1/2$ and $b' < \min\{b, 1/3\}$, finds a b'-balanced cut of weight $O((\log n)/(b - b'))$ times the weight of the minimum b-balanced cut.*

The algorithm solves the sparsest cut problem on the given graph, puts aside the smaller-weight shore of the cut, and recurses on the larger-weight shore until both shores of the sparsest cut found have weight at most $(1-b')\pi(G)$. Now the larger-weight shore of the last iteration's sparsest cut is returned as one shore of the balanced cut, and everything else as the other shore. Since the sparsest cut problem is itself NP-hard, Leighton and Rao first required an approximation algorithm for this problem.

Theorem 2 *There is a polynomial-time algorithm with approximation ratio $O(\log p)$ for product sparsest cut (Problem 2), where p denotes the number of nonzero-weight nodes in the graph.*

This algorithm follows immediately from Theorem 3.

Theorem 3 *There is a polynomial-time algorithm that finds a cut $(S : V \setminus S)$ with ratio-cost $(c(\delta(S)))/(\pi(S)\pi(V \setminus S)) \in O(f \log p)$, where*

f is the max-flow for the product multicommodity flow and p the number of nodes with nonzero weight.

The proof of Theorem 3 is based on solving a linear programming formulation of the multicommodity flow problem and using the solution to construct a sparse cut.

Related Results

Shahrokhi and Matula [27] gave a max-flow min-cut theorem for a special case of the multicommodity flow problem and used a similar LP-based approach to prove their result. An $O(\log n)$ upper bound for arbitrary demands was proved by Aumann and Rabani [6] and Linial et al. [26]. In both cases, the solution to the dual of the multicommodity flow linear program is interpreted as a finite metric and embedded into ℓ_1 with distortion $O(\log n)$, using an embedding due to Bourgain [10]. The resulting ℓ_1 metric is a convex combination of cut metrics, from which a cut can be extracted with sparsity ratio at least as good as that of the combination.

Arora et al. [5] gave an $O(\sqrt{\log n})$ pseudo-approximation algorithm for (uniform or product-weight) balanced separators, based on a semidefinite programming relaxation. For the nonuniform version, the best bound is $O(\sqrt{\log n}\log\log n)$ due to Arora et al. [4]. Khot and Vishnoi [18] showed that, for the nonuniform version of the problem, the semidefinite relaxation of [5] has an integrality gap of at least $(\log\log n)^{1/6-\delta}$ for any $\delta > 0$, and further, assuming their Unique Games Conjecture, that it is NP-hard to (pseudo)-approximate the balanced separator problem to within any constant factor. The SDP integrality gap was strengthened to $\Omega(\log\log n)$ by Krauthgamer and Rabani [20]. Devanur et al. [11] show an $\Omega(\log\log n)$ integrality gap for the SDP formulation even in the uniform case.

Implementation

The bottleneck in the balanced separator algorithm is solving the multicommodity flow linear program. There exists a substantial amount of work on fast approximate solutions to such linear programs [19, 22, 25]. In most of the following results, the algorithm produces a $(1+\epsilon)$-approximation, and its hidden constant depends on ϵ^{-2}. Garg and Könemann [15], Fleischer [14] and Karakostas [16] gave efficient approximation schemes for multicommodity flow and related problems, with running times $\tilde{O}((k+m)m)$ [15] and $\tilde{O}(m^2)$ [14, 16]. Benczúr and Karger [7] gave an $O(\log n)$ approximation to sparsest cut based on randomized minimum cut and running in time $\tilde{O}(n^2)$. The current fastest $O(\log n)$ sparsest cut (balanced separator) approximation is based on a primal-dual approach to semidefinite programming due to Arora and Kale [3], and runs in time $O(m+n^{3/2})(\tilde{O}(m+n^{3/2})$, respectively). The same paper gives an $O(\sqrt{\log n})$ approximation in time $O(n^2)(\tilde{O}(n^2)$, respectively), improving on a previous $\tilde{O}(n^2)$ algorithm of Arora et al. [2]. If an $O(\log^2 n)$ approximation is sufficient, then sparsest cut can be solved in time $\tilde{O}(n^{3/2})$, and balanced separator in time $\tilde{O}(m+n^{3/2})$ [17].

Applications

Many problems can be solved by using a balanced separator or sparsest cut algorithm as a subroutine. The approximation ratio of the resulting algorithm typically depends directly on the ratio of the underlying subroutine. In most cases, the graph is recursively split into pieces of balanced size. In addition to the $O(\log n)$ approximation factor required by the balanced separator algorithm, this leads to another $O(\log n)$ factor due to the recursion depth. Even et al. [12] improved many results based on balanced separators by using *spreading metrics*, reducing the approximation guarantee to $O(\log n\log\log n)$ from $O(\log^2 n)$.

Some applications are listed here; where no reference is given, and for further examples, see [24].

- Minimum cut linear arrangement and minimum feedback arc set. One single algorithm provides an $O(\log^2 n)$ approximation for both of these problems.

- Minimum chordal graph completion and elimination orderings [1]. Elimination orderings are useful for solving sparse symmetric linear systems. The $O(\log^2 n)$ approximation algorithm of [1] for chordal graph completion has been improved to $O(\log n \log\log n)$ by Even et al. [12].
- Balanced node cuts. The cost of a balanced cut may be measured in terms of the weight of nodes removed from the graph. The balanced separator algorithm can be easily extended to this node-weighted case.
- VLSI layout. Bhatt and Leighton [8] studied several optimization problems in VLSI layout. Recursive partitioning by a balanced separator algorithm leads to polylogarithmic approximation algorithms for crossing number, minimum layout area and other problems.
- Treewidth and pathwidth. Bodlaender et al. [9] showed how to approximate treewidth within $O(\log n)$ and pathwidth within $O(\log^2 n)$ by using balanced node separators.
- Bisection. Feige and Krauthgamer [13] gave an $O(\alpha \log n)$ approximation for the minimum bisection, using any α-approximation algorithm for sparsest cut.

Experimental Results

Lang and Rao [21] compared a variant of the sparsest cut algorithm from [24] to methods used in graph decomposition for VLSI design.

Cross-References

- ▸ Fractional Packing and Covering Problems
- ▸ Minimum Bisection
- ▸ Sparsest Cut

Recommended Reading

Further details and pointers to additional results may be found in the survey [28].

1. Agrawal A, Klein PN, Ravi R (1993) Cutting down on fill using nested dissection: provably good elimination orderings. In: Brualdi RA, Friedland S, Klee V (eds) Graph theory and sparse matrix computation. IMA volumes in mathematics and its applications. Springer, New York, pp 31–55
2. Arora S, Hazan E, Kale S (2004) $O(\sqrt{\log n})$ approximation to sparsest cut in $\tilde{O}(n^2)$ time. In: FOCS '04: proceedings of the 45th annual IEEE symposium on foundations of computer science (FOCS'04). IEEE Computer Society, Washington, pp 238–247
3. Arora S, Kale S (2007) A combinatorial, primal-dual approach to semidefinite programs. In: STOC '07: proceedings of the 39th annual ACM symposium on theory of computing. ACM, pp 227–236
4. Arora S, Lee JR, Naor A (2005) Euclidean distortion and the sparsest cut. In: STOC '05: proceedings of the thirty-seventh annual ACM symposium on theory of computing. ACM, New York, pp 553–562
5. Arora S, Rao S, Vazirani U (2004) Expander flows, geometric embeddings and graph partitioning. In: STOC '04: proceedings of the thirty-sixth annual ACM symposium on theory of computing. ACM, New York, pp 222–231
6. Aumann Y, Rabani Y (1998) An (log) approximate min-cut maxflow theorem and approximation algorithm. SIAM J Comput 27(1):291–301
7. Benczúr AA, Karger DR (1996) Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In: STOC '96: proceedings of the twenty-eighth annual ACM symposium on theory of computing. ACM, New York, pp 47–55
8. Bhatt SN, Leighton FT (1984) A framework for solving vlsi graph layout problems. J Comput Syst Sci 28(2):300–343
9. Bodlaender HL, Gilbert JR, Hafsteinsson H, Kloks T (1995) Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. J Algorithms 18(2):238–255
10. Bourgain J (1985) On Lipshitz embedding of finite metric spaces in Hilbert space. Isr J Math 52:46–52
11. Devanur NR, Khot SA, Saket R, Vishnoi NK (2006) Integrality gaps for sparsest cut and minimum linear arrangement problems. In: STOC '06: proceedings of the thirty-eighth annual ACM symposium on theory of computing. ACM, New York, pp 537–546
12. Even G, Naor JS, Rao S, Schieber B (2000) Divide-and-conquer approximation algorithms via spreading metrics. J ACM 47(4):585–616
13. Feige U, Krauthgamer R (2002) A polylogarithmic approximation of the minimum bisection. SIAM J Comput 31(4):1090–1118
14. Fleischer L (2000) Approximating fractional multicommodity flow independent of the number of commodities. SIAM J Discret Math 13(4):505–520
15. Garg N, Könemann J (1998) Faster and simpler algorithms for multicommodity flow and other fractional packing problems. In: FOCS '98: proceedings of the 39th annual symposium on foundations of computer science. IEEE Computer Society, Washington, p 300

16. Karakostas G (2002) Faster approximation schemes for fractional multicommodity flow problems. In: SODA '02: proceedings of the thirteenth annual ACM-SIAM symposium on discrete algorithms. Society for Industrial and Applied Mathematics, Philadelphia, pp 166–173
17. Khandekar R, Rao S, Vazirani U (2006) Graph partitioning using single commodity flows. In: STOC '06: proceedings of the thirty-eighth annual ACM symposium on theory of computing. ACM, New York, pp 385–390
18. Khot S, Vishnoi NK (2005) The unique games conjecture, integrality gap for cut problems and embeddability of negative type metrics into l_1. In: FOCS '07: proceedings of the 46th annual IEEE symposium on foundations and computer science. IEEE Computer Society, pp 53–62
19. Klein PN, Plotkin SA, Stein C, Tardos É (1994) Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. SIAM J Comput 23(3):466–487
20. Krauthgamer R, Rabani Y (2006) Improved lower bounds for embeddings into l_1. In: SODA '06: proceedings of the seventeenth annual ACM-SIAM symposium on discrete algorithm. ACM, New York, pp 1010–1017
21. Lang K, Rao S (1993) Finding near-optimal cuts: an empirical evaluation. In: SODA '93: proceedings of the fourth annual ACM SIAM symposium on discrete algorithms. Society for Industrial and Applied Mathematics, Philadelphia, pp 212–221
22. Leighton FT, Makedon F, Plotkin SA, Stein C, Stein É, Tragoudas S (1995) Fast approximation algorithms for multicommodity flow problems. J Comput Syst Sci 50(2):228–243
23. Leighton T, Rao S (1988) An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In: Proceedings of the 29th annual symposium on foundations of computer science. IEEE Computer Society, Washington, DC, pp 422–431
24. Leighton T, Rao S (1999) Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. J ACM 46(6):787–832
25. Leong T, Shor P, Stein C (1991) Implementation of a combinatorial multicommodity flow algorithm. In: Johnson DS, McGeoch CC (eds) Network flows and matching. DIMACS series in discrete mathematics and theoretical computer science, vol 12, AMS, Providence, pp 387–406
26. Linial N, London E, Rabinovich Y (1995) The geometry of graphs and some of its algorithmic applications. Combinatorica 15(2):215–245
27. Shahrokhi F, Matula DW (1990) The maximum concurrent flow problem. J ACM 37(2):318–334
28. Shmoys DB (1997) Cut problems and their applications to divide-and-conquer. In: Hochbaum DS (ed) Approximation algorithms for NP-hard problems. PWS Publishing Company, pp 192–235

Sequence and Spatial Motif Discovery in Short Sequence Fragments

Jie Liang[1] and Ronald Jackups[2]
[1]Department of Bioengineering, University of Illinois, Chicago, IL, USA
[2]Department of Pediatrics, Washington University, St. Louis, MO, USA

Keywords

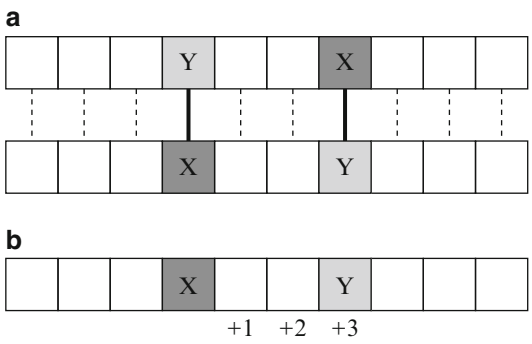
Internally random model; Permutation model; Positional null model; Sequence motif; Spatial motif; String pairing pattern; String pattern

Years and Authors of Summarized Original Work

2005; Jackups, Jr and Liang
2006; Jackups, Jr, Cheng, Liang
2010; Jackups, Jr and Liang

Problem Definition

The problem is to detect specific patterns in string and patterns in string pairs for discovery of sequence and spatial motifs in membrane proteins. A *spatial interaction* motif of residue pair X-Y is defined as a pattern in which a character or residue of type X is found interacting with a residue of type Y on two strings or sequences (Fig. 1a). We define a *sequence pair* XYk as a pattern in which a residue of type Y is found at the k-th position from a residue of type X along a single sequence (Fig. 1b). The propensity $P(X, Y)$ of residue pairing XY is $P(X, Y) = \frac{f_{\text{obs}}(X,Y)}{\mathbb{E}[f(X,Y)]}$, where $f_{\text{obs}}(X, Y)$ is the observed count of XY patterns and $\mathbb{E}[f(X, Y)]$ is the expected count of XY patterns according to some random null model. We define a *motif* as a residue pair with propensity >1.0 (or greater than some other predefined limit) and statistically significant.



Sequence and Spatial Motif Discovery in Short Sequence Fragments, Fig. 1 Examples of spatial and sequence patterns. (**a**) Two X-Y spatial patterns on interacting sequences. (**b**) An $XY3$ sequence pattern

The null model for calculating $\mathbb{E}[f(X,Y)]$ is critical for motif detection. For short sequence fragments, the null model for spatial motif detection cannot be the χ^2 distribution as was used in [13], since the assumption of Gaussian distribution is not valid for short sequences. The null model for sequence motif detection cannot be the binomial distribution as was used in [4, 10], since the assumption of drawing from a universal population with replacement is unrealistic for short sequence fragments. Instead, we use a combinatorial model called the *permutation model* more effective for discoveries of motifs [5–7]. This null model is similar for both pair types: the residues within each sequence are exhaustively and independently permuted *without replacement*, and each permutation occurs with equal probability. This model has been called the *internally random* model [6]. This permutation model is further extended to positional null model to correct position-specific bias in residue distributions [6].

Objective. Our task is to determine explicit formulas to calculate $\mathbb{E}[f(X,Y)]$ for each pair type under different conditions. Explicit probability distributions for $f(X,Y)$ can also be found for many special cases, which will allow for the calculation of statistical significance p-values. These formulas can also be used to study whole datasets of short sequences.

Key Results

Spatial Motifs by Permutation Model

Expectation for Interacting Residues of the Same Type

For cases in which X is the same as Y (i.e., X-X pairs), let x_1 be the number of residues of type X in the first sequence, x_2 the number of residues of type X in the second sequence, and l the common length of the sequence pair. The probability $\mathbb{P}_{XX}(i)$ of exactly $i = f(X,X)$ number of X-X contacts follows a hypergeometric distribution: $\mathbb{P}_{XX}(i) = \binom{x_1}{i}\binom{l-x_1}{x_2-i}/\binom{l}{x_2}$. Its expectation $\mathbb{E}[f(X,X)]$ is then:

$$\mathbb{E}[f(X,X)] = \frac{x_1 x_2}{l}.$$

Expectation for Interacting Residues of Different Types

When $X \neq Y$, the number of X-Y contacts in the permutation model for one sequence pair is the sum of two dependent hypergeometric variables, one variable for type X residues in the first sequence s_1 and type Y in the second sequence s_2, and another variable for type Y residues in s_1 and type X in s_2. The expected number of X-Y contacts $\mathbb{E}[f(X,Y)]$ is the sum of the two expected values $\mathbb{E}[f(X,Y|X \in s_1, Y \in s_2)] + \mathbb{E}[f(X,Y|Y \in s_1, X \in s_2)]$:

$$\mathbb{E}[f(X,Y)] = \frac{x_1 y_2}{l} + \frac{y_1 x_2}{l},$$

where x_1 and x_2 are the numbers of residues of type X in the first and second sequence, respectively, y_1 and y_2 are the numbers of residues of type Y in the first and second sequence, respectively, and l is the length of the sequence pair.

Significance of Spatial Motifs

To calculate the statistical significance in the form of p-value of interacting residues of the same type, two-tailed p-values can be calculated using the hypergeometric distribution for a dataset of sequence pairs.

For interacting residues of different types, the formula to determine the p-value for a specific

observed number of X-Y contacts is more complex because of the dependency. We define a 3-element multinomial function $M(a,b,c) \equiv \frac{a!}{b!c!(a-b-c)!}$, where $M(a,b,c) = 0$ if $a-b-c < 0$. This represents the number of distinct permutations, without replacement, in a multiset of size a containing three different types of elements, with number count b, c, and $a-b-c$ of each of the three element types.

The probability $\mathbb{P}(h,i,j,k)$ of inter-sequence matches, namely, the probability of h X-X contacts, i X-Y contacts, j Y-X contacts, and k Y-Y contacts occurring in a random permutation is (Fig. 2)

$$\mathbb{P}(h,i,j,k) = \frac{M(x_1,h,i)\cdot M(y_1,j,k)\cdot M(l-x_1-y_1, x_2-h-j, y_2-i-k)}{M(l,x_2,y_2)}.$$

The marginal probability $\mathbb{P}_{XY}(m)$ that there are a total of $i+j=m$ X-Y contacts is

$$\mathbb{P}_{XY}(m) = \sum_{h=0}^{x_1} \sum_{i=0}^{x_1-h} \sum_{k=0}^{y_1-(m-i)} \mathbb{P}(h,i,m-i,k).$$

There are x_1 possible values for h, one for each residue of type X on sequence 1; x_1-h possible values for i, once h has been determined; and $y_1-j = y_1-(m-i)$ possible values for k, once i has been determined. The i number of X-Y contacts plus the $m-i$ number of Y-X contacts will sum to the m number of contacts desired.

This closed-form formula allows calculation of p-values analytically. The running time is $O(l^4)$, due to the presence of 3 summations and $l!$ in the summand. For short sequences, the computing cost is not prohibitive.

Sequences of Different Lengths

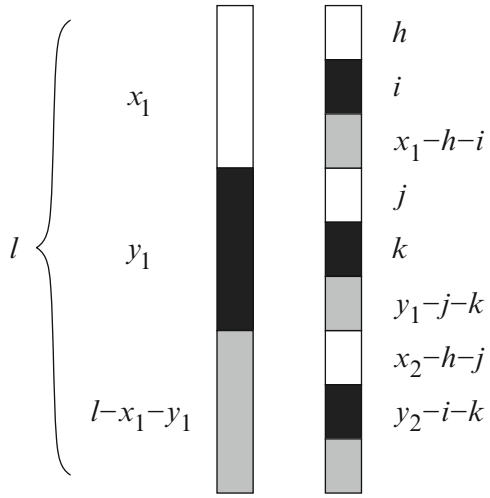
The requirement for interacting sequences to be of the same length may be relaxed by introducing a 21st "dummy" amino acid type. All unpaired residues in the longer member of a sequence pair will be paired to this extra amino acid type, and our standard method can be applied to determine the propensity of unpaired amino acids (i.e., residues paired with the "dummy" amino acid type).

Sequence Motifs by Permutation Model

The propensity $P(X,Y|k)$ for the XYk pattern of two ordered intrasequence residues of type X and type Y that are k positions away on the same sequence (Fig. 1b) is $P(X,Y|k) = \frac{f_{\text{obs}}(X,Y|k)}{\mathbb{E}[f(X,Y|k)]}$, where $f_{\text{obs}}(X,Y|k)$ is the observed count of XYk patterns, and $\mathbb{E}[f(X,Y|k)]$ is the expected count of XYk patterns.

Expectation of XYk and XXk Two-Residue Motifs

We can regard $f(X,Y|k)$ as the sum of identical Bernoulli variables $f_t(X,Y|k)$, each of which equals 1 if one of the x number of residues of type X occurs at position t in the sequence and one of the y number of residues of type Y occurs



Sequence and Spatial Motif Discovery in Short Sequence Fragments, Fig. 2 Division of residues in spatial motif analysis when $X \neq Y$. *White* = X, *black* = Y, *gray* = "neither" X or Y

at position $t + k$ or equals 0 otherwise. Since an XYk pattern cannot occur if $t > l - k$, we concern ourselves only with the first $l - k$ positions. We have: $\mathbb{E}[f_t(X, Y|k)] = \mathbb{P}[f_t(X, Y|k) = 1] = \frac{x}{l} \cdot \frac{y}{(l-1)}$ if $t \leq l-k$. There are $l-k$ such identical variables, and their expectations may be summed as

$$\mathbb{E}[f(X, Y|k)] = (l - k)\frac{xy}{l(l-1)}, \quad (1)$$

where l is the length of the sequence, x is the number of residues of type X, and y is the number of residues of type Y.

For XXk patterns, the expectation is calculated as

$$\mathbb{E}[f(X, X|k)] = (l - k)\frac{x(x-1)}{l(l-1)}, \quad (2)$$

as there will be $x - 1$ residues available to place the second X residue at position $t + k$ after the first X residue is placed at t. Although these Bernoulli random variables are dependent (i.e., the placement of one XYk pattern will affect the probability of another XYk pattern), their expectations may be summed, because expectation is a linear operator.

Significance of XYk and XXk Two-Residue Sequence Motifs

To calculate statistical significance p-values, several formulas have been derived to determine $\mathbb{P}_{XYk}(i)$, the probability of the occurrence of $i = f(X, Y|k)$ XYk patterns for different k values.

1. **Sequence motifs when $k = 1$.** We have

$$\mathbb{P}_{XY1}(i) = \frac{\binom{l-y}{x}\binom{x}{i}\binom{l-x}{y-i}}{\frac{l!}{x!y!(l-x-y)!}} = \frac{\binom{x}{i}\binom{l-x}{y-i}}{\binom{l}{y}}, \quad \text{and}$$

$$\mathbb{P}_{XX1}(i) = \frac{\binom{l-x+1}{x-i}\binom{x-1}{i}}{\binom{l}{x}},$$

with the convention that $\binom{n}{r} = 0$ if $n < r$.

2. **Sequence motifs with residues of different types and if $x \leq 2$ or $y \leq 2$.**
   - If either $x = 1$ or $y = 1$, we have

$$\mathbb{P}_{XYk}(1) = (l - k)\frac{xy}{l(l-1)}.$$

   For $i = 0$, we have simply $\mathbb{P}_{XYk}(0) = 1 - \mathbb{P}_{XYk}(1)$.
   - If $x = 2$ or $y = 2$, the probability of two XYk patterns is

$$\mathbb{P}_{XYk}(2) = \frac{\left[\binom{l-k}{2} - (l - 2k)\right]}{\frac{l(l-1)(l-2)(l-3)}{x(x-1)y(y-1)}}.$$

   We also have for the probabilities of exactly one XYk pattern or zero pattern:

$$\mathbb{P}_{XYk}(1) = \mathbb{E}[f(XYk)] - 2\mathbb{P}_{XYk}(2) \quad \text{and}$$

$$\mathbb{P}_{XYk}(0) = 1 - [\mathbb{P}_{XYk}(1) + \mathbb{P}_{XYk}(2)].$$

3. **Sequence motifs with residues of the same type if $x \leq 3$.**
   - If $x = 2$, the probability of one XXk pattern is

$$\mathbb{P}_{XXk}(1) = \mathbb{E}[f(XXk)] = (l-k)\frac{x(x-1)}{l(l-1)},$$

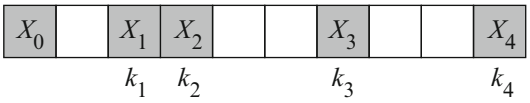
   The probability of no XXk pattern is

$$\mathbb{P}_{XXk}(0) = 1 - \mathbb{P}_{XXk}(1).$$

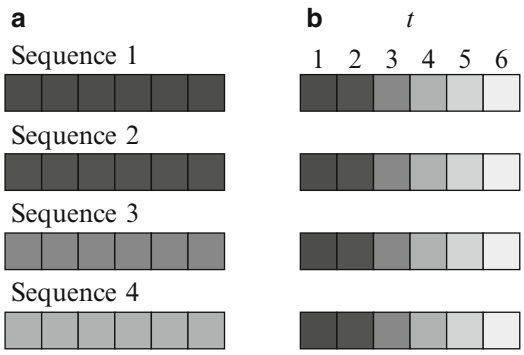
   - If $x = 3$, the probability of exactly two XXk patterns is

$$\mathbb{P}_{XXk}(2) = \frac{l - 2k}{\binom{l}{x}},$$

4. **Sequence motifs with $k > 1$, $x > 2$, and $y > 2$.** When $k > 1$, $x > 2$, and $y > 2$, the analytical formulas for $\mathbb{P}_{XYk}(i)$ become very complicated. However, when the sequences in the dataset used are short, it is possible to fully enumerate all permutations of a sequence and calculate $\mathbb{P}_{XYk}(i)$ and p-values exactly, as shown by Senes et al. [11]. Because x and y are usually small in short sequences, the computation time needed for motif analysis of short sequences is not prohibitive.



Sequence and Spatial Motif Discovery in Short Sequence Fragments, Fig. 3 Example of a multi-residue sequence pattern as described in the text. This pattern contains five specified residues in a span of ten residues. Here, X_0, X_1, X_2, X_3, and X_4 are specified amino acid types, and the corresponding k values are counted as the distance from the first position of the sequence (i.e., the position occupied by X_0). Thus, $k_1 = 2$, $k_2 = 3$, $k_3 = 6$, and $k_4 = 9$. All other residues (in *white*) are unspecified and may be any amino acid type. This pattern is written as $(X_0, X_1, X_2, X_3, X_4 \mid 2, 3, 6, 9)$



Sequence and Spatial Motif Discovery in Short Sequence Fragments, Fig. 4 Difference between (**a**) a *permutation* null model for sequence motif analysis and (**b**) a *position-dependent* null model. In both cases, only residues of the same shade are permuted with each other. In (**a**), residues are permuted only within each sequence individually, while in (**b**), residues are permuted across sequences but only within their specified position t

Propensity of Multi-residue Sequence Motifs

We now discuss the expected number $\mathbb{E}[f(X_0, X_1, X_2, \ldots, X_n | k_1, k_2, \ldots, k_n)]$ of a specific pattern containing $n+1$ residues placed in a contiguous subsequence of $k_n + 1$ residues ($k_n \geq n$). Here X_i is the residue type of the i-th fixed residue in the pattern and k_i is the position of this residue from the 0-th residue ($k_0 = 0$). Positions not specified by k_i can be any residue type. For example, the pattern $(A, L, Y | 2, 4)$ is written as AL2Y4 and represents AxLxY. A graphic example is shown in Fig. 3. Many examples of these multi-residue sequence motifs in proteins have been discovered, including the GxGxxG NADH binding motif [1] and the RSxSxP 14-3-3 binding motif [14].

The expected value can be calculated as:

$$\mathbb{E}[f(X_0, X_1, X_2, \ldots, X_n | k_1, k_2, \ldots, k_n)] = (l - k_n) \frac{\prod_{i=0}^{n} [x_i - \#(\mathbb{I}(X_i))]}{\frac{l!}{(l-n-1)!}}, \quad (3)$$

where x_i is the number of residues of type X_i, l is the length of the sequence, and $\#(\mathbb{I}(X_i))$ is the number of times residue type X_i appears in the "subpattern" $\{X_0, X_1, X_2, \ldots, X_{i-1}\}$.

Remark

The above discussions are for determining motifs in a single short sequence or sequence pair. This can be extended so analysis can be performed on a dataset of multiple short sequences to attain sufficient statistical significance. This has the advantage of capturing within-sequence relationships on a scale large enough to obtain reliable p-values. Details can be found in [6].

Spatial Motifs by Positional Null Model

When there are significant biases in residue preferences for certain positions in a sequence known a priori, e.g., the enrichment of aromatic residues at either end of a transmembrane α-helix or β-strand [12], these single-residue biases may confound two-residue propensities. The *positional null model* should be used for motif detection in such cases [6]. Instead of permuting residues across all positions within individual sequences, we permute residues across all sequences in a dataset within specific positions (Fig. 4).

Expectation and Significance of Interacting Residue Pairs

We allocate residues into regions, which do not overlap. Regions may have different lengths along the sequences. Interacting regions within a sequence pair are assumed to have equal length. If a residue in region r interacts with a residue in region s on a spatially adjacent sequence fragment, all residues in region r in the dataset

must only interact with residues in region s. For example, for interacting antiparallel β-strands, we divide each strand into three regions, the N-terminal, central core, and C-terminal regions, and all interacting strand pairs into two spatial pair types, N-terminal with C-terminal and core with core. We require that no core residue interact with an N-terminal or C-terminal residue.

The null model for position-dependent spatial motifs differs depending on whether paired residues are from the same region ($r = s$) or different regions ($r \neq s$), and whether the residue types in the pair are the same ($X = Y$) or different ($X \neq Y$).

1. **When $r = s$ and $X = Y$.** The expected value of X-X pairs in region r is

$$\mathbb{E}(X, X|rr) = \frac{\binom{x_r}{2}}{\binom{n_r}{2}} \cdot \frac{n_r}{2} = \frac{x_r(x_r - 1)}{2(n_r - 1)},$$

where n_r is the number of residues in region r. The probability $\mathbb{P}_{XX|rr}(i)$ of i X-X interacting pairs in region r in the dataset for p-values calculation calculated is

$$\mathbb{P}_{XX|rr}(i) = \frac{M(\frac{n_r}{2}, i, x_r - 2i) \cdot 2^{x_r - 2i}}{\binom{n_r}{x_r}},$$

where the 3-element multinomial function $M(a, b, c)$ is as defined before.

2. **When $r = s$ and $X \neq Y$.** The expected value when $X \neq Y$ is

$$\mathbb{E}(XY|rr) = \frac{x_r y_r}{\binom{n_r}{2}} \cdot \frac{n_r}{2} = \frac{x_r y_r}{n_r - 1}.$$

The probability $\mathbb{P}(i, j, k)$ of each combination of i, j, and k pairs of type X-Y, X-X, and Y-Y interactions, respectively, is

$$\mathbb{P}(i, j, k) = \frac{M(\frac{n_r}{2}, i, j, k, x_r - i - 2j, y_r - i - 2k) \cdot 2^{x_r + y_r - i - 2j - 2k}}{M(n_r, x_r, y_r)},$$

where the 6-variable multinomial function $M(a,b,c,d,e,f) \equiv \frac{a!}{b!c!d!e!f!(a-b-c-d-e-f)!}$. The probability $\mathbb{P}_{XY|rr}(i)$ of i X-Y pairs in the dataset is then

$$\mathbb{P}_{XY|rr}(i) = \sum_{j=0}^{\frac{x_r - i}{2}} \sum_{k=0}^{\frac{y_r - i}{2}} \mathbb{P}(i, j, k).$$

3. **When $r \neq s$.** We distinguish X_r, a residue of type X occurring in region r in one sequence, and X_s, a residue of type X occurring in region s in the other sequence. Thus, an X-Y pair, which we define as an $X_r - Y_s$ pair, is different from a Y-X pair, which is $Y_r - X_s$. Because there is a one-to-one correspondence between residues in region r and region s, $n_r = n_s$ is the total number of $r - s$ pairs.

In order for exactly i X-Y pairs to occur, i X_r residues must be drawn from a possible x_r residues of type X to match i Y_s residues drawn from a possible y_s residues of type Y. This can be modeled with a simple hypergeometric distribution. The expected value can be calculated as

$$\mathbb{E}(XY|rs) = \frac{x_r y_s}{n_r}.$$

The $\mathbb{P}_{XY|rs}(i)$ of i X-Y pairs is

$$\mathbb{P}_{XY|rs}(i) = \frac{\binom{x_r}{i}\binom{n_r - x_r}{y_s - i}}{\binom{n_r}{y_s}}.$$

Expectation and Significance of Sequence Motifs

We define the *positional residue frequency* x_t as the number of residues of type X occupying the t-th position of all sequences in the dataset. If sequences of different lengths are represented in the dataset, it is necessary to normalize t to be within an appropriate range $[1, l]$, to approximate

an average or predetermined sequence length of l:

$$t = \lceil \frac{l(t_{\text{obs}} - 0.5)}{l_{\text{obs}}} \rceil,$$

where $t_{\text{obs}} \in \{1, 2, 3, \cdots, l_{\text{obs}}\}$ is the actual position of the residue within its sequence, l_{obs} is the actual length of the sequence, $\lceil x \rceil$ represents the ceiling function, equal to the lowest integer greater than or equal to x, and the 0.5 factor is a correction for continuity to round to the next integer. This ensures that $1 \leq t \leq l$, no residues are removed from the model by truncation, and each position t will be represented by nearly the same number of residues.

For sequence motif, we use the model of permutation within each position in a sequence *with replacement* across all sequences. Although all other null models in this study rely on permutation without replacement, this model is based on datasets of multiple sequences instead of individual sequences, and the approximation of sampling without replacement will not be problematic once a sufficiently large sample of sequences is assembled.

1. XYk **motif at position** t**.** When $t \leq 1-k$, the probability of an XYk pattern at position t is

$$\mathbb{P}(X, Y|k, t) = \frac{x_t}{n_t} \cdot \frac{y_{t+k}}{n_{t+k}},$$

where x_t is the number of residues of type X in position t on all sequences, y_t is the number of residues of type Y in position t, and n_t is the number of all residues of all types in position t. This null model can be represented as a binomial distribution.

The expected frequency of XYk patterns at position t is

$$\mathbb{E}[f(X, Y|k, t)] = n_t \cdot \mathbb{P}(X, Y|k, t).$$

The probability of i XYk patterns at position t in the dataset is

$$\mathbb{P}_{XYk|t}(i) = \binom{n_t}{i} \mathbb{P}(X, Y|k, t)^i [1 - \mathbb{P}(X, Y|k, t)]^{n_t - i}.$$

Note that the probability that an XYk pattern appears at position t is 0 if $t > l - k$, as an XYk pattern would span across the end of a sequence of length l.

2. XYk **motif at any arbitrary position.** To calculate the dataset-wide probability of an XYk pattern at any arbitrary position of the sequence, we average $\mathbb{P}(X, Y|k, t)$ over all $l - k$ possible positions:

$$\mathbb{P}(X, Y|k) = \frac{1}{l-k} \sum_{t=1}^{l-k} \mathbb{P}(X, Y|k, t).$$

This can similarly be represented as a binomial distribution with probability distribution function: $\mathbb{P}_{XYk}(i) = \binom{n_k}{i} \mathbb{P}(X, Y|k)^i [1 - \mathbb{P}(X, Y|k)]^{n_k - i}$, where n_k is the number of all pairs of all residue types k residues apart in the dataset. The expected value is

$$\mathbb{E}[f(X, Y|k)] = n_k \cdot \mathbb{P}(X, Y|k).$$

Unlike the situation where only one position t is concerned, this distribution represents the sum of dependent Bernoulli variables. Methods of accounting for this dependence can be found in Robin et al. [10].

Applications

Several spatial and sequence motifs have been discovered using the approach discussed here [5–7]. The estimated propensities have also been used to develop empirical potential function for prediction of oligomerization stated [8], protein-protein interaction interfaces [3, 8], engineering of thermal resistance [2], and in predicting structures of β-barrel membrane proteins [9].

Open Problems

General analytical formulas for calculating probabilities of two-residue and multi-residue motifs under the permutation model are unknown.

Recommended Reading

1. Baker PJ, Britton KL, Rice DW, Rob A, Stillman TJ (1992) Structural consequences of sequence patterns in the fingerprint region of the nucleotide binding fold. Implications for nucleotide specificity. J Mol Biol 228(2):662–671
2. Gessmann D, Mager F, Naveed H, Arnold T, Weirich S, Linke D, Liang J, Nussberger S (2011) Improving the resistance of a eukaryotic beta-barrel protein to thermal and chemical perturbations. J Mol Biol 413(1):150–161. doi:10.1016/j.jmb.2011.07.054
3. Geula S, Naveed H, Liang J, Shoshan-Barmatz V (2012) Structure-based analysis of VDAC1 protein: defining oligomer contact sites. J Biol Chem 287(3):2179–2190
4. Hart R, Royyuru A, Stolovitzky G, Califano A (2000) Systematic and fully automated identification of protein sequence patterns. J Comput Biol 7(3–4):585–600
5. Jackups R Jr, Liang J (2005) Interstrand pairing patterns in beta-barrel membrane proteins: the positive-outside rule, aromatic rescue, and strand registration prediction. J Mol Biol 354(4):979–993
6. Jackups R Jr, Liang J (2010) Combinatorial analysis for sequence and spatial motif discovery in short sequence fragments. IEEE/ACM Trans Comput Biol Bioinform 7(3):524–536
7. Jackups R Jr, Cheng S, Liang J (2006) Sequence motifs and antimotifs in beta-barrel membrane proteins from a genome-wide analysis: the ala-tyr dichotomy and chaperone binding motifs. J Mol Biol 363(2):611–623
8. Naveed H, Jackups R Jr, Liang J (2009) Predicting weakly stable regions, oligomerization state, and protein-protein interfaces in transmembrane domains of outer membrane proteins. Proc Natl Acad Sci USA 106(31):12735–12740. doi:10.1073/pnas.0902169106
9. Naveed H, Xu Y, Jackups R, Liang J (2012) Predicting three-dimensional structures of transmembrane domains of β-barrel membrane proteins. J Am Chem Soc 134(3):1775–1781
10. Robin S, Rodophe F, Schbath S (2005) DNA, words and models: statistics of exceptional words. Cambridge University Press, Cambridge/New York
11. Senes A, Gerstein M, Engelman DM (2000) Statistical analysis of amino acid patterns in transmembrane helices: the GxxxG motif occurs frequently and in association with β-branched residues at neighboring positions. J Mol Biol 296:921–936
12. Wimley WC (2002) Toward genomic identification of β-barrel membrane proteins: composition and architecture of known structures. Protein Sci 11:301–312
13. Wouters MA, Curmi PM (1995) An analysis of side chain interactions and pair correlations within antiparallel β-sheets: the differences between backbone hydrogen-bonded and non-hydrogen-bonded residue pairs. Proteins 22:119–131
14. Yaffe MB, Rittinger K, Volinia S, Caron PR, Aitken A, Leffers H, Gamblin SJ, Smerdon SJ, Cantley LC (1997) The structural basis for 14-3-3:phosphopeptide binding specificity. Cell 91(7):961–971

Sequential Circuit Technology Mapping

Peichen Pan
Xilinx, Inc., San Jose, CA, USA

Keywords

Boolean network; EDA; FPGA; Retiming; Technology mapping; VLSI CAD

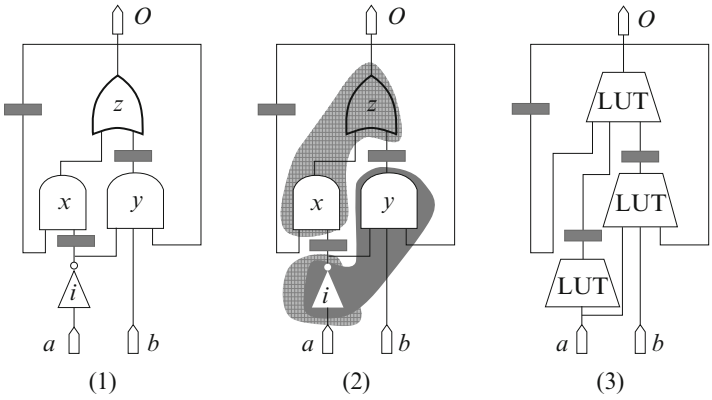
Years and Authors of Summarized Original Work

1996; Pan, Liu
1998; Pan, Liu
1998; Pan, Lin

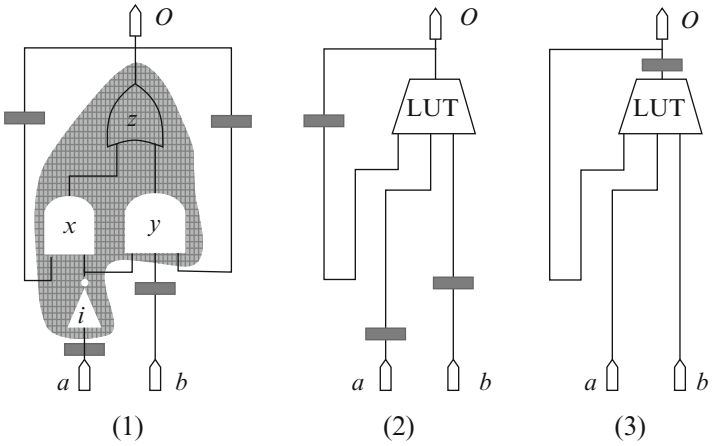
Problem Definition

One of the key steps in a VLSI design flow is technology mapping that converts a Boolean network of technology-independent logic gates and D-flipflops (FFs) into an equivalent one comprised of cells from a technology library [1, 4]. Technology mapping can be formulated as a covering problem where logic gates are covered by cells from the technology library. For ease of discussion, it is assumed that the cell library contains only one cell, a K-input lookup table (K-LUT) with one unit of delay. A K-LUT can implement any Boolean function with up to K inputs as is the case in field-programmable gate arrays (FPGAs) [1, 3].

Figure 1 shows an example of technology mapping. The original network in (1) with three FFs and four gates is covered by three 3-input cones as indicated in (2). The corresponding



Sequential Circuit Technology Mapping, Fig. 1 Technology mapping: (**1**) original network, (**2**) covering, (**3**) mapping solution



Sequential Circuit Technology Mapping, Fig. 2 Retiming and mapping: (**1**) retiming and covering, (**2**) mapping solution, (**3**) retimed solution

mapping solution using 3-LUTs is shown in (3). Note that gate i is covered by two cones so it will be replicated. The mapping solution has a *cycle time* (or *clock period*) of two units, which is the maximum number of LUTs on all paths without FFs.

Retiming relocates FFs in a network by moving FFs across logic nodes backward or forward [5]. Retiming does not alter the functionality of a network. Figure 2 (1) shows the network obtained from the one in Fig. 1 (1) by moving the FFs at the output of gates y and i to their inputs. It can now be covered with just one 3-input cone as indicated in (1). The corresponding mapping solution shown in (2) is better in both cycle time and area than the one in Fig. 1 (3) obtained without retiming.

A *K-bounded* network is one in which each gate has at most K inputs. The sequential circuit technology mapping problem can be defined as follows: *Given a K-bounded Boolean network N and a target cycle time ϕ, find a mapping solution with a cycle time of ϕ, assuming FFs can be relocated using retiming*.

Key Results

The first polynomial time algorithm for the problem was proposed in [9, 10]. An improved algorithm was proposed in [2] to reduce runtime. Both algorithms are based on min-cost flow computation.

In [8], an efficient algorithm was proposed to take advantage of the fact that K is a small integer usually between 3 and 6. The algorithm is based

Sequential Circuit Technology Mapping, Fig. 3 Cut enumeration procedure

FindAllCuts(N, K)

foreach node v in N **do** C(v) $\Leftarrow \{\{v^0\}\}$

while (*new cuts discovered*) **do**

foreach node v in N **do** $C(v) \Leftarrow merge(C(u_1),....,C(u_t))$

on enumerating all K-input cones for each gate and will be described next.

Cut Enumeration

A Boolean network can be represented as an edge-weighted directed graph where the nodes denote logic gates and primary inputs/outputs. There is a directed edge (u, v) with weight d if u, after going through d FFs, drives v.

A *cone* for a node can be captured by a *cut* consisting of inputs to the cone. An element in a cut for v consists of the driving node u and the total weight d on the paths from u to v, denoted by u^d. If u can reach v on several paths with different FF counts, u will appear in the cut multiple times with different ds. For the cone for z in Fig. 2 (2), the corresponding cut is $\{z^1, a^1, b^1\}$. A cut of size K is called a K-cut.

Let (u_i, v), where $i = 1, \ldots, t$, be all input edges to v in N. Further assume the weight of (u_i, v) is d_i and $C(u_i)$ is a set of K-cuts for u_i. Let $merge(C(u_1), \ldots, C(u_t))$ denote the following set operation:

$$\{\{v^0\}\} \cup \{c_1^{d_1} \cup \cdots \cup c_t^{d_t} | c_1 \in (u_1), \ldots, c_t \in C(u_t), |c_1^{d_1} \cup \cdots \cup c_t^{d_t}| \leq K\}$$

where $c_i^{d_i} = \{u^{d+d_i} | u^d \in c_i\}$ for $i = 1, \ldots, t$. It is obvious that $merge(C(u_1), \ldots, C(u_t))$ is a set of K-cuts for v.

If the network N does not contain cycles, the K-cuts of all nodes can be determined using the merge operation in a topological order starting from the PIs. For general networks, Fig. 3 outlines the iterative cut computation procedure proposed in [8].

Figure 4 depicts the iterations in enumerating the 3-cuts for the network in Fig. 1 (1) where cuts are merged in the order i, x, y, z, and o. At the beginning, every node has a trivial cut formed by itself (Row 0). Row 1 shows the new cuts discovered in the first iteration. In second iteration, two more cuts are discovered (for x). After that, further merging does not yield any new cut and the procedure stops.

Lemma 1 *After at most Kn iterations, the cut enumeration procedure will find all the K-cuts for every node in N.*

Techniques have been proposed to speed up the procedure [8]. For practical networks, the cut enumeration procedure typically converges in just a few iterations.

Label Computation

After obtaining all K-cuts, the cuts are evaluated based on sequential arrival times (or l-values), which is an extension of traditional arrival times, to consider the effect of retiming [7, 9].

The labeling procedure tries to find a label for each node as outlined in Fig. 5, where w_v denotes the weight of the shortest paths from PIs to node v.

Figure 6 shows the iterations for label computation for the network in Fig. 1 (1), assuming that the target cycle time $\phi = 1$ and the nodes are evaluated in the order of i, x, y, z, and o. In the table, the current label as well as a corresponding cut for each node is listed. In this example, after the first iteration, none of the labels will change and the procedure stops.

It can be shown that the labeling procedure will stop after at most $n(n-1)$ iterations [10]. The following lemma relates labels to mapping:

Iter	*a*	*B*	*I*	*x*	*y*	*z*	*o*
0	$\{a^0\}$	$\{b^0\}$	$\{i^0\}$	$\{x^0\}$	$\{y^0\}$	$\{z^0\}$	$\{o^0\}$
1			$\{a^0\}$	$\{i^1,z^1\}$ $\{a^1,z^1\}$	$\{i^0,b^0,z^0\}$ $\{a^0,b^0,z^0\}$	$\{x^0,y^1\}$ $\{i^1,z^1,b^1\}$ $\{a^1,z^1,b^1\}$ $\{i^1,z^1,y^1\}$ $\{a^1,z^1,y^1\}$	$\{z^0\}$
2				$\{i^1,x^1,y^2\}$ $\{a^1,x^1,y^2\}$			

Sequential Circuit Technology Mapping, Fig. 4 Cut enumeration example

Sequential Circuit Technology Mapping, Fig. 5 Label computation procedure

FindMinLabels(N **)**

foreach node v in N **do** $l(v) \Leftarrow -w_v \cdot \phi$

while (*there are label updates*) **do**

foreach node v in N **do**

$l(v) \Leftarrow \min_{c \in C(v)} \{\max\{l(u) - d \cdot \phi + 1 \mid u^d \in c\}\}$

if v is a primary output and $l(v) > \phi$, **return** failure

return success

iter	*a*	*B*	*I*	*x*	*y*	*z*	*o*
0	$\{a^0\}$:0	$\{b^0\}$:0	$\{i^0\}$:0	$\{x^0\}$:-1	$\{y^0\}$:0	$\{z^0\}$:-1	$\{o^0\}$:-1
1			$\{a^0\}$:1	$\{a^1,z^1\}$:0	$\{a^0,b^0,z^0\}$:1	$\{a^1,z^1,b^1\}$:0	$\{z^0\}$:0

Sequential Circuit Technology Mapping, Fig. 6 Label computation example

Lemma 2 *N has a mapping solution with cycle time* ϕ iff the labeling procedure returns "success."

Mapping Solution Generation

Once the labels for all nodes are computed successfully, a mapping solution can be constructed starting from primary outputs. At each node v, the procedure selects the cut that realizes the label of the node and then moves on to select a cut for u if u^d is in the cut selected for v. On the edge from u to v, d FFs are inserted. For the network in Fig. 1 (1), the mapping solution generated based on the labels found in Fig. 6 is exactly the one in Fig. 2 (2).

To obtain a mapping solution with the target cycle time ϕ, v will be retimed by a value of $\lceil l(v)/\phi \rceil - 1$. For the network in Fig. 1 (1), the final mapping solution after retiming is shown in Fig. 2 (3) which has a cycle time of 1.

Applications

The algorithm can be used to map a technology-independent Boolean network to a network consisting of cells from a target technology library. The concepts and framework are generally enough to be adapted to study other circuit optimizations such as sequential circuit clustering and sequential circuit restructuring [6].

Cross-References

- ▸ Circuit Retiming
- ▸ Circuit Retiming: An Incremental Approach
- ▸ FPGA Technology Mapping
- ▸ Technology Mapping

Recommended Reading

1. Cong J, Ding Y (1994) FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. IEEE Trans Comput Aided Des Integr Circuits Syst 13(1):1–12
2. Cong J, Wu C (1997) FPGA synthesis with retiming and pipelining for clock period minimization of sequential circuits. In: ACM/IEEE design automation conference, Anaheim
3. Cong J, Wu C, Ding Y (1999) Cut ranking and pruning: enabling a general and efficient FPGA mapping solution. In: ACM international symposium on field-programmable gate arrays, Monterey
4. Keutzer K (1987) DAGON: technology binding and local optimization by DAG matching. In: ACM/IEEE design automation conference, Miami Beach
5. Leiserson CE, Saxe JB (1991) Retiming synchronous circuitry. Algorithmica 6:5–35
6. Mishchenko A, Chatterjee S, Brayton R, Pan P (2006) Integrating logic synthesis, technology mapping, and retiming. ERL technical report, EECS Department, UC Berkeley
7. Pan P (1997) Continuous retiming algorithms and applications. In: IEEE international conference on computer design, Austin
8. Pan P, Lin CC (1998) A new retiming-based technology mapping algorithm for LUT-based FPGAs. In: ACM international symposium on field-programmable gate arrays, Monterey
9. Pan P, Liu CL (1996) Optimal clock period FPGA technology mapping for sequential circuits. In: ACM/IEEE design automation conference, Las Vegas
10. Pan P, Liu CL (1998) Optimal clock period FPGA technology mapping for sequential circuits. ACM Trans Des Autom Electron Syst 3(3):437–462

Set Agreement

Michel Raynal
Institut Universitaire de France and IRISA, Université de Rennes, Rennes, France

Keywords

Distributed coordination

Years and Authors of Summarized Original Work

1993; Chaudhuri

Problem Definition

Short History

The k-set agreement problem is a paradigm of coordination problems. Defined in the setting of systems made up of processes prone to failures, it is a simple generalization of the consensus problem (that corresponds to the case $k = 1$). That problem was introduced in 1993 by Chaudhuri [2] to investigate how the number of choices (k) allowed for the processes is related to the maximum number of processes that can crash. (After it has crashed, a process executes no more steps: a crash is a premature halting.)

Definition

Let S be a system made up of n processes where up to t can crash and where each process has an input value (called a *proposed* value). The problem is defined by the three following properties (i.e., any algorithm that solves that problem has to satisfy these properties):

1. Termination. Every nonfaulty process decides a value.
2. Validity. A decided value is a proposed value.
3. Agreement. At most k different values are decided.

The Trivial Case

It is easy to see that this problem can be trivially solved if the upper bound on the number of process failures t is smaller than the allowed number of choices k, also called the *coordination degree*. (The trivial solution consists in having $t + 1$ predetermined processes that send their proposed values to all the processes, and a process deciding the first value it ever receives.) So, $k \leq$ t is implicitly assumed in the following.

Key Results

Key Results in Synchronous Systems

The Synchronous Model

In this computation model, each execution consists of a sequence of rounds. These are identified by the successive integers $1, 2$, etc. For the processes, the current round number appears as a global variable whose global progress entails their own local progress.

During a round, a process first broadcasts a message, then receives messages, and finally executes local computation. The fundamental synchrony property the a synchronous system provides the processes with is the following: a message sent during a round r is received by its destination process during the very same round r. If during a round, a process crashes while sending a message, an arbitrary subset (not known in advance) of the processes receive that message.

Main Results

The k-set agreement problem can always be solved in a synchronous system. The main result is for the minimal number of rounds (R_t) that are needed for the nonfaulty processes to decide in the worst-case scenario (this scenario is when exactly k processes crash in each round). It was shown in [3] that $R_t = \lfloor \frac{t}{k} \rfloor + 1$. A very simple algorithm that meets this lower bound is described in Fig. 1.

Although failures do occur, they are rare in practice. Let f denote the number of processes that crash in a given run, $0 \leq f \leq t$. We are interested in synchronous algorithms that terminate in at most R_t rounds when t processes crash in the current run, but that allow the nonfaulty processes to decide in far fewer rounds when there are few failures. Such algorithms are called *early-deciding* algorithms. It was shown in [4] that, in the presence of f process crashes, any early-deciding k-set agreement algorithm has runs in which no process decides before the round $R_f = \min(\lfloor \frac{f}{k} \rfloor + 2, \lfloor \frac{t}{k} \rfloor + 1)$. This lower bound shows an inherent tradeoff linking the coordination degree k, the maximum number of process failures t, the actual number of process failures f, and the best time complexity that can be achieved. Early-deciding k-set agreement algorithms for the synchronous model can be found in [4, 12].

Other Failure Models

In the send omission failure model, a process is faulty if it crashes or forgets to send messages. In the general omission failure model, a process is faulty if it crashes, forgets to send messages, or forgets to receive messages. (A send omission failure models the failure of an output buffer, while a receive omission failure models the failure of an input buffer.) These failure models were introduced in [11].

The notion of *strong* termination for set agreement problems was introduced in [13]. Intuitively, that property requires that as many processes as possible decide. Let a *good* process be a process that neither crashes nor commits receive omission failures. A set agreement algorithm is strongly terminating if it forces all the good processes to decide. (Only the processes that crash during the execution of the algorithm, or that do not receive enough messages, can be prevented from deciding.)

An early-deciding k-set agreement algorithm for the general omission failure model was described in [13]. That algorithm, which requires $t < n/2$, directs a good process to decide and stop in at most $R_f = \min(\lfloor \frac{f}{k} \rfloor + 2, \lfloor \frac{t}{k} \rfloor + 1)$ rounds. Moreover, a process that is not a good

Set Agreement, Fig. 1
A simple k-set agreement synchronous algorithm (code for p_i)

```
Function k-set_agreement (v_i)
(1) est_i ← v_i;
(2) when r = 1, 2, ..., ⌊t/k⌋ + 1 do % r: round number %
(3) begin_round
(4)       send (est_i) to all; % including p_i itself %
(5)       est_i ← min({est_j values received during
                                    the current round r});
(6) end_round;
(7) return (est_i)
```

process executes at most $R_f(not\ good)$) $\min(\lceil \frac{f}{k} \rceil + 2, \lfloor \frac{t}{k} \rfloor + 1)$ rounds.

As R_f is a lower bound for the number of rounds in the crash failure model, the previous algorithm shows that R_f is also a lower bound for the nonfaulty processes to decide in the more severe general omission failure model. Proving that $R_f(not\ good)$ is an upper bound for the number of rounds that a nongood process has to execute remains an open problem.

It was shown in [13] that, for a given coordination degree k, $t < \frac{k}{k+1}n$ is an upper bound on the number of process failures when one wants to solve the k-set agreement problem in a synchronous system prone to process general omission failures. A k-set agreement algorithm that meets this bound was described in [13]. That algorithm requires the processes execute $R = t + 2 - k$ rounds to decide. Proving (or disproving) that R is a lower bound when $t < \frac{k}{k+1}n$ is an open problem. Designing an early-deciding k-set agreement algorithm for $t < \frac{k}{k+1}n$ and $k > 1$ is another problem that remains open.

Key Results in Asynchronous Systems

Impossibility

A fundamental result of distributed computing is the impossibility to design a deterministic algorithm that solves the k-set agreement problem in asynchronous systems when $k \leq$ t [1, 7, 15]. Compared with the impossibility of solving asynchronous consensus despite one process crash, that impossibility is based on deep combinatorial arguments. This impossibility has opened new research directions for the connection between distributed computing and topology. This topology approach has allowed the discovery of links relating asynchronous k-set agreement with other distributed computing problems such as the *renaming* problem [5].

Circumventing the Impossibility

Several approaches have been investigated to circumvent the previous impossibility. These approaches are the same as those that have been used to circumvent the impossibility of asynchronous consensus despite process crashes.

One approach consists in replacing the "deterministic algorithm" by a "randomized algorithm." In that case, the termination property becomes "the probability for a correct process to decide tends to 1 when the number of rounds tends to $+\infty$." That approach was investigated in [9].

Another approach that has been proposed is based on failure detectors. Roughly speaking, a failure detector provides each process with a list of processes suspected to have crashed. As an example, the class of failure detectors denoted $\diamond\mathcal{S}_x$ includes all the failure detectors such that, after some finite (but unknown) time, (1) any list contains the crashed processes and (2) there is a set Q of x processes such that Q contains one correct process and that correct process is no longer suspected by the processes of Q (let us observe that correct processes can be suspected intermittently or even forever). Tight bounds for the k-set agreement problem in asynchronous systems equipped with such failure detectors, conjectured in [9], were proved in [6]. More precisely, such

a failure detector class allows the k-set agreement problem to be solved for $k \geq t - x + 2$ [9], and cannot solve it when $k < t - x + 2$ [6].

Another approach that has been investigated is the combination of failure detectors and conditions [8]. A condition is a set of input vectors, and each input vector has one entry per process. The entries of the input vector associated with a run contain the values proposed by the processes in that run. Basically, such an approach guarantees that the nonfaulty processes always decide when the actual input vector belongs to the condition the k-set algorithm has been instantiated with.

Applications

The set agreement problem was introduced to study how the number of failures and the synchronization degree are related in an asynchronous system; hence, it is mainly a theoretical problem. That problem is used as a canonical problem when one is interested in asynchronous computability in the presence of failures. Nevertheless, one can imagine practical problems the solutions of which are based on the set agreement problem (e.g., allocating a small shareable resources – such as broadcast frequencies – in a network).

Cross-References

- ▸ Asynchronous Consensus Impossibility
- ▸ Failure Detectors
- ▸ Renaming
- ▸ Topology Approach in Distributed Computing

Recommended Reading

1. Borowsky E, Gafni E (1993) Generalized FLP impossibility results for t-resilient asynchronous computations. In: Proceedings of the 25th ACM symposium on theory of computation, California, pp 91–100
2. Chaudhuri S (1993) More choices allow more faults: set consensus problems in totally asynchronous systems. Inf Comput 105:132–158
3. Chaudhuri S, Herlihy M, Lynch N, Tuttle M (2000) Tight bounds for k set agreement. J ACM 47(5):912–943
4. Gafni E, Guerraoui R, Pochon B (2005) From a static impossibility to an adaptive lower bound: the complexity of early deciding set agreement. In: Proceedings of the 37th ACM symposium on theory of computing (STOC 2005). ACM, New York, pp 714–722
5. Gafni E, Rajsbaum S, Herlihy M (2006) Subconsensus tasks: renaming is weaker than set agreement. In: Proceedings of the 20th international symposium on distributed computing (DISC'06). LNCS, vol 4167. Springer, Berlin, pp 329–338
6. Herlihy MP, Penso LD (2005) Tight bounds for k set agreement with limited scope accuracy failure detectors. Distrib Comput 18(2):157–166
7. Herlihy MP, Shavit N (1999) The topological structure of asynchronous computability. J ACM 46(6):858–923
8. Mostefaoui A, Rajsbaum S, Raynal M (2005) The combined power of conditions and failure detectors to solve asynchronous set agreement. In: Proceedings of the 24th ACM symposium on principles of distributed computing (PODC'05). ACM, New York, pp 179–188
9. Mostefaoui A, Raynal M (2000) k set agreement with limited scope accuracy failure detectors. In: Proceedings of the 19th ACM symposium on principles of distributed computing. ACM, New York, pp 143–152
10. Mostefaoui A, Raynal M (2001) Randomized set agreement. In: Proceedings of the 13th ACM symposium on parallel algorithms and architectures (SPAA'01), Hersonissos (Crete). ACM, New York, pp 291–297
11. Perry KJ, Toueg S (1986) Distributed agreement in the presence of processor and communication faults. IEEE Trans Softw Eng SE-12(3):477–482
12. Raipin Parvedy P, Raynal M, Travers C (2005) Early-stopping k-set agreement in synchronous systems prone to any number of process crashes. In: Proceedings of the 8th international conference on parallel computing technologies (PaCT'05). LNCS, vol 3606. Springer, Berlin, pp 49–58
13. Raipin Parvedy P, Raynal M, Travers C (2006) Strongly-terminating early-stopping k-set agreement in synchronous systems with general omission failures. In: Proceedings of the 13th colloquium on structural information and communication complexity (SIROCCO'06). LNCS, vol 4056. Springer, Berlin, pp 182–196
14. Raynal M, Travers C (2006) Synchronous set agreement: a concise guided tour (including a new algorithm and a list of open problems). In: Proceedings of the 12th international IEEE pacific rim dependable computing symposium (PRDC'2006). IEEE Computer Society, Los Alamitos, pp 267–274
15. Saks M, Zaharoglou F (2000) Wait-free k-set agreement is impossible: the topology of public knowledge. SIAM J Comput 29(5):1449–1483

Set Cover with Almost Consecutive Ones

Michael Dom
Department of Mathematics and Computer Science, University of Jena, Jena, Germany

Keywords

Hitting set

Years and Authors of Summarized Original Work

2004; Mecke, Wagner

Problem Definition

The SET COVER problem has as input a set R of m items, a set C of n subsets of R and a weight function $w: C \to \mathbb{Q}$. The task is to choose a subset $C' \subseteq C$ of minimum weight whose union contains all items of R.

The sets R and C can be represented by an $m \times n$ binary matrix A that consists of a row for every item in R and a column for every subset of R in C, where an entry $a_{i,j}$ is 1 iff the ith item in R is part of the jth subset in C. Therefore, the SET COVER problem can be formulated as follows.

> **Input**: An $m \times n$ binary matrix A and a weight function w on the columns of A.
> **Task:** Select some columns of A with minimum weight such that the submatrix A' of A that is induced by these columns has at least one 1 in every row.

While SET COVER is NP-hard in general [4], it can be solved in polynomial time on instances whose columns can be permuted in such a way that in every row the ones appear consecutively, that is, on instances that have the *consecutive ones property* (*C1P*). (The C1P can be defined symmetrically for columns; this article focuses on rows. SET COVER on instances with the C1P can be solved in polynomial time, e.g., with a linear programming approach, because the corresponding coefficient matrices are totally unimodular (see [9]).

Motivated by problems arising from railway optimization, Mecke and Wagner [7] consider the case of SET COVER instances that have "almost the C1P". Having almost the C1P means that the corresponding matrices are similar to matrices that have been generated by starting with a matrix that has the C1P and replacing randomly a certain percentage of the 1's by 0's [7]. For Ruf and Schöbel [8], in contrast, having almost the C1P means that the average number of blocks of consecutive 1's per row is much smaller than the number of columns of the matrix. This entry will also mention some of their results.

Notation

Given an instance (A, w) of SET COVER, let R denote the row set of A and C its column set. A column c_j *covers* a row r_i, denoted by $r_i \in c_j$, if $a_{i,j} = 1$.

A binary matrix has the *strong C1P* if (without any column permutation) the 1's appear consecutively in every row. A *block of consecutive 1's* is a maximal sequence of consecutive 1's in a row. It is possible to determine in linear time if a matrix has the C1P, and if so, to compute a column permutation that yields the strong C1P [2, 3, 6]. However, note that it is NP-hard to permute the columns of a binary matrix such that the number of blocks of consecutive 1's in the resulting matrix is minimized [1, 4, 5].

A *data reduction rule* transforms in polynomial time a given instance I of an optimization problem into an instance I' of the same problem such that $|I'| < |I|$ and the optimal solution for I' has the same value (e.g., weight) as the optimal solution for I. Given a set of data reduction rules, *to reduce* a problem instance means to repeatedly apply the rules until no rule is applicable; the resulting instance is called *reduced*.

Key Results

Data Reduction Rules

For SET COVER there exist well-known data reduction rules:

Row domination rule: If there are two rows $r_{i_1}, r_{i_2} \in R$ with $\forall c \in C: r_{i_1} \in c \text{implies} r_{i_2} \in c$, then r_{i_2} is *dominated* by r_{i_1}. Remove row r_{i_2} from A.

Column domination rule: If there are two columns $c_{j_1}, c_{j_2} \in C$ with $w(c_{j_1}) \geq w(c_{j_2})$ and $\forall r \in R: r \in c_{j_1}$ implies $r \in c_{j_2}$, then c_{j_1} is *dominated* by c_{j_2}. Remove c_{j_1} from A.

In addition to these two rules, a column $c_{j_1} \in C$ can also be dominated by a subset $C' \subseteq C$ of the columns instead of a single column: If there is a subset $C' \subseteq C$ with $w(c_{j_1}) \geq \sum_{c \in C'} w(c)$ and $\forall r \in R: r \in c_{j_1}$ implies $(\exists c \in C': r \in c)$, then remove c_{j_1} from A. Unfortunately, it is NP-hard to find a dominating subset C' for a given set c_{j_1}. Mecke and Wagner [7], therefore, present a restricted variant of this generalized column domination rule.

For every row $r \in R$, let $c_{\min}(r)$ be a column in C that covers r and has minimum weight under this property. For two columns $c_{j_1}, c_{j_2} \in C$, define $X(c_{j_1}, c_{j_2}) := \{c_{\min}(r) \mid r \in c_{j_1} \wedge r \notin c_{j_2}\}$. The new data reduction rule then reads as follows.

Advanced column domination rule: If there are two columns $c_{j_1}, c_{j_2} \in C$ and a row that is covered by both c_{j_1} and c_{j_2}, and if $w(c_{j_1}) \geq w(c_{j_2}) + \sum_{c \in X(c_{j_1}, c_{j_2})} w(c)$, then c_{j_1} is *dominated* by $\{c_{j_2}\} \cup X(c_{j_1}, c_{j_2})$. Remove c_{j_1} from A.

Theorem 1 ([7]) *A matrix A can be reduced in O(Nn) time with respect to the column domination rule, in O(Nm) time with respect to the row domination rule, and in O(Nmn) time with respect to all three data reduction rules described above, when N is the number of 1's in A.*

In the databases used by Ruf and Schöbel [8], matrices are represented by the column indices of the first and last 1's of its blocks of consecutive 1's. For such matrix representations, a fast data reduction rule is presented [8], which eliminates "unnecessary" columns and which, in the implementations, replaces the column domination rule. The new rule is faster than the column domination rule (a matrix can be reduced in $O(mn)$ time with respect to the new rule), but not as powerful: Reducing a matrix A with the new rule can result in a matrix that has more columns than the matrix resulting from reducing A with the column domination rule.

Algorithms

Mecke and Wagner [7] present an algorithm that solves SET COVER by enumerating all feasible solutions.

Given a row r_i of A, a *partial solution for the rows* $r_1, \ldots, r_i$ is a subset $C' \subseteq C$ of the columns of A such that for each row r_j with $j \in \{1, \ldots, i\}$ there is a column in C' that covers row r_j.

The main idea of the algorithm is to find an optimal solution by iterating over the rows of A and updating in every step a data structure S that keeps *all* partial solutions for the rows considered so far. More exactly, in every iteration step the algorithm considers the first row of A and updates the data structure S accordingly. Thereafter, the first row of A is deleted. The following code shows the algorithm.

```
1 Repeat m times: {
2   for every partial solution C' in S that does not
                                cover the first row of A: {
3     for every column c of A that covers the first row
                                                of A: {
4       Add {c} ∪ C' to S; }
5 Delete C' from S; }
6 Delete the first row of A; }
```

This straightforward enumerative algorithm could create a set S of exponential size. Therefore, the data reduction rules presented above are used to delete after each iteration step partial solutions that are not needed any more. To this end, a matrix B is associated with the set S, where every row corresponds to a row of A and every column corresponds to a partial solution in S–an entry $b_{i,j}$ of B is 1 iff the jth partial solution of B contains a column of A that covers the row r_i. The algorithm uses the matrix $C := \left(\begin{array}{c|c} A & B \\ \hline 0 \ldots 0 & 1 \ldots 1 \end{array}\right)$, which is updated together with S in every iteration step. (The last row of C allows to distinguish the columns belonging to A from those belonging

to B.) Line 6 of the code shown above is replaced by the following two lines:

6 Delete the first row of the matrix C;
7 Reduce the matrix C and update S accordingly;}

At the end of the algorithm, S contains exactly one solution, and this solution is optimal. Moreover, if the SET COVER instance is nicely structured, the algorithm has polynomial running time:

Theorem 2 ([7]) *If A has the strong C1P, is reduced, and its rows are sorted in lexicographic order, then the algorithm has a running time of $O(M^{3n})$ where M is the maximum number of 1's per row and per column.*

Theorem 3 ([7]) *If the distance between the first and the last 1 in every column is at most k, then at any time throughout the algorithm the number of columns in the matrix B is $O(2^{kn})$, and the running time is $O(2^{2k}\, kmn^2)$.*

Ruf and Schöbel [8] present a branch and bound algorithm for SET COVER instances that have a small average number of blocks of consecutive 1's per row.

The algorithm considers in each step a row r_i of the current matrix (which has been reduced with data reduction rules before) and branches into bl_i cases, where bl_i is the number of blocks of consecutive 1's in r_i. In each case, one block of consecutive 1's in row r_i is selected, and the 1's of all other blocks in this row are replaced by 0's. Thereafter, a lower and an upper bound on the weight of the solution for each resulting instance is computed. If a lower bound differs by a factor of more than $1 + \epsilon$, for a given constant ε, from the best upper bound achieved so far, the corresponding instance is subjected to further branchings. Finally, the best upper bound that was found is returned.

In each branching step, the bl_i instances that are newly generated are "closer" to have the (strong) C1P than the instance from which they descend. If an instance has the C1P, the lower and upper bound can easily be computed by exactly solving the problem. Otherwise, standard heuristics are used.

Applications

SET COVER instances occur e.g., in railway optimization, where the task is to determine where new railway stations should be built. Each row then corresponds to an existing settlement, and each column corresponds to a location on the existing trackage where a railway station could be build. A column c covers a row r, if the settlement corresponding to r lies within a given radius around the location corresponding to c.

If the railway network consisted of one straight line rail track only, the corresponding SET COVER instance would have the C1P; instances arising from real world data are close to have the C1P [7, 8].

Experimental Results

Mecke and Wagner [7] make experiments on real-world instances as described in the Applications section and on instances that have been generated by starting with a matrix that has the C1P and replacing randomly a certain percentage of the 1's by 0's. The real-world data consists of a railway graph with 8,200 nodes and 8,700 edges, and 30,000 settlements. The generated instances consist of 50–50,000 rows with 10–200 1's per row. Up to 20 % of the 1's are replaced by 0's.

In the real-world instances, the data reduction rules decrease the number of 1's to between 1 % and 25 % of the original number of 1's without and to between 0.2 % and 2.5 % with the advanced column reduction rule. In the case of generated instances that have the C1P, the number of 1's is decreased to about 2 % without and to 0.5 % with the advanced column reduction rule. In instances with 20 % perturbation, the number of 1's is decreased to 67 % without and to 20 % with the advanced column reduction rule.

The enumerative algorithm has a running time that is almost linear for real-world instances and most generated instances. Only in the case of generated instances with 20 % perturbation, the running time is quadratic.

Ruf and Schöbel [8] consider three instance types: real-world instances, instances arising

from Steiner triple systems, and randomly generated instances. The latter have a size of 100×100 and contain either 1–5 blocks of consecutive 1's in each row, each one consisting of between one and nine 1's, or they are generated with a probability of 3 % or 5 % for any entry to be 1.

The data reduction rules used by Ruf and Schöbel turn out to be powerful for the real-world instances (reducing the matrix size from about $1{,}100 \times 3{,}100$ to 100×800 in average), whereas for all other instance types the sizes could not be reduced noticeably.

The branch and bound algorithm could solve almost all real-world instances up to optimality within a time of less than a second up to one hour. In all cases where an optimal solution has been found, the first generated subproblem had already provided a lower bound equal to the weight of the optimal solution.

Cross-References

▸ Greedy Set-Cover Algorithms

Recommended Reading

1. Atkins JE, Middendorf M (1996) On physical mapping and the consecutive ones property for sparse matrices. Discret Appl Math 71(1–3):23–40
2. Booth KS, Lueker GS (1976) Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. J Comput Syst Sci 13:335–379
3. Fulkerson DR, Gross OA (1965) Incidence matrices and interval graphs. Pac J Math 15(3):835–855
4. Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, New York
5. Goldberg PW, Golumbic MC, Kaplan H, Shamir R (1995) Four strikes against physical mapping of DNA. J Comput Biol 2(1):139–152
6. Hsu WL, McConnell RM (2003) PC trees and circular-ones arrangements. Theor Comput Sci 296(1):99–116
7. Mecke S, Wagner D (2004) Solving geometric covering problems by data reduction. In: Proceedings of the 12th annual European symposium on algorithms (ESA '04). LNCS, vol 3221. Springer, Berlin, pp 760–771
8. Ruf N, Schöbel A (2004) Set covering with almost consecutive ones property. Discret Optim 1(2):215–228
9. Schrijver A (1986) Theory of linear and integer programming. Wiley, Chichester

Shadowless Solutions for Fixed-Parameter Tractability of Directed Graphs

Rajesh Chitnis and Mohammad Taghi Hajiaghayi
Department of Computer Science, University of Maryland, College Park, MD, USA

Keywords

Directed graphs; Fixed-parameter tractability; Important Separators; Shadowless solutions; Transversal problems

Years and Authors of Summarized Original Work

2012; Chitnis, Hajiaghayi, Marx
2012; Chitnis, Cygan, Hajiaghayi, Marx

Problem Definition

The study of the parameterized complexity of problems on directed graphs has been hitherto relatively unexplored. Usually the directed version of the problems require significantly different and more involved ideas than the ones for the undirected version. Furthermore, for directed graphs there are no known algorithmic meta-techniques: for example, there is no known algorithmic analogue of the Graph Minor Theory of Robertson and Seymour for directed graphs. As a result, the fixed-parameter tractability status of the directed versions of several fundamental problems such as Multiway Cut, Multicut, Feedback Vertex Set, etc., was open for a long time. The problem of Feedback Vertex Set best illustrates this gulf between undirected and directed graphs with respect to parameterized complexity. In this problem, we are given a graph and the question is whether there exists a set of size at most k whose deletion makes the graph acyclic. The undirected version was known to be FPT

since 1984 [10]. However, the directed version was a long-standing open problem until it was shown to be FPT in 2008 [1].

The framework of *shadowless solutions* aims to bridge this gap by providing an important first step in designing FPT algorithms for a general class of transversal problems on directed graphs. In undirected graphs, the framework of *shadowless solutions* was introduced in [9] and has since been used in [4, 6, 7]. It was adapted and generalized to directed graphs in [2, 3] for the following general class of problems:

Finding an $\mathcal{F}$-transversal for some T-connected $\mathcal{F}$
Input: A directed graph $G = (V, E)$, a positive integer k, a set $T \subseteq V$, and a set $\mathcal{F} = \{F_1, F_2, \ldots, F_q\}$ of subgraphs such that $\mathcal{F}$ is T-connected, i.e., $\forall\, i \in [q]$ each vertex of F_i can reach some vertex of T by a walk completely contained in $G[F_i]$ and is reachable from some vertex of T by a walk completely contained in $G[F_i]$.
Parameter: k
Question: Is there an $\mathcal{F}$-transversal $W \subseteq V$ with $|W| \leq k$, i.e., a set W such that $F_i \cap W \neq \emptyset$ for every $i \in [q]$?

The collection $\mathcal{F}$ is implicitly defined in a problem-specific way and need not be given explicitly in the input. In fact, it is possible that $\mathcal{F}$ is exponentially large. The *shadow* of a solution X is the set of vertices that are disconnected from T (in either direction) after the removal of X. More formally, the reverse shadow of X is given by $r_T(X) = \{v : X \text{is a } v \to T \text{ separator}\}$. Similarly, the forward shadow of X is given by $f_T(X) = \{v : X \text{is a } T \to v \text{ separator}\}$. The shadow of X is given by the union of its reverse and forward shadows, i.e., $\text{shadow}(X) = r(X) \cup f(X)$. A set X is said to be *shadowless* if its shadow is empty.

The aim is to ensure first that there is a solution whose shadow is empty, as finding such a shadowless solution can be a significantly easier task.

Key Results

For the $\mathcal{F}$-transversal problem defined above, [2] shows how to invoke the technique of *random sampling of important separators* and obtain a set Z which is disjoint from a minimum solution X and covers its shadow.

Theorem 1 (randomized covering of the shadow) *Let $T \subseteq V(G)$. There is an algorithm $\texttt{RandomSet}(G, T, k)$ that runs in $4^k \cdot n^{O(1)}$ time and returns a set $Z \subseteq V(G)$ such that for any set $\mathcal{F}$ of T-connected subgraphs, if there exists an $\mathcal{F}$-transversal of size $\leq k$, then the following holds with probability $2^{-2^{O(k)}}$: there is an $\mathcal{F}$-transversal X of size $\leq k$ such that*

1. *$X \cap Z = \emptyset$ and*
2. *Z covers the shadow of X, i.e., $r(X) \cup f(X) \subseteq Z$.*

The set $\mathcal{F}$ is *not* an input of the algorithm described by Theorem 1: the set Z constructed in the above theorem works for *every* T-connected set $\mathcal{F}$ of subgraphs. Therefore, issues related to the representation of $\mathcal{F}$ do not arise. Theorem 1 can be derandomized using the theory of splitters [11]:

Theorem 2 (deterministic covering of the shadow) *Let $T \subseteq V(G)$. We can construct a set $\{Z_1, Z_2, \ldots, Z_t\}$ with $t = 2^{2^{O(k)}} \cdot \log^2 n$ in time $2^{2^{O(k)}} \cdot n^{O(1)}$ such that for any set $\mathcal{F}$ of T-connected, if there exists an $\mathcal{F}$-transversal of size $\leq k$, then there is an $\mathcal{F}$-transversal X of size $\leq k$ such that for at least one $1 \leq i \leq t$ we have*

1. *$X \cap Z_i = \emptyset$ and*
2. *Z_i covers the shadow of X, i.e., $r(X) \cup f(X) \subseteq Z_i$.*

Consider one such set Z_i for some $1 \leq i \leq 2^{2^{O(k)}} \cdot \log^2 n$. Since this set Z_i is disjoint from a minimum solution X, it can be removed from the graph. However, we need to remember the

structure that the set Z_i imposed on the problem. This structure is problem specific, and the reduced (equivalent) instance is obtained on a supergraph of $G \setminus Z_i$ via the *torso operation*. It can be shown that the original instance G has a solution if and only if the reduced instance has a shadowless solution. Therefore, one can focus on the simpler task of finding a shadowless solution or more precisely, finding any solution under the guarantee that a shadowless solution exists.

Applications

The first FPT algorithms for the Directed Multiway Cut problem [3] and the Directed Subset Feedback Vertex Set problem [2] were obtained via the framework of shadowless solutions.

Directed Multiway Cut

In the Directed Multiway Cut problem, given a directed graph $G = (V, E)$, an integer k, and a set of terminals $T = \{t_1, t_2, \ldots, t_p\}$, the objective is to find whether there exists a set $X \subseteq V(G)$ of size at most k such that $G \setminus X$ has no $t_i \to t_j$ path for any $1 \leq i \neq j \leq p$. Let $\mathcal{F}$ be the set of all paths between pairs of (distinct) terminals. Then it is easy to show that $\mathcal{F}$ is T-connected, and the problem of finding an $\mathcal{F}$-transversal is exactly the same as the Directed Multiway Cut problem. It is shown in [3] that a shadowless solution of Directed Multiway Cut is also a solution of the underlying undirected instance of Multiway Cut, which is known to be FPT [8] parameterized by k. Combining with Theorem 2, this gives an FPT algorithm for the Directed Multiway Cut problem.

Directed Subset Feedback Vertex Set

In the Directed Subset Feedback Vertex Set problem, given a directed graph $G = (V, E)$, an integer k, and a set $S \subseteq V(G)$, the objective is to find whether there exists a set $X \subseteq V(G)$ of size at most k such that $G \setminus X$ has no S-cycles, i.e., cycles containing at least one vertex of S. The special case when $S = V(G)$ is the Directed Feedback Vertex Set problem. Let $\mathcal{F}$ be the set of all S-cycles and T be a solution of size $k + 1$ (which can be obtained via *iterative compression*). Then it is easy to show that $\mathcal{F}$ is T-connected, and the problem of finding an $\mathcal{F}$-transversal is exactly the same as the Directed Subset Feedback Vertex Set problem. It is shown in [2] that a shadowless solution of Directed Subset Feedback Vertex Set can be found in FPT time. Combining with Theorem 2, this gives an FPT algorithm for the Directed Subset Feedback Vertex Set problem. This generalizes the FPT algorithm for Directed Feedback Vertex Set [1].

Open Problems

The two main open problems which fit within the framework of "Finding an $\mathcal{F}$-transversal for some T-connected $\mathcal{F}$" are Directed Multicut and Directed Odd Cycle Transversal. Unfortunately, the structure of shadowless solutions is not yet understood well enough to be able to find them in FPT time.

Directed Multicut

In the Directed Multicut problem, given a directed graph $G = (V, E)$, an integer k, and a set of terminal pairs $T = \{(s_1, t_1), (s_2, t_2), \ldots, (s_p, t_p)\}$, the objective is to find whether there exists a set $X \subseteq V(G)$ of size at most k such that $G \setminus X$ has no $s_i \to t_i$ path for any $1 \leq i \leq p$. Let $\mathcal{F}$ be the union of set of all $s_i \to t_i$ paths for $1 \leq i \leq p$. Then it is easy to show that $\mathcal{F}$ is T-connected, and the problem of finding an $\mathcal{F}$-transversal is exactly the same as the Directed Multicut problem. It is known [9] that Directed Multicut parameterized by k is W[1]-hard. However, for the special case of $p = 2$ terminal pairs, the problem can be reduced to Directed Multiway Cut and is hence FPT parameterized by k [3]. The complexity for $p = 3$ parameterized by k is an important open problem. With respect to the bigger parameter $p + k$, the problem is known [5] to be FPT on directed acyclic graphs. However, this algorithm heavily uses the properties of a topological ordering, and the complexity parameterized by

$p + k$ on general graphs is another important open problem.

Directed Odd Cycle Transversal

In the Directed Odd Cycle Transversal problem, given a directed graph $G = (V, E)$ and an integer k, the objective is to find whether there exists a set $X \subseteq V(G)$ of size at most k such that $G \setminus X$ has no cycle of odd length. Let $\mathcal{F}$ be the set of all odd cycles in G and T be a solution of size $k + 1$ (which can be obtained via *iterative compression* [12]). Then it is easy to show that $\mathcal{F}$ is T-connected, and the problem of finding an $\mathcal{F}$-transversal is exactly the same as the Directed Odd Cycle Transversal problem. The complexity parameterized by k is open. Moreover, it is known that Directed Odd Cycle Transversal problem generalizes the Directed Feedback Vertex Set problem [1] and the Undirected Odd Cycle Transversal problem [12]. Hence, an FPT algorithm for Directed Odd Cycle Transversal would have to generalize the ideas used to obtain FPT algorithms for these two problems.

Cross-References

- Bidimensionality
- Undirected Feedback Vertex Set

Recommended Reading

1. Chen J, Liu Y, Lu S, O'Sullivan B, Razgon I (2008) A fixed-parameter algorithm for the directed feedback vertex set problem. In: STOC, Victoria, pp 177–186
2. Chitnis RH, Cygan M, Hajiaghayi MT, Marx D (2012) Directed subset feedback vertex set is fixed-parameter tractable. In: ICALP (1), Warwick, pp 230–241
3. Chitnis RH, Hajiaghayi M, Marx D (2012) Fixed-parameter tractability of directed multiway cut parameterized by the size of the cutset. In: SODA, Kyoto, pp 1713–1725
4. Chitnis RH, Egri L, Marx D (2013) List H-coloring a graph by removing few vertices. In: ESA, Sophia Antipolis, pp 313–324
5. Kratsch S, Pilipczuk M, Pilipczuk M, Wahlström M (2012) Fixed-parameter tractability of multicut in directed acyclic graphs. In: ICALP (1), Warwick, pp 581–593
6. Lokshtanov D, Marx D (2011) Clustering with local restrictions. In: ICALP (1), Zurich, pp 785–797
7. Lokshtanov D, Ramanujan MS (2012) Parameterized tractability of multiway cut with parity constraints. In: ICALP (1), Warwick, pp 750–761
8. Marx D (2006) Parameterized graph separation problems. Theor Comput Sci 351(3):394–406
9. Marx D, Razgon I (2011) Fixed-parameter tractability of multicut parameterized by the size of the cutset. In: STOC, San Jose, pp 469–478
10. Mehlhorn K (1984) Data structures and algorithms 2: graph algorithms and NP-completeness. Springer, Berlin/New York
11. Naor M, Schulman LJ, Srinivasan A (1995) Splitters and near-optimal derandomization. In: FOCS, Milwaukee, pp 182–191
12. Reed BA, Smith K, Vetta A (2004) Finding odd cycle transversals. Oper Res Lett 32(4):299–301

Shortest Elapsed Time First Scheduling

Nikhil Bansal
Eindhoven University of Technology, Eindhoven, The Netherlands

Keywords

Feedback queues; MLF algorithm; Response time; Scheduling with unknown job sizes; Sojourn time

Years and Authors of Summarized Original Work

2003; Bansal, Pruhs

Problem Definition

The problem is concerned with scheduling dynamically arriving jobs in the scenario when the processing requirements of jobs are unknown to the scheduler. The lack of knowledge of how long a job will take to execute is a particularly attractive assumption in real systems where such

information might be difficult or impossible to obtain. The goal is to schedule jobs to provide good quality of service to the users. In particular the goal is to design algorithms that have good average performance and are also fair in the sense that no subset of users experiences substantially worse performance than others.

Notations

Let $\mathcal{J} = \{1, 2, \ldots, n\}$ denote the set of jobs in the input instance. Each job j is characterized by its release time r_j and its processing requirement p_j. In the online setting, job j is revealed to the scheduler only at time r_j. A further restriction is the *non-clairvoyant* setting, where only the existence of job j is revealed at r_j, in particular the scheduler does not know p_j until the job meets its processing requirement and leaves the system. Given a schedule, the completion time c_j of a job is the earliest time at which job j receives p_j amount of service. The flow time f_j of j is defined as $c_j - r_j$. The stretch of a job is defined as the ratio of its flow time divided by its size. Stretch is also referred to as normalized flow time or slowdown and is a natural measure of fairness as it measures the waiting time of a job per unit of service received. A schedule is said to be preemptive, if a job can be interrupted arbitrarily, and its execution can be resumed later from the point of interruption without any penalty. It is well known that preemption is necessary to obtain reasonable guarantees for flow time even in the offline setting [6].

Recall that the online shortest remaining processing time (SRPT) algorithm that at any time works on the job with the least remaining processing is optimum for minimizing average flow time. However, a common critique of SRPT is that it may lead to starvation of jobs, where some jobs may be delayed indefinitely. For example, consider the sequence where a job of size 3 arrives at time $t = 0$ and one job of size 1 arrives every unit of time starting $t = 1$ for a long time. Under SRPT, the size 3 job will be delayed until the size 1 jobs stop arriving. On the other hand, if the goal is to minimize the maximum flow time, then it is easily seen that first in first out (FIFO) is the optimum algorithm. However, FIFO can perform very poorly with respect to average flow time (e.g., many small jobs could be stuck behind a very large job that arrived just earlier). A natural way to balance both the average and worst case performance is to consider the ℓ_p norms of flow time and stretch, where the ℓ_p norm of the sequence $x_1, \ldots, x_n$ is defined as $\left(\sum_i x_i^p\right)^{1/p}$.

The shortest elapsed time first (SETF) is a non-clairvoyant algorithm that at any time works on the job that has received the least amount of service thus far. This is a natural way to favor short jobs given the lack of knowledge of job sizes. In fact, SETF is the continuous version of the multilevel feedback (MLF) algorithm. Unfortunately, SETF (or any other deterministic non-clairvoyant algorithm) performs poorly in the framework of competitive analysis, where an algorithm is called c-competitive if for every input instance, its performance is no worse than c times that of the optimum offline (clairvoyant) solution for that instance [7]. However, competitive analysis can be overly pessimistic in its guarantee. A way around this problem was proposed by Kalyanasundaram and Pruhs [5] who allowed the online scheduler a slightly faster processor to make up for its lack of knowledge of future arrivals and job sizes. Formally, an algorithm Alg is said to be s-speed, c-speed competitive where c is the worst case ratio over all instance I, of $\text{Alg}_s(I)/\text{Opt}_1(I)$, where Alg_s is the value of solution produced by Alg when given an s-speed processor, and Opt_1 is the optimum value using a speed 1 processor. Typically the most interesting results are those where c is small and $s = (1+\epsilon)$ for any arbitrary $\epsilon > 0$.

Key Results

In their seminal paper [5], Kalyanasundaram and Pruhs showed the following.

Theorem 1 ([5]) *SETF is a $(1+\epsilon)$-speed, $(1 + 1/\epsilon)$-competitive non-clairvoyant algorithm for minimizing the average flow time on a single machine with preemptions.*

For minimizing the average stretch, Muthukrishnan, Rajaraman, Shaheen, and Gehrke [6] considered the clairvoyant setting and showed that SRPT is 2-competitive for a single machine and 14-competitive for multiple machines. The non-clairvoyant setting was consider by Bansal, Dhamdhere, Konemann, and Sinha [7]. They showed that

Theorem 2 ([1]) *SETF is a* $(1+\epsilon)$*-speed,* $0(\log^2 P)$*-competitive for minimizing average stretch, where* P *is the ratio of the maximum to minimum job size. On the other hand, even with* $O(1)$*-speed, any non-clairvoyant algorithm is at least* $\Omega(\log P)$*-competitive. Interestingly, in terms of* n*, any non-clairvoyant algorithm must be* $\Omega(n)$*-competitive even with* $O(1)$*-speedup. Moreover, SETF is* $O(n)$*-competitive (even without extra speedup). For the special case when all jobs arrive at time 0, SETF is optimum up to constant factors. It is* $O(\log P)$*-competitive (without any extra speedup). Moreover, any non-clairvoyant must be* $\Omega(\log P)$*-competitive even with factor* $O(1)$*-speedup.*

The key idea of the above result was a connection between SETF and SRPT. First, at the expense of $(1+\epsilon)$-speedup, it can be seen that SETF is no worse than MLF where the thresholds are powers of $(1+\epsilon)$. Second, the behavior of MLF on an instance I can be related to the behavior of shortest job first (SJF) algorithm on another instance I' that is obtained from/by dividing each job into logarithmically many jobs with geometrically increasing sizes. Finally, the performance of SJF is related to SRPT using another $(1+\epsilon)$ factor speedup.

Bansal and Pruhs [2] considered the problem of minimizing the ℓ_p norms of flow time and stretch on a single machine. They showed the following.

Theorem 3 ([2]) *In the clairvoyant setting, SRPT and SJF are* $(1+\epsilon)$*-speed,* $O(1/\epsilon)$*-competitive for minimizing the* ℓ_p *norms of both flow time and stretch. On the other hand, for* $1 < p < \infty$*, no online algorithm (possibly clairvoyant) can be* $O(1)$*-competitive for minimizing* ℓ_p *norms of stretch or flow time without speedup. In particular, any randomized online algorithm is at least* $\Omega(n^{(p-1)/3p^2})$*-competitive for* ℓ_p *norms of stretch and is at least* $\Omega(n^{(p-1)/p(3p-1)})$*-competitive for* ℓ_p *norms of flow time.*

The above lower bounds are somewhat surprising, since SRPT and FIFO are optimum for the case $p = 1$ and $p = \infty$ for flow time.

Bansal and Pruhs [2] also consider the non-clairvoyant case.

Theorem 4 ([2]) *In the non-clairvoyant setting, SETF is* $(1+\epsilon)$*-speed,* $O(1/\epsilon^{2+2/p})$*-competitive for minimizing the* ℓ_p *norms of flow time. For minimizing* ℓ_p *norms of stretch, SETF is* $(1+\epsilon)$*-speed,* $O(1/\epsilon^{3+1/p} \cdot \log^{1+1/p} P)$*-competitive*

Finally, Bansal and Pruhs also consider round robin (RR) or processor sharing that at any time splits the processor equally among the unfinished jobs. RR is considered to be an ideal fair strategy since it treats all unfinished jobs equally. However, they show that

Theorem 5 *For any* $p \geq 1$*, there is an* $\epsilon > 0$ *such that even with a* $(1+\epsilon)$ *times faster processor, RR is not* $n^{o(1)}$*-competitive for minimizing the* ℓ_p *norms of flow time. In particular, for* $\epsilon < 1/2p$*, RR is* $(1+\epsilon)$*-speed,* $\Omega(n^{(1-2\epsilon p)/p})$*-competitive. For* ℓ_p *norms of stretch, RR is* $\Omega(n)$*-competitive as is in fact any randomized non-clairvoyant algorithm.*

The results above have been extended in a couple of directions. Bansal and Pruhs [3] extend these results to *weighted* ℓ_p norms of flow time and stretch. Chekuri, Khanna, Kumar, and Goel [4] have extended these results to the multiple machines case. Their algorithms are particularly elegant: Each job is assigned to some machine at random, and all jobs at a particular machine are processed using SRPT or SETF (as applicable).

Applications

SETF and its variants such as MLF are widely used in operating systems [9,10]. Note that SETF is not really practical since each job could be

preempted infinitely often. However, variants of SETF with fewer preemptions are quite popular.

Open Problems

It would be interesting to explore other notions of fairness in the dynamic scheduling setting. In particular, it would be interesting to consider algorithms that are both fair and have a good average performance.

An immediate open problem is whether the gap between $O(\log^2 P)$ and $\Omega(\log P)$ can be closed for minimizing the average stretch in the non-clairvoyant setting.

Cross-References

- ▸ Flow Time Minimization
- ▸ Minimum Flow Time
- ▸ Multilevel Feedback Queues

Recommended Reading

1. Bansal N, Dhamdhere K, Konemann J, Sinha A (2004) Non-clairvoyant scheduling for minimizing mean slowdown. Algorithmica 40(4):305–318
2. Bansal N, Pruhs K (2003) Server scheduling in the Lp norm: a rising tide lifts all boat. In: Symposium on theory of computing (STOC), San Diego, pp 242–250
3. Bansal N, Pruhs K (2004) Server scheduling in the weighted Lp norm. In: LATIN, Buenos Aires, pp 434–443
4. Chekuri C, Goel A, Khanna S, Kumar A (2004) Multi-processor scheduling to minimize flow time with epsilon resource augmentation. In: Symposium on theory of computing (STOC), Chicago, pp 363–372
5. Kalyanasundaram B, Pruhs K (2000) Speed is as powerful as clairvoyance. J ACM 47(4):617–643
6. Kellerer H, Tautenhahn T, Woeginger GJ (1999) Approximability and nonapproximability results for minimizing total flow time on a single machine. SIAM J Comput 28(4):1155–1166
7. Motwani R, Phillips S, Torng E (1994) Non-clairvoyant scheduling. Theor Comput Sci 130(1):17–47
8. Muthukrishnan S, Rajaraman R, Shaheen A, Gehrke J (2004) Online scheduling to minimize average stretch. SIAM J Comput 34(2):433–452
9. Nutt G (1999) Operating system projects using Windows NT. Addison Wesley, Reading
10. Tanenbaum AS (1992) Modern operating systems. Prentice-Hall, Englewood Cliffs

Shortest Paths Approaches for Timetable Information

Riko Jacob
Institute of Computer Science,
Technical University of Munich, Munich, Germany
IT University of Copenhagen, Copenhagen, Denmark

Keywords

Journey planner; Passenger information system; Timetable lookup; Trip planner

Years and Authors of Summarized Original Work

2004; Pyrga, Schulz, Wagner, Zaroliagis

Problem Definition

Consider the route-planning task for passengers of scheduled public transportation. Here, the running example is that of a train system, but the discussion applies equally to bus, light-rail and similar systems. More precisely, the task is to construct a timetable information system that, based upon the detailed schedules of all trains, provides passengers with good itineraries, including the transfer between different trains.

Solutions to this problem consist of a model of the situation (e.g., can queries specify a limit on the number of transfers?), an algorithmic approach, its mathematical analysis (does it always return the best solution? Is it guaranteed to work fast in all settings?), and an evaluation in the real world (Can travelers actually use the produced itineraries? Is an implementation fast enough on current computers and real data?).

Key Results

The problem is discussed in detail in a recent survey article [6].

Modeling

In a simplistic model, it is assumed that a transfer between trains does not take time. A more realistic model specifies a certain minimum transfer time per station. Furthermore, the objective of the optimization problem needs to be defined. Should the itinerary be as fast as possible, or as cheap as possible, or induce the least possible transfers? There are different ways to resolve this as surveyed in [6], all originating in multi-objective optimization, like resource constraints or Pareto-optimal solutions. From a practical point of view, the preferences of a traveler are usually difficult to model mathematically, and one might want to let the user choose the best option among a set of reasonable itineraries himself. For example, one can compute all itineraries that are not inferior to some other itinerary in all considered aspects. As it turns out, in real timetables the number of such itineraries is not too big, such that this approach is computationally feasible and useful for the traveler [5]. Additionally, the fare structure of most railways is fairly complicated [4], mainly because fares usually are not additive, i.e., are not the sum of fares of the parts of a trip.

Algorithmic Models

The current literature establishes two main ideas how to transform the situation into a shortest path problem on a graph. As an example, consider the simplistic modeling where transfer takes no time, and where queries specify starting time and station to ask for an itinerary that achieves the earliest arrival time at the destination.

In the time-expanded model [11], every arrival and departure event of the timetable is a vertex of the directed graph. The arcs of the graph represent consecutive events at one station, and direct train connections. The length of an arc is given by the time difference of its end vertices. Let s be the vertex at the source station whose time is directly after the starting time. Now, a shortest path from s to any vertex of the destination station is an optimal itinerary.

In the time-dependent model [3, 7, 9, 10], the vertices model stations, and the arcs stand for the existence of a direct (non-stop) train connection. Instead of edge length, the arcs are labeled with edge-traversal functions that give the arrival time at the end of the arc in dependence on the time a passenger starts at the beginning of the arc, reflecting the times when trains actually run. To solve this time-dependent shortest path problem, a modification of Dijkstra's algorithm can be used. Further exploiting the structure of this situation, the graph can be represented in a way that allows constant time evaluation of the link traversal functions [3]. To cope with more realistic transfer models, a more complicated graph can be used.

Additionally, many of the speed-up techniques for shortest path computations can be applied to the resulting graph queries.

Applications

The main application are timetable information systems for scheduled transit (buses, trains, etc.). This extends to route planning where trips in such systems are allowed, as for example in the setting of fine-grained traffic simulation to compute fastest itineraries [2].

Open Problems

Improve computation speed, in particular for fully integrated timetables and the multi-criteria case. Extend the problem to the dynamic case, where the current real situation is reflected, i.e., delayed or canceled trains, and otherwise temporarily changed timetables are reflected.

Experimental Results

In the cited literature, experimental results usually are part of the contribution [2, 4, 5, 6, 7, 8, 9, 10, 11]. The time-dependent approach can

be significantly faster than the time-expanded approach. In particular for the simplistic models speed-ups in the range 10–45 are observed [8, 10]. For more detailed models, the performance of the two approaches becomes comparable [6].

Cross-References

- ▸ Implementation Challenge for Shortest Paths
- ▸ Routing in Road Networks with Transit Nodes
- ▸ Single-Source Shortest Paths

Acknowledgments I want to thank Matthias Müller-Hannemann, Dorothea Wagner, and Christos Zaroliagis for helpful comments on an earlier draft of this entry.

Recommended Reading

1. Gerards B, Marchetti-Spaccamela A (eds) (2004) Proceedings of the 3rd workshop on algorithmic methods and models for optimization of railways (ATMOS'03) 2003. Electronic notes in theoretical computer science, vol 92. Elsevier
2. Barrett CL, Bisset K, Jacob R, Konjevod G, Marathe MV (2002) Classical and contemporary shortest path problems in road networks: implementation and experimental analysis of the TRANSIMS router. In: Algorithms – ESA 2002: 10th annual European symposium, Rome, 17–21 Sept 2002. Lecture notes computer science, vol 2461. Springer, Berlin, pp 126–138
3. Brodal GS, Jacob R (2003) Time-dependent networks as models to achieve fast exact time-table queries. In: Proceedings of the 3rd workshop on algorithmic methods and models for optimization of railways (ATMOS'03), [1], pp 3–15
4. Müller-Hannemann M, Schnee M (2006) Paying less for train connections with MOTIS. In: Kroon LG, Möhring RH (eds) Proceedings of the 5th workshop on algorithmic methods and models for optimization of railways (ATMOS'05), Dagstuhl, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl. Dagstuhl Seminar Proceedings, no. 06901
5. Müller-Hannemann M, Schnee M (2007) Finding all attractive train connections by multi-criteria pareto search. In: Geraets F, Kroon LG, Schöbel A, Wagner D, Zaroliagis CD (eds) Algorithmic methods for railway optimization, international Dagstuhl workshop, Dagstuhl Castle, 20–25 June 2004, 4th international workshop, ATMOS 2004, Bergen, 16–17 Sept 2004, revised selected papers. Lecture notes in computer science, vol 4359. Springer, Berlin, pp 246–263
6. Müller-Hannemann M, Schulz F, Wagner D, Zaroliagis CD (2007) Timetable information: models and algorithms. In: Geraets F, Kroon LG, Schöbel A, Wagner D, Zaroliagis CD (eds) Algorithmic methods for railway optimization, international Dagstuhl workshop, Dagstuhl Castle, 20–25 June 2004, 4th International Workshop, ATMOS 2004, Bergen, 16–17 Sept 2004, revised selected papers. Lecture notes in computer science, vol 4359. Springer, pp 67–90
7. Nachtigall K (1995) Time depending shortest-path problems with applications to railway networks. Eur J Oper Res 83:154–166
8. Pyrga E, Schulz F, Wagner D, Zaroliagis C (2004) Experimental comparison of shortest path approaches for timetable information. In: Proceedings 6th workshop on algorithm engineering and experiments (ALENEX). Society for Industrial and Applied Mathematics, pp 88–99
9. Pyrga E, Schulz F, Wagner D, Zaroliagis C (2003) Towards realistic modeling of time-table information through the time-dependent approach. In: Proceedings of the 3rd workshop on algorithmic methods and models for optimization of railways (ATMOS'03), [1], pp 85–103
10. Pyrga E, Schulz F, Wagner D, Zaroliagis C (2007) Efficient models for timetable information in public transportation systems. J Exp Algorithm 12:2.4
11. Schulz F, Wagner D, Weihe K (2000) Dijkstra's algorithm on-line: an empirical case study from public railroad transport. J Exp Algorithm 5:1–23

Shortest Paths in Planar Graphs with Negative Weight Edges

Jittat Fakcharoenphol[1] and Satish Rao[2]
[1]Department of Computer Engineering, Kasetsart University, Bangkok, Thailand
[2]Department of Computer Science, University of California, Berkeley, CA, USA

Keywords

Shortest paths in planar graphs with arbitrary arc weights; Shortest paths in planar graphs with general arc weights

Years and Authors of Summarized Original Work

2001; Fakcharoenphol, Rao

Problem Definition

This problem is to find shortest paths in planar graphs with general edge weights. It is known that shortest paths exist only in graphs that contain no negative weight cycles. Therefore, algorithms that work in this case must deal with the presence of negative cycles, i.e., they must be able to detect negative cycles.

In general graphs, the best known algorithm, the Bellman-Ford algorithm, runs in time $O(mn)$ on graphs with n nodes and m edges, while algorithms on graphs with no negative weight edges run much faster. For example, Dijkstra's algorithm implemented with the Fibonacchi heap runs in time $O(m + n \log n)$, and, in case of integer weights Thorup's algorithm runs in linear time. Goldberg [5] also presented an $O(m\sqrt{n} \log L)$-time algorithm where L denotes the absolute value of the most negative edge weights. Note that his algorithm is weakly polynomial.

Notations

Given a directed graph $G = (V, E)$ and a weight function w: $E \to \mathbb{R}$ on its directed edges, a *distance labeling* for a source node s is a function d: $V \to \mathbb{R}$ such that $d(v)$ is the minimum length over all s-to-v paths, where the *length of path* P is $\sum_{e \in P} w(e)$.

Problem 1 (Single-Source-Shortest-Path)
INPUT: A directed graph $G = (V, E)$, weight function w: $E \to \mathbb{R}$, source node $s \in V$.
OUTPUT: If G does not contain negative length cycles, output a distance labeling d for source node s. Otherwise, report that the graph contains some negative length cycle.

The algorithm by Fakcharoenphol and Rao [4] deals with the case when G is planar. They gave an $O(n \log^3 n)$-time algorithm, improving on an $O(n^{3/2})$-time algorithm by Lipton, Rose, and Tarjan [9] and an $O(n^{4/3} \log nL)$-time algorithm by Henzinger, Klein, Rao, and Subramanian [6].

Their algorithm, as in all previous algorithms, uses a recursive decomposition and constructs a data structure called a dense distance graph, which shall be defined next.

A *decomposition* of a graph is a set of subsets $P_1, P_2, \ldots, P_k$ (not necessarily disjoint) such that the union of all the sets is V and for all $e = (u, v) \in E$, there is a unique P_i that contains e. A node v is a *border node* of a set P_i if $v \in P_i$ and there exists an edge $e = (v, x)$ where $x \notin P_i$. The subgraph induced on a subset P_i is referred to as a *piece* of the decomposition.

The algorithm works with a *recursive decomposition* where at each level, a piece with n nodes and r border nodes is divided into two subpieces such that each subpiece has no more than $2n/3$ nodes and at most $2r/3 + c\sqrt{n}$ border nodes, for some constant c. In this recursive context, a border node of a subpiece is defined to be any border node of the original piece or any new border node introduced by the decomposition of the current piece.

With this recursive decomposition, the *level of a decomposition* can be defined in the natural way, with the entire graph being the only piece in the level 0 decomposition, the pieces of the decomposition of the entire graph being the level 1 pieces in the decomposition, and so on.

For each piece of the decomposition, the all-pair shortest path distances between all its border nodes along paths that lie entirely inside the piece are recursively computed. These all-pair distances form the edge set of a non-planar graph representing shortest paths between border nodes. The dense distance graph of the planar graph is the union of these graphs over all the levels.

Using the dense distance graph, the shortest distance queries between pairs of nodes can be answered.

Problem 2 (Shortest-Path-Distance-Data-Structure)
INPUT: A directed graph $G = (V, E)$, weight function w: $E \to \mathbb{R}$, source node $s \in V$.
OUTPUT: If G does not contain negative length cycles, output a data structure that support distance queries between pairs of nodes. Otherwise, report that the graph contains some negative length cycle.

The algorithm of Fakcharoenphol and Rao relies heavily on planarity, i.e., it exploits properties regarding how shortest paths on each piece intersect. Therefore, unlike previous algorithms that require only that the graph can be recursively decomposed with small numbers of border nodes [10], their algorithm also requires that each piece has a nice embedding.

Given an embedding of the piece, a *hole* is a bounded face where all adjacent nodes are border nodes. Ideally, one would hope that there is a planar embedding of any piece in the recursive decomposition where all the border nodes are on a single face and are circularly ordered, i.e., there is no holes in each piece. Although this is not always true, the algorithm works with any decomposition with a constant number of holes in each piece. This decomposition can be found in $O(n \log n)$ time using the simple cycle separator algorithm by Miller [12].

Key Results

Theorem 1 *Given a recursive decomposition of a planar graph such that each piece of the decomposition contains at most a constant number of holes, there is an algorithm that constructs the dense distance graph is $O(n \log^3 n)$ time.*

Given the procedure that constructs the dense distance graph, the shortest paths from a source s can be computed by first adding s as a border node in every piece of the decomposition, computing the dense distance graph, and then extending the distances into all internal nodes on every piece. This can be done in time $O(n \log^3 n)$.

Theorem 2 *The single-source shortest path problem for an n-node planar graph with negative weight edges can be solved in time $O(n \log^3 n)$.*

The dense distance graph can be used to answer distance queries between pairs of nodes.

Theorem 3 *Given the dense distance graph, the shortest distance between any pair of nodes can be found in $O(\sqrt{n} \log^2 n)$ time.*

It can also be used as a dynamic data structure that answers shortest path queries and allows edge cost updates.

Theorem 4 *For planar graphs with only non-negative weight edges, there is a dynamic data structure that supports distance queries and update operations that change edge weights in amortized $O(n^{2/3} \log^{7/3} n)$ time per operation. For planar graph with negative weight edges, there is a dynamic data structures that supports the same set of operations in amortized $O(n^{4/5} \log^{13/5} n)$ time per operation.*

Note that the dynamic data structure does not support edge insertions and deletions, since these operations might destroy the recursive decomposition.

Applications

The shortest path problem has long been studied and continues to find applications in diverse areas. There are a many problems that reduce to the shortest path problem where negative weight edges are required, for example the minimum-mean length directed circuit. For planar graphs, the problem has wide application even when the underlying graph is a grid. For example, there are recent image segmentation approaches that use negative cycle detection [2, 3]. Some of other applications for planar graphs include separator algorithms [13] and multi-source multi-sink flow algorithms [11].

Open Problems

Klein [8] gives a technique that improves the running time of the construction of the dense distance graph to $O(n \log^2 n)$ when all edge weights are non-negative; this also reduces the amortized running time for the dynamic case down to $O(n^{2/3} \log^{5/3} n)$. Also, for planar graphs with no negative weight edges, Cabello [1] gives a faster algorithm for computing the shortest distances between k pairs of nodes. However, the problem

for improving the bound of $O(n \log^3 n)$ for finding shortest paths in planar graphs with general edge weights remains opened.

It is not known how to handle edge insertions and deletions in the dynamic data structure. A new data structure might be needed instead of the dense distance graph, because the dense distance graph is determined by the decomposition.

Cross-References

- All Pairs Shortest Paths in Sparse Graphs
- All Pairs Shortest Paths via Matrix Multiplication
- Approximation Schemes for Planar Graph Problems
- Decremental All-Pairs Shortest Paths
- Fully Dynamic All Pairs Shortest Paths
- Implementation Challenge for Shortest Paths
- Negative Cycles in Weighted Digraphs
- Planarity Testing
- Shortest Paths Approaches for Timetable Information
- Single-Source Shortest Paths

Recommended Reading

1. Cabello S (2006) Many distances in planar graphs. In: SODA '06: proceedings of the seventeenth annual ACM-SIAM symposium on discrete algorithm. ACM, New York, pp 1213–1220
2. Cox IJ, Rao SB, Zhong Y (1996) 'Ratio Regions': a technique for image segmentation. In: Proceedings international conference on pattern recognition, Aug 1996. IEEE, pp 557–564
3. Geiger LCD, Gupta A, Vlontzos J (1995) Dynamic programming for detecting, tracking and matching elastic contours. IEEE Trans Pattern Anal Mach Intell
4. Fakcharoenphol J, Rao S (2006) Planar graphs, negative weight edges, shortest paths, and near linear time. J Comput Syst Sci 72:868–889
5. Goldberg AV (1992) Scaling algorithms for the shortest path problem. SIAM J Comput 21:140–150
6. Henzinger MR, Klein PN, Rao S, Subramanian S (1997) Faster shortest-path algorithms for planar graphs. J Comput Syst Sci 55:3–23
7. Johnson D (1977) Efficient algorithms for shortest paths in sparse networks. J Assoc Comput Mach 24:1–13
8. Klein PN (2005) Multiple-source shortest paths in planar graphs. In: Proceedings of the 16th ACM-SIAM symposium on discrete algorithms, pp 146–155
9. Lipton R, Rose D, Tarjan RE (1979) Generalized nested dissection. SIAM J Numer Anal 16:346–358
10. Lipton RJ, Tarjan RE (1979) A separator theorem for planar graphs. SIAM J Appl Math 36:177–189
11. Miller G, Naor J (1995) Flow in planar graphs with multiple sources and sinks. SIAM J Comput 24:1002–1017
12. Miller GL (1986) Finding small simple cycle separators for 2-connected planar graphs. J Comput Syst Sci 32:265–279
13. Rao SB (1992) Faster algorithms for finding small edge cuts in planar graphs (extended abstract). In: Proceedings of the twenty-fourth annual ACM symposium on the theory of computing, May 1992, pp 229–240
14. Thorup M (2004) Compact oracles for reachability and approximate distances in planar digraphs. J ACM 51:993–1024

Shortest Vector Problem

Daniele Micciancio
Department of Computer Science, University of California, San Diego, La Jolla, CA, USA

Keywords

Closest vector problem; Lattice basis reduction; LLL algorithm; Nearest vector problem; Minimum distance problem

Years and Authors of Summarized Original Work

1982; Lenstra, Lenstra, Lovasz

Problem Definition

A *point lattice* is the set of all integer linear combinations

$$\mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n) = \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i : x_1, \ldots, x_n \in \mathbb{Z} \right\}$$

of n linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^m$ in m-dimensional Euclidean space. For computational purposes, the lattice vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n$ are often assumed to have integer (or rational)

entries, so that the lattice can be represented by an integer matrix $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{m \times n}$ (called *basis*) having the generating vectors as columns. Using matrix notation, lattice points in $\mathcal{L}(\mathbf{B})$ can be conveniently represented as $\mathbf{Bx}$ where $\mathbf{x}$ is an integer vector. The integers m and n are called the *dimension* and *rank* of the lattice respectively. Notice that any lattice admits multiple bases, but they all have the same rank and dimension.

The main computational problems on lattices are the *Shortest Vector Problem*, which asks to find the shortest nonzero vector in a given lattice, and the *Closest Vector Problem*, which asks to find the lattice point closest to a given target. Both problems can be defined with respect to any norm, but the Euclidean norm $\|\mathbf{v}\| = \sqrt{\sum_i v_i^2}$ is the most common. Other norms typically found in computer science applications are the ℓ_1 norm $\|\mathbf{v}\|_1 = \sum_i |v_i|$ and the *max* norm $\|\mathbf{v}\|_\infty = \max_i |v_i|$. This entry focuses on the Euclidean norm.

Since no efficient algorithm is known to solve SVP and CVP exactly in arbitrary high dimension, the problems are usually defined in their approximation version, where the approximation factor $\gamma \geq 1$ can be a function of the dimension or rank of the lattice.

Definition 1 (Shortest Vector Problem, SVP_γ) Given a lattice $\mathcal{L}(\mathbf{B})$, find a nonzero lattice vector $\mathbf{Bx}$ (where $\mathbf{x} \in \mathbb{Z}^n \setminus \{\mathbf{0}\}$) such that $\|\mathbf{Bx}\| \leq \gamma \cdot \|\mathbf{By}\|$ for any $\mathbf{y} \in \mathbb{Z}^n \setminus \{\mathbf{0}\}$.

Definition 2 (Closest Vector Problem, CVP_γ) Given a lattice $\mathcal{L}(\mathbf{B})$ and a target point $\mathbf{t}$, find a lattice vector $\mathbf{Bx}$ (where $\mathbf{x} \in \mathbb{Z}^n$) such that $\|\mathbf{Bx} - \mathbf{t}\| \leq \gamma \cdot \|\mathbf{By} - \mathbf{t}\|$ for any $\mathbf{y} \in \mathbb{Z}^n$.

Lattices have been investigated by mathematicians for centuries in the equivalent language of quadratic forms, and are the main object of study in the *geometry of numbers*, a field initiated by Minkowski as a bridge between geometry and number theory. For a mathematical introduction to lattices see [3]. The reader is referred to [6, 12] for an introduction to lattices with an emphasis on computational and algorithmic issues.

Key Results

The problem of finding an efficient (polynomial time) solution to SVP_γ for lattices in arbitrary dimension was first solved by the celebrated *lattice reduction* algorithm of Lenstra, Lenstra and Lovász [11], commonly known as the *LLL* algorithm.

Theorem 1 *There is a polynomial time algorithm to solve* SVP_γ *for* $\gamma = (2/\sqrt{3})^n$*, where* n *is the rank of the input lattice.*

The LLL algorithm achieves more than just finding a relatively short lattice vector: it finds a so-called *reduced basis* for the input lattice, i.e., an entire basis of relatively short lattice vectors. Shortly after the discovery of the LLL algorithm, Babai [2] showed that reduced bases can be used to efficiently solve CVP_γ as well within similar approximation factors.

Corollary 1 There is a polynomial time algorithm to solve CVP_γ for $\gamma = O(2/\sqrt{3})^n$, where n is the rank of the input lattice.

The reader is referred to the original papers [2, 11] and [12, chap. 2] for details. Introductory presentations of the LLL algorithm can also be found in many other texts, e.g., [5, chap. 16] and [15, chap. 27]. It is interesting to note that CVP is at least as hard as SVP (see [12, chap 2]) in the sense that any algorithm that solves CVP_γ can be efficiently adapted to solve SVP_γ within the same approximation factor.

Both SVP_γ and CVP_γ are known to be NP-hard in their exact ($\gamma = 1$) or even approximate versions for small values of γ, e.g., constant γ independent of the dimension. (See [13, chaps. 3 and 4] and [4, 10] for the most recent results.) So, no efficient algorithm is likely to exist to solve the problems exactly in arbitrary dimension. For any fixed dimension n, both SVP and CVP can be solved exactly in polynomial time using an algorithm of Kannan [9]. However, the dependency of the running time on the lattice dimension is $n^{O(n)}$. Using randomization, exact SVP can be

solved probabilistically in $2^{O(n)}$ time and space using the *sieving* algorithm of Ajtai, Kumar and Sivakumar [1].

As for approximate solutions, the LLL lattice reduction algorithm has been improved both in terms of running time and approximation guarantee. (See [14] and references therein.) Currently, the best (randomized) polynomial time approximation algorithm achieves approximation factor $\gamma = 2^{O(n \log\log n / \log n)}$.

Applications

Despite the large (exponential in n) approximation factor, the LLL algorithm has found numerous applications and lead to the solution of many algorithmic problems in computer science. The number and variety of applications is too large to give a comprehensive list. Some of the most representative applications in different areas of computer science are mentioned below.

The first motivating applications of lattice basis reduction were the solution of integer programs with a fixed number of variables and the factorization of polynomials with rationals coefficients. (See [11, 8], and [15, chap. 16].) Other classic applications are the solution of random instances of low-density subset-sum problems, breaking (truncated) linear congruential pseudorandom generators, simultaneous Diophantine approximation, and the disproof of Mertens' conjecture. (See [8] and [5, chap. 17].)

More recently, lattice basis reduction has been extensively used to solve many problems in cryptanalysis and coding theory, including breaking several variants of the RSA cryptosystem and the DSA digital signature algorithm, finding small solutions to modular equations, and list decoding of CRT (Chinese Reminder Theorem) codes. The reader is referred to [7, 13] for a survey of recent applications, mostly in the area of cryptanalysis.

One last class of applications of lattice problems is the design of cryptographic functions (e.g., collision resistant hash functions, public key encryption schemes, etc.) based on the apparent intractability of solving SVP_γ within small approximation factors. The reader is referred to [12, chap. 8] and [13] for a survey of such applications, and further pointers to relevant literature. One distinguishing feature of many such lattice based cryptographic functions is that they can be proved to be hard to break *on the average*, based on a *worst-case* intractability assumption about the underlying lattice problem.

Open Problems

The main open problems in the computational study of lattices is to determine the complexity of approximate SVP_γ and CVP_γ for approximation factors $\gamma = n^c$ polynomial in the rank of the lattice. Specifically,

- Are there polynomial time algorithm that solve SVP_γ or CVP_γ for polynomial factors $\gamma = n^c$? (Finding such algorithms even for very large exponent c would be a major breakthrough in computer science.)
- Is there an $\epsilon > 0$ such that approximating SVP_γ or CVP_γ to within $\gamma = n^\epsilon$ is NP-hard? (The strongest known inapproximability results [4] are for factors of the form $n^{O(1/\log\log n)}$ which grow faster than any poly-logarithmic function, but slower than any polynomial.)

There is theoretical evidence that for large polynomials factors $\gamma = n^c$, SVP_γ and CVP_γ are not NP-hard. Specifically, both problems belong to complexity class coAM for approximation factor $\gamma = O(\sqrt{n/\log n})$. (See [12, chap. 9].) So, the problems cannot be NP-hard within such factors unless the polynomial hierarchy PH collapses.

URL to Code

The LLL lattice reduction algorithm is implemented in most library and packages for computational algebra, e.g.,

- GAP (http://www.gap-system.org)
- LiDIA (http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/)
- Magma (http://magma.maths.usyd.edu.au/magma/)
- Maple (http://www.maplesoft.com/)
- Mathematica (http://www.wolfram.com/products/mathematica/index.html)
- NTL (http://shoup.net/ntl/).

NTL also includes an implementation of Block Korkine-Zolotarev reduction that has been extensively used for cryptanalysis applications.

Cross-References

- ▸ Cryptographic Hardness of Learning
- ▸ Knapsack
- ▸ Learning Heavy Fourier Coefficients of Boolean Functions
- ▸ Quantum Algorithm for the Discrete Logarithm Problem
- ▸ Quantum Algorithm for Factoring
- ▸ Sphere Packing Problem

Recommended Reading

1. Ajtai M, Kumar R, Sivakumar D (2001) A sieve algorithm for the shortest lattice vector problem. In: Proceedings of the thirty-third annual ACM symposium on theory of computing – STOC 2001, Heraklion, July 2001. ACM, New York, pp 266–275
2. Babai L (1986) On Lovasz' lattice reduction and the nearest lattice point problem. Combinatorica 6(1):1–13, Preliminary version in STACS 1985
3. Cassels JWS (1971) An introduction to the geometry of numbers. Springer, New York
4. Dinur I, Kindler G, Raz R, Safra S (2003) Approximating CVP to within almost-polynomial factors is NP-hard. Combinatorica 23(2):205–243, Preliminary version in FOCS 1998
5. von zur Gathen J, Gerhard J (2003) Modern computer algebra, 2nd edn. Cambridge
6. Grotschel M, Lovász L, Schrijver A (1993) Geometric algorithms and combinatorial optimization. Algorithms and combinatorics, vol 2, 2nd edn. Springer
7. Joux A, Stern J (1998) Lattice reduction: a toolbox for the cryptanalyst. J Cryptol 11(3):161–185
8. Kannan R (1987) Algorithmic geometry of numbers. In: Annual reviews of computer science, vol 2. Annual Review, Palo Alto, pp 231–267
9. Kannan R (1987) Minkowski's convex body theorem and integer programming. Math Oper Res 12(3):415–440
10. Khot S (2005) Hardness of approximating the shortest vector problem in lattices. J ACM 52(5):789–808, Preliminary version in FOCS 2004
11. Lenstra AK, Lenstra HW Jr, Lovász L (1982) Factoring polynomials with rational coefficients. Math Ann 261:513–534
12. Micciancio D, Goldwasser S (2002) Complexity of lattice problems: a cryptographic perspective, vol 671, The Kluwer international series in engineering and computer science. Kluwer Academic, Boston
13. Nguyen P, Stern J (2001) The two faces of lattices in cryptology. In: Silverman J (ed) Cryptography and lattices conference – CaLC 2001, Providence, Mar 2001. Lecture notes in computer science, vol 2146. Springer, Berlin, pp 146–180
14. Schnorr CP (2006) Fast LLL-type lattice reduction. Inf Comput 204(1):1–25
15. Vazirani VV (2001) Approximation algorithms. Springer

Similarity Between Compressed Strings

Jin Wook Kim[1], Amihood Amir[2,5], Gad M. Landau[3], and Kunsoo Park[4]
[1]HM Research, Seoul, Korea
[2]Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel
[3]Department of Computer Science, University of Haifa, Haifa, Israel
[4]School of Computer Science and Engineering, Seoul National University, Seoul, Korea
[5]Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA

Keywords

Alignment between compressed strings; Compressed approximate string matching; Similarity between compressed strings

Years and Authors of Summarized Original Work

2005; Kim, Amir, Landau, Park

Similarity Between Compressed Strings, Table 1 Various scoring metrics

Metric	Match	Mismatch	Indel	Indel of k characters
Longest common subsequence	1	0	0	0
Levenshtein distance	0	1	1	k
Weighted edit distance	0	δ	μ	$k\mu$
Affine gap penalty	1	$-\delta$	$-\gamma-\mu$	$-\gamma-k\mu$

Problem Definition

The problem of computing similarity between two strings is concerned with comparing two strings using some scoring metric. There exist various scoring metrics and a popular one is the Levenshtein distance (or edit distance) metric. The standard solution for the Levenshtein distance metric was proposed by Wagner and Fischer [13], which is based on dynamic programming. Other widely used scoring metrics are the longest common subsequence metric, the weighted edit distance metric, and the affine gap penalty metric. The affine gap penalty metric is the most general, and it is a quite complicated metric to deal with. Table 1 shows the differences between the four metrics.

The problem considered in this entry is the similarity between two compressed strings. This problem is concerned with efficiently computing similarity without decompressing two strings. The compressions used for this problem in the literature are run-length encoding and Lempel-Ziv (LZ) compression [14].

Run-Length Encoding

A string S is run-length encoded if it is described as an ordered sequence of pairs (σ, i), often denoted "σ^i", each consisting of an alphabet symbol, σ, and an integer, i. Each pair corresponds to a *run* in S, consisting of i consecutive occurrences of σ. For example, the string *aaabbbbbaccccbb* can be encoded $a^3b^4a^1c^4b^2$ or, equivalently, $(a,3)(b,4)(a,1)(c,4)(b,2)$. Let A and B be two strings with lengths n and m, respectively. Let A' and B' be the run-length encoded strings of A and B, and n' and m' be the lengths of A' and B', respectively.

Problem 1
INPUT: Two run-length encoded strings A' and B', a scoring metric d.
OUTPUT: The similarity between A' and B' using d.

LZ Compression

Let X and Y be two strings with length $O(n)$. Let X' and Y' be the LZ compressed strings of X and Y, respectively. Then the lengths of X' and Y' are $O(hn/\log n)$, where $h \leq 1$ is the entropy of strings X and Y.

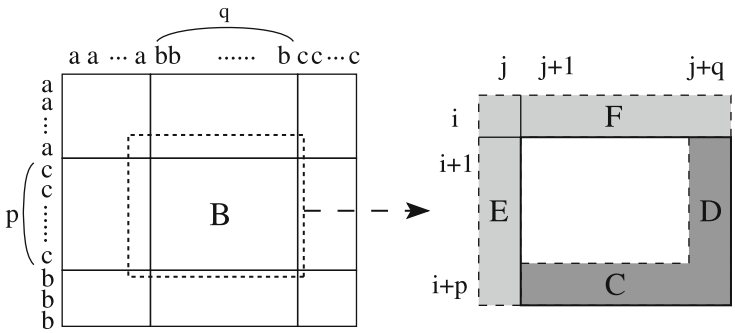
Problem 2
INPUT: Two LZ compressed strings X' and Y', a scoring metric d.
OUTPUT: The similarity between X' and Y' using d.

Block Computation

To compute similarity between compressed strings efficiently, one can use a block computation method. Dynamic programming tables are divided into submatrices, which are called "*blocks*". For run-length encoded strings, a block is a submatrix made up of two runs – one of A and one of B. For LZ compressed strings, a block is a submatrix made up of two phrases – one phrase from each string. See [5] for more details. Then, blocks are computed from left to right and from top to bottom. For each block, only the bottom row and the rightmost column are computed. Figure 1 shows an example of block computation.

Key Results

The problem of computing similarity of two run-length encoded strings, A' and B', has been



Similarity Between Compressed Strings, Fig. 1 Dynamic programming table for strings $a^r c^p b^t$ and $a^s b^q c^u$ is divided into 9 blocks. For one of the blocks, e.g., B, only the bottom row C and the rightmost column D are computed from E and F

studied for various scoring metrics. Bunke and Csirik [4] presented the first solution to Problem 1 using the longest common subsequence metric. The algorithm is based on block computation of the dynamic programming table.

Theorem 1 (Bunke and Csirik [4]) *A longest common subsequence of run-length encoded strings A' and B' can be computed in $O(nm' + n'm)$ time.*

For the Levenshtein distance metric, Arbell, Landau, and Mitchell [2] and Mäkinen, Navarro, and Ukkonen [10] presented $O(nm' + n'm)$ time algorithms, independently. These algorithms are extensions of the algorithm of Bunke and Csirik.

Theorem 2 (Arbell, Landau, and Mitchell [2], Mäkinen, Navarro, and Ukkonen [10]) *The Levenshtein distance between run-length encoded strings A' and B' can be computed in $O(nm' + n'm)$ time.*

For the weighted edit distance metric, Crochemore, Landau, and Ziv-Ukelson [6] and Mäkinen, Navarro, and Ukkonen [11] gave $O(nm' + n'm)$ time algorithms using techniques completely different from each other. The algorithm of Crochemore, Landau, and Ziv-Ukelson [6] is based on the technique which is used in the LZ compressed pattern matching algorithm [6], and the algorithm of Mäkinen, Navarro, and Ukkonen [11] is an extension of the algorithm for the Levenshtein distance metric.

Theorem 3 (Crochemore, Landau, and Ziv-Ukelson [6] Mäkinen, Navarro, and Ukkonen [11]) *The weighted edit distance between run-length encoded strings A' and B' can be computed in $O(nm' + n'm)$ time.*

For the affine gap penalty metric, Kim, Amir, Landau, and Park [8] gave an $O(nm' + n'm)$ time algorithm. To compute similarity in this metric efficiently, the problem is converted into a path problem on a directed acyclic graph and some properties of maximum paths in this graph are used. It is not necessary to build the graph explicitly since they came up with new recurrences using the properties of the graph.

Theorem 4 (Kim, Amir, Landau, and Park [8]) *The similarity between run-length encoded strings A' and B' in the affine gap penalty metric can be computed in $O(nm' + n'm)$ time.*

The above results show that comparison of run-length encoded strings using the longest common subsequence metric is successfully extended to more general scoring metrics.

For the longest common subsequence metric, there exist improved algorithms. Apostolico, Landau, and Skiena [1] gave an $O(n'm' \log(n'm'))$ time algorithm. This algorithm is based on tracing specific optimal paths.

Theorem 5 (Apostolico, Landau, and Skiena [1]) *A longest common subsequence of run-length encoded strings A' and B' can be computed in $O(n'm' \log(n' + m'))$ time.*

Mitchell [12] obtained an $O((d + n' + m') \log(d + n' + m'))$ time algorithm, where d is the number of matches of compressed characters. This algorithm is based on computing geometric

shortest paths using special convex distance functions.

Theorem 6 (Mitchell [12]) *A longest common subsequence of run-length encoded strings A' and B' can be computed in $O((d + n' + m')\log(d + n' + m'))$ time, where d is the number of matches of compressed characters.*

Mäkinen, Navarro, and Ukkonen [11] conjectured an $O(n'm')$ time algorithm on average under the assumption that the lengths of the runs are equally distributed in both strings.

Conjecture 1 (Mäkinen, Navarro, and Ukkonen [11]) A longest common subsequence of run-length encoded strings A' and B' can be computed in $O(n'm')$ time on average.

For Problem 2, Crochemore, Landau, and Ziv-Ukelson [6] presented a solution using the additive gap penalty metric. The additive gap penalty metric consists of 1 for match, $-\delta$ for mismatch, and $-\mu$ for indel, which is almost the same as the weighted edit distance metric.

Theorem 7 (Crochemore, Landau, and Ziv-Ukelson [6]) *The similarity between LZ compressed strings X' and Y' in the additive gap penalty metric can be computed in $O(hn^2/\log n)$ time, where $h \leq 1$ is the entropy of strings X and Y.*

Applications

Run-length encoding serves as a popular image compression technique, since many classes of images (e.g., binary images in facsimile transmission or for use in optical character recognition) typically contain large patches of identically-valued pixels. Approximate matching on images can be a useful tool to handle distortions. Even a one-dimensional compressed approximate matching algorithm would be useful to speed up two-dimensional approximate matching allowing mismatches and even rotations [3, 7, 9].

Open Problems

The worst-case complexity of the problem is not fully understood. For the longest common subsequence metric, there exist some results whose time complexities are better than $O(nm' + n'm)$ to compute the similarity of two run-length encoded strings [1, 11, 12]. It remains open to extend these results to the Levenshtein distance metric, the weighted edit distance metric and the affine gap penalty metric.

In addition, for the longest common subsequence metric, it is an open problem to prove Conjecture 1.

Cross-References

- Approximate String Matching
- Local Alignment (with Affine Gap Weights)
- Multidimensional Compressed Pattern Matching

Recommended Reading

1. Apostolico A, Landau GM, Skiena S (1999) Matching for run length encoded strings. J Complex 15(1):4–16
2. Arbell O, Landau GM, Mitchell J (2002) Edit distance of run-length encoded strings. Inf Process Lett 83(6):307–314
3. Baeza-Yates R, Navaro G (2000) New models and algorithms for multidimensional approximate pattern matching. J Discret Algorithm 1(1):21–49
4. Bunke H, Csirik H (1995) An improved algorithm for computing the edit distance of run length coded strings. Inf Process Lett 54:93–96
5. Crochemore M, Landau GM, Schieber B, Ziv-Ukelson M (2005) Re-use dynamic programming for sequence alignment: an algorithmic toolkit. In: Iliopoulos CS, Lecroq T (eds) String algorithmics. King's College London Publications, London, pp 19–59
6. Crochemore M, Landau GM, Ziv-Ukelson M (2003) A subquadratic sequence alignment algorithm for unrestricted scoring matrices. SIAM J Comput 32(6):1654–1673
7. Fredriksson K, Navarro G, Ukkonen E (2005) Sequential and indexed two-dimensional combinatorial template matching allowing rotations. Theor Comput Sci 347(1–2):239–275

8. Kim JW, Amir A, Landau GM, Park K (2005) Computing similarity of run-length encoded strings with affine gap penalty. In: Proceedings of the 12th symposium on string processing and information retrieval (SPIRE'05), LNCS, vol 3772, pp 440–449
9. Krithivasan K, Sitalakshmi R (1987) Efficient two-dimensional pattern matching in the presence of errors. Inf Sci 43:169–184
10. Mäkinen V, Navarro G, Ukkonen E (2001) Approximate matching of run-length compressed strings. In: Proceedings of the 12th symposium on combinatorial pattern matching (CPM'01). LNCS, vol 2089, pp 31–49
11. Mäkinen V, Navarro G, Ukkonen E (2003) Approximate matching of run-length compressed strings. Algorithmica 35:347–369
12. Mitchell J (1997) A geometric shortest path problem, with application to computing a longest common subsequence in run-length encoded strings. Technical report, Department of Applied Mathematics, SUNY Stony Brook
13. Wagner RA, Fischer MJ (1974) The string-to-string correction problem. J ACM 21(1):168–173
14. Ziv J, Lempel A (1978) Compression of individual sequences via variable rate coding. IEEE Trans Inf Theory 24(5):530–536

Simple Algorithms for Spanners in Weighted Graphs

Surender Baswana[1] and Sandeep Sen[2]
[1]Department of Computer Science and Engineering, Indian Institute of Technology (IIT), Kanpur, Kanpur, India
[2]Indian Institute of Technology (IIT) Delhi, Hauz Khas, New Delhi, India

Keywords

Graph algorithms; Randomized algorithms; Shortest path; Spanner

Years and Authors of Summarized Original Work

2003; Baswana, Sen

Problem Definition

A spanner is a *sparse* subgraph of a given undirected graph that preserves approximate distance between each pair of vertices. More precisely, a t-spanner of a graph $G = (V, E)$ is a subgraph (V, E_S), $E_S \subseteq E$ such that, for any pair of vertices, their distance in the subgraph is at most t times their distance in the original graph, where t is called the *stretch factor*. The spanners were defined formally by Peleg and Schäffer [15] though the associated notion was used implicitly by Awerbuch [3] in the context of network synchronizers.

Computing t-spanner of smallest size for a given graph is a well-motivated combinatorial problem with many applications. However, computing t-spanner of smallest size for a graph is NP-hard. In fact, for $t > 2$, it is NP-hard [11] even to approximate the smallest size of t-spanner of a graph with ratio $O(2^{(1-\mu)\ln n})$ for any $\mu > 0$. Having realized this fact, researchers have pursued another direction which is quite interesting and useful. Let S_G^t be the size of the sparsest t-spanner of a graph G, and let $\mathcal{S}_n^t$ be the maximum value of S_G^t over all possible graphs on n vertices. Does there exist a polynomial time algorithm which computes, for any weighted graph and parameter t, its t-spanner of size $O(\mathcal{S}_n^t)$? Such an algorithm would be the best one can hope for given the hardness of the original t-spanner problem. Naturally, the question arises as to how large can $\mathcal{S}_n^t$ be ? A 43-year-old girth lower bound conjecture by Erdös [13] implies that there are graphs on n vertices whose $2k-$ as well as $(2k-1)$-spanner will require $\Omega(n^{1+1/k})$ edges. This conjecture has been proved for $k = 1, 2, 3$, and 5. Note that a $(2k-1)$-spanner is also a $2k$-spanner, and the lower bound on the size is the same for both $2k$-spanner and $(2k-1)$-spanner. So the objective is to design an algorithm that, for any weighted graph on n vertices, computes a $(2k-1)$-spanner of $O(n^{1+1/k})$ size. Needless to say, one would like to design the fastest algorithm for this problem, and the most ambitious aim would be to achieve the linear time complexity.

Key Results

The key results of this entry are two very simple algorithms which compute a $(2k-1)$-spanner of a given weighted graph $G = (V, E)$. Let n and m

denote, respectively, the number of vertices and edges of G. The first algorithm, due to Althöfer et al. [2], is based on a greedy strategy and runs in $O(mn^{1+1/k})$ time. The second algorithm [6] is based on a very local approach and runs in an expected $O(m)$ time. To start with, consider the following simple observation. Suppose there is a subset $E_S \subset E$ that ensures the following proposition for every edge $(x, y) \in E \backslash E_S$.

> $\mathcal{P}_t(x, y)$: the vertices x and y are connected in the subgraph (V, E_S) by a path consisting of at most t edges, and the weight of each edge on this path is not more than that of the edge (x, y).

It follows easily that the subgraph (V, E_S) will be a t-spanner of G. The two algorithms for computing $(2k-1)$-spanner eventually compute such set E_S based on two completely different approaches.

Algorithm I

This algorithm selects edges for spanner in a greedy fashion and is similar to Kruskal's algorithm for computing a minimum spanning tree. The edges of the graph are processed in the increasing order of their weights. To begin with, the spanner $E_S = \emptyset$; and the algorithm adds edges to it gradually. The decision as to whether an edge, say (u, v), has to be added (or not) to E_S is made as follows:

> If the distance between u and v in the subgraph induced by the current spanner edges E_S is more than $t \cdot weight(u, v)$, then add the edge (u, v) to E_S; otherwise, discard the edge.

It follows that $\mathcal{P}_t(x, y)$ would hold for each edge of E missing in E_S, and so at the end, the subgraph (V, E_S) will be a t-spanner. A well-known result in elementary graph theory states that a graph with more than $n^{1+1/k}$ edges must have a cycle of length at most $2k$. It follows from the above algorithm that the length of any cycle in the subgraph (V, E_S) has to be at least $t + 1$. Hence, for $t = 2k - 1$, the number of edges in the subgraph (V, E_S) will be less than $n^{1+1/k}$. Thus, the algorithm I described above computes a $(2k-1)$-spanner of size $O(n^{1+1/k})$, which is indeed optimal based on the lower bound mentioned earlier.

A simple $O(mn^{1+1/k})$ implementation of algorithm I follows based on Dijkstra's algorithm. Cohen [10] and later Thorup and Zwick [19] designed algorithms for $(2k-1)$-spanner with an improved running time of $O(kmn^{1+1/k})$. These algorithms relied on several calls to Dijkstra's single-source Shortest path algorithm for distance computation and therefore were far from achieving linear time. On the other hand, since a spanner must approximate all-pairs distances in a graph, it appears difficult to compute a spanner by avoiding explicit distance information. Somewhat surprisingly, algorithm II, described in the following section, avoids any sort of distance computation and achieves expected linear time.

Algorithm II

This algorithm employs a novel clustering based on a very local approach and establishes the following result for the spanner problem:

> Given a weighted graph $G = (V, E)$ and an integer $k > 1$, a spanner of $(2k-1)$ stretch and $O(kn^{1+1/k})$ size can be computed in expected $O(km)$ time.

The algorithm executes in $O(k)$ rounds, and in each round it essentially explores adjacency list of each vertex to prune dispensable edges. As a testimony of its simplicity, we will present the entire algorithm for 3-spanner and its analysis in the following section. The algorithm can be easily adapted in other computational models (parallel, external memory, distributed) with nearly optimal performance (see [6] for more details).

Computing a 3-Spanner in Linear Time

To meet the size constraint of a 3-spanner, a vertex, on an average, contributes $\sqrt{n}$ edges to the spanner. So the vertices with degree $O(\sqrt{n})$ are easy to handle since all their edges can be selected in the spanner. For vertices with higher degree, a clustering (groupings) scheme is employed to tackle this problem which has its basis in *dominating sets*.

To begin with, there is a set of edges E' initialized to E and empty spanner E_S. The algorithm processes the edges E', moves some of them to the spanner E_S, and discards the remaining ones. It does so in the following two phases:

1. *Forming the clusters*
   A sample $\mathcal{R} \subset V$ is chosen by picking each vertex independently with probability $\frac{1}{\sqrt{n}}$. The clusters will be formed around these sampled vertices. Initially, the clusters are $\{\{u\}|u \in \mathcal{R}\}$. Each $u \in \mathcal{R}$ is called the *center* of its cluster. Each unsampled vertex $v \in V - \mathcal{R}$ is processed as follows:
   (a) If v is not adjacent to any sampled vertex, then every edge incident on v is moved to E_S.
   (b) If v is adjacent to one or more sampled vertices, let $\mathcal{N}(v, \mathcal{R})$ be the sampled neighbor that is nearest (Ties can be broken arbitrarily. However, it helps conceptually to assume that all weights are distinct) to v. The edge $(v, \mathcal{N}(v, \mathcal{R}))$ along with every edge that is incident on v with weight less than this edge is moved to E_S. The vertex v is added to the cluster centered at $\mathcal{N}(v, \mathcal{R})$.
   As a last step of the first phase, all those edges (u, v) from E' where u and v are not sampled and belong to the same cluster are discarded.
   Let V' be the set of vertices corresponding to the endpoints of the edges E' left after the first phase. It follows that each vertex from V' is either a sampled vertex or adjacent to some sampled vertex, and step 1(b) has partitioned V' into disjoint clusters each centered around some sampled vertex. Also note that, as a consequence of the last step, each edge of the set E' is an intercluster edge. The graph (V', E'), and the corresponding clustering of V', is passed onto the following (second) phase.
2. *Joining vertices with their neighboring clusters*
   Each vertex v of graph (V', E') is processed as follows. Let $E'(v, c)$ be the edges from the set E' incident on v from a cluster c. For each cluster c neighboring to v, the least-weight edge from $E'(v, c)$ is moved to E_S, and the remaining edges are discarded.

The number of edges added to the spanner E_S during the algorithm described above can be bounded as follows. Note that the sample set $\mathcal{R}$ is formed by picking each vertex randomly independently with probability $\frac{1}{\sqrt{n}}$. It thus follows from elementary probability that for each vertex $v \in V$, the expected number of incident edges with weight less than that of $(v, \mathcal{N}(v, \mathcal{R}))$ is at most $\sqrt{n}$. Thus, the expected number of edges contributed to the spanner by each vertex in the first phase of the algorithm is at most $\sqrt{n}$. The number of edges added to the spanner in the second phase is $O(n|\mathcal{R}|)$. Since the expected size of the sample $\mathcal{R}$ is $\sqrt{n}$, therefore, the expected number of edges added to the spanner in the second phase is at most $n^{3/2}$. Hence, the expected size of the spanner E_S at the end of the algorithm described above is at most $2n^{3/2}$. The algorithm is repeated if the size of the spanner exceeds $3n^{3/2}$. It follows using Markov's inequality that the expected number of such repetitions will be $O(1)$.

We now establish that E_S is a 3-spanner. Note that for every edge $(u, v) \notin E_S$, the vertices u, v belong to some cluster in the first phase. There are two cases now.

Case 1 : *(u and v belong to the same cluster)*

Let u and v belong to the cluster centered at $x \in \mathcal{R}$. It follows from the first phase of the algorithm that there is a 2-edge path $u - x - v$ in the spanner with each edge not heavier than the edge (u, v). (This provides a justification for discarding all intracluster edges at the end of the first phase.)

Case 2 : *(u and v belong to different clusters)*

Clearly, the edge (u, v) was removed from E' during phase 2, and suppose it was removed while processing the vertex u. Let v belong to the cluster centered at $x \in \mathcal{R}$.

In the beginning of the second phase, let $(u, v') \in E'$ be the least-weight edge among all the edges incident on u from the vertices of the cluster centered at x. So it

must be that $weight(u, v') \leq weight(u, v)$. The processing of vertex u during the second phase of our algorithm ensures that the edge (u, v') gets added to E_S. Hence, there is a path $\Pi_{uv} = u - v' - x - v$ between u and v in the spanner E_S, and its weight can be bounded as $weight(\Pi_{uv}) = weight(u, v') + weight(v', x) + weight(x, v)$. Since (v', x) and (v, x) were chosen in the first phase, it follows that $weight(v', x) \leq weight(u, v')$ and $weight(x, v) \leq weight(u, v)$. It follows that the spanner (V, E_S) has stretch 3. Moreover, both phases of the algorithm can be executed in $O(m)$ time using elementary data structures and bucket sorting.

The algorithm for computing a $(2k - 1)$-spanner executes k iterations where each iteration is similar to the first phase of the 3-spanner algorithm. For details and formal proofs, the reader may refer to [6].

Other Related Works

The notion of a spanner has been generalized in the past by many researchers.

Additive Spanners

A t-spanner as defined above approximates pairwise distances with multiplicative error and can be called a multiplicative spanner. In an analogous manner, one can define spanners that approximate pairwise distances with additive error. Such a spanner is called an additive spanner, and the corresponding error is called *surplus*. Aingworth et al. [1] presented the first additive spanner of size $O(n^{3/2} \log n)$ with surplus 2. Baswana et al. [7] presented a construction of $O(n^{4/3})$-size additive spanner with surplus 6. Recently, Chechik [9] presented a construction of $O(n^{7/5})$-size additive spanner with surplus 4. It is a major open problem if there exists any sparser additive spanner.

(α, β)-Spanner

Elkin and Peleg [12] introduced the notion of (α, β)-spanner for unweighted graphs, which can be viewed as a hybrid of multiplicative and additive spanners. An (α, β)-spanner is a subgraph such that the distance between any pair of vertices $u, v \in V$ in this subgraph is bounded by $\alpha\delta(u, v) + \beta$, where $\delta(u, v)$ is the distance between u and v in the original graph. Elkin and Peleg showed that an $(1 + \epsilon, \beta)$-spanner of size $O(\beta n^{1+\delta})$, for arbitrarily small $\epsilon, \delta > 0$, can be computed at the expense of sufficiently large surplus β. Recently, Thorup and Zwick [20] introduced a spanner where the additive error is sublinear in terms of the *distance* being approximated.

Other interesting variants of spanner include *distance preserver* proposed by Bollobás et al. [8] and *lightweight* spanner proposed by Awerbuch et al. [4]. A subgraph is said to be a d-preserver if it preserves exact distances for each pair of vertices which are separated by distance at least d. A lightweight spanner tries to minimize the number of edges as well as the total edge weight. A *lightness* parameter is defined for a subgraph as the ratio of total weight of all its edges and the weight of the minimum spanning tree of the graph. Awerbuch et al. [4] showed that for any weighted graph and integer $k > 1$, there exists a polynomially constructible $O(k)$-spanner with $O(k\rho n^{1+1/k})$ edges and $O(k\rho n^{1/k})$ lightness, where $\rho = \log(diameter)$.

In addition to the above work on the generalization of spanners, a lot of work has also been done on computing spanners for special classes of graphs, e.g., chordal graphs, unweighted graphs, and Euclidean graphs. For chordal graphs, Peleg and Schäffer [15] designed an algorithm that computes a 2-spanner of size $O(n^{3/2})$ and a 3-spanner of size $O(n \log n)$. For unweighted graphs, Halperin and Zwick [14] gave an $O(m)$ time algorithm for this problem. Salowe [18] presented an algorithm for computing a $(1 + \epsilon)$-spanner of a d-dimensional complete Euclidean graph in $O(n \log n + \frac{n}{\epsilon^d})$ time. However, none of the algorithms for these special classes of graphs seem to extend to general weighted undirected graphs.

Applications

Spanners are quite useful in various applications in the area of distributed systems and

communication networks. In these applications, spanners appear as the underlying graph structure. In order to build compact routing tables [17], many existing routing schemes use the edges of a sparse spanner for routing messages. In distributed systems, spanners play an important role in designing *synchronizers*. Awerbuch [3] and Peleg and Ullman [16] showed that the quality of a spanner (in terms of stretch factor and the number of spanner edges) is very closely related to the time and communication complexity of any synchronizer for the network. The spanners have also been used implicitly in a number of algorithms for computing all-pairs approximate shortest paths [5, 10, 19, 21]. For a number of other applications, please refer to the papers [2, 3, 15, 17].

Open Problems

The running time as well as the size of the $(2k-1)$-spanner computed by the algorithm described above are away from their respective worst-case lower bounds by a factor of k. For any constant value of k, both these parameters are optimal. However, for the extreme value of k, that is, for $k = \log n$, there is deviation by a factor of $\log n$. Is it possible to get rid of this multiplicative factor of k from the running time of the algorithm and/or the size of the $(2k-1)$-spanner computed? It seems that a more careful analysis coupled with advanced probabilistic tools might be useful in this direction.

Recommended Reading

1. Aingworth D, Chekuri C, Indyk P, Motwani R (1999) Fast estimation of diameter and shortest paths(without matrix multiplication). SIAM J Comput 28:1167–1181
2. Althöfer I, Das G, Dobkin DP, Joseph D, Soares J (1993) On sparse spanners of weighted graphs. Discret Comput Geom 9:81–100
3. Awerbuch B (1985) Complexity of network synchronization. J Assoc Comput Mach 32(4):804–823
4. Awerbuch B, Baratz A, Peleg D (1992) Efficient broadcast and light weight spanners. Tech. Report CS92-22, Weizmann Institute of Science
5. Awerbuch B, Berger B, Cowen L, Peleg D (1998) Near-linear time construction of sparse neighborhod covers. SIAM J Comput 28:263–277
6. Baswana S, Sen S (2007) A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. Random Struct Algorithms 30(4):532–563
7. Baswana S, Kavitha T, Mehlhorn K, Pettie S (2010) Additive spanners and (alpha, beta)-spanners. ACM Trans Algorithms 7(1):5
8. Bollobás B, Coppersmith D, Elkin M (2003) Sparse distance preserves and additive spanners. In Proceedings of the 14th annual ACM-SIAM symposium on discrete algorithms (SODA), Baltimore, pp 414–423
9. Chechik S (2013) New additive spanners. In Proceedings of the twenty-fourth annual ACM-SIAM symposium on discrete algorithms, SODA 2013, New Orleans, 6–8 Jan 2013, pp 498–512
10. Cohen E (1998) Fast algorithms for constructing t-spanners and paths with stretch t. SIAM J Comput 28:210–236
11. Elkin M, Peleg D (2000) The hardness of approximating spanner problems. In STACS 2000, 17th annual symposium on theoretical aspects of computer science, Lille, Feb 2000, proceedings, pp 370–381
12. Elkin M, Peleg D (2004) $(1+\epsilon, \beta)$-spanner construction for general graphs. SIAM J Comput 33:608–631
13. Erdős P (1964) Extremal problems in graph theory. In Theory of graphs and its applications (Proc. Sympos. Smolenice,1963), pp 29–36, Publ. House Czechoslovak Acad. Sci., Prague
14. Halperin S, Zwick U (1996) Linear time deterministic algorithm for computing spanners for unweighted graphs. Unpublished manuscript
15. Peleg D, Schaffer AA (1989) Graph spanners. J Graph Theory 13:99–116
16. Peleg D, Ullman JD (1989) An optimal synchronizer for the hypercube. SIAM J Comput 18:740–747
17. Peleg D, Upfal E (1989) A trade-off between space amd efficiency for routing tables. J Assoc Comput Mach 36(3):510–530
18. Salowe JD (1991) Construction of multidimensional spanner graphs, with application to minimum spanning trees. In ACM symposium on computational geometry, North Conway, pp 256–261
19. Thorup M, Zwick U (2005) Approximate distance oracles. J Assoc Comput Mach 52:1–24
20. Thorup M, Zwick U (2006) Spanners and emulators with sublinear distance errors. In Proceedings of 17th annual ACM-SIAM symposium on discrete algorithms, Miami, 22-26 Jan 2006, pp 802–809

21. Wulff-Nilsen C (2012) Approximate distance oracles with improved preprocessing time. In Proceedings of the twenty-third annual ACM-SIAM symposium on discrete algorithms, SODA 2012, Kyoto, 17–19 Jan 2012, pp 202–208

Simpler Approximation for Stable Marriage

Zoltán Király
Department of Computer Science, Eötvös Loránd University, Budapest, Hungary
Egerváry Research Group (MTA-ELTE), Eötvös Loránd University, Budapest, Hungary

Keywords

Approximation; Local algorithm; SMTI; Stable marriage

Years and Authors of Summarized Original Work

2013; Király

Introduction

We have a two-sided market, one side is a set U of men, the other side is a set V of women. The first part of the input also contains the mutually acceptable man-woman pairs E. This makes up a bipartite graph $G(U \cup V, E)$. The second part of the input contains the preference lists of each person, that is a weak order (may contain ties) on his/her acceptable pairs.

A *matching* is a set of mutually disjoint acceptable man-woman pairs. Given a matching M, a man m and a woman w form a *blocking* pair, if they are an acceptable pair but are not partners in M, and they both prefer each other to their partner, or have no partner in M. That is either w is unmatched in M or w prefers m to her M-partner, *and* either m is unmatched in M or m prefers w to his M-partner. A matching M is *stable* if there are no blocking pairs.

We consider a two-sided market under incomplete preference lists with ties (SMTI), where the goal is to find a maximum size stable matching (MAX-SMTI).

Problem Definition

Problem 1 (MAX-SMTI)

INPUT: Set U of men, and set V of women and each person's preference list.
OUTPUT: A stable matching of maximum size.

Input format A list of an agent a consists of pairs (a_1, p_1), $(a_2, p_2), \ldots, (a_d, p_d)$, where a_i are the acceptable persons from the other gender and $1 \leq p_i \leq \max(|U|, |V|)$ are integers with ordering $p_1 \geq p_2 \geq \cdots \geq p_d$. Agent a strictly prefers a_i to a_j if $p_i > p_j$ and is indifferent between a_i and a_j if $p_i = p_j$. Moreover women needs a black-box procedure, which on input a_i outputs in constant time p_i (we assume that this procedure is also a part of the input). The size of the input is the number of agents plus the total length of the lists.

Definition of approximation ratios A goodness measure of an approximation algorithm A for a maximization problem is defined as follows: the *approximation ratio* of A is $\max\{\text{opt}(I)/A(I)\}$ over all instances I, where $\text{opt}(I)$ and $A(I)$ are the size of the optimal and the algorithm's solution on instance I, respectively.

Short history It was shown in [4] that finding the optimal solution is NP-hard; moreover, it is APX-hard [3]. The original Deferred Acceptance Algorithm of Gale and Shapley gives a 2-approximation; the first approximation algorithm with a strictly better ratio was presented in [5], where the approximation ratio was 15/8. This was improved in [6] to a 5/3-approximation and later in [9] to a 3/2-approximation; this latter algorithm had nonlinear running time. Recently in [10] and

in [7], linear time 3/2-approximation algorithms were given.

Key Results

A simple variation of the famous Deferred Acceptance Algorithm of Gale and Shapley is presented; which also runs in linear time and gives a 3/2-approximation for the problem MAX-SMTI. This algorithm is local; no central agent or knowledge about the global input is needed.

Algorithm

Preliminary Definitions and Concepts for the Algorithm

During the algorithm, the agents may have different statuses, and some Boolean properties described below, and also varying actual preferences.

A status of a man can be either a **lad** or a **bachelor** or an **old bachelor**. A man can be **active** or inactive. A man is active, if he is not an old bachelor and he is not **engaged** (i.e., he has actually no partner). A man can also be **uncertain**, described later. Initially every man is an active lad.

A status of a woman can be either **maiden** or **engaged**. An engaged woman is **flighty**, if her fiancé is *uncertain*. Initially every woman is maiden.

The actual preferences a man m is described as follows. If women w_1 and w_2 are indifferent on m's list, and w_1 is maiden but w_2 is engaged, then m **prefers maiden** w_1 to **engaged** w_2. An engaged lad is **uncertain** if his list contains a woman he prefers to his actual fiancée (this can happen, if there were two maidens with the same highest priority on m's list, and m became engaged to one of them).

The actual preferences a woman w is described as follows. If there are two men, m_1 and m_2 with the same priority in w's list, and m_1 is a lad, but m_2 is a bachelor, then w **prefers bachelor** m_2 to **lad** m_1. If w is flighty, then she prefers a man who is not uncertain, to a man who is uncertain (regardless of her original preferences).

The Algorithm

> While there exists an active man m, he proposes to his favorite woman w. If w accepts his proposal, they become engaged. If w rejects him, m deletes w from his list and remain active.
> When a woman w gets a new proposal from man m, she accepts this proposal if she (actually) prefers m to her current fiancé. Otherwise she rejects m.
> If w accepted m, then she rejects her previous fiancé, if there was one (breaks off her engagement), and becomes engaged to m.
> If m was engaged to a woman w and later w rejects him, then m becomes active again and deletes w from his list, except if m is uncertain, in this case m keeps w on the list.
> If the list of m becomes empty for the first time, he turns into a bachelor, his original list is recovered, and he reactivates himself. If the list of m becomes empty for the second time, he will turn into an old bachelor and will remain inactive forever.

After the algorithm finishes, the engaged pairs get married and form matching M.

Theorem 1 ([7]) *The algorithm always gives a stable matching M and it is $3/2$-approximating, i.e., the stable matching given has size at least $2/3$ of the maximum size stable matching.*

Running Time, Locality

This algorithm runs in linear time using the assumptions on the input format. Though it is clear that along every edge at most three proposals happen, the technical details must be worked out; see [7] for details.

Local algorithm Each agent (a man or woman) always makes a greedy decision based only on local information (his/her preference list, and provided by some communication with his/her acceptable partners). A local algorithm is linear if

every agent communicates with each acceptable partner only a constant time during the algorithm.

The algorithm presented is a linear time local algorithm (using the appropriate data structures); see [7] for details.

Cross-References

- ▸ Hospitals/Residents Problem
- ▸ Maximum Cardinality Stable Matchings
- ▸ Stable Marriage
- ▸ Stable Marriage with One-Sided Ties
- ▸ Stable Marriage with Ties and Incomplete Lists

Recommended Reading

1. Gale D, Shapley LS (1962) College admissions and the stability of marriage. Am Math Mon 69:9–15
2. Gusfield D, Irving RW (1989) The stable marriage problem: structure and algorithms. MIT, Boston
3. Halldórsson MM, Irving RW, Iwama K, Manlove DF, Miyazaki S, Morita Y, Scott S (2003) Approximability results for stable marriage problems with ties. Theor Comput Sci 306:431–447
4. Iwama K, Manlove DF, Miyazaki S, Morita Y (1999) Stable marriage with incomplete lists and ties. In: Proceedings of the 26th international colloquium on automata, languages and programming (ICALP 1999), Prague. LNCS, vol 1644, pp 443–452
5. Iwama K, Miyazaki S, Yamauchi N (2007) A 1.875-approximation algorithm for the stable marriage problem. In: SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on discrete algorithms, pp 288–297
6. Király Z (2009(online), 2011) Better and simpler approximation algorithms for the stable marriage problem. Algorithmica 60(1):3–20
7. Király Z (2013) Linear time local approximation algorithm for maximum stable marriage. Algorithms 6(3):471–484
8. Manlove D (2013) Algorithmics of matching under preferences. World Scientific Publishing, Singapore
9. McDermid EJ (2009) A $\frac{3}{2}$-approximation algorithm for general stable marriage. In: Proceedings of the 36th international colloquium automata, languages and programming (ICALP 2009), Rhodes, pp 689–700
10. Paluch K (2014) Faster and simpler approximation of stable matchings. Algorithms 7(2):189–202

Single and Multiple Buffer Processing

Sergey I. Nikolenko[1] and Kirill Kogan[2]
[1]Laboratory of Mathematical Logic, Steklov Institute of Mathematics, St. Petersburg, Russia
[2]IMDEA Networks, Madrid, Spain

Keywords

Admission control; Buffer management policies; Online algorithms

Years and Authors of Summarized Original Work

2004; Kesselman, Mansour
2012; Keslassy, Kogan, Scalosub, Segal
2012; Kesselman, Kogan, Segal
2012, 2013; Kogan, López-Ortiz, Nikolenko, Sirotkin
2014; Eugster, Kogan, Nikolenko, Sirotkin

Problem Definition

Buffer management policies are online algorithms that control a limited buffer of packets with homogeneous or heterogeneous characteristics, deciding whether to accept new packets when they arrive, which packets to process and transmit, and possibly whether to push out packets already residing in the buffer. Although settings differ, the problem is always to achieve the best possible competitive ratio, i.e., find a policy with good worst-case guarantees in comparison with an optimal offline clairvoyant algorithm. The policies themselves are often simple, simplicity being an important advantage for implementation in switches; the hard problem is to find proofs of lower and especially upper bounds for their competitive

The work of Sergey Nikolenko was partially supported by the Government of the Russian Federation (grant 14.Z50.31.0030).

ratios. Thus, this problem is more theoretical in nature, although the resulting throughput guarantees are important tools in the design of network elements. Comprehensive surveys of this field have been given in the past by Goldwasser [9] and Epstein and van Stee [7].

General Model Description

We assume discrete slotted time. A packet is *fully processed* if the processing unit has scheduled the packet for processing for at least its required number of cycles. Each packet may have the following characteristics: (i) *required processing*, i.e., how many processing cycles the packet has to go through before it can be transmitted; (ii) *value*, i.e., how much the packet contributes to the objective function when it is transmitted; (iii) *output port*, i.e., where the packet is headed (in settings with multiple output ports, it is usually assumed that processing occurs independently at each port, so it becomes advantageous to have more busy output ports at a time); and (iv) *size*, i.e., how many slots (bytes) a packet occupies in the buffer. The objective of a buffer management policy is to maximize the total value of transmitted packets. Different settings may assume that some characteristics are uniform.

Competitive Analysis

Competitive analysis provides a uniform throughput guarantee for online algorithms across all traffic patterns. An online algorithm ALG is said to be α-*competitive* with respect to some objective function f (for some $\alpha \geq 1$ which is called the *competitive ratio*) if for any arrival sequence σ the objective function value on the result of ALG is at least $1/\alpha$ times the objective function value on the solution obtained by an offline clairvoyant algorithm, denoted OPT.

Problem 1 (Competitive Ratio) For a given switch architecture, packet characteristics, and an online algorithm ALG in a given setting, prove lower and upper bounds on its competitive ratio with respect to weighted throughput (total value of packets transmitted by an algorithm).

Key Results

Policies and lower and upper bounds on their competitive ratios are outlined according to problem settings; the latter differ in which packet characteristics they assume to be uniform and which are allowed to vary, and additional restrictions may be imposed on admission, processing and/or transmission order, and admissible packet characteristics.

Uniform Processing, Uniform Value, Shared Memory Switch

Since all packets are identical, the problem for a single queue with one output port is trivial. We consider an $M \times N$ shared memory switch that can hold B packets, with a separate processor on each output port. All packets require a single processing cycle and have equal value; the goal is to maximize the number of transmitted packets. Each packet is labeled with an output port where it has to be processed and transmitted.

Non-Push-Out Policies

Kesselman and Mansour [14] show an adversarial logarithmic lower bound: no non-push-out policy can achieve competitive ratio better than $d/2$ for $d = \log_d N$. On the positive side, they present the Harmonic policy that allocates approximately $1/i$ of the buffer to the ith largest queue and, for its variant, the Parametric Harmonic policy, show an upper bound of $c \log_c N + 1$.

Push-Out Policies

The best known policy is Longest Queue Drop (LQD): accept packets greedily if the buffer is not full; if it is, accept the new packet and then drop a packet from the longest queue (destined to the output port with the most packets assigned to it). Aiello et al. [1,10] show that the competitive ratio of LQD is between $\sqrt{2}$ and 2; they also provide nonconstant lower bounds for other popular policies and a general adversarial lower bound of $\frac{4}{3}$ on the competitive ratio of any online algorithm.

Uniform Processing, Uniform Value, Multiple Separated Queues

In an $N \times 1$ switch where each of N input queues has a separate independent buffer of size B, a policy must select which input queue to take a packet from and set admission policies for input queues. For uniform values, the problem was closed by Azar and Litichevskey [3] with a deterministic policy with competitive ratio converging to $\frac{e}{e-1} \approx 1.582$ for arbitrary B; a matching lower bound was shown by Azar and Richter [4].

Uniform Processing, Variable Values, Single Queue

Here, there is only one output port (a single queue), and each packet is fully processed in one cycle; however, packets have different values, making it desirable to drop packets with smaller value and process packets of larger value. It is easy to show that the Priority Queue (PQ) policy that sorts packets with respect to values and pushes out smaller values for larger ones is optimal. Research has concentrated on models with additional constraints: non-push-out policies that are not allowed to push admitted packets out and the FIFO model where packets have to be transmitted in order of arrival. Another important special case considers two possible values: 1 and $V > 1$.

Non-Push-Out Policies

Aiello et al. [2] consider five online policies for the two-valued case, considering the specific cases of $V = 1$, $V = 2$, and $V = \infty$. Andelman, Mansour, and Zhu provide a deterministic policy (Ratio Partition) that achieves optimal $\left(2 - \frac{1}{V}\right)$-competitiveness [26]. In the case of arbitrary values between 1 and $V > 1$, they show that the optimal competitive ratio is $\ln V$, proving tightly matching bounds of $1 + \ln V$ and $2 + \ln V + O(\ln^2 V/B)$ [2,26].

Push-Out Policies

In the FIFO model, there has been a line of adversarial lower bounds culminating in the lower bound of 1.419 shown by Kesselman, Mansour, and van Stee [18] that applies to all algorithms, with a stronger bound of 1.434 for $B = 2$ [2,26]. As for upper bounds, in this simple model the FIFO greedy push-out policy (accept every packet to end of queue, then push out the packet with smallest value if buffer has overflown) has been shown by Kesselman et al. to be 2-competitive [17]; in the two-valued case, they provide an adversarial lower bound of 1.282, and a long line of improvements for the upper bound has led to the optimal Account Strategy policy of Englert and Westermann [6]. They show an adversarial lower bound of $r = \frac{1}{2}(\sqrt{13} - 1) \approx 1.303$ for any $B \geq 2$ and $r_\infty = \sqrt{2} - \frac{1}{2}(\sqrt{5 + 4\sqrt{2}} - 3) \approx 1.282$ for $B \to \infty$ and show that Account Strategy achieves competitive ratio r for arbitrary B and r_∞ for $B \to \infty$. Thus, in the push-out two-valued case, the gap between lower and upper bounds has been closed completely.

Uniform Processing, Variable Values, Multiple Separated Queues

Kawahara et al. [11] consider an $N \times 1$ switch with N separated queues, each of which has a distinct buffer of size B and has a value α_j associated with it, $1 = \alpha_1 \leq \ldots \leq \alpha_N = \alpha$. A policy selects one of N queues, maximizing total transmitted value; [11] provides matching lower and upper bounds for the PQ policy as $1 + \frac{\sum_{j=1}^{n'} \alpha_j}{\sum_{j=1}^{n'+1} \alpha_j}$, where $n' = \arg\max_n \frac{\sum_{j=1}^{n} \alpha_j}{\sum_{j=1}^{n+1} \alpha_j}$, and an adversarial lower bound $1 + \frac{\alpha^3+\alpha^2+\alpha}{\alpha^4+4\alpha^3+3\alpha^2+4\alpha+1}$ for any online algorithm. Azar and Richter [4] show that any r-competitive policy for a FIFO queue with variable values yields a $2r$-competitive policy for multiple queues. Kobayashi et al. [21] show that an r-competitive policy for unit values and multiple queues yields a $\min\left\{Vr, \frac{Vr(2-r)+r^2-2r+2}{V(2-r)+r-1}\right\}$-competitive policy for the two-valued case.

Uniform Processing, Variable Values, Shared Memory Switch

Several output queues, each with a processor, share a buffer of size B, and each unit-sized packet is labeled with an output port

and an intrinsic value from 1 to V. Eugster, Kogan, Nikolenko, and Sirotkin [8] show a $(\sqrt[3]{V} - o(\sqrt[3]{V}))$ lower bound for the LQD (Longest Queue Drop) policy, an $\frac{1}{2}(\min\{V, B\} - 1)$ lower bound for the MVD (Minimal Value Drop) policy, and a $\frac{4}{3}$ lower bound for the MRD (Maximal Ratio Drop) policy.

Uniform Processing, CIOQ Switches

In CIOQ (Combined Input–Output Queued) switches, one maintains at each input a separate queue for each output (also called Virtual Output Queuing, VOQ). To get delay guarantees of an input queuing (IQ) switch closer to those of an output queuing switch (OQ), one usually assumes increased *speedup* S: the switching fabric runs S times faster than each of the input or the output ports. Hence, an OQ switch has a speedup of N (where N is the number of input/output ports), whereas an IQ switch has a speedup of 1; for $1 < S < N$, packets need to be buffered at the inputs before switching as well as at the outputs after switching. This architecture is called a CIOQ switch.

Uniform Values

Consider an $N \times N$ CIOQ switch with speedup S. Packets of equal size arrive at input ports, each labeled with the output port where it has to leave the switch. Each packet is placed in the input queue corresponding to its output port; when it crosses the switch fabric, it is placed in the output queue and resides there until it is sent on the output link. For unit-valued packets, Kesselman and Rosén [15] proposed a non-push-out policy which is 3-competitive for any S and 2-competitive for $S = 1$. Kesselman, Kogan, and Segal [13] show an upper bound of 4 on the competitiveness of a simple greedy policy.

Variable Values

For up to m packet values in $[1, V]$, Kesselman and Rosén [15] show two push-out policies to be $4S$- and $8\min\{m, 2\log V\}$-competitive. Azar and Richter [5] propose a push-out policy β-PG with parameter β; Kesselman et al. [20] show that the competitive ratio of β-PG is at most 7.5 for $\beta = 3$ and at most 7.47 for $\beta = 2.8$. Kesselman and Rosén [16] consider CIOQ switches with PQ buffers (transmit the highest value packet) and show that this policy is 6-competitive for any S.

Uniform Processing, Crossbar Switches

In the buffered crossbar switch architecture, a small buffer is placed on each crosspoint in addition to input and output queues, which greatly simplifies the scheduling process. For packets with unit length and value, Kesselman et al. [20] introduce a greedy switch policy with competitive ratio between $\frac{3}{2}$ and 4 and show a general lower bound of $\frac{3}{2}$ for unit-size buffers. For variable values and PQ buffers, they propose a push-out greedy switch policy with preemption factor β with competitive ratio between $(2\beta-1)/(\beta-1)$ (3.87 for $\beta = 1.53$) and $(\beta + 2)^2 + 2/(\beta - 1)$ (16.24 for $\beta = 1.53$). For variable values and FIFO buffers, they propose a β-push-out greedy switching policy with competitive ratio $6 + 4\beta + \beta^2 + 3/(\beta - 1)$ (19.95 for $\beta = 1.67$) [19].

Uniform Values, Variable Processing, Single Queue

In this setting, each packet contributes one unit to the objective function, but different packets have different processing requirements, i.e., they spend a different number of time slots at the processor. We denote maximal possible required processing by k.

Non-Push-Out Policies

For a single queue and packets with heterogeneous processing, non-push-out policies have not been considered in any detail. Kogan, López-Ortiz, Nikolenko, and Sirotkin [23] have shown that any greedy non-push-out policy is at least $\frac{1}{2}(k + 1)$-competitive. It remains an open problem to find non-push-out policies with sublinear competitive ratios or show that none exists.

Push-Out Policies

Keslassy et al. [12] showed that again, for a single queue, PQ (Priority Queue) that sorts packets with respect to required processing (smallest first) is optimal; research has concentrated on the FIFO case, where packets have to be transmitted in order of arrival. Kogan et al. [24] introduced

lazy policies that process packets down to a single cycle but then delay their transmission until the entire queue consists of such packets; then all packets are transmitted out in as many time slots as there are packets in the queue. In [24], LPO (Lazy Push-Out) was proven to be at most $(\max\{1, \ln k\} + 2 + o(1))$-competitive; [24] also provides a lower bound of $\lfloor \log_B k \rfloor + 1 - O(1/B)$ for both PO (push-out FIFO) and LPO; for large k this bound matches the upper bound up to a factor of $\log B$. Proving a matching upper bound for the PO policy remains an important open problem. In the two-valued case, when packets may have required processing only 1 or k, LPO has a lower bound of $2 - \frac{1}{k}$ and a matching upper bound of $2 + \frac{1}{B}$ [24]. Kogan, López-Ortiz, Nikolenko, and Sirotkin [23] introduce *semi-FIFO* policies, separating processing order from transmission order so that transmission can conform to FIFO constraints while processing order remains arbitrary. Lazy policies thus become a special case of semi-FIFO policies. The authors show a general upper bound of $\frac{1}{B} \log_{\frac{B}{B-1}} k + 3$ on the competitive ratio of any lazy policy and a matching lower bound of $\frac{1}{B} \log_{\frac{B}{B-1}} k + 1$ for several processing orders. In the two-valued case, when processing is only 1 or k, this upper bound improves to $2 + \frac{1}{B}$, so any lazy policy has constant competitiveness. LPQ (Lazy Priority Queue) also falls in the semi-FIFO class; its competitiveness is between $\left(2 - \frac{1}{B}\left\lceil \frac{B}{k} \right\rceil\right)$ and 2 even for arbitrary processing requirements. Kogan et al. [22] consider a generalization with packets of varying size, considering several natural policies and showing an upper bound of $4L$ for one of PO policies, where L is the maximal packet size.

Copying Cost

An important generalization of the heterogeneous processing model was introduced by Keslassy et al. [12]. They attach a penalty α called copying cost to admitting a packet in the queue; thus, the objective function is now $T - \alpha A$, where T is the number of transmitted packets and A is the number of accepted ones, and it becomes less advantageous to push packets out. To deal with copying cost, the authors propose to use β-push-out policies that push a packet out only if its required processing is at least $\beta > 1$ times less than the required processing of a packet which is being pushed out. Keslassy et al. [12] consider the PQ_β policy (Priority Queue with β-push-out) and show that it is at most $\frac{1}{1-\alpha \log_\beta k}\left(1 + \log_{\frac{\beta}{\beta-1}} \frac{k}{2} + 2\log_\beta k\right)(1-\alpha)$-competitive. Kogan, López-Ortiz, Nikolenko, and Sirotkin [23] show that for any processing order, a β-push-out lazy policy LA_β has competitive ratio at most $\left(3 + \frac{1}{B}\log_{\frac{\beta B}{\beta B - 1}} k\right)\frac{1-\alpha}{1-\alpha\log_\beta k}$. They show a lower bound $\frac{1-\alpha}{1-\alpha\log_\beta k}$ on the competitive ratio of any β-push-out policy, which matches the additional factor in the upper bound. In the two-valued case, the upper bound becomes $\left(2 + \frac{1}{B}\right)\frac{1-\alpha}{1-2\alpha}$, and the authors also show a matching lower bound of $\frac{(2B-2)(1-\alpha)}{(B-1)(1-2\alpha)+(1-\alpha)}$.

Uniform Values, Variable Processing, Multiple Separated Queues

Consider k separate queues of size B each; packets with required processing i fall into the ith queue, and the processor chooses which queue to process on a given time slot. Push-out is irrelevant since queues are independent and packets in a queue are identical. Kogan, López-Ortiz, Nikolenko, and Sirotkin [25] show linear lower bounds for several seemingly attractive policies: $\frac{1}{2}\min\{k, B\}$ for LQF (Longest Queue First), k for SQF (Shortest Queue First), $\frac{3k(k+2)}{4k+16}$ for PRR (Packet Round Robin), and an almost linear lower bound of $\frac{k}{H(k)}$, where $H(k) = \sum_{i=1}^{k} \frac{1}{i} \approx \ln k + \gamma$, for CRR (Cycle Round Robin). They introduce a policy called MQF (Minimal Queue First) that processes packets from a nonempty queue with minimal processing requirement. They show that MQF is at least $\left(1 + \frac{k-1}{2k}\right)$-competitive and prove a constant upper bound of 2. For the two-valued case with two queues, 1 and k, Kogan et al. [25] show exactly matching lower and upper bounds for MQF of $1 + \left(1 + \left\lfloor \frac{aB-1}{b} \right\rfloor\right) / \left(B + \left\lceil \frac{1}{a}\left(b\left\lfloor \frac{aB-1}{b} \right\rfloor + 1\right)\right\rceil\right)$.

Uniform Values, Variable Processing, Shared Memory Switch

In this setting, multiple queues with shared memory are implemented in the same way as for

uniform processing and heterogeneous values: there are N output ports, each output port manages a single output queue Q_i, and each output queue collects packets with the same processing requirement (so packets in a given queue are identical).

Non-Push-Out Policies

Eugster, Kogan, Nikolenko, and Sirotkin [8] consider non-push-out policies and show that NHST (Non-Push-Out Harmonic with Static Threshold: $|Q_i|$ is bounded by $\frac{B}{r_i Z}$) is $(kZ + o(kZ))$-competitive, NEST (Non-Push-Out with Equal Static Threshold: $|Q_i|$ is bounded by B/n) is $(N + o(N))$-competitive, NHDT (Non-Push-Out with Harmonic Dynamic Threshold: accept into Q_i if $\sum_{s=1}^{m} |Q_{j_s}| < \frac{B}{H_k}\left(1 + \frac{1}{2} + \ldots + \frac{1}{m}\right)$, where $j_1 \ldots j_m = i$ are queues for which $|Q_j| \geq |Q_i|$) is $(\frac{1}{2}\sqrt{k \ln k} - o(\sqrt{k \ln k}))$-competitive; finding better non-push-out policies is an open problem.

Push-Out Policies

The work [8] also shows lower bounds on the competitive ratio of well-known policies: $(\sqrt{k} - o(\sqrt{k}))$ for LQD (Longest Queue Drop), $(\ln k + \gamma)$ for BQD (Biggest Packet Drop), and $\left(\frac{4}{3} - \frac{6}{B}\right)$ for LWD (Largest Work Drop). The main result of [8] is that LWD is at most 2-competitive.

Open Problems

1. Close the gap between competitive ratios $\frac{4}{3}$ (lower bound for any policy) and 2 (upper bound for LQD) in the uniform processing, uniform value case.
2. Do there exist policies with constant competitive ratio in the uniform processing, variable values, shared memory multiple output queues setting?
3. Do there exist non-push-out policies with sublinear competitive ratio in the case of a single queue with packets with variable processing and uniform values?
4. Prove an upper bound on the competitiveness of PO (push-out) policy in the single-queue FIFO model with heterogeneous required processing and uniform values.
5. Do there exist non-push-out policies with logarithmic competitive ratio in the case of multiple output ports with shared memory that contain packets with variable processing and uniform values?
6. Design efficient policies for CIOQ and crossbar switches with packets with heterogeneous processing and uniform values; prove bounds on their competitive ratios.
7. Design efficient policies and prove bounds on their competitive ratios for the case of packets with both variable values and heterogeneous processing requirements in all of the above settings.

Cross-References

- ▸ Packet Switching in Multi-queue Switches
- ▸ Packet Switching in Single Buffer

Recommended Reading

1. Aiello W, Kesselman A, Mansour Y (2008) Competitive buffer management for shared-memory switches. ACM Trans Algorithms 5(1):1–16
2. Andelman N, Mansour Y, Zhu A (2003) Competitive queueing policies for QoS switches. In: Proceedings of the 4th annual ACM-SIAM symposium on discrete algorithms, Baltimore, pp 761–770
3. Azar Y, Litichevskey A (2006) Maximizing throughput in multi-queue switches. Algorithmica 45(1):69–90
4. Azar Y, Richter Y (2005) Management of multi-queue switches in QoS networks. Algorithmica 43(1-2):81–96
5. Azar Y, Richter Y (2006) An improved algorithm for CIOQ switches. ACM Trans Algorithms 2(2):282–295
6. Englert M, Westermann M (2009) Lower and upper bounds on FIFO buffer management in QoS switches. Algorithmica 53(4):523–548
7. Epstein L, van Stee R (2004) Buffer management problems. SIGACT News 35(3):58–66
8. Eugster P, Kogan K, Nikolenko SI, Sirotkin AV (2014) Shared-memory buffer management for heterogeneous packet processing. In: Proceedings of the 34th international conference on distributed computing systems, Madrid
9. Goldwasser M (2010) A survey of buffer management policies for packet switches. SIGACT News 41(1):100–128

10. Hahne EL, Kesselman A, Mansour Y (2001) Competitive buffer management for shared-memory switches. In: 13th ACM symposium on parallel algorithms and architectures, Crete Island, pp 53–58
11. Kawahara J, Kobayashi K, Maeda T (2012) Tight analysis of priority queuing policy for egress traffic. CoRR abs/1207.5959
12. Keslassy I, Kogan K, Scalosub G, Segal M (2012) Providing performance guarantees in multipass network processors. IEEE/ACM Trans Netw 20(6):1895–1909
13. Kesselman A, Kogan K, Segal M (2008) Best effort and priority queuing policies for buffered crossbar switches. In: Structural information and communication complexity, 15th international colloquium (SIROCCO 2008), Villars-sur-Ollon, 170–184 http://dx.doi.org/10.1007/978-3-540-69355-0_15
14. Kesselman A, Mansour Y (2004) Harmonic buffer management policy for shared memory switches. Theor Comput Sci 324(2-3):161–182
15. Kesselman A, Rosén A (2006) Scheduling policies for CIOQ switches. J Algorithms 60(1):60–83
16. Kesselman A, Rosén A (2008) Controlling CIOQ switches with priority queuing and in multistage interconnection networks. J Interconnect Netw 9(1/2):53–72
17. Kesselman A, Lotker Z, Mansour Y, Patt-Shamir B, Schieber B, Sviridenko M (2004) Buffer overflow management in QoS switches. SIAM J Comput 33(3):563–583
18. Kesselman A, Mansour Y, van Stee R (2005) Improved competitive guarantees for QoS buffering. Algorithmica 43(1-2):63–80
19. Kesselman A, Kogan K, Segal M (2010) Packet mode and QoS algorithms for buffered crossbar switches with FIFO queuing. Distrib Comput 23(3):163–175
20. Kesselman A, Kogan K, Segal M (2012) Improved competitive performance bounds for CIOQ switches. Algorithmica 63(1–2):411–424
21. Kobayashi K, Miyazaki S, Okabe Y (2009) Competitive buffer management for multi-queue switches in QoS networks using packet buffering algorithms. In: Proceedings of the 21st ACM symposium on parallelism in algorithms and architectures (SPAA), Portland, OR, USA, pp 328–336
22. Kogan K, López-Ortiz A, Nikolenko S, Scalosub G, Segal M (2014) Balancing Work and Size with Bounded Buffers. In: Proceedings of the 6th international conference on communication systems and networks (COMSNETS 2014), Bangalore, pp 1–8
23. Kogan K, López-Ortiz A, Nikolenko SI, Sirotkin AV (2012) A taxonomy of semi-FIFO policies. In: Proceedings of the 31st IEEE international performance computing and communications conference (IPCCC2012), Austin, pp 295–304
24. Kogan K, López-Ortiz A, Nikolenko SI, Sirotkin AV, Tugaryov D (2012) FIFO queueing policies for packets with heterogeneous processing. In: Proceedings of the 1st Mediterranean conference on algorithms (MedAlg 2012), Ein Gedi. Lecture notes in computer science, vol 7659, pp 248–260
25. Kogan K, López-Ortiz A, Nikolenko SI, Sirotkin A (2013) Multi-queued network processors for packets with heterogeneous processing requirements. In: Proceedings of the 5th international conference on communication systems and networks (COMSNETS 2013), Bangalore, pp 1–10
26. Zhu A (2004) Analysis of queueing policies in QoS switches. J Algorithms 53(2):137–168

Single-Source Fully Dynamic Reachability

Camil Demetrescu[1,2] and Giuseppe F. Italiano[1,2]
[1]Department of Computer and Systems Science, University of Rome, Rome, Italy
[2]Department of Information and Computer Systems, University of Rome, Rome, Italy

Keywords

Single-source fully dynamic transitive closure

Years and Authors of Summarized Original Work

2005; Demetrescu, Italiano

Problem Definition

A dynamic graph algorithm maintains a given property $\mathcal{P}$ on a graph subject to dynamic changes, such as edge insertions, edge deletions and edge weight updates. A dynamic graph algorithm should process queries on property $\mathcal{P}$ quickly, and perform update operations faster than recomputing from scratch, as carried out by the fastest static algorithm. An algorithm is *fully dynamic* if it can handle both edge insertions and edge deletions and *partially dynamic* if it can handle either edge insertions or edge deletions, but not both.

Given a graph with n vertices and m edges, the *transitive closure* (or *reachability*) problem

consists of building an $n \times n$ Boolean matrix M such that $M[x, y] = 1$ if and only if there is a directed path from vertex x to vertex y in the graph. The fully dynamic version of this problem can be defifined as follows:

Definition 1 (Fully dynamic reachability problem) The *fully dynamic reachability problem* consists of maintaining a directed graph under an intermixed sequence of the following operations:

- `insert`(u,v): insert edge (u,v) into the graph.
- `delete`(u,v): delete edge (u,v) from the graph.
- `reachable`(x,y): return *true* if there is a directed path from vertex x to vertex y, and *false* otherwise.

This entry addresses the *single-source* version of the fully-dynamic reachability problem, where one is only interested in queries with a fixed source vertex s. The problem is defined as follows:

Definition 2 (Single-source fully dynamic reachability problem) The *fully dynamic single-source reachability problem* consists of maintaining a directed graph under an intermixed sequence of the following operations:

- `insert`(u,v): insert edge (u,v) into the graph.
- `delete`(u,v): delete edge (u,v) from the graph.
- `reachable`(y): return *true* if there is a directed path from the source vertex s to vertex y, and *false* otherwise.

Approaches

A simple-minded solution to the problem of Definition would be to keep explicit reachability information from the source to all other vertices and update it by running any graph traversal algorithm from the source s after each insert or delete. This takes $O(m + n)$ time per operation, and then reachability queries can be answered in constant time.

Another simple-minded solution would be to answer queries by running a point-to-point reachability computation, without the need to keep explicit reachability information up to date after each insertion or deletion. This can be done in $O(m + n)$ time using any graph traversal algorithm. With this approach, queries are answered in $O(m + n)$ time and updates require constant time. Notice that the time required by the slowest operation is $O(m+n)$ for both approaches, which can be as high as $O(n^2)$ in the case of dense graphs.

The first improvement upon these two basic solutions is due to Demetrescu and Italiano, who showed how to support update operations in $O(n^{1.575})$ time and reachability queries in $O(1)$ time [1] in a directed acyclic graph. The result is based on a simple reduction of the single-source problem of Definition to the all-pairs problem of Definition. Using a result by Sankowski [2], the bounds above can be extended to the case of general directed graphs.

Key Results

This Section presents a simple reduction presented in [1] that allows it to keep explicit single-source reachability information up to date in subquadratic time per operation in a directed graph subject to an intermixed sequence of edge insertions and edge deletions. The bounds reported in this entry were originally presented for the case of directed acyclic graphs, but can be extended to general directed graphs using the following theorem from [2]:

Theorem 1 *Given a general directed graph with n vertices, there is a data structure for the fully dynamic reachability problem that supports each insertion/deletion in $O(n^{1.575})$ time and each reachability query in $O(n^{0.575})$ time. The algorithm is randomized with one-sided error.*

The idea described in [1] is to maintain reachability information from the source vertex s to all other vertices explicitly by keeping a Boolean array R of size n such that $R[y] = 1$ if and

only if there is a directed path from s to y. An instance D of the data structure for fully dynamic reachability of Theorem is also maintained. After each insertion or deletion, it is possible to update D in $O(n^{1.575})$ time and then rebuild R in $O(n \cdot n^{0.575}) = O(n^{1.575})$ time by letting $R[y] \leftarrow D.$ `reachable` (s,y) for each vertex y. This yields the following bounds for the single-source fully dynamic reachability problem:

Theorem 2 *Given a general directed graph with n vertices, there is a data structure for the single-source fully dynamic reachability problem that supports each insertion/deletion in $O(n^{1.575})$ time and each reachability query in $O(1)$ time.*

Applications

The graph reachability problem is particularly relevant to the field of databases for supporting transitivity queries on dynamic graphs of relations [3]. The problem also arises in many other areas such as compilers, interactive verification systems, garbage collection, and industrial robotics.

Open Problems

An important open problem is whether one can extend the result described in this entry to maintain fully dynamic single-source shortest paths in subquadratic time per operation.

Cross-References

▸ Trade-Offs for Dynamic Graph Problems

Recommended Reading

1. Demetrescu C, Italiano G (2005) Trade-offs for fully dynamic reachability on dags: breaking through the $O(n^2)$ barrier. J Assoc Comput Mach 52: 147–156
2. Sankowski P (2004) Dynamic transitive closure via dynamic matrix inverse. In: FOCS '04: proceedings of the 45th annual IEEE symposium on foundations of computer science (FOCS'04). IEEE Computer Society, Washington, DC, pp 509–517
3. Yannakakis M (1990) Graph-theoretic methods in database theory. In: Proceedings of the 9-th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, Nashville, pp 230–242

Single-Source Shortest Paths

Seth Pettie
Electrical Engineering and Computer Science (EECS) Department, University of Michigan, Ann Arbor, MI, USA

Keywords

Shortest route; Quickest route

Years and Authors of Summarized Original Work

1999; Thorup

Problem Definition

The *single-source* shortest path problem (SSSP) is, given a graph $G = (V, E, l)$ and a *source* vertex $s \in V$, to find the shortest path from s to every $v \in V$. The difficulty of the problem depends on whether the graph is directed or undirected and the assumptions placed on the length function ℓ. In the most general situation, $l : E \to \mathbb{R}$ assigns arbitrary (positive and negative) real lengths. The algorithms of Bellman-Ford and Edmonds [1, 4] may be applied in this situation and have running times of roughly $O(mn)$, (Edmonds's algorithm works for undirected graphs and presumes that there are no negative length simple cycles.) where $m = |E|$ and $n = |V|$ are the number of edges and vertices. If ℓ assigns only *nonnegative* real edge lengths, then the

algorithms of Dijkstra and Pettie-Ramachandran [4,13] may be applied on directed and undirected graphs, respectively. These algorithms include a *sorting bottleneck* and, in the worst case, take $\Omega(m + n \log n)$ time. (The [13] algorithm actually runs in $O(m + n \log \log n)$ time if the ratio of any two edge lengths is polynomial in n).

A common assumption is that ℓ assigns *integer* edge lengths in the range $\{0, \ldots, 2^w - 1\}$ or $\{-2^{w-1}, \ldots, 2^{w-1} - 1\}$ and that the machine is a w-bit *word RAM*; that is, each edge length fits in one register. For general integer edge lengths, the best SSSP algorithms improve on Bellman-Ford and Edmonds by a factor of roughly $\sqrt{n}$ [6]. For nonnegative integer edge lengths, the best SSSP algorithms are faster than Dijkstra and Pettie-Ramachandran by up to a logarithmic factor. They are frequently based on integer priority queues [9].

Key Results

Thorup's primary result [16] is an optimal linear time SSSP algorithm for undirected graphs with integer edge lengths. This is the first and only linear time shortest path algorithm that does not make serious assumptions on the class of input graphs.

Theorem 1 *There is a SSSP algorithm for integer-weighted undirected graphs that runs in $O(m)$ time.*

Thorup avoids the sorting bottleneck inherent in Dijkstra's algorithm by precomputing (in linear time) a *component hierarchy*. The algorithm of [16] operates in a manner similar to Dijkstra's algorithm [4] but uses the component hierarchy to identify groups of vertices that can be visited in any order. In later work, Thorup [17] extended this approach to work when the edge lengths are floating-point numbers. (There is some flexibility in the definition of *shortest path* since floating-point addition is neither commutative nor associative).

Thorup's hierarchy-based approach has since been extended to directed and/or real-weighted graphs and to solve the *all pairs* shortest path (APSP) problem [11–13]. The generalizations to related SSSP problems are summarized below. See [11,12] for hierarchy-based APSP algorithms.

Theorem 2 (Hagerup [8], 2000) *A component hierarchy for a directed graph $G = (V, E, l)$, where $l : E \to \{0, \ldots, 2^w - 1\}$, can be constructed in $O(m \log w)$ time. Thereafter, SSSP from any source can be computed in $O(m + n \log \log n)$ time.*

Theorem 3 (Pettie and Ramachandran [13], 2005) *A component hierarchy for an undirected graph $G = (V, E, l)$, where $l : E \to \mathbb{R}^+$, can be constructed in $O(m\alpha(m, n) + \min\{n \log \log r, n \log n\})$ time, where r is the ratio of the maximum-to-minimum edge length. Thereafter, SSSP from any source can be computed in $O(m \log \alpha(m, n))$ time.*

The algorithms of Hagerup [8] and Pettie-Ramachandran [13] take the same basic approach as Thorup's algorithm: use some kind of component hierarchy to identify groups of vertices that can safely be visited in any order. However, the assumption of directed graphs [8] and real edge lengths [13] renders Thorup's hierarchy inapplicable or inefficient. Hagerup's component hierarchy is based on a directed analogue of the minimum spanning tree. The Pettie-Ramachandran algorithm enforces a certain degree of balance in its component hierarchy and, when computing SSSP, uses a specialized priority queue to take advantage of this balance.

Applications

Shortest path algorithms are frequently used as a subroutine in other optimization problems, such as flow and matching problems [1] and facility location [18]. A widely used commercial application of shortest path algorithms is finding efficient routes on road networks, e.g., as provided by Google Maps, MapQuest, or Yahoo Maps.

Open Problems

Thorup's SSSP algorithm [16] runs in linear time and is therefore optimal. The main open problem is to find a linear time SSSP algorithm that works on *real*-weighted *directed* graphs. For real-weighted undirected graphs, the best running time is given in Theorem 3. For integer-weighted directed graphs, the fastest algorithms are based on Dijkstra's algorithm (not Theorem 2) and run in $O(m\sqrt{\log\log n})$ time (randomized) and deterministically in $O(m + n\log\log n)$ time.

Problem 1 Is there an $O(m)$ time SSSP algorithm for integer-weighted directed graphs?

Problem 2 Is there an $O(m) + o(n\log n)$ time SSSP algorithm for real-weighted graphs, either directed or undirected?

The complexity of SSSP on graphs with positive and negative edge lengths is also open.

Experimental Results

Asano and Imai [2] and Pettie et al. [14] evaluated the performance of the hierarchy-based SSSP algorithms [13, 16]. There have been a number of studies of SSSP algorithms on integer-weighted directed graphs; see [7] for the latest and references to many others. The trend in recent years is to find practical preprocessing schemes that allow for very quick point-to-point shortest path queries. See [3, 10, 15] for recent work in this area.

Data Sets

See [5] for a number of US and European road networks.

URL to Code

See [5].

Cross-References

▸ All Pairs Shortest Paths via Matrix Multiplication

Recommended Reading

1. Ahuja RK, Magnati TL, Orlin JB (1993) Network flows: theory, algorithms, and applications. Prentice Hall, Englewood Cliffs
2. Asano Y, Imai H (2000) Practical efficiency of the linear-time algorithm for the single source shortest path problem. J Oper Res Soc Jpn 43(4):431–447
3. Bast H, Funke S, Matijevic D, Sanders P, Schultes D (2007) In transit to constant shortest-path queries in road networks. In: Proceedings 9th workshop on algorithm engineering and experiments (ALENEX), New Orleans
4. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms. MIT, Cambridge
5. Demetrescu C, Goldberg AV, Johnson D (2006) 9th DIMACS implementation challenge-shortest paths. http://www.dis.uniroma1.it/~challenge9/
6. Goldberg AV (1995) Scaling algorithms for the shortest paths problem. SIAM J Comput 24(3):494–504
7. Goldberg AV (2001) Shortest path algorithms: engineering aspects. In: Proceedings of the 12th international symposium on algorithms and computation (ISAAC), Christchurch. LNCS, vol 2223. Springer, Berlin, pp 502–513
8. Hagerup T (2000) Improved shortest paths on the word RAM. In: Proceedings of the 27th international colloquium on automata, languages, and programming (ICALP), Geneva. LNCS, vol 1853. Springer, Berlin, pp 61–72
9. Han Y, Thorup M (2002) Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In: Proceedings of the 43rd symposium on foundations of computer science (FOCS), Vancouver, pp 135–144
10. Knopp S, Sanders P, Schultes D, Schulz F, Wagner D (2007) Computing many-to-many shortest paths using highway hierarchies. In: Proceedings of the 9th workshop on algorithm engineering and experiments (ALENEX), New Orleans
11. Pettie S (2002) On the comparison-addition complexity of all-pairs shortest paths. In: Proceedings of the 13th international symposium on algorithms and computation (ISAAC), Vancouver, pp 32–43
12. Pettie S (2004) A new approach to all-pairs shortest paths on real-weighted graphs. Theor Comput Sci 312(1):47–74
13. Pettie S, Ramachandran V (2005) A shortest path algorithm for real-weighted undirected graphs. SIAM J Comput 34(6):1398–1431
14. Pettie S, Ramachandran V, Sridhar S (2002) Experimental evaluation of a new shortest path algorithm. In: Proceedings of the 4th workshop on algorithm

engineering and experiments (ALENEX), San Francisco, pp 126–142
15. Sanders P, Schultes D (2006) Engineering highway hierarchies. In: Proceedings of the 14th European symposium on algorithms (ESA), Zurich, pp 804–816
16. Thorup M (1999) Undirected single-source shortest paths with positive integer weights in linear time. J ACM 46(3):362–394
17. Thorup M (2000) Floats, integers, and single source shortest paths. J Algorithms 35:189–201
18. Thorup M (2003) Quick and good facility location. In: Proceedings of the 14th annual ACM-SIAM symposium on discrete algorithms (SODA), Baltimore, pp 178–185

Ski Rental Problem

Mark S. Manasse
Microsoft Research, Mountain View, CA, USA

Keywords

Metrical task systems; Oblivious adversaries; Worst-case approximation

Years and Authors of Summarized Original Work

1990; Karlin, Manasse, McGeogh, Owicki

Problem Definition

The ski rental problem was developed as a pedagogical tool for understanding the basic concepts in some early results in online algorithms. (In the interest of full disclosure, the earliest presentations of these results described the problem as the wedding-tuxedo-rental problem. Objections were presented that this was a gender-biased name for the problem, since while groomsmen can rent their wedding apparel, bridesmaids usually cannot. A further complication, owing to the difficulty of instantaneously producing fitted garments or ski equipment outlined below, suggests that some complications could have been avoided by focusing on the dilemma of choosing between daily lift passes or season passes, although this leads to the pricing complexities of purchasing season passes well in advance of the season, as opposed to the higher cost of purchasing them at the mountain during the ski season. A similar problem could be derived from the question as to whether to purchase the daily newspaper at a newsstand or to take a subscription, after adding the challenge that one's peers will treat one contemptuously if one has not read the news on days on which they have.) The ski rental problem considers the plight of one consumer who, in order to socialize with peers, is forced to engage in a variety of athletic activities, such as skiing, bicycling, windsurfing, rollerblading, sky diving, scuba-diving, tennis, soccer, and ultimate Frisbee, each of which has a set of associated apparatus, clothing, or protective gear.

In all of these, it is possible either to purchase the accoutrements needed or to rent them. For the purpose of this problem, it is assumed that one-time rental is less expensive than purchasing. It is also assumed that purchased items are durable, and suitable for reuse for future activities of the same type without further expense, until the items wear out (which occurs at the same rate for all users), are outgrown, become unfashionable, or are disposed of to make room for other purchased items. The social consumer must make the decision to rent or buy for each event, although it is assumed that the consumer is sufficiently parsimonious as to abjure rental if already in possession of serviceable purchased equipment. Whether purchases are as easy to arrange as rentals, or whether some advance planning is required (e.g., to mount bindings on a ski) is a further detail considered in this problem. It is assumed that the social consumer has no particular independent interest in these activities, and engages in these activities only to socialize with peers who choose to engage in these activities disregarding the consumer's desires.

These putative peers are more interested in demonstrating the superiority of their financial acumen to that of the social consumer in question than they are in any particular activity. To that end, the social consumer is taunted mercilessly based on the ratio of his/her total expenses on

rentals and purchases to theirs. Consequently, the peers endeavor to invite the social consumer to engage in events while they are costly to him/her, and once the activities are free to the social consumer, if continued activity would be costly to them, cease. But, to present an illusion of fairness, skis, both rented and purchased, have the same cost for the peers as they do for the social consumer in question. The ski rental problem takes a very restricted setting. It assumes that purchased ski equipment never needs replacement, and that there are no costs to a ski trip other than the skis (thus, no cost for the gasoline, for the lift and/or speeding tickets, for the hot chocolates during skiing, or for the après-ski liqueurs and meals). It is assumed that the social consumer experiences no physical disabilities preventing him/her from skiing and has no impending restrictions to his/her participation in ski trips (obviously, a near-term-fatal illness or an anticipated conviction leading to confinement for life in a penitentiary would eliminate any potential interest in purchasing alpine equipment – when the ratio of purchase to rental exceeds the maximum need for equipment, one should always rent). It is assumed that the social consumer's peers have disavowed any interest in activities other than skiing, and that the closet, basement, attic, garage, or storage locker included in the social consumer's rent or mortgage (or necessitated by other storage needs) has sufficient capacity to hold purchased ski equipment without entailing the disposal of any potentially useful items. Bringing these complexities into consideration brings one closer to the hardware-based problems which initially inspired this work.

The impact of invitations issued with sufficient time allowed for purchasing skis, as well as those without, will be considered.

Given all of that, what ratio of expenses can the social consumer hope to attain? What ratio can the social consumer not expect to beat? These are the basic questions of competitive analysis.

The impact of keeping secrets from one's peers is further considered. Rather than a fixed strategy for when to purchase skis, the social consumer may introduce an element of chance into the process. If the peers are able to observe his/her ski equipment and notice when it changes from rented skis to purchased skis, and change their schedule for alpine recreation in light of this observation, randomness provides no advantages. If, on the other hand, the social consumer announces to the peers, in advance of the first trip, how he/she will decide when the time is right for purchasing skis, including any use of probabilistic techniques, and they then decide on the schedule for ski trips for the coming winter, a deterministic decision procedure generally produces a larger competitive ratio than does a randomized procedure.

Key Results

Given an unbounded sequence of skiing trips, one should eventually purchase skis if the cost of renting skis, r, is positive. In particular, let the cost of purchasing skis be some number $p \geq r$. If one never intends to make a purchase, one's cost for the season will be rn, where n is the number of ski trips in which one participates. If n exceeds p/r, one's cost will exceed the price of purchasing skis; as n continues to increase, the ratio of one's costs to those of one's peers increases to nr/p, which grows unboundedly with n, since your peers, knowing that n exceeds p/r, will have purchased skis prior to the first trip.

On the other hand, if one rushes out to purchase skis upon being told that the ski season is approaching, one's peers will decide that this season looks inopportune, and that skiing is passé, leaving their costs at zero, and one's costs at p, leaving an infinite ratio between one's costs and theirs; if one chooses to defer the purchase until after one's first ski trip, this produces the less unfavorable ratio p/r or $1 + p/r$, depending on whether the invitation left one time to purchase skis before the first trip or not.

Suppose one chooses, instead, to defer one's purchase until after one has made k rentals, but before ski trip $k+1$. One's costs are then bounded by $kr + p$. After k ski trips, the cost to one's peers will be the lesser of kr and p (as one's peers will have decided whether to rent or buy for

the season upon knowing one's plans, which in this case amounts to knowing k), for a ratio equal to the larger of $1 + kr/p$ and $1 + p/kr$. Were they to choose to terminate the activity earlier (so $n < k$), the ratio would be only the greater of kr/p and 1, which is guaranteed to be less than the sum of the two – one's peers would be shirking their opportunity to make one's behavior look foolish were they to allow one to stop skiing prior to one's purchase of a pair of skis!

It is certain, since kr/p and p/kr are reciprocals, that one of them is at least equal to 1, ensuring that one will be compelled to spend at least twice as much as one's peers.

The analysis above applies to the case where ski trips are announced without enough warning to leave one time to buy skis. Purchases in that case are not instantaneous; in contrast, if one is able to purchase skis on demand, the cost to one's peers changes to the lesser of $(k+1)r$ and p. The overall results are not much different; the ratio choices become the larger of $1 + kr/p$ and $1 + (p - r)/((k + 1)r)$.

When probabilistic algorithms are considered with oblivious frenemies (those who know the way in which random choices will affect one's purchasing decisions, but who do not take time to notice that one's skis are no longer marked with the name and phone number of a rental agency), one can appear more thrifty.

A randomized algorithm can be viewed as a distribution over deterministic algorithms. No good algorithm can purchase skis prior to the first invitation, lest it exhibit infinite regrettability (some positive cost compared to zero). A good algorithm must purchase skis by the time one's peers will have; otherwise, one's cost ratio continues to increase with the number of ski trips. Moreover, the ratio should be the same after every ski trip; if not, then there is an earliest ratio not equal to the largest, and probabilities can be adjusted to change this earliest ratio to be closer to the largest while decreasing all larger ratios.

Consider, for example, the case of $p = 2r$, with purchases allowed at the time of an invitation. The best deterministic ratio in this case is 1.5. It is only necessary to choose a probability q, the probability of purchasing at the time of the first invitation. The cost after one trip is then $(1-q)r+2qr = r(1+q)$, for a ratio of $1+q$, and after two trips the cost is $q(2r) + (1 - q)(3r) = 3 - q)r$, producing a ratio of $(3 - q)/2$. Setting these to be equal yields $q = 1/3$, for a ratio of 4/3.

If insufficient time is allowed for purchases before skiing, the best deterministic ratio is 2. Purchasing after the first ski trip with probability q (and after the second with probability $1 - q$) leads to expected costs of $(1-q)r+3qr = r(1+2q)$ after the first trip, and $(1-q)(2+2)r+3qr = r(4 + q)$, leading to a ratio of $2 - q/2$. Setting $1 + 2q = 2 - q/2$ yields $q = 2/5$, for a ratio of 9/5.

More careful analysis, for which readers are referred to the references and the remainder of this volume, shows that the best achievable ratio approaches $\epsilon/(\epsilon - 1) \approx 1.58197$ as p/r increases, approaching the limit from below if sufficient warning time is offered, and from above otherwise.

Applications

The primary initial results were directed towards problems of computer architecture; in particular, design questions for capacity conflicts in caches, and shared memory design in the presence of a shared communication channel. The motivation for these analyses was to find designs which would perform reasonably well on as-yet-unknown workloads, including those to be designed by competitors who may have chosen alternative designs which favor certain cases. While it is probably unrealistic to assume that precisely the least-desirable workloads will occur in ordinary practice, it is not unreasonable to assume that extremal workloads favoring either end of a decision will occur.

History and Further Reading

This technique of finding algorithms with bounded worst-case performance ratios is common in analyzing approximation algorithms.

The initial proof techniques used for such analyses (the method of amortized analysis) were first presented by Sleator and Tarjan.

The reader is advised to consult the remainder of this volume for further extensions and applications of the principles of competitive online algorithms.

Cross-References

- ▸ Algorithm DC-TREE for k-Servers on Trees
- ▸ Metrical Task Systems
- ▸ Online List Update
- ▸ Online Paging and Caching
- ▸ Work-Function Algorithm for k-Servers

Recommended Reading

1. Karlin AR, Manasse MS, Rudolph L, Sleator DD (1988) Competitive snoopy caching. Algorithmica 3:77–119 (Conference version: FOCS 1986, pp 244–254)
2. Karlin AR, Manasse MS, McGeoch LA, Owicki SS (1994) Competitive randomized algorithms for nonuniform problems. Algorithmica 11(6):542–571 (Conference version: SODA 1990, pp 301–309)
3. Reingold N, Westbrook J, Sleator DD (1994) Randomized competitive algorithms for the list update problem. Algorithmica 11(1):15–32 (Conference version included author Irani S: SODA 1991, pp 251–260)

Slicing Floorplan Orientation

Evangeline F.Y. Young
Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China

Keywords

Shape curve computation

Years and Authors of Summarized Original Work

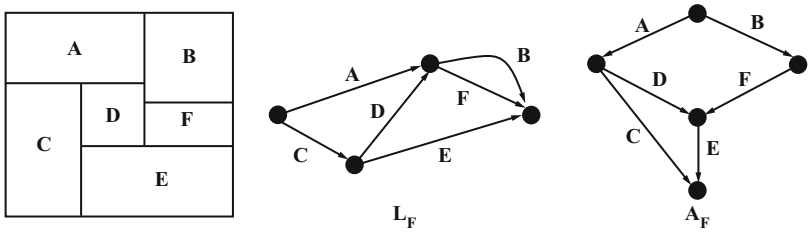
1983; Stockmeyer

Problem Definition

This problem is about finding the optimal orientations of the cells in a slicing floorplan to minimize the total area. In a floorplan, cells represent basic pieces of the circuit which are regarded as indivisible. After performing an initial placement, for example, by repeated application of a min-cut partitioning algorithm, the relative positions between the cells on a chip are fixed. Various optimizations can then be done on this initial layout to optimize different cost measures such as chip area, interconnect length, routability, etc. One such optimization, as mentioned in Lauther [3], Otten [4], and Zibert and Saal [13], is to determine the best orientation of each cell to minimize the total chip area. This work by Stockmeyer [8] gives a polynomial time algorithm to solve the problem optimally in a special type of floorplans called *slicing floorplans* and shows that this orientation optimization problem in general non-slicing floorplans is NP-complete.

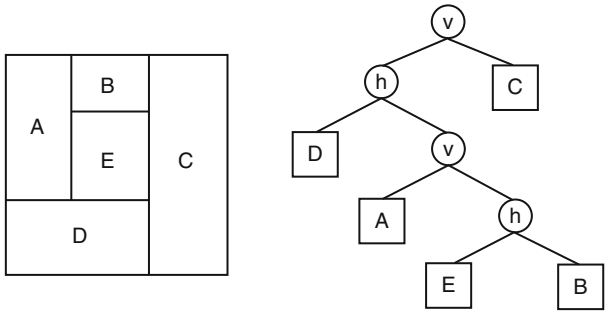
Slicing Floorplan

A floorplan consists of an enclosing rectangle subdivided by horizontal and vertical line segments into a set of non-overlapping *basic rectangles*. Two different line segments can meet but not cross. A floorplan F is characterized by a pair of planar acyclic directed graphs A_F and L_F defined as follows. Each graph has one source and one sink. The graph A_F captures the "above" relationships and has a vertex for each horizontal line segment, including the top and the bottom of the enclosing rectangle. For each basic rectangle R, there is an edge e_r directed from segment σ to segment σ' if and only if σ (or part of σ) is the top of R and σ' (or part of σ') is the bottom of R. There is a one-to-one correspondence between the basic rectangles and the edges in A_F. The graph L_F is defined similarly for the



Slicing Floorplan Orientation, Fig. 1 A floorplan F and its A_F and L_F representing the above and left relationships

Slicing Floorplan Orientation, Fig. 2 A slicing floorplan F and its slicing tree representation



"left" relationships of the vertical segments. An example is shown in Fig. 1. Two floorplans F and G are equivalent if and only if $A_F = A_G$ and $L_F = L_G$. A floorplan F is slicing if and only if both its A_F and L_F are series parallel.

Slicing Tree

A slicing floorplan can also be described naturally by a rooted binary tree called *slicing tree*. In a slicing tree, each internal node is labeled by either an h or a v, indicating a horizontal or a vertical slice respectively. Each leaf corresponds to a basic rectangle. An example is shown in Fig. 2. There can be several slicing trees describing the same slicing floorplan, but this redundancy can be removed by requiring the label of an internal node to differ from that of its right child [12]. For the algorithm presented in this work, a tree of smallest depth should be chosen, and this depth minimization process can be done in $O(n \log n)$ time using the algorithm by Golumbic [2].

Orientation Optimization

In optimization of a floorplan layout, some freedom in moving the line segments and in choosing the dimensions of the rectangles are allowed. In the input, each basic rectangle R has two positive integers a_R and b_R, representing the dimensions of the cell that will be fit into R. Each cell has two possible orientations resulting in either the side of length a_R or b_R being horizontal. Given a floorplan F and an orientation p, each edge e in A_F and L_F is given a label $l(e)$ representing the height or the width of the cell corresponding to e depending on its orientation. Define an (F, ρ)-*placement* to be a labeling l of the vertices in A_F and L_F such that (i) the sources are labeled by zero and (ii) if e is an edge from vertex σ to $\sigma', l(\sigma') \geq l(\sigma) + l(e)$. Intuitively, if σ is a horizontal segment, $l(\sigma)$ is the distance of σ from the top of the enclosing rectangle, and the inequality constraint ensures that the basic rectangle corresponding to e is tall enough for the cell contained in it and similarly for the vertical segments. Now, $h_F(\rho)$ (resp. $w_F(\rho)$) is defined to be the minimum label of the sink in $A_F(\rho)$ (resp. $L_F(\rho)$) over all (F, ρ)-placements, where $A_F(\rho)$ (resp. $L_F(\rho)$) is obtained from A_F (resp. L_F) by labeling the edges and vertices as described above. Intuitively, $h_F(\rho)$ and $w_F(\rho)$ give the minimum height and width of a floorplan F given an orientation ρ of all the cells such that each cell fits well into its associated basic rectangle.

The orientation optimization problem can be defined formally as follows:

Problem 1 (Orientation Optimization Problem for Slicing Floorplan)
INPUT: A slicing floorplan F of n cells described by a slicing tree T, the widths and heights of the cells a_i and b_i for $i = 1...n$, and a cost function $\psi(h, w)$.
OUTPUT: An orientation ρ of all the cells that minimizes the objective function $\psi(h_F(\rho), w_F(\rho))$ over all orientations ρ.

For this problem, Lauther [3] has suggested a greedy heuristic. Zibert and Saal [13] use integer programming methods to do rotation optimization and several other optimization simultaneously for general floorplans. In the following sections, an efficient algorithm will be given to solve the problem optimally in $O(nd)$ time where n is the number of cells and d is the depth of the given slicing tree.

Key Results

In the following algorithm, $F(u)$ denotes the floorplan described by the subtree rooted at u in the given slicing tree T, and let $L(u)$ be the set of leaves in that subtree. For each node u of T, the algorithm constructs recursively a list of pairs:

$$\{(h_1, w_1), (h_2, w_2), \ldots, (h_m, w_m)\}$$

where (1) $m \leq |L(u)| + 1$, (2) $h_i > h_{i+1}$ and $w_i < w_{i+1}$ for $i = 1...m - 1$, (3) there is an orientation ρ of the cells in $L(u)$ such that $(h_i, w_i) = (h_F(u)(\rho), w_F(u)(\rho))$ for each $i = 1...m$, and (4) for each orientation ρ of the cells in $L(u)$, there is a pair (h_i, w_i) in the list such that $h_i \leq h_{F(u)}(\rho)$ and $w_i \leq w_{F(u)}(\rho)$.

$L(u)$ is thus a non-redundant list of all possible dimensions of the floorplan described by the subtree rooted at u. Since the cost function ψ is non-decreasing, it can be minimized over all orientations by finding the minimum $\psi(h_i, w_i)$ over all the pairs (h_i, w_i) in the list constructed at the root of T. At the beginning, a list is constructed at each leaf node of T representing the possible dimensions of the cell. If a leaf cell has dimensions a and b with $a > b$, the list is $\{(a, b), (b, a)\}$. If $a = b$, there will just be one pair (a, b) in the list. (If the cell has a fixed orientation, there will also be just one pair as defined by the fixed orientation.) Notice that the condition (1) above is satisfied in these leaf node lists. The algorithm then works its way up the tree and constructs the list at each node recursively. In general, assume that u is an internal node with children v and v' and u represents a vertical slice. Let $\{(h_1, w_1) \ldots (h_k, w_k)\}$ and $\{(h'_1, w'_1) \ldots (h'_m, w'_m)\}$ be the lists at v and v' respectively where $k \leq |L(v)| + 1$ and $m \leq |L(v')| + 1$. A pair (h_i, wi) from v can be put together by a vertical slice with a pair (h'_j, w'_j) from v' to give a pair:

$$\text{join}((h_i, w_i), (h'_j, w'_j)) = (\max(h_i, h'_j), w_i + w'_j)$$

in the list of u (see Fig. 3). The key fact is that most of the km pairs are sub-optimal and do not need to be considered. For example, if $h_i > h'_j$, there is no need to join (h_i, w_i) with (h'_z, w'_z) for any $z > j$ since

$$\max(h_i, h'_z) = \max(h_i, h'_j) = h_i,$$
$$w_i + w'_z > w_i + w'_j$$

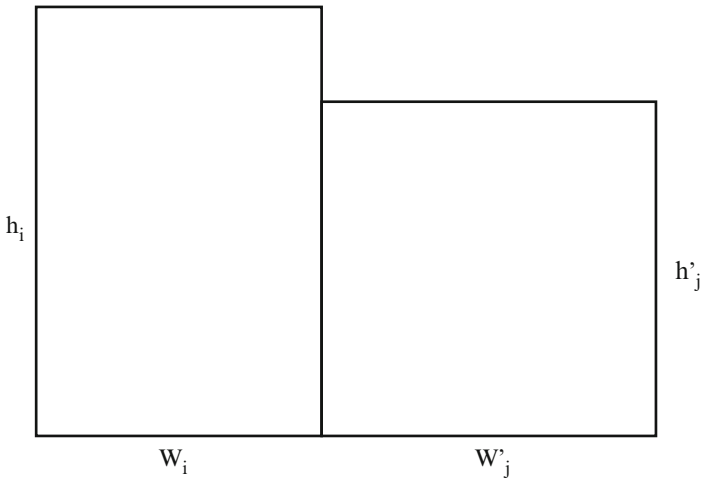
Similarly, if node u represents a horizontal slice, the join operation will be

$$\text{join}((h_i, w_i), (h'_j, w'_j)) = (h_i + h'_j, \max(w_i, w'_j))$$

The algorithm also keeps two pointers for each element in the lists in order to construct back the optimal orientation at the end. The algorithm is summarized by the following pseudocode:

Pseudocode *Stockmeyer()*

1. Initialize the list at each leaf node.
2. Traverse the tree in postorder. At each internal node u with children v and v', construct a list at node u as follows:
3. Let $\{(h_1, w_1) \ldots (h_k, w_k)\}$ and $\{(h'_1, w'_1) \ldots (h'_m, w'_m)\}$ be the lists at v and v' respectively.
4. Initialize i and j to one.



Slicing Floorplan Orientation, Fig. 3 An illustration of the merging step

5. If $i > k$ or $j > m$, the whole list at u is constructed.
6. Add join$((h_i, w_i), (h'_j, w'_j))$ to the list with pointers pointing to (h_i, w_i) and (h'_j, w'_j) in $L(v)$ and $L(v')$ respectively.
7. If $h_i > h'_j$, increment i by 1.
8. If $h_i > h'_j$, increment j by 1.
9. If $h_i > h'_j$, increment both i and j by 1.
10. Go to step 5
11. Compute $\psi(h_i, w_i)$ for each pair (h_i, w_i) in the list L_r at the root r of T.
12. Return the minimum $\psi(h_i, w_i)$ for all (h_i, w_i) in L_r and construct back the optimal orientation by following the pointers.

Correctness

The algorithm is correct since at each node u, a list is constructed that records all the possible non-redundant dimensions of the floorplan described by the subtree rooted at u. This can be proved easily by induction starting from the leaf nodes and working up the tree recursively. Since the cost function ψ is non-decreasing, it can be minimized over all orientations of the cells by finding the minimum $\psi(h_i, w_i)$ over all the pairs (h_i, w_i) in the list L_r constructed at the root r of T.

Runtime

At each internal node u with children v and v'. If the lengths of the lists at v and v' are k and m respectively, the time spent at u to combine the two lists is $O(k + m)$. Each possible dimension of a cell will thus invoke one unit of execution time at each node on its path up to the root in the postorder traversal. The total runtime is thus $O(d \times N)$ where N is the total number of realizations of all the n cells, which is equal to $2n$ in the orientation optimization problem. Therefore, the runtime of this algorithm is $O(nd)$.

Theorem 1 *Let $\psi(h, w)$ be non-decreasing in both arguments, i.e., if $h \leq h'$ and $w \leq w'$, $\psi(h, w) \leq \psi(h', w')$, and computable in constant time. For a slicing floorplan F described by a binary slicing tree T, the problem of minimizing $\psi(h_F(\rho), w_F(\rho))$ over all orientations ρ can be solved in time $O(nd)$ time, where n is the number of leaves of T (equivalently, the number of cells of F) and d is the depth of T.*

Applications

Floorplan design is an important step in the physical design of VLSI circuits. Stockmeyer's optimal orientation algorithm [8] has been generalized to solve the area minimization problem in slicing floorplans [7], in hierarchical non-slicing floorplans of order five [6,9], and in general floorplans [5]. The floorplan area minimization problem is similar except that each *soft cell* now has a number of possible realizations, instead of just two different orientations. The same technique

can be applied immediately to solve optimally the area minimization problem for slicing floorplans in $O(nd)$ time where n is the total number of realizations of all the cells in a given floorplan F and d is the depth of the slicing tree of F. Shi [7] has further improved this result to $O(n \log n)$ time. This is done by storing the list of non-redundant pairs at each node in a balanced binary search tree structure called *realization tree* and using a new merging algorithm to combine two such trees to create a new one. It is also proved in [7] that this $O(n \log n)$ time complexity is the lower bound for this area minimization problem in slicing floorplans.

For hierarchical non-slicing floorplans, Pan et al. [6] prove that the problem is NP-complete. Branch-and-bound algorithms are developed by Wang and Wong [9], and pseudopolynomial time algorithms are developed by Wang and Wong [10] and Pan et al. [6]. For general floorplans, Stockmeyer [8] has shown that the problem is strongly NP-complete. It is therefore unlikely to have any pseudopolynomial time algorithm. Wimer et al. [11] and Chong and Sahni [1] propose branch-and-bound algorithms. Pan et al. [5] develop algorithms for general floorplans that are approximately slicing.

Recommended Reading

1. Chong K, Sahni S (1993) Optimal realizations of floorplans. IEEE Trans Comput Aided Des 12(6):793–901
2. Golumbic MC (1976) Combinatorial merging. IEEE Trans Comput C-25:1164–1167
3. Lauther U (1980) A Min-cut placement algorithm for general cell assemblies based on a graph representation. J Digit Syst 4:21–34
4. Otten RHJM (1982) Automatic floorplan design. In: Proceedings of the 19th design automation conference, Las Vegas, pp 261–267
5. Pan P, Liu CL (1995) Area minimization for floorplans. IEEE Trans Comput Aided Des 14(1):123–132
6. Pan P, Shi W, Liu CL (1996) Area minimization for hierarchical floorplans. Algorithmica 15(6):550–571
7. Shi W (1996) A fast algorithm for area minimization of slicing floorplan. IEEE Trans Comput Aided Des 15(12):1525–1532
8. Stockmeyer L (1983) Optimal orientations of cells in slicing floorplan designs. Infect Control 59:91–101
9. Wang TC, Wong DF (1992) Optimal floorplan area optimization. IEEE Trans Comput Aided Des 11(8):992–1002
10. Wang TC, Wong DF (1993) A note on the complexity of Stockmeyer's floorplan optimization technique. In: Algorithmic aspects of VLSI layout. Lecture notes series on computing, vol 2. World Scientific, Singapore, pp 309–320
11. Wimer S, Koren I, Cederbaum I (1989) Optimal aspect ratios of building blocks in VLSI. IEEE Trans Comput Aided Des 8(2):139–145
12. Wong DF, Liu CL (1986) A new algorithm for floorplan design. In: Proceedings of the 23rd ACM/IEEE design automation conference, Las Vegas, pp 101–107
13. Zibert K, Saal R (1974) On computer aided hybrid circuit layout. In: Proceedings of the IEEE international symposium on circuits and systems, San Francisco, pp 314–318

Sliding Window Algorithms

Vladimir Braverman
Department of Computer Science, Johns Hopkins University, Baltimore, MD, USA

Keywords

Data streams; Histograms; Randomized algorithms; Sampling; Sketching

Years and Authors of Summarized Original Work

2007; Braverman, Ostrovsky

Problem Definition

In the last decade, the theoretical study of the sliding window model was developed to advance applications with very large input and time-sensitive output. In some practical situations, input might be seen as an ordered sequence, and it is useful to restrict computations to recent portions of the input. Examples include the analysis of recent tweets and time series of the stock market.

To address the aforementioned practical situations, Datar et al. [20] introduced the sliding window model that assumes that the input is a stream (i.e., the ordered sequence) of data elements and divides the data elements into two categories: *active* elements and *expired* elements. Typically, a recent portion (i.e., a suffix) of the stream defines the window of active elements, and the reminder (i.e., a complimenting prefix) of the stream defines the set of expired elements. When a new data element arrives, the set of active elements expands to include the new element, but the set might also shrink by discarding some portion of oldest active elements. This process of additions and expirations reminds one of the movements of an interval (or a window) along a line and explains the name of the model. The number of active elements N is often called a *size of the sliding window*. There are two popular variants of the sliding window model. The variant of a *sequence-based* window fixes the number of active elements N, and every insertion (or arrival) of a new element corresponds to a deletion (or expiration) of the oldest active element (after the size of the stream becomes larger than N). For example, a sequence-based window on a stream of IP packets is a set of last N packets. The variant of a *timestamp-based* window associates each element with a nondecreasing timestamp, and the window contains all elements with timestamps larger than a certain value. Thus, there is no obvious dependence between the number of elements that arrive and expire. In the previous example, the timestamp-based window might be defined as a set of all packets that arrived within the last t seconds.

Formal Definition

We denote the stream D by a sequence of elements $\{p_i\}_{i=1}^{m}$ where $p_i \in [n]$. It is important to note that m is incremented for each new arrival. A *bucket* $B(x, y) = \{p_i, i \in [x, y]\}$ is the set of all stream elements between p_x and p_y, inclusively. A sequence-based window is defined $W = B(m - N + 1, m)$ where N is a predefined parameter. Consider a nondecreasing *timestamp* function $T : [m] \to R$ and let t be a parameter. Given T and t, a timestamp-based window is defined as $W = B(l(t), m)$ where $l(t) = \min\{i : T(i) \geq T(m) - t\}$. Consider function f that is defined on buckets. An algorithm maintains a $(1 \pm \epsilon)$-approximation of f on W if, at any moment, the algorithm outputs X s.t. $|f(W) - X| \leq \epsilon f(W)$. Similarly, a randomized algorithm maintains a $(1 \pm \epsilon, \delta)$-approximation if $P(|f(W) - X| > \epsilon f(W)) \leq \delta$. It is often the case that f can be computed precisely if the entire window is available, but sublinear-space approximations, i.e., computation when the size of the available memory is $o(N + n)$, might be challenging. For example, Datar et al. [20] show linear space is required to maintain a $(1 \pm \epsilon, \delta)$-approximation of a sum of active elements if $p_i \in \{1, 0, -1\}$. A typical question in the sliding window model is the following: given function f, what are the upper and lower bounds on the space complexity of maintaining $(1 \pm \epsilon, \delta)$-approximation of f.

History

In their pioneering papers, Datar et al. [20, 21] and Babcock et al. [3] gave the first formal definition of the sliding window model. The model arose in the context of relational databases as a special case of time-sensitive queries in temporal databases [3]. Below we give a short survey of a subset of known results. A survey of Datar and Motwani [1] provides additional details. Datar et al. [20] gave the first algorithms for estimating the count and sum of positive integers, average, L_p for $p \in [1, 2]$, and a wide class of *weakly additive functions*. Gibbons and Tirthapura [24] provided further improvements to count and sum and gave the first methods for distributed computations. Lee and Ting [29] provided an optimal solution for a relaxed version of the counting problem, where the correct answer is provided only if it is comparable with the window's size. Braverman and Ostrovsky [6, 7] extended the results in [20] to a wider class of *smooth functions*. Chi et al. [15] considered a problem of frequent itemsets. Arasu and Manku [2], Lee and Ting [30], and Golab et al. [26] considered the problem of finding frequent elements, frequency counts, and quantiles. Babcock, Datar, Motwani, and O'Callaghan [5] pro-

vided first algorithms for variance and k-medians problems. Feigenbaum, Kannan and Zhang [22] presented an efficient solution for the diameter of a data set in multidimensional space. Later, Chan and Sadjad [23] presented optimal solutions for this and other geometric problems. Babcock, Datar and Motwani [4] presented algorithms for uniform random sampling from sliding windows.

Recently, Crouch et al. [17] presented the first approximation algorithms for important graph problems such as combinatorial sparsifiers and spanners, graph matching, and minimum spanning tree. Among other results, the methods in [17] allow non-smooth statistics using a modified smooth histogram to be computed. McGregor provided a detailed survey of these and other graph algorithms [32]. Datar and Muthukrishnan [19] solved problems of rarity and similarity. Braverman et al. [11] gave improved algorithms for rarity, similarity, and L_2-heavy hitters. Cormode and Yi developed several first algorithms for sliding windows in distributed streams [16]. Babcock et al. [4] gave the first method of sampling an element with constant expected space complexity. Braverman et al. [9, 10] gave a solution with a space complexity that is a constant in the worst case. Tatbul and Zdonik [35] considered the problem of load shedding for aggregation queries. Golab and Özsu [25] gave the first algorithm for approximating multi-joins. Recently, Braverman et al. [13] extended the zero-one law for increasing frequency-based functions [8] to sliding windows.

Key Results

Smooth Histogram

Extending the results in [20], Braverman and Ostrovsky [6, 7] introduced a notion of a *smooth function* and presented techniques for approximating smooth functions over sliding windows. Denote by $B \subseteq_r A$ the event when bucket B is a suffix of A; i.e., if $A = \{p_{n_1}, \ldots, p_{n_2}\}$ (for some $n_1 < n_2$), then $B = \{p_{n_3}, \ldots, p_{n_2}\}$, where $n_1 \leq n_3 \leq n_2$. Denote by $A \cup C$ the union of adjacent buckets A and C.

Definition 1 Function f is (α, β)-*smooth* if it preserves the following properties:

1. $f(A) \geq 0$.
2. $f(A) \geq f(B)$ for $B \subseteq_r A$.
3. $f(A) \leq poly(|A|)$.
4. For any $0 < \epsilon < 1$, there exist $\alpha = \alpha(\epsilon, f)$ and $\beta = \beta(\epsilon, f)$ such that
   - $0 < \beta \leq \alpha < 1$.
   - If $B \subseteq_r A$ and $(1-\beta) f(A) \leq f(B)$, then $(1-\alpha) f(A \cup C) \leq f(B \cup C)$ for any adjacent C.

In other words, a nonnegative, nondecreasing, and polynomially bounded function f is (α, β)-smooth if the following is true. If $f(B)$ is a $(1 \pm \beta)$-approximation of $f(A)$, then $f(B \cup C)$ is $(1 \pm \alpha)$-approximation of $f(A \cup C)$ for any $B \subseteq_r A$ and C. The main technical result of [7] is a new data structure called "smooth histogram" that allows algorithms for insertion-only streams to be extended to sliding windows with space complexity increased by a polylogarithmic factor. If there exists an algorithm that computes f precisely using g space and h time per element, then a smooth histogram can be used to maintain a $(1 \pm \alpha)$-approximation of f over sliding windows, using $O\left(\frac{1}{\beta} \log n (g + \log n)\right)$ bits and $O\left(\frac{1}{\beta} h \log n\right)$ time. Further, $(1 \pm \rho)$-approximation of f on D results in $(1 \pm (\alpha + \rho))$-approximation of f over sliding windows. Examples of smooth functions include sum, count, min, diameter, weakly additive functions, L_p norms, frequency moments, length of longest subsequence, and geometric mean.

Let f be (α, β)-smooth for which there exists an algorithm Λ that calculates f on D using g space and h operation per element. To maintain f on sliding windows, we construct a data structure that we call *smooth histogram*. It consists of a set of indexes $x_1 < x_2 < \cdots < x_s = N$ and instances of Λ for each bucket $B(x_i, N)$. Informally, the smooth histogram ensures the following properties of the sequence. The first two elements of the sequence always

"sandwich" the window, i.e., $x_1 \leq N - n < x_2$. This requirement and the monotonicity of f give us useful bounds for the sliding window W: $f(x_2, N) \leq f(W) \leq f(x_1, N)$. Also, f should slowly but constantly decrease with i, i.e., $f(x_{i+2}, N) < (1 - \beta) f(x_i, N)$. This gradual decrease, together with the fact that f is polynomially bounded, ensures that the sequence is short, i.e., $s = O\left(\frac{1}{\beta} \log n\right)$. Finally, the values of f on successive buckets were close in the past, i.e., $f(x_{i+1}, N') \geq (1 - \beta) f(x_i, N')$ for some $N' \leq N$. This represents our key idea and exploits the properties of smoothness. Indeed, $f(x_2, N') \geq (1-\beta) f(x_1, N')$ for some $N' \leq N$; thus, by the (α, β)-smoothness of f, we have $f(x_2, N) \geq (1 - \alpha) f(x_1, N) \geq (1 - \alpha) f(W)$. We refer a reader to [7] for further technical details.

Applications

There are several applications of the theoretical methods for the sliding window model, for example, [15, 18, 31, 33, 36].

Open Problems

We list several interesting open problems. It would be important to understand the difference between the sliding window model and other streaming models such as the insertion-only model, the turnstile, and decay models. This is perhaps one of the most important unresolved open problems; see, e.g., Sohler [34]. In particular, it would be nice to understand the exact space complexity of the frequency moments that are well understood in the other streaming models [12, 27, 28]. Also, it would be interesting to extend the coreset methods [14] to sliding windows, obtain polylogarithmic solutions for clustering, and improve the first clustering algorithm in [5]. Also, it would be nice to further develop graph methods [17]. Improving the approximation ratio of the maximum matching and obtaining the $O(n^{1+1/t})$ space bound for $(2t - 1)$-spanners are important open problems.

Acknowledgments This material is based upon work supported in part by the National Science Foundation under Grant No. 1447639.

Recommended Reading

1. Aggarwal C (2007) Data streams: models and algorithms. Advances in database systems. http://www.springer.com/west/home/default?SGWID=4-40356-22-107949228-0
2. Arasu A, Manku GS (2004) Approximate counts and quantiles over sliding windows. In: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems (PODS'04). ACM, New York, pp 286–296. doi:10.1145/1055558.1055598
3. Babcock B, Babu S, Datar M, Motwani R, Widom J (2002) Models and issues in data stream systems. In: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems (PODS'02). ACM, New York, pp 1–16. doi:10.1145/543613.543615
4. Babcock B, Datar M, Motwani R (2002) Sampling from a moving window over streaming data. In: Proceedings of the thirteenth annual ACM-SIAM symposium on discrete algorithms (SODA'02). Society for Industrial and Applied Mathematics, Philadelphia, pp 633–634. http://dl.acm.org/citation.cfm?id=545381.545465
5. Babcock B, Datar M, Motwani R, O'Callaghan L (2003) Maintaining variance and k-medians over data stream windows. In: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'03). ACM, New York, pp 234–243. doi:10.1145/773153.773176
6. Braverman V, Ostrovsky R (2007) Smooth histograms for sliding windows. In: Proceedings of the 48th annual IEEE symposium on foundations of computer science (FOCS'07). IEEE Computer Society, Washington, DC, pp 283–293. doi:10.1109/FOCS.2007.63
7. Braverman V, Ostrovsky R (2010) Effective computations on sliding windows. SIAM J Comput 39(6):2113–2131. doi:10.1137/090749281
8. Braverman V, Ostrovsky R (2010) Zero-one frequency laws. In: Proceedings of the 42nd ACM symposium on theory of computing (STOC'10). ACM, New York, pp 281–290. doi:10.1145/1806689.1806729
9. Braverman V, Ostrovsky R, Zaniolo C (2009) Optimal sampling from sliding windows. In: Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-

SIGART symposium on principles of database systems (PODS'09). ACM, New York, pp 147–156. doi:10.1145/1559795.1559818
10. Braverman V, Ostrovsky R, Zaniolo C (2012) Optimal sampling from sliding windows. J Comput Syst Sci 78(1):260–272. doi:10.1016/j.jcss.2011.04.004
11. Braverman V, Gelles R, Ostrovsky R (2013) How to catch l 2-heavy-hitters on sliding windows. In: Du DZ, Zhang G (eds) Computing and combinatorics. Lecture notes in computer science, vol 7936. Springer, Berlin/Heidelberg, pp 638–650. doi:10.1007/978-3-642-38768-5_56
12. Braverman V, Katzman J, Seidell C, Vorsanger G (2014) An optimal algorithm for large frequency moments using $O(n^{1-2/k})$ bits. In: Proceedings of the 18th international workshop on randomization and computation (RANDOM'2014)
13. Braverman V, Ostrovsky R, Roytman A (2014) Universal Streaming. ArXiv e-prints 1408.2604
14. Chen K (2009) On coresets for k-median and k-means clustering in metric and euclidean spaces and their applications. SIAM J Comput 39(3):923–947. doi:http://dx.doi.org/10.1137/070699007
15. Chi Y, Wang H, Yu PS, Muntz RR (2004) Moment: Maintaining closed frequent itemsets over a stream sliding window. In: ICDM, pp 59–66
16. Cormode G, Yi K (2012) Tracking distributed aggregates over time-based sliding windows. In: Proceedings of the 24th international conference on scientific and statistical database management (SSDBM'12). Springer, Berlin/Heidelberg, pp 416–430. doi:10.1007/978-3-642-31235-9_28
17. Crouch MS, McGregor A, Stubbs D (2013) Dynamic graphs in the sliding-window model. In: ESA, pp 337–348
18. Dang XH, Lee VC, Ng WK, Ong KL (2009) Incremental and adaptive clustering stream data over sliding window. In: Proceedings of the 20th international conference on database and expert systems applications (DEXA'09). Springer, Berlin/Heidelberg, pp 660–674. doi:10.1007/978-3-642-03573-9_55
19. Datar M, Muthukrishnan MS (2002) Estimating rarity and similarity over data stream windows. In: Proceedings of the 10th annual European symposium on algorithms (ESA'02). Springer, London, pp 323–334. http://dl.acm.org/citation.cfm?id=647912.740833
20. Datar M, Gionis A, Indyk P, Motwani R (2002) Maintaining stream statistics over sliding windows. SIAM J Comput 31(6):1794–1813
21. Datar M, Gionis A, Indyk P, Motwani R (2002) Maintaining stream statistics over sliding windows: (extended abstract). In: Proceedings of the thirteenth annual ACM-SIAM symposium on discrete algorithms (SODA'02). Society for Industrial and Applied Mathematics, Philadelphia, pp 635–644. http://dl.acm.org/citation.cfm?id=545381.545466
22. Feigenbaum J, Kannan S, Zhang J (2004) Computing diameter in the streaming and sliding-window models. Algorithmica 41(1):25–41
23. Feigenbaum J, Kannan S, Zhang J (2005) Computing diameter in the streaming and sliding-window models. Algorithmica 41:25–41
24. Gibbons PB, Tirthapura S (2002) Distributed streams algorithms for sliding windows. In: Proceedings of the fourteenth annual ACM symposium on parallel algorithms and architectures (SPAA'02). ACM, New York, pp 63–72. doi:10.1145/564870.564880
25. Golab L, Özsu MT (2003) Processing sliding window multi-joins in continuous queries over data streams. In: Proceedings of the 29th international conference on Very large data bases (VLDB'03), vol 29. VLDB Endowment, pp 500–511. http://dl.acm.org/citation.cfm?id=1315451.1315495
26. Golab L, DeHaan D, Demaine ED, Lopez-Ortiz A, Munro JI (2003) Identifying frequent items in sliding windows over on-line packet streams. In: Proceedings of the 3rd ACM SIGCOMM conference on internet measurement (IMC'03). ACM, New York, pp 173–178. doi:10.1145/948205.948227
27. Kane DM, Nelson J, Woodruff DP (2010) On the exact space complexity of sketching and streaming small norms. In: Proceedings of the 21st annual ACM-SIAM symposium on discrete algorithms (SODA'10)
28. Kane DM, Nelson J, Woodruff DP (2010) An optimal algorithm for the distinct elements problem. In: Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'10). ACM, New York, pp 41–52. doi:10.1145/1807085.1807094
29. Lee LK, Ting HF (2006) Maintaining significant stream statistics over sliding windows. In: SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm. ACM, New York, pp 724–732. doi:http://doi.acm.org/10.1145/1109557.1109636
30. Lee LK, Ting HF (2006) A simpler and more efficient deterministic scheme for finding frequent items over sliding windows. In: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'06). ACM, New York, pp 290–297. doi:http://doi.acm.org/10.1145/1142351.1142393
31. Li J, Maier D, Tufte K, Papadimos V, Tucker PA (2005) No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD Rec 34(1):39–44. doi:10.1145/1058150.1058158
32. McGregor A (2014) Graph stream algorithms: A survey. SIGMOD Rec 43(1):9–20. doi:10.1145/2627692.2627694
33. Ren J, Ma R, Ren J (2009) Density-based data streams clustering over sliding windows. In: Proceedings of the 6th international conference on Fuzzy systems and knowledge discovery (FSKD'09), vol 5. IEEE Press, Piscataway, pp 248–252. http://dl.acm.org/citation.cfm?id=1801874.1801929
34. Sohler C (2006) List of open problems in sublinear algorithms: problem **20**. http://sublinear.info/20

35. Tatbul N, Zdonik S (2006) Window-aware load shedding for aggregation queries over data streams. In: Proceedings of the 32nd international conference on very large data bases (VLDB'06). VLDB Endowment, pp 799–810. http://dl.acm.org/citation.cfm?id=1182635.1164196
36. Zhang L, Li Z, Yu M, Wang Y, Jiang Y (2005) Random sampling algorithms for sliding windows over data streams. In: Proceedings of the 11th joint international computer conference, pp 572–575

Smooth Surface and Volume Meshing

Tamal Krishna Dey
Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA

Keywords

Delaunay mesh; Delaunay refinement; Surface mesh; Topology; Volume mesh

Years and Authors of Summarized Original Work

2001; Cheng, Dey, Edelsbrunner, Sullivan
2003; Boissonnat, Oudot
2004; Cheng, Dey, Ramos
2005; Oudot, Rienau, Yvinec
2012; Cheng, Dey, Shewchuk

Problem Definition

Given a smooth surface $S \subset \mathbb{R}^3$, we are required to compute a set of points $P \subset S$ and connect them with edges and triangles so that the resulted triangulation T is *geometrically* close and is *topologically* equivalent to S.

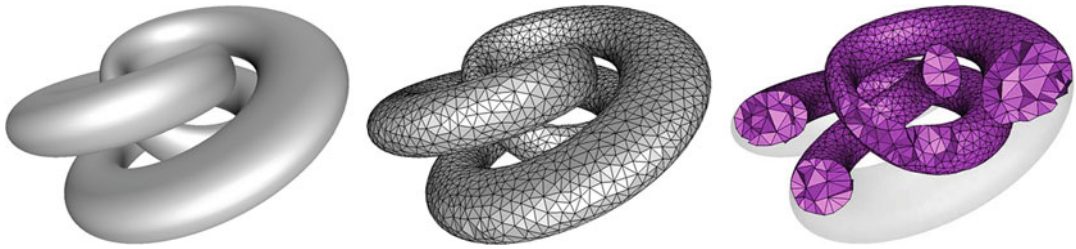
The output triangulation T is a simplicial 2-complex whose vertices are the points in P. Its underlying space, which is the pointwise union of the simplices (vertices, edges, triangles), is denoted with $|T|$. Geometric proximity is often characterized by Hausdorff distance between S and the underlying space $|T|$ of T. It is also desired that the triangle normals in T closely approximate the surface normals at its vertices. Topological equivalence is characterized by the existence of a *homeomorphism* between S and $|T|$. In some cases, the topological guarantee can be given in terms of *isotopy* which is stronger than homeomorphism. It is important to notice that, unlike polyhedral surfaces, a smooth surface cannot be represented *exactly* and hence needs to be approximated with a finite triangulation. This approximation requires that the mesh generation algorithms guarantee topological fidelity in addition to the geometric proximity.

In volume mesh generation, the space bounded by a smooth surface S is required to be tessellated with tetrahedra which form a simplicial 3-complex T. Similar to the surface case, it is required that the underlying space $|T|$ is geometrically close and topologically equivalent to the space bounded by S. It turns out that if the underlying space of the boundary 2-complex of T is geometrically close and has an isotopy to S, then so is $|T|$.

In both surface and volume meshes, it is desirable that the triangles and tetrahedra have good aspect ratio. This is often achieved by bounding the circumradius to shortest edge length ratios for triangles. Unfortunately, for tetrahedra, a bounded radius-edge ratio does not necessarily imply a bounded aspect ratio though most poor quality tetrahedra except slivers [4] are eliminated by bounded radius-edge ratio. Figure 1 shows an example of a surface and a volume mesh.

Key Results

Theoretically sound algorithms for surface meshing use the technique of Delaunay refinement originally proposed by Chew [8]. For a point set $P \subset \mathbb{R}^3$, let Vor P and Del P denote the Voronoi diagram and Delaunay triangulation of P, respectively. A typical Delaunay refinement algorithm iteratively samples the space to be meshed with a *locally furthest point* strategy that inserts points where a Voronoi face of appropriate dimension intersects the space. The decision of



Smooth Surface and Volume Meshing, Fig. 1 A knotted torus, its surface mesh, and its volume mesh

which points to be inserted is guided by certain desirable properties of the output such as topological equivalence, simplex radius-edge ratios, geometric proximity, and so on.

In both surface and volume meshing, the features of the surface S play an important role because regions of small features need to be sampled relatively densely to capture the geometry and topology of S. The definition of *local feature size* and ε-sample given by Amenta, Bern, and Eppstein [2] captures this idea.

Let S be a smooth, closed surface, that is, S is compact, C^2-smooth, and has no boundary. The *medial axis* $M(S)$ of S is defined as the closure of the set of points $x \in \mathbb{R}^3$ so that the distance $d(x, S)$ is realized by two or more points in S. The *local feature size* is defined as

$$f(x) = d(x, M).$$

A set of points $P \subset S$ is called an ε-sample of S if every point $x \in S$ has a sample point in P within $\varepsilon f(x)$ distance.

It turns out that if P is an ε-sample of S for a sufficiently small value of ε, a subcomplex of the Delaunay triangulation of this sample captures the topology of S. We define this subcomplex in generality and then specialize it to S.

Let V_ξ denote the dual Voronoi face of a Delaunay simplex ξ in Del P. The restricted Voronoi face of V_ξ with respect to $\mathbb{X} \subset \mathbb{R}^3$ is the intersection $V_\xi|_{\mathbb{X}} = V_\xi \cap \mathbb{X}$. The *restricted Voronoi diagram* and *restricted Delaunay triangulation* of P with respect to $\mathbb{X}$ are

$$\begin{aligned}\text{Vor } P|_{\mathbb{X}} &= \{V_\xi|_{\mathbb{X}} \mid V_\xi|_{\mathbb{X}} \neq \emptyset\} \text{ and Del } P|_{\mathbb{X}} \\ &= \{\xi \mid V_\xi|_{\mathbb{X}} \neq \emptyset\} \text{ respectively.}\end{aligned}$$

In words, Del $P|_{\mathbb{X}}$ consists of those Delaunay simplices in Del P whose dual Voronoi face intersects $\mathbb{X}$. We call these simplices *restricted*.

Now consider a sample P on the surface S. The restricted Delaunay triangulation of P with respect to S is Del $P|_S$. It is known that if P is an ε-sample of S for $\varepsilon \leq 0.09$, then Del $P|_S$ has its underlying space homeomorphic to S [1, 9]. To use this result one requires computing an ε-sample of S. A computation of local feature size or its approximation is necessary to determine if a sample is an ε-sample for a predetermined ε. Even if one is allowed to assume the availability of the local feature size at any given point, it is not immediately obvious how to place points on S so that they become ε-sample for a given $\varepsilon > 0$.

Surface Meshing

The following theorem about the fidelity of the restricted Delaunay triangulation of a dense sample on a smooth closed surface is the basis of provable surface meshing algorithms. It has been proved in various versions in [1, 5, 7, 9].

Theorem 1 *Let P be an ε-sample of a smooth, compact, boundary-less surface $S \subset \mathbb{R}^3$. The restricted Delaunay complex $T =$ Del $P|_S$ satisfies the following properties for $\varepsilon \leq 0.09$:*

1. *The underlying space $|T|$ is homeomorphic to S (actually, there is an ambient isotopy taking $|T|$ to S).*
2. *Every point in $|T|$ has a point $x \in S$ so that $d(p, x) \leq O(\varepsilon) f(x)$. Similarly, every point x in S has a point p in $|T|$ so that $d(p, x) \leq O(\varepsilon) f(x)$.*

3. *Each triangle $t \in T$ has a normal making an angle $O(\varepsilon)$ with the normal to the surface S at any of its vertices.*

Cheng, Dey, Edelsbrunner, and Sullivan [5] applied Chew's furthest point placement strategy [8] to maintain a dynamic surface mesh of a special type of surface called *skin surface* for which they computed the local feature size explicitly. The above theorem then allowed them to argue the geometric and topological fidelity of the output. Boissonnat and Oudot [3] used similar point placement strategy assuming that the local feature sizes are available, but they suggested how to initialize the meshing procedure for general surfaces. For a restricted triangle $t \in \text{Del } P|_S$, the dual Voronoi edge intersects S possibly at multiple points. Each ball centering such an intersection point and circumscribing vertices of t is called a *surface Delaunay ball* of t. Boissonnat and Oudot observed that if every surface Delaunay ball of each restricted triangle has small radius, say at most 0.05 times the local feature size at the center, then P a 0.09-sample of S. It follows that $\text{Del } P|_S$ at this point satisfies the properties stated in Theorem 1. The deduction of this conclusion also requires that every component of S has at least one Voronoi edge intersecting it which Boissonnat and Oudot ensure with *persistent triangles*.

When local feature sizes are not known, we cannot use the method of Boissonnat and Oudot [3]. Instead, we fall back upon a different strategy to drive the Delaunay refinement. A result of Edelsbrunner and Shah [10] says that if Voronoi faces intersect S in a closed topological ball of appropriate dimension, then the underlying space of the restricted Delaunay triangulation becomes homeomorphic to S. In fact, this is the basis of the proof of Theorem 1. Therefore, a Delaunay refinement driven by the violation of the topological ball conditions provides a viable strategy for meshing with topological guarantees. This strategy is followed by Cheng, Dey, Ramos, and Ray [6].

The algorithm of Cheng et al. avoids computing local feature sizes or their approximation; however, it needs to compute critical points of certain functions on the surface, which may not be easily computable. In a recent book on Delaunay mesh generation [7], Cheng, Dey, and Shewchuk have suggested a strategy that is more practical which leverages on both algorithms of Boissonnat and Oudot [3] and Cheng et al. [6]. It operates with an input parameter $\lambda > 0$. As long as the surface Delaunay balls of the restricted triangles are not all smaller than a ball of radius λ, the algorithm refines. It also refines if the restricted triangles around each vertex do not form a topological disk. The algorithm can be shown to terminate and has the following guarantees.

Theorem 2 ([7]) *There is a Delaunay refinement algorithm that runs with a parameter $\lambda > 0$ on an input smooth, compact, boundary-less surface S with the following guarantees:*

1. *The output mesh is a Delaunay subcomplex and is a 2-manifold for all values of λ.*
2. *If λ is sufficiently small, then the output mesh has similar guarantees with respect to the input surface S as in Theorem 1 (replace ε with λ).*

It should be noted that in any of the above algorithms, one may introduce the condition that the output triangles have radius-edge ratio of at most 1 without loosing any of the geometric or topological guarantees. Even a graded mesh can be guaranteed by supplying an appropriate grading function as input. For details see [7].

Volume Meshing

Let $\mathcal{O}$ denote the volume enclosed by a smooth surface S. Consider the surface mesh of S produced by one of the algorithms mentioned above. The volume enclosed by this surface mesh is already triangulated with Delaunay tetrahedra. We can further refine them for quality using the radius-edge ratio condition. The circumcenters of skinny tetrahedra can be added as long as they do not disturb the surface triangulation. One easy approach is to skip adding those circumcenters who encroach the surface Delaunay balls meaning that they lie inside these balls. This ensures that all

surface triangles remain intact. The trade-off of this easy fix is that the tetrahedra near the boundary may not have bounded radius-edge ratios. To ensure the quality for all tetrahedra, additional effort is required to maintain the surface. Oudot, Rineau, and Yvinec [11] proposed an algorithm for guaranteed quality volume meshing.

The algorithm first runs the algorithm of [3] to obtain a surface triangulation with a vertex set P on the surface. It uses two parameters ε and ρ where ε controls the level of refinement and ρ controls the aspect ratios of the tetrahedra and triangles. It ensures that all restricted triangles on the surface have vertices from S. It refines surface triangles as in surface meshing algorithm. Then, it refines the tetrahedra. Refinement of surface triangles is given priority over the tetrahedra. Oudot et al. [11] prove that their algorithm terminates and has the following geometric and topological guarantees.

Theorem 3 ([11]) *Given a volume $\mathcal{O}$ bounded by a smooth surface S, for $\varepsilon \leq 0.05$ and $\rho > 1$, there is an algorithm that produces $T = \text{Del}\, P|_{\mathcal{O}}$ where each tetrahedron in T has radius-edge ratio at most ρ and $|T|$ is homeomorphic (isotopic) to $\mathcal{O}$ and the boundary of T is $\text{Del}\, P|_S$. Furthermore, the isotopy moves a point $x \in S$ by at most $O(\varepsilon^2) f(x)$ distance.*

An improved version of the algorithm and its analysis in presented in the book [7].

URLs to Code and Data Sets

CGAL(http://cgal.org), a library of geometric algorithms, contains software for surface and volume mesh generation. The DelPSC software that implements the surface and volume meshing algorithms as described in [7] is also available from http://web.cse.ohio-state.edu/~tamaldey/delpsc.html.

Cross-References

- Manifold Reconstruction
- Meshing Piecewise Smooth Complexes
- Surface Reconstruction

Recommended Reading

1. Amenta N, Bern M (1999) Surface reconstruction by Voronoi filtering. Discret Comput Geom 22:481–504
2. Amenta N, Bern M, Eppstein D (1998) The crust and the beta-skeleton: combinatorial curve reconstruction. Graph Models Image Process 60(2:2):125–135
3. Boissonnat J-D, Oudot S (2005) Provably good surface sampling and meshing of surfaces. Graph Models 67:405–451. Conference version 2003
4. Cheng S-W, Dey TK, Edelsbrunner H, Teng SH (2000) Sliver exudation. J ACM 47:883–904
5. Cheng H-L, Dey TK, Edelsbrunner H, Sullivan J (2001) Dynamic skin triangulation. Discret Comput Geom 25:525–568
6. Cheng S-W, Dey TK, Ramos EA, Ray T (2007) Sampling and meshing a surface with guaranteed topology and geometry. SIAM J Comput 37:1199–1227. Conference version 2004
7. Cheng S-W, Dey TK, Shewchuk JR (2012) Delaunay mesh generation. CRC Press, Boca Raton
8. Chew LP (1993) Guaranteed-quality mesh generation for curved surfaces. In: Proceedings of the 9th annual symposium on computational geometry, San Diego, pp 274–280
9. Dey TK (2006) Curve and surface reconstruction: algorithms with mathematical analysis. Cambridge University Press, New York
10. Edelsbrunner H, Shah N (1997) Triangulating topological spaces. Int J Comput Geom Appl 7:365–378
11. Oudot S, Rineau L, Yvinec M (2005) Meshing volumes bounded by smooth surfaces. In: Proceedings of the 14th international meshing roundtable, pp 203–219

Smoothed Analysis

Heiko Röglin
Department of Computer Science, University of Bonn, Bonn, Germany

Keywords

Computational complexity; Linear programming; Probabilistic analysis

Years and Authors of Summarized Original Work

2001; Spielman, Teng
2004; Beier, Vöcking

Problem Definition

Smoothed analysis has originally been introduced by Spielman and Teng [22] in 2001 to explain why the simplex method is usually fast in practice despite its exponential worst-case running time. Since then it has been applied to a wide range of algorithms and optimization problem. In smoothed analysis, inputs are generated in two steps: first, an adversary chooses an arbitrary instance, and then this instance is slightly perturbed at random. The smoothed performance of an algorithm is defined to be the worst expected performance the adversary can achieve. This model can be viewed as a less pessimistic worst-case analysis, in which the randomness rules out pathological worst-case instances that are rarely observed in practice but dominate the worst-case analysis. If the smoothed running time of an algorithm is low (i.e., the algorithm is efficient in expectation on any perturbed instance) and inputs are subject to a small amount of random noise, then it is unlikely to encounter an instance on which the algorithm performs poorly. In practice, random noise can stem, for example, from measurement errors, numerical imprecision, or rounding errors. It can also model arbitrary influences, which we cannot quantify exactly, but for which there is also no reason to believe that they are adversarial. After its invention smoothed analysis has been applied in a variety of different contexts, e.g., linear programming [8, 19, 21, 23], multi-objective optimization [5, 10, 17, 18], online and approximation algorithms [4, 7, 20], searching and sorting [3, 12, 15, 16], game theory [9, 11], and local search [1, 2, 13, 14].

Key Results

Simplex Method

Spielman and Teng [22] considered linear programs of the form

$$\begin{aligned} &\text{maximize} \quad c^T x \\ &\text{subject to} \quad (\overline{A} + G)x \leq (\overline{b} + h), \end{aligned}$$

where $\overline{A} \in \mathbb{R}^{n \times d}$ and $\overline{b} \in \mathbb{R}^n$ are chosen arbitrarily by an adversary and the entries of the matrix $G \in \mathbb{R}^{n \times d}$ and the vector $h \in \mathbb{R}^n$ are independent Gaussian random variables that represent the perturbation. These Gaussian random variables have mean 0 and standard deviation $\sigma \cdot (\max_i \|(\overline{b}_i, \overline{a}_i)\|)$, where the vector $(\overline{b}_i, \overline{a}_i) \in \mathbb{R}^{d+1}$ consists of the i-th component of $\overline{b}$ and the i-th row of $\overline{A}$ and $\|\cdot\|$ denotes the Euclidean norm. Without loss of generality, we can scale the linear program specified by the adversary and assume that $\max_i \|(\overline{b}_i, \overline{a}_i)\| = 1$. Then the perturbation consists of adding an independent Gaussian random variable with standard deviation σ to each entry of $\overline{A}$ and $\overline{b}$. The smaller σ is chosen, the more concentrated are the random variables, and hence, the better worst-case instances can be approximated by the adversary. Intuitively, σ can be seen as a measure specifying how close the analysis is to a worst-case analysis.

Spielman and Teng analyzed the smoothed running time of the simplex algorithm using the *shadow vertex pivot rule*. This pivot rule has a simple and intuitive geometric description which makes probabilistic analyses feasible. Let x_0 denote the given initial vertex of the polytope $\mathcal{P}$ of feasible solutions. Since x_0 is a vertex of the polytope, there exists an objective function $u^T x$ which is maximized by x_0 subject to the constraint $x \in \mathcal{P}$. In the first step, the shadow vertex pivot rule computes an objective function $u^T x$ with this property. If x_0 is not an optimal solution of the linear program, then the vectors c and u are linearly independent and span a plane. The shadow vertex method projects the polytope $\mathcal{P}$ onto this plane. The *shadow*, that is, the projection of $\mathcal{P}$ onto this plane is a possibly open polygon. One can show that both x_0 and the optimal solution x^* are projected onto vertices of the polygon and that each path between the projections of x_0 and x^* in the polygon corresponds to a path between x_0 and x^* in the polytope. Hence, one only needs to follow the edges of the polygon starting from the projection of x_0 to (the projection of) x^*.

The number of steps performed by the simplex method with shadow vertex pivot rule is upper bounded by the number of vertices of the two-dimensional projection of the polytope. Hence, bounding the expected number of vertices on the

polygon is the crucial step for bounding the expected running time of the simplex method with shadow vertex pivot rule. Spielman and Teng first consider the case that the polytope $\mathcal{P}$ is projected onto a fixed plane specified by two fixed vectors c and u. They show that the expected number of vertices of the polygon is polynomially bounded in d, n, and $1/\sigma$. Though this result is the main ingredient of the analysis, alone it does not yield a polynomial bound on the smoothed running time of the simplex method. We have, for example, not yet described how the initial solution x_0 is found. It is also problematic that the vector u is not independent of the constraints because it is determined by x_0 which in turn is determined by a subset of the constraints. Spielman and Teng showed in a very involved analysis the following theorem.

Theorem 1 *The smoothed running time of the shadow vertex simplex method is bounded polynomially in d, n, and $1/\sigma$.*

Later, this analysis was substantially improved and simplified by Vershynin [23], who proved that the smoothed running time is even polynomially bounded in d, $\log n$, and $1/\sigma$.

Binary Optimization Problems

Beier and Vöcking [6] studied the question which *linear binary optimization problems* have *polynomial smoothed complexity*. Intuitively these are the problems that can be solved efficiently on perturbed inputs. An instance I of such an optimization problem Π consists of a set of feasible solutions $\mathcal{S} \subseteq \{0,1\}^n$ and a linear objective function $f : \{0,1\}^n \to \mathbb{R}$ of the form maximize (or minimize) $f(x) = c^T x$ for some $c \in \mathbb{R}^n$. Many well-known optimization problems can be formulated this way, e.g., the problem of finding a *Minimum Spanning Tree*, the *Knapsack Problem*, and the *Traveling Salesman Problem*.

It is assumed that an adversary is allowed to choose the coefficients of the objective function from the interval $[-1, 1]$. In the second step, these coefficients are perturbed by adding independent Gaussian random variables with mean 0 and standard deviation σ to them. Naturally one might define that a problem Π has *polynomial smoothed complexity* if there exists an algorithm A for Π whose expected running time $\mathbf{E}[T_A(I)]$ is bounded polynomially in the input size $|I|$ and $1/\sigma$. This definition, however, is not sufficiently robust as it depends on the machine model. An algorithm with expected polynomial running time on one machine model might have expected exponential running time on another machine model even if the former can be simulated by the latter in polynomial time. In contrast, the definition from [6] yields a notion of polynomial smoothed complexity that does not vary among classes of machines admitting polynomial time simulations among each other. It states that a problem Π has polynomial smoothed complexity if there exists an algorithm A for Π and some $\alpha > 0$ such that $\mathbf{E}[T_A(I)^\alpha]$ is bounded polynomially in the input size $|I|$ and $1/\sigma$.

Beier and Vöcking proved the following theorem that characterizes the class of linear binary optimization problems with polynomial smoothed complexity.

Theorem 2 *A linear binary optimization problem Π has polynomial smoothed complexity if and only if there exists a randomized algorithm for solving Π whose expected worst-case running time is pseudo-polynomial with respect to the coefficients in the objective function.*

For example, the knapsack problem, which can be solved by dynamic programming in pseudo-polynomial time, has polynomial smoothed complexity even if the weights are fixed and only the profits are randomly perturbed. Moreover, the traveling salesman problem does not have polynomial smoothed complexity when only the distances are randomly perturbed, unless P = NP, since a simple reduction from Hamiltonian cycle shows that it is strongly NP-hard.

Open Problems

An interesting open question is whether or not other pivot rules for the simplex method also have polynomial smoothed running time. It would also be interesting to see whether the insights gained from smoothed analysis can be used to improve existing algorithms.

Cross-References

▸ Knapsack

Recommended Reading

1. Arthur D, Vassilvitskii S (2009) Worst-case and smoothed analysis of the ICP algorithm, with an application to the k-means method. SIAM J Comput 39(2):766–782
2. Arthur D, Manthey B, Röglin H (2011) Smoothed analysis of the k-means method. J ACM 58(5)
3. Banderier C, Beier R, Mehlhorn K (2003) Smoothed analysis of three combinatorial problems. In: Proceedings of the 28th international symposium on mathematical foundations of computer science (MFCS), Bratislava. Lecture notes in computer science, vol 2747. Springer, pp 198–207
4. Becchetti L, Leonardi S, Marchetti-Spaccamela A, Schäfer G, Vredeveld T (2006) Average case and smoothed competitive analysis of the multilevel feedback algorithm. Math Oper Res 31(1):85–108
5. Beier R, Vöcking B (2004) Random knapsack in expected polynomial time. J Comput Syst Sci 69(3):306–329
6. Beier R, Vöcking B (2006) Typical properties of winners and losers in discrete optimization. SIAM J Comput 35(4):855–881
7. Bläser M, Manthey B, Rao BVR (2011) Smoothed analysis of partitioning algorithms for Euclidean functionals. In: Proceedings of the 12th workshop on algorithms and data structures (WADS), New York. Lecture notes in computer science. Springer, pp 110–121
8. Blum AL, Dunagan JD (2002) Smoothed analysis of the perceptron algorithm for linear programming. In: Proceedings of the 13th annual ACM-SIAM symposium on discrete algorithms (SODA), San Francisco. SIAM, pp 905–914
9. Boros E, Elbassioni K, Fouz M, Gurvich V, Makino K, Manthey B (2011) Stochastic mean payoff games: smoothed analysis and approximation schemes. In: Proceedings of the 38th international colloquium on automata, languages and programming (ICALP), Zurich, Part I. Lecture notes in computer science, vol 6755. Springer, pp 147–158
10. Brunsch T, Röglin H (2012) Improved smoothed analysis of multiobjective optimization. In: Proceedings of the 44th annual ACM symposium on theory of computing (STOC), New York, pp 407–426
11. Chen X, Deng X, Teng SH (2009) Settling the complexity of computing two-player Nash equilibria. J ACM 56(3)
12. Damerow V, Manthey B, auf der Heide FM, Räcke H, Scheideler C, Sohler C, Tantau T (2012) Smoothed analysis of left-to-right maxima with applications. ACM Trans Algorithms 8(3): Article no 30
13. Englert M, Röglin H, Vöcking B (2007) Worst case and probabilistic analysis of the 2-Opt algorithm for the TSP. In: Proceedings of the 18th annual ACM-SIAM symposium on discrete algorithms (SODA), New Orleans. SIAM, pp 1295–1304
14. Etscheid M, Röglin H (2014) Smoothed analysis of local search for the maximum-cut problem. In: Proceedings of the 25th annual ACM-SIAM symposium on discrete algorithms (SODA), Portland pp 882–889
15. Fouz M, Kufleitner M, Manthey B, Zeini Jahromi N (2012) On smoothed analysis of quicksort and Hoare's find. Algorithmica 62(3–4):879–905
16. Manthey B, Reischuk R (2007) Smoothed analysis of binary search trees. Theor Comput Sci 378(3):292–315
17. Moitra A, O'Donnell R (2012) Pareto optimal solutions for smoothed analysts. SIAM J Comput 41(5):1266–1284
18. Röglin H, Teng SH (2009) Smoothed analysis of multiobjective optimization. In: Proceedings of the 50th annual IEEE symposium on foundations of computer science (FOCS), Atlanta. IEEE, pp 681–690
19. Sankar A, Spielman DA, Teng SH (2006) Smoothed analysis of the condition numbers and growth factors of matrices. SIAM J Matrix Anal Appl 28(2):446–476
20. Schäfer G, Sivadasan N (2005) Topology matters: smoothed competitiveness of metrical task systems. Theor Comput Sci 241(1–3):216–246
21. Spielman DA, Teng SH (2003) Smoothed analysis of termination of linear programming algorithms. Math Program 97(1–2):375–404
22. Spielman DA, Teng SH (2004) Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time. J ACM 51(3):385–463
23. Vershynin R (2009) Beyond Hirsch conjecture: walks on random polytopes and smoothed complexity of the simplex method. SIAM J Comput 39(2):646–678

Snapshots in Shared Memory

Eric Ruppert
Department of Computer Science and Engineering, York University, Toronto, ON, Canada

Keywords

Atomic scan

Years and Authors of Summarized Original Work

1993; Afek, Attiya, Dolev, Gafni, Merritt, Shavit

Problem Definition

Implementing a snapshot object is an abstraction of the problem of obtaining a consistent view of several shared variables while other processes are concurrently updating those variables.

In an asynchronous shared-memory distributed system, a collection of n processes communicate by accessing shared data structures, called *objects*. The system provides basic types of shared objects; other needed types must be built from them. One approach uses locks to guarantee exclusive access to the basic objects, but this approach is not fault-tolerant, risks deadlock or livelock, and causes delays when a process holding a lock runs slowly. Lock-free algorithms avoid these problems but introduce new challenges. For example, if a process reads two shared objects, the values it reads may not be consistent if the objects were updated between the two reads.

A *snapshot object* stores a vector of m values, each from some domain D. It provides two operations: scan and update(i, v), where $1 \leq i \leq m$ and $v \in D$. If the operations are invoked sequentially, an update(i, v) operation changes the value of the ith component of the stored vector to v, and a scan operation returns the stored vector.

Correctness when snapshot operations by different processes overlap in time is described by the *linearizability* condition, which says operations should appear to occur instantaneously. More formally, for every execution, one can choose an instant of time for each operation (called its *linearization point*) between the invocation and the completion of the operation. (An incomplete operation may either be assigned no linearization point or given a linearization point at any time after its invocation.) The responses returned by all completed operations in the execution must return the same result as they would if all operations were executed sequentially in the order of their linearization points.

An implementation must also satisfy a progress property. *Wait-freedom* requires that each process completes each scan or update in a finite number of its own steps. The weaker *non-blocking* progress condition says the system cannot run forever without some operation completing.

This article describes implementations of snapshots from more basic types, which are also linearizable, without locks. Two types of snapshots have been studied. In a *single-writer* snapshot, each component is owned by a process, and only that process may update it. (Thus, for single-writer snapshots, $m = n$.) In a *multi-writer* snapshot, any process may update any component. There also exist algorithms for *single-scanner* snapshots, where only one process may scan at a time [10, 13, 14, 16]. Snapshots were introduced by Afek et al. [1], Anderson [2] and Aspnes and Herlihy [4].

Space complexity is measured by the number of basic objects used and their size (in bits). Time complexity is measured by the maximum number of steps a process must do to finish a scan or update, where a step is an access to a basic shared object. (Local computation and local memory accesses are usually not counted.) Complexity bounds will be stated in terms of $n, m, d = \log |D|$ and k, the number of operations invoked in an execution. Ordinarily, there is no bound on k.

Most of the algorithms below use read-write registers, the most elementary shared object type. A *single-writer* register may only be written by one process. A *multi-writer* register may be written by any process. Some algorithms using stronger types of basic objects are discussed in section "Wait-Free Implementations from Small, Stronger Objects".

Key Results

A Simple Non-blocking Implementation from Small Registers

Suppose each component of a single-writer snapshot object is represented by a single-writer register. Process i does an update(i, v) by writing v and a sequence number into register i, and incrementing its sequence number. Performing a scan operation is more difficult than merely reading each of the m registers, since some registers

might change while these reads are done. To scan, a process repeatedly reads all the registers. A sequence of reads of all the registers is called a *collect*. If two collects return the same vector, the scan returns that vector (with the sequence numbers stripped away). The sequence numbers ensure that, if the same value is read in a register twice, the register had that value during the entire interval between the two reads. The scan can be assigned a linearization point between the two identical collects, and updates are linearized at the write. This algorithm is non-blocking, since a scan continues running only if at least one update operation is completed during each collect. A similar algorithm, with process identifiers appended to the sequence numbers, implements a non-blocking multi-writer snapshot from m multi-writer registers.

Wait-Free Implementations from Large Registers

Afek et al. [1] described how to modify the non-blocking single-writer snapshot algorithm to make it wait-free using scans embedded within the updates. An update(i, v) first does a scan and then writes a triple containing the scan's result, v and a sequence number into register i. While a process P is repeatedly performing collects to do a scan, either two collects return the same vector (which P can return) or P will eventually have seen three different triples in the register of some other process. In the latter case, the third triple that P saw must contain a vector that is the result of a scan that started after P's scan, so P's scan outputs that vector. Updates and scans that terminate after seeing two identical collects are assigned linearization points as before. If one scan obtains its output from an embedded scan, the two scans are given the same linearization point. This is a wait-free single-writer snapshot implementation from n single-writer registers of $(n+1)d + \log k$ bits each. Operations complete within $O(n^2)$ steps. Afek et al. [1] also describe how to replace the unbounded sequence numbers with handshaking bits. This requires $n\Theta(nd)$-bit registers and n^2 1-bit registers. Operations still complete in $O(n^2)$ steps.

The same idea can be used to build multi-writer snapshots from multi-writer registers. Using unbounded sequence numbers yields a wait-free algorithm that uses m registers storing $\Theta(nd + \log k)$ bits each, in which each operation completes within $O(mn)$ steps. (This algorithm is given explicitly in [9].) No algorithm can use fewer than m registers if $n \geq m$ [9]. If handshaking bits are used instead, the multi-writer snapshot algorithm uses n^2 1-bit registers, $m(d + \log n)$-bit registers and n (md)-bit registers, and each operation uses $O(nm + n^2)$ steps [1].

Guerraoui and Ruppert [12] gave a similar wait-free multi-writer snapshot implementation that is anonymous, i.e., it does not use process identifiers and all processes are programmed identically.

Anderson [3] gave an implementation of a multi-writer snapshot from a single-writer snapshot. Each process stores its latest update to each component of the multi-writer snapshot in the single-writer snapshot, with associated timestamp information computed by scanning the single-writer snapshot. A scan is done using just one scan of the single-writer snapshot. An update requires scanning and updating the single-writer snapshot twice. The implementation involves some blow-up in the size of the components, i.e., to implement a multi-writer snapshot with domain D requires a single-writer snapshot with a much larger domain D'. If the goal is to implement multi-writer snapshots from single-writer registers (rather than multi-writer registers), Anderson's construction gives a more efficient solution than that of Afek et al.

Attiya, Herlihy and Rachman [7] defined the *lattice agreement* object, which is very closely linked to the problem of implementing a single-writer snapshot when there is a known upper bound on k. Then, they showed how to construct a single-writer snapshot (with no bound on k) from an infinite sequence of lattice agreement objects. Each snapshot operation accesses the lattice agreement object twice and does $O(n)$ additional steps. Their implementations of lattice agreement are discussed in section "Wait-Free Implementations from Small, Stronger Objects".

Attiya and Rachman [8] used a similar approach to give a single-writer snapshot implementation from large single-writer registers using $O(n \log n)$ steps per operation. Each update has an associated sequence number. A scanner traverses a binary tree of height $\log k$ from root to leaf (here, a bound on k is required). Each node has an array of n single-writer registers. A process arriving at a node writes its current vector into a single-writer register associated with the node and then gets a new vector by combining information read from all n registers. It proceeds to the left or right child depending on the sum of the sequence numbers in this vector. Thus, all scanners can be linearized in the order of the leaves they reach. Updates are performed by doing a similar traversal of the tree. The bound on k can be removed as in [7]. Attiya and Rachman also give a more direct implementation that achieves this by recycling the snapshot object that assumes a bound on k. Their algorithm has also been adapted to solve condition-based consensus [15].

Attiya, Fouren and Gafni [6] described how to adapt the algorithm of Attiya and Rachman [8] so that the number of steps required to perform an operation depends on the number of processes that actually access the object, rather than the number of processes in the system.

Attiya and Fouren [5] solve lattice agreement in $O(n)$ steps. (Here, instead of using the terminology of lattice agreement, the algorithm is described in terms of implementing a snapshot in which each process does at most one snapshot operation.) The algorithm uses, as a data structure, a two-dimensional array of $O(n^2)$ *reflectors*. A reflector is an object that can be used by two processes to exchange information. Each reflector is built from two large single-writer registers. Each process chooses a path through the array of reflectors, so that at most two processes visit each reflector. Each reflector in column i is used by process i to exchange information with one process $j < i$. If process i reaches the reflector first, process j learns about i's update (if any). If process j reaches it first, then process i learns all the information that j has already gathered. (If both reach it at about the same time, both processes learn the information described above.) As the processes move from column $i-1$ to column i, a process that enters column i at some row r will have gathered all the information that has been gathered by any process that enters column i below row r (and possibly more). This invariant is maintained by ensuring that if process i passes information to any process $j < i$ in row r of column i, it also passes that information to all processes that entered column i above row r. Furthermore, process i exits column i at a row that matches the amount of information it learns while traveling through the column. When processes have reached the rightmost column of the array, the ones in higher rows know strictly more than the ones in lower rows. Thus, the linearization order of their scans is the order in which they exit the rightmost column, from bottom to top. The techniques of Attiya, Herlihy and Rachman [7, 8], mentioned above, can be used to remove the restriction that each process performs at most one operation. The number of steps per operation is still $O(n)$.

Wait-Free Implementations from Small, Stronger Objects

All of the wait-free implementations described above use registers that can store $\Omega(m)$ bits each, and are therefore not practical when m is large. Some implementations from smaller objects equipped with stronger synchronization operations, rather than just reads and writes, are described in this section. An object is considered to be small if it can store $O(d + \log n + \log k)$ bits. This means that it can store a constant number of component values, process identifiers and sequence numbers.

Attiya, Herlihy and Rachman [7] gave an elegant divide-and-conquer recursive solution to the lattice agreement problem. The division of processes into groups for the recursion can be done dynamically using test&set objects. This provides a snapshot algorithm that runs in $O(n)$ time per operation, and uses $O(kn^2 \log n)$ small single-writer registers and $O(kn \log^2 n)$ test&set objects. (This requires modifying their implementation to replace those registers that are large, which are written only once, by many small

registers.) Using randomization, each test&set object can be replaced by single-writer registers to give a snapshot implementation from registers only with $O(n)$ expected steps per operation.

Jayanti [13] gave a multi-writer snapshot implementation from $O(mn^2)$ small compare & swap objects where updates take $O(1)$ steps and scans take $O(m)$ steps. He began with a very simple single-scanner, single-writer snapshot implementation from registers that uses a secondary array to store a copy of recent updates. A scan clears that array, collects the main array, and then collects the secondary array to find any overlooked updates. Several additional mechanisms are introduced for the general, multi-writer, multi-scanner snapshot. In particular, compare & swap operations are used instead of writes to coordinate writers updating the same component and multiple scanners coordinate with one another to simulate a single scanner. Jayanti's algorithm builds on an earlier paper by Riany, Shavit and Touitou [16], which gave an implementation that achieved similar complexity, but only for a single-writer snapshot.

Applications

Applications of snapshots include distributed databases, storing checkpoints or backups for error recovery, garbage collection, deadlock detection, debugging distributed programmes and obtaining a consistent view of the values reported by several sensors. Snapshots have been used as building blocks for distributed solutions to randomized consensus and approximate agreement. They are also helpful as a primitive for building other data structures. For example, consider implementing a counter that stores an integer and provides increment, decrement and read operations. Each process can store the number of increments it has performed minus the number of its decrements in its own component of a single-writer snapshot object, and the counter may be read by summing the values from a scan. See [10] for references on many of the applications mentioned here.

Open Problems

Some complexity lower bounds are known for implementations from registers [9], but there remain gaps between the best known algorithms and the best lower bounds. In particular, it is not known whether there is an efficient wait-free implementation of snapshots from small registers.

Experimental Results

Riany, Shavit and Touitou gave performance evaluation results for several implementations [16].

Cross-References

- ▸ Implementing Shared Registers in Asynchronous Message-Passing Systems
- ▸ Linearizability
- ▸ Registers

Recommended Reading

See also Fich's survey paper on the complexity of implementing snapshots [11].

1. Afek Y, Attiya H, Dolev D, Gafni E, Merritt M, Shavit N (1993) Atomic snapshots of shared memory. J Assoc Comput Mach 40:873–890
2. Anderson JH (1993) Composite registers. Distrib Comput 6:141–154
3. Anderson JH (1994) Multi-writer composite registers. Distrib Comput 7:175–195
4. Aspnes J, Herlihy M (1990) Wait-free data structures in the asynchronous PRAM model. In: Proceedings of the 2nd ACM symposium on parallel algorithms and architectures, Crete, July 1990. ACM, New York, pp 340–349
5. Attiya H, Fouren A (2001) Adaptive and efficient algorithms for lattice agreement and renaming. SIAM J Comput 31:642–664
6. Attiya H, Fouren A, Gafni E (2002) An adaptive collect algorithm with applications. Distrib Comput 15:87–96
7. Attiya H, Herlihy M, Rachman O (1995) Atomic snapshots using lattice agreement. Distrib Comput 8:121–132
8. Attiya H, Rachman O (1998) Atomic snapshots in O(n log n) operations. SIAM J Comput 27:319–340

9. Ellen F, Fatourou P, Ruppert E (2007) Time lower bounds for implementations of multi-writer snapshots. J Assoc Comput Mach 54(6), 30
10. Fatourou P, Kallimanis ND (2006) Single-scanner multi-writer snapshot implementations are fast! In: Proceedings of the 25th ACM symposium on principles of distributed computing, Colorado, July 2006. ACM, New York, pp 228–237
11. Fich FE (2005) How hard is it to take a snapshot? In: SOFSEM 2005: theory and practice of computer science, Liptovský Ján, Jan 2005. LNCS, vol 3381. Springer, pp 28–37
12. Guerraoui R, Ruppert E (2007) Anonymous and fault-tolerant shared-memory computing. Distrib Comput 20(3):165–177
13. Jayanti P (2005) An optimal multi-writer snapshot algorithm. In: Proceedings of the 37th ACM symposium on theory of computing, Baltimore, May 2005. ACM, New York, pp 723–732
14. Kirousis LM, Spirakis P, Tsigas P (1996) Simple atomic snapshots: a linear complexity solution with unbounded time-stamps. Inf Process Lett 58:47–53
15. Mostéfaoui A, Rajsbaum S, Raynal M, Roy M (2004) Conditionbased consensus solvability: a hierarchy of conditions and efficient protocols. Distrib Comput 17:1–20
16. Riany Y, Shavit N, Touitou D (2001) Towards a practical snapshot algorithm. Theor Comput Sci 269:163–201

Sorting by Transpositions and Reversals (Approximate Ratio 1.5)

Chin Lung Lu
Institute of Bioinformatics and Department of Biological Science and Technology, National Chiao Tung University, Hsinchu, Taiwan

Keywords

Genome rearrangements

Problem Definition

One of the most promising ways to determine evolutionary distance between two organisms is to compare the order of appearance of identical (e.g., orthologous) genes in their genomes. The resulting genome rearrangement problem calls for finding a shortest sequence of rearrangement operations that sorts one genome into the other. In this work [8], Hartman and Sharan provide a 1.5-approximation algorithm for the problem of sorting by transpositions, transreversals, and revrevs, improving on a previous 1.75 ratio for this problem. Their algorithm is also faster than current approaches and requires $O(n^{3/2}\sqrt{\log n})$ time for n genes.

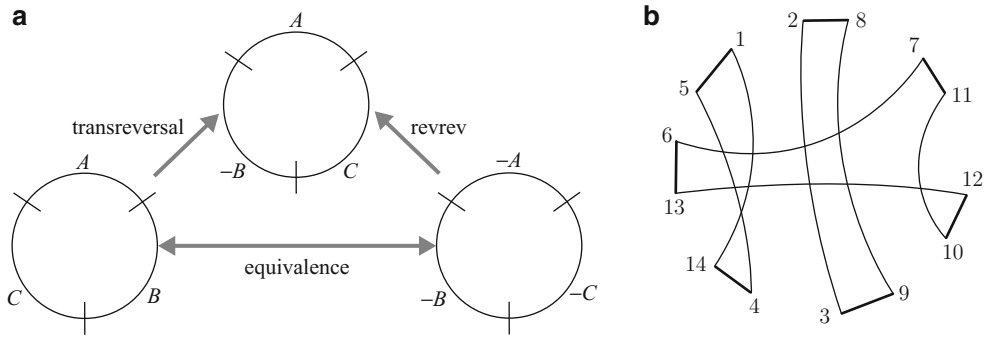
Notations and Definition

A *signed permutation* $\pi = [\pi_1, \pi_2, \ldots, \pi_n]$ on $n(\pi) \equiv n$ elements is a permutation in which each element is labeled by a sign of plus or minus. A *segment* of π is a sequence of consecutive elements $\pi_i, \pi_{i+1}, \ldots, \pi_k$, where $1 \leq i \leq k \leq n$. A *reversal* ρ is an operation that reverses the order of the elements in a segment and also flips their signs. Two segments $\pi_i, \pi_{i+1}, \ldots, \pi_k$ and $\pi_j, \pi_{j+1}, \ldots, \pi_l$ are said to be *contiguous* if $j = k+1$ or $i = l+1$. A *transposition* τ is an operation that exchanges two contiguous (disjoint) segments. A *transreversal* $\tau\rho_{A,B}$ (respectively, $w\tau\rho_{B,A}$) is a transposition that exchanges two segments A and B and also reverses A (respectively, B). A *revrev* operation $\rho\rho$ reverses each of the two contiguous segments (without transposing them). The problem of finding a shortest sequence of transposition, transreversal, and revrev operations that transforms a permutation into the identity permutation is called *sorting by transpositions, transreversals, and revrevs*. The *distance* of a permutation π, denoted by $d(\pi)$, is the length of the shortest sorting sequence.

Key Results

Linear vs. Circular Permutations

An operation is said to *operate* on the affected segments as well as on the elements in those segments. Two operations μ and μ' are *equivalent* if they have the same rearrangement result, i.e., $\mu \cdot \pi = \mu' \cdot \pi$ for all π. In this work [8], Hartman and Sharan showed that for an element x of a circular permutation π, if μ is an operation that operates on x, then there exists an equivalent operation μ' that does not operate on x. Based on this property, they further proved that the problem of sorting by transpositions, transreversals, and revrevs is



Sorting by Transpositions and Reversals (Approximate Ratio 1.5), Fig. 1 (**a**) The equivalence of transreversal and revrev on circular permutations. (**b**) The breakpoint graph $G(\pi)$ of the permutation $\pi = [1, -4, 6, -5, 2, -7, -3]$, for which $f(\pi) = [1, 2, 8, 7, 11, 12, 10, 9, 3, 4, 14, 13, 6, 5]$. It is convenient to draw $G(\pi)$ on a circle such that *black edges* (i.e., *thick lines*) are on the circumference and *gray edges* (i.e., *thin lines*) are chords

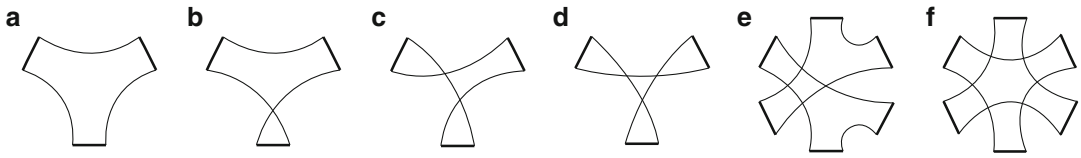
equivalent for linear and circular permutations. Moreover, they observed that revrevs and transreversals are equivalent operations for circular permutations (as illustrated in Fig. 1a), implying that the problem of sorting a linear/circular permutation by transpositions, transreversals, and revrevs can be reduced to that of sorting a circular permutation by transpositions and transreversals only.

The Breakpoint Graph

Given a signed permutation π on {1,2,...,n} of n elements, it is transformed into an unsigned permutation $f(\pi) = \pi' = [\pi'_1, \pi'_2, \ldots, \pi'_{2n}]$ on {1,2,...,2 n} of $2n$ elements by replacing each positive element i with two elements $2i - 1, 2i$ (in this order) and each negative element $-i$ with $2i, 2i - 1$. The extended $f(\pi)$ is considered here as a circular permutation by identifying $2n + 1$ and 1 in both indices and elements. To ensure that every operation on $f(\pi)$ can be mimicked by an operation on π, only operations that cut before odd position are allowed for $f(\pi)$. The *breakpoint graph* $(G\pi)$ is an edge-colored graph on $2n$ vertices $\{1, 2, \ldots, 2n\}$, in which for every $1 \le i \le n, \pi'_{2i}$ is joined to π'_{2i+1} by a black edge and $2i$ is joined to $2i + 1$ by a gray edge (e.g., see Fig. 1b). Since the degree of each vertex in $G(\pi)$ is exactly 2, $G(\pi)$ uniquely decomposes into cycles. A *k-cycle* (i.e., a cycle of *lengthk*) is a cycle with k black edges, and it is *odd* if k is odd. The number of odd cycles in $G(\pi)$ is denoted by $c_{\text{odd}}(\pi)$. It is not hard to verify that $G(\pi)$ consists of n 1-cycles, and hence, $c_{\text{odd}}(\pi) = n$, if π is an identity permutation $[1, 2, \ldots, n]$. Gu et al. [5] have shown that $c_{\text{odd}}(\mu \cdot \pi) \le c_{\text{odd}}(\pi) + 2$ for all linear permutations π and operations μ. In this work [8], Hartman and Sharan further noted that the above result holds also for circular permutations and proved that the lower bound of $d(\pi)$ is $(n(\pi) - c_{\text{odd}}(\pi))/2$.

Transformation into 3-Permutations

A permutation is called *simple* if its breakpoint graph contains only k-cycle, where $k \le 3$. A simple permutation is also called a *3-permutation* if it contains no 2-cycles. A transformation from π to $\hat{\pi}$ is said to be *safe* if $n(\pi) - c_{\text{odd}}(\pi) = n(\hat{\pi}) - c_{\text{odd}}(\hat{\pi})$. It has been shown that every permutation π can be transformed into a simple one π' by safe transformations and, moreover, every sorting of π' mimics a sorting of π with the same number of operations [6, 11]. Here, Hartman and Sharan [8] further showed that every simple permutation π' can be transformed into a 3-permutation $\hat{\pi}$ by safe paddings (of transforming those 2-cycles into 1-twisted 3-cycles) and, moreover, every sorting of $\hat{\pi}$ mimics a sorting of π' with the same number of operations. Hence, based on these two properties, an arbitrary permutation π can be transformed into a 3-permutation $\hat{\pi}$ such that every sorting of $\hat{\pi}$ mimics a sorting of π with the



Sorting by Transpositions and Reversals (Approximate Ratio 1.5), Fig. 2 Configurations of 3-cycles. (**a**) Unoriented, 0-twisted 3-cycle. (**b**) Unoriented, 1-twisted 3-cycle. (**c**) Oriented, 2-twisted 3-cycle. (**d**) Oriented, 3-twisted 3-cycle. (**e**) A pair of intersecting 3-cycles. (**f**) A pair of interleaving 3-cycles

same number of operations, suggesting that one can restrict attention to circular 3-permutations only.

Cycle Types

An operation that cuts some black edges is said to *act* on these edges. An operation is further called a *k-operation* if it increases the number of odd cycles by k. A (0, 2,2)-*sequence* is a sequence of three operations, of which the first is a 0-operation and the next two are 2-operations. An odd cycle is called *oriented* if there is a 2-operation that acts on three of its black edges; otherwise, it is *unoriented*. A *configuration* of cycles is a subgraph of the breakpoint graph that contains one ore more cycles. As shown in Fig. 2a–d, there are four possible configurations of single 3-cycles. A black edge is called *twisted* if its two adjacent gray edges cross each other in the circular breakpoint graph. A cycle is k-twisted if k of its black edges is twisted. For example, the 3-cycles in Fig. 2a–d are 0-, 1-, 2-, and 3-twisted, respectively. Hartman and Sharan observed that a 3-cycle is oriented if and only if it is 2- or 3-twisted.

Cycle Configurations

Two pairs of black edges are called *intersecting* if they alternate in the order of their occurrence along the circle. A pair of black edges *intersects* with cycle C, if it intersects with a pair of black edges that belong to C. Cycles C and *Dintersect* if there is a pair of black edges in C that intersects with D (see Fig. 2e). Two intersecting cycles are called *interleaving* if their black edges alternate in their order of occurrence along the circle (see Fig. 2f). Clearly, the relation between two cycles is one of (1) nonintersecting, (2) intersecting but non-interleaving, and (3) interleaving. A pair of black edges is *coupled* if they are connected by a gray edge and when reading the edges along the cycle, they are read in the same direction. For example, all pairs of black edges in Fig. 2a are coupled. Gu et al. [5] have shown that given a pair of coupled black edges (b_1, b_2), there exists a cycle C that intersects with (b_1, b_2). A *1-twisted pair* is a pair of 1-twisted cycles, whose twists are consecutive on the circle in a configuration that consists of these two cycles only. A 1-twisted cycle is called *closed* in a configuration if its two coupled edges intersect with some other cycle in the configuration. A configuration is *closed* if at least one of its 1-twisted cycles is closed; otherwise, it is called *open*.

The Algorithm

The basic ideas of the Hartman and Sharan's 1.5-approximation algorithm [8] for the problem of sorting by transpositions, transreversals, and revrevs are as follows. Hartman and Sharan reduced the problem to that of sorting a circular 3-permutation by transpositions and transreversals only and then focused on transforming the 3-cycles into 1-cycles in the breakpoint graph of this 3-permutation. By definition, an oriented (i.e., 2- or 3-twisted) 3-cycle admits a 2-operation and, therefore, they continued to consider unoriented (i.e., 0- or 1-twisted) 3-cycles only. Since configurations involving only 0-twisted 3-cycles were handled with (0,2,2)-sequences in [7], Hartman and Sharan restricted their attention to those configurations that consist of 0- and 1-twisted 3-cycles. They showed that these configurations are all closed and that it can be sorted by a (0,2,2)-sequence of operations for each of the following five possible closed configurations: (1) a closed configuration with two unoriented, interleaving 3-cycles that do not form a 1-twisted pair; (2) a closed configuration with two intersecting, 0-twisted 3-cycles; (3) a closed configuration with two intersecting, 1-twisted 3-cycles; (4) a closed

configuration with a 0-twisted 3-cycles that intersects with the coupled edges of a 1-twisted 3-cycle; and (5) a closed configuration that contains $k \geq 2$ mutually interleaving 1-twisted 3-cycles such that all their twists are consecutive on the circle and k is maximal with this property. As a result, the sequence of operations used by Hartman and Sharan in their algorithm contains only 2-operations and (0,2,2)-sequences. Since every sequence of three operations increases the number of odd cycles by at least 4 out of 6 possible in 3 steps, the ratio of their approximation algorithm is 1.5. Furthermore, Hartman and Sharan showed that their algorithm can be implemented in $O(n^{3/2}\sqrt{\log n})$ time using the data structure of Kaplan and Verbin [10], where n is the number of elements in the permutation.

Theorem 1 *The problem of sorting linear permutations by transpositions, transreversals, and revrevs is linearly equivalent to the problem of sorting circular permutations by transpositions, transreversals, and revrevs.*

Theorem 2 *There is a 1.5-approximation algorithm for sorting by transpositions, transreversals, and revrevs, which runs in $O(n^{3/2}\sqrt{\log n})$ time.*

Applications

When trying to determine evolutionary distance between two organisms using genomic data, biologists may wish to reconstruct the sequence of evolutionary events that have occurred to transform one genome into the other. One of the most promising ways to do this phylogenetic study is to compare the order of appearance of identical (e.g., orthologous) genes in two different genomes [9, 12]. This comparison of computing global rearrangement events (such as reversals, transpositions, and transreversals of genome segments) may provide more accurate and robust clues to the evolutionary process than the analysis of local point mutations (i.e., substitutions, insertions, and deletions of nucleotides/amino acids). Usually, the two genomes being compared are represented by signed permutations, with each element standing for a gene and its sign representing the (transcriptional) direction of the corresponding gene on a chromosome. Then the goal of the resulting genome rearrangement problem is to find a shortest sequence of rearrangement operations that transforms (or, equivalently, *sorts*) one permutation into the other. Previous work focused on the problem of sorting a permutation by reversals. This problem has been shown by Capara [2] to be NP-hard, if the considered permutation is unsigned. However, for signed permutations, this problem becomes tractable and Hannenhalli and Pevzer [6] gave the first polynomial-time algorithm for it. On the other hand, there has been less progress on the problem of sorting by transpositions. Thus far, the complexity of this problem is still open, although several 1.5-approximation algorithms [1, 3, 7] have been proposed for it. Recently, the approximation ratio of sorting by transpositions was further improved to 1.375 by Elias and Hartman [4]. Gu et al. [5] and Lin and Xue [11] gave quadratic-time 2-approximation algorithms for sorting signed, linear permutations by transpositions and transreversals. In [11], Lin and Xue considered the problem of sorting signed, linear permutations by transpositions, transreversals, and revrevs and proposed a quadratic-time 1.75-approximation algorithm for it. In this work [8], Hartman and Sharan further showed that this problem is equivalent for linear and circular permutations and can be reduced to that of sorting signed, circular permutations by transpositions and transreversals only. In addition, they provided a 1.5-approximation algorithm that can be implemented in $O(n^{3/2}\sqrt{\log n})$ time.

Cross-References

- Sorting Signed Permutations by Reversal (Reversal Distance)
- Sorting Signed Permutations by Reversal (Reversal Sequence)

Recommended Reading

1. Bafna V, Pevzner PA (1998) Sorting by transpositions. SIAM J Discret Math 11:224–240

2. Caprara A (1999) Sorting permutations by reversals and Eulerian cycle decompositions. SIAM J Discret Math 12:91–110
3. Christie DA (1999) Genome rearrangement problems. Ph.D. thesis, Department of Computer Science. University of Glasgow, U.K.
4. Elias I, Hartman T (2006) A 1.375-approximation algorithm for sorting by transpositions. IEEE/ACM Trans Comput Biol Bioinform 3:369–379
5. Gu QP, Peng S, Sudborough H (1999) A 2-approximation algorithm for genome rearrangements by reversals and transpositions. Theor Comput Sci 210:327–339
6. Hannenhalli S, Pevzner PA (1999) Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. J Assoc Comput Mach 46:1–27
7. Hartman T, Shamir R (2006) A simpler and faster 1.5-approximation algorithm for sorting by transpositions. Inf Comput 204:275–290
8. Hartman T, Sharan R (2004) A 1.5-approximation algorithm for sorting by transpositions and transreversals. In: Proceedings of the 4th workshop on algorithms in bioinformatics (WABI'04), Bergen, pp 50–61, 17–21 Sept (2004)
9. Hoot SB, Palmer JD (1994) Structural rearrangements, including parallel inversions, within the chloroplast genome of Anemone and related genera. J Mol Evol 38:274–281
10. Kaplan H, Verbin E (2003) Efficient data structures and a new randomized approach for sorting signed permutations by reversals. In: Proceedings of the 14th annual symposium on combinatorial pattern matching (CPM'03), Morelia, pp 170–185, 25–27 June (2003)
11. Lin GH, Xue G (2001) Signed genome rearrangements by reversals and transpositions: models and approximations. Theor Comput Sci 259:513–531
12. Palmer JD, Herbon LA (1986) Tricircular mitochondrial genomes of Brassica and Raphanus: reversal of repeat configurations by inversion. Nucleic Acids Res 14:9755–9764

Sorting Signed Permutations by Reversal (Reversal Distance)

David A. Bader
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Keywords

Inversion distance; Reversal distance; Sorting by reversals

Years and Authors of Summarized Original Work

2001; Bader, Moret, Yan

Problem Definition

This entry describes algorithms for finding the minimum number of steps needed to sort a signed permutation (also known as inversion distance, reversal distance). This is a real-world problem and, for example, is used in computational biology.

Inversion distance is a difficult computational problem that has been studied intensively in recent years [1, 4, 6–10]. Finding the inversion distance between unsigned permutations is NP-hard [7], but with signed ones, it can be done in linear time [1].

Key Results

Bader et al. [1] present the first worst-case linear-time algorithm for computing the reversal distance that is simple and practical and runs faster than previous methods. Their key innovation is a new technique to compute connected components of the overlap graph using only a stack, which results in the simple linear-time algorithm for computing the inversion distance between two signed permutations. Bader et al. provide ample experimental evidence that their linear-time algorithm is efficient in practice as well as in theory: they coded it as well as the algorithm of Berman and Hannenhalli, using the best principles of algorithm engineering to ensure that both implementations would be as efficient as possible and compared their running times on a large range of instances generated through simulated evolution.

Bafna and Pevzner introduced the cycle graph of a permutation [3], thereby providing the basic data structure for inversion distance computations. Hannenhalli and Pevzner then developed the basic theory for expressing the inversion distance in easily computable terms

(number of breakpoints minus number of cycles plus number of hurdles plus a correction factor for a fortress [3, 15]-hurdles and fortresses are easily detectable from a connected component analysis). They also gave the first polynomial-time algorithm for sorting signed permutations by reversals [9]; they also proposed a $O(n^4)$ implementation of their algorithm which runs in quadratic time when restricted to distance computation. Their algorithm requires the computation of the connected components of the overlap graph, which is the bottleneck for the distance computation. Berman and Hannenhalli later exploited some combinatorial properties of the cycle graph to give a $O(n\alpha(n))$ algorithm to compute the connected components, leading to a $O(n^2\alpha(n))$ implementation of the sorting algorithm [6], where α is the inverse Ackerman function. (The later Kaplan-Shamir-Tarjan (KST) algorithm [10] reduces the time needed to compute the shortest sequence of inversions, but uses the same algorithm for computing the length of that sequence.)

No algorithm that actually builds the overlap graph can run in linear time, since that graph can be of quadratic size. Thus, Bader's key innovation is to construct an *overlap forest* such that two vertices belong to the same tree in the forest exactly when they belong to the same connected component in the overlap graph. An overlap forest (the composition of its trees is unique, but their structure is arbitrary) has exactly one tree per connected component of the overlap graph and is thus of linear size. The linear-time step for computing the connected components scans the permutation twice. The first scan sets up a trivial forest in which each node is its own tree, labeled with the beginning of its cycle. The second scan carries out an iterative refinement of this first forest, by adding edges and so merging trees in the forest; unlike a Union-Find, however, this algorithm does not attempt to maintain the trees within certain shape parameters. This step is the key to Bader's linear-time algorithm for computing the reversal distance between signed permutations.

Applications

Some organisms have a single chromosome or contain single-chromosome organelles (such as mitochondria or chloroplasts), the evolution of which is largely independent of the evolution of the nuclear genome. Given a particular strand from a single chromosome, whether linear or circular, we can infer the ordering and directionality of the genes, thus representing each chromosome by an ordering of oriented genes. In many cases, the evolutionary process that operates on such single-chromosome organisms consists mostly of inversions of portions of the chromosome; this finding has led many biologists to reconstruct phylogenies based on gene orders, using as a measure of evolutionary distance between two genomes the inversion distance, i.e., the smallest number of inversions needed to transform one signed permutation into the other [11, 12, 14].

The linear-time algorithm is in wide use (as it has been cited nearly 200 times within the first several years of its publication). Examples include the handling multichromosomal genome rearrangements [16], genome comparison [5], parsing RNA secondary structure [13], and phylogenetic study of the HIV-1 virus [2].

Open Problems

Efficient algorithms for computing minimum distances with weighted inversions, transpositions, and inverted transpositions are open.

Experimental Results

Bader et al. give experimental results in [1].

URL to Code

An implementation of the linear-time algorithm is available as *C* code from www.cc.gatech.edu/~bader. Two other dominated implementations are available that are designed to compute the shortest sequence of inversions as well as its length:

one, due to Hannenhalli that implements his first algorithm [9], which runs in quadratic time when computing distances, while the other, a Java applet written by Mantin (http://www.math.tau.ac.il/~rshamir/GR/), that implements the KST algorithm [10], but uses an explicit representation of the overlap graph and thus also takes quadratic time. The implementation due to Hannenhalli is very slow and implements the original method of Hannenhalli and Pevzner and not the faster one of Berman and Hannenhalli. The KST applet is very slow as well since it explicitly constructs the overlap graph.

Cross-References

For finding the actual sorting sequence, see the entry:

- ▸ Sorting Signed Permutations by Reversal (Reversal Sequence)

Recommended Reading

1. Bader DA, Moret BME, Yan M (2001) A linear-time algorithm for computing inversion distance between signed permutations with an experimental study. J Comput Biol 8(5):483–491. An earlier version of this work appeared In: The proceedings of 7th Int'l workshop on algorithms and data structures (WADS 2001)
2. Badimo A, Bergheim A, Hazelhurst S, Papathanasopolous M, Morris L (2003) The stability of phylogenetic tree construction of the HIV-1 virus using genome-ordering data versus env gene data. In: Proceedings of ACM annual research conference of the South African institute of computer scientists and information technologists on enablement through technology (SAICSIT 2003), Port Elizabeth, Sept 2003, vol 47. ACM, Fourways, South Africa, pp 231–240
3. Bafna V, Pevzner PA (1993) Genome rearrangements and sorting by reversals. In: Proceedings of 34th annual IEEE symposium on foundations of computer science (FOCS93), Palo Alto, CA, pp 148–157. IEEE Press
4. Bafna V, Pevzner PA (1996) Genome rearrangements and sorting by reversals. SIAM J Comput 25:272–289
5. Bergeron A, Stoye J (2006) On the similarity of sets of permutations and its applications to genome comparison. J Comput Biol 13(7):1340–1354
6. Berman P, Hannenhalli S (1996) Fast sorting by reversal. In: Hirschberg DS, Myers EW (eds) Proceedings of 7th annual symposium combinatorial pattern matching (CPM96), Laguna Beach, June 1996. Lecture notes in computer science, vol 1075. Springer, pp 168–185
7. Caprara A (1997) Sorting by reversals is difficult. In: Proceedings of 1st conference on computational molecular biology (RECOMB97), Santa Fe. ACM, pp 75–83
8. Caprara A (1999) Sorting permutations by reversals and Eulerian cycle decompositions. SIAM J Discret Math 12(1):91–110
9. Hannenhalli S, Pevzner PA (1995) Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In: Proceedings of 27th annual symposium on theory of computing (STOC95), Las Vegas. ACM, pp 178–189
10. Kaplan H, Shamir R, Tarjan RE (1999) A faster and simpler algorithm for sorting signed permutations by reversals. SIAM J Comput 29(3):880–892. First appeared In: Proceedings of 8th annual symposium on discrete algorithms (SODA97), New Orleans. ACM, pp 344–351
11. Olmstead RG, Palmer JD (1994) Chloroplast DNA systematics: a review of methods and data analysis. Am J Bot 81:1205–1224
12. Palmer JD (1992) Chloroplast and mitochondrial genome evolution in land plants. In: Herrmann R (ed) Cell organelles. Springer, Vienna, pp 99–133
13. Rastegari B, Condon A (2005) Linear time algorithm for parsing RNA secondary structure. In: Casadio R, Myers E (eds) Proceedings of 5th workshop algorithms in bioinformatics (WABI'05), Mallorca. Lecture notes in computer science, vol 3692. Springer, Mallorca, Spain, pp 341–352
14. Raubeson LA, Jansen RK (1992) Chloroplast DNA evidence on the ancient evolutionary split in vascular land plants. Science 255:1697–1699
15. Setubal JC, Meidanis J (1997) Introduction to computational molecular biology. PWS, Boston
16. Tesler G (2002) Efficient algorithms for multichromosomal genome rearrangements. J Comput Syst Sci 63(5):587–609

Sorting Signed Permutations by Reversal (Reversal Sequence)

Eric Tannier
LBBE Biometry and Evolutionary Biology, INRIA Grenoble Rhône-Alpes, University of Lyon, Lyon, France

Keywords

Bioinformatics; Computational biology; Genome evolution; Genome rearrangements; Inversion distance; Sorting by inversions

Years and Authors of Summarized Original Work

2004; Tannier, Sagot

Problem Definition

A *signed permutation* π of size n is a permutation over $\{-n, \ldots, -1, 1 \ldots n\}$, where $\pi_{-i} = -\pi_i$ for all i. We note $\pi = (\pi_1, \ldots, \pi_n)$.

The *reversal* $\rho = \rho_{i,j} (1 \leq i \leq j \leq n)$ is an operation that reverses the order and flips the signs of the elements $\pi_i, \ldots, \pi_j$ in a permutation π:

$$\pi \cdot \rho = (\pi_1, \ldots, \pi_{i-1}, -\pi_j, \ldots, -\pi_i \pi_{j+1}, \ldots, \pi_n).$$

If $\rho_1, \ldots, \rho_k$ is a sequence of reversals, it is said to *sort* a permutation π if $\pi \cdots \rho_1 \cdots \rho_k = Id$, where $Id = (1, 2, \ldots, n)$ is the identity permutation. The length of a shortest sequence of reversals sorting π is called the *reversal distance* of π and is denoted by $d(\pi)$.

If the computation of $d(\pi)$ is solved in linear time [3] (see the entry ▸ Sorting Signed Permutations by Reversal (Reversal Distance)), the computation of a sequence $\rho^1, \ldots, \rho^k$ of size $k = d(\pi)$ that sorts π is more complicated, and no linear time algorithm is known so far. The best complexity is currently achieved by the subquadratic solution of Tannier and Sagot [17], which has later been improved by Tannier, Bergeron and Sagot [18], and Han [9].

Key Results

The $O(n^4)$ Self-Reduction

Recall there is a linear algorithm to compute the reversal distance thanks to the formula $d(\pi) = n + 1 - c(\pi) + t(\pi) + h(\pi) + f(\pi)$, where $c(\pi)$ is the number of cycles in the breakpoint graph and $h(\pi) + f(\pi)$ is computed from the unoriented components of the permutation (see the entry ▸ Sorting Signed Permutations by Reversal (Reversal Distance)). Once this is known, the self-reduction technique trivially computes a sequence of size $d(\pi)$: try every possible reversal ρ at one step, until you find one such that $d(\pi \cdot \rho) = d(\pi) - 1$. Such a reversal is called a *sorting reversal*. This necessitates $O(n)$ computations for every possible reversal. There are at most $n(n+1)/2 = O(n^2)$ reversals to try, so iterating this to find a sequence yields an $O(n^4)$ algorithm.

The first polynomial algorithm by Hannenhalli and Pevzner [10] was not achieving a better complexity, and the algorithmic study of finding the shortest sequences of reversals began its history.

The Quadratic Roof

All the published solutions for the computations of a sorting sequence are divided into two, following the division of the distance formula into its parameters: a first part computes a sequence of reversals so that the resulting permutation has no unoriented component, and a second part sorts all oriented components.

The first part was given its best solution by Kaplan, Shamir, and Tarjan [12], whose algorithm runs in linear time when coupled with the linear distance computation [3], and it is based on Hannenhalli and Pevzner's [10] early results.

The second part is the bottleneck of the whole procedure. At this point, if there is no unoriented component, the distance is $d(\pi) = n + 1 - c(\pi)$, so a sorting reversal is one that increases $c(\pi)$ and does not create unoriented components.

A reversal that increases $c(\pi)$ is called *oriented*. Finding an oriented reversal is an easy part: any two consecutive numbers that have different signs in the permutation define one. This can easily be done in linear time or sublinear with ad hoc data structures to maintain the permutation during the scenario. The hard part is to make sure it does not create unoriented components.

The quadratic solutions (see, e.g., the one of Kaplan, Shamir, and Tarjan [12]) are based on the linear recognition of sorting reversals. No better algorithm is known so far to recognize sorting reversals, and it seemed that a lower bound had been reached, as witnessed by a survey

of Ozery-Flato and Shamir [15] in which they wrote that "a central question in the study of genome rearrangements is whether one can obtain a subquadratic algorithm for sorting by reversals." This was obtained by Tannier and Sagot [17], who proved that the recognition of sorting reversal at each step is not necessary, but only the recognition of oriented reversals.

A Promising New but Still Quadratic Method

The algorithm is based on the following theorem, taken from [18]. A sequence of oriented reversals $\rho_1, \ldots, \rho_k$ is said to be *maximal* if there is no oriented reversal in $\pi \cdot \rho_1 \ldots \rho_k$. In particular a sorting sequence is maximal, but the converse is not true.

Theorem 1 *If S is a maximal but not a sorting sequence of oriented reversals for a permutation, then there exists a nonempty sequence S' of oriented reversals such that S may be split into two parts $S = S_1, S_2$, and S_1, S', S_2 is a sequence of oriented reversal.*

This allows to construct sequences of oriented reversals instead of sorting reversals, increase their size by adding reversals inside the sequence instead of at the end, and obtain a sorting sequence.

This algorithm, with a classical data structure to represent permutations (e.g., as an array), has still an $O(n^2)$ complexity, because at each step it has to test the presence of an oriented reversal and apply it to the permutation.

Composing with Data Structures

The slight modification of a data structure invented by Kaplan and Verbin [11] allows to pick and apply an oriented reversal in $O(\sqrt{n \log n})$, and using this, Tannier and Sagot's algorithm achieves $O(n^{3/2}\sqrt{\log n})$ time complexity.

Han [9] announced another data structure that allows to pick and apply an oriented reversal in $O(\sqrt{n})$ time, and integrating this to the algorithm can plausibly decrease the complexity of the overall method to $O(n^{3/2})$. Swenson et al. [16] gave an $O(n \log n)$ solution for picking oriented reversals, but their attempts of integrating it to the overall procedure seems to fail on worst cases.

Extensions

Once sorting by reversals has reached its best solutions, there are natural extensions guided by the main motivation for the problem in computational biology: sample among optimal solutions, and handle several permutations and more operations than just the reversal.

Counting optimal solutions is conjectured to be #P-complete [14], but sampling almost uniformly from the solution space is still open, and has been given a heuristic solution [14], including suboptimal solutions in the sample.

Algorithms to enumerate all sorting reversals at one step have also been worked out [4], which provides a way for enumeration. A structure of the solution space was proposed, but with a possibly exponential number of objects to enumerate [5].

The median problem consists in handling more than one permutation and is a particular case of the so-called small parsimony problem, which consists in reconstructing ancestral states in a phylogenetic context. Additional operations can be transpositions, duplications, or many others. Many generalizations and variants have been listed in a book on Combinatorics of Genome Rearrangements [8]. Almost all are NP-hard.

Applications

The motivation as well as the main application of this problem is in computational biology. Signed permutations are an adequate object to model the relative position and orientation of homologous segments of DNA in two species.

Reversal scenarios were used to test some evolutionary properties, like the propension of rearrangement to cut around the replication origin [1] or the fragility of certain genomic regions [2]. But evolutionary hypotheses can hardly be

tested from a single optimal solution; this would necessitate a better view of the solution space.

The gain of complexity for sorting by reversals inspired many other algorithmic works, and several problems in genome rearrangement found a better solution thanks to the subquadratic gain described here. But the computational difficulties of the problem (parameters $h(\pi)$ and $f(\pi)$, additional complexity for generating a scenario compared to the distance calculation, NP-completeness of every generalization with more operations, more permutations, more realistic models) lead most computational biologists to progressively abandon the reversal model for simpler ones (DCJ [19], SCJ [7]).

Sometimes heroic gains in complexity are worth for computer science but seem just like going a bit further in a dead end for applications. Research consists in breaking walls without always knowing if behind there is a space for a community to work in or another thicker wall.

Open Problems

Still there are a couple of questions that remain unsolved before closing (or reopening?) this entry:

- I conjecture that the "real" complexity of giving a reversal scenario is $O(n \log n)$. It is more or less what Swenson et al. [16] also claim, but without giving a full proof.
- Counting and sampling, even approximately, are open. I learned this interesting conjecture from Istvan Miklos: is it possible to walk in the entire space of sequences of sorting reversals by small transformations of scenarios, consisting at each step to change at most 4 reversals? This would be a first step to design an almost uniform sampler.

Experimental Results

To my knowledge the data structure that allows the subquadratic complexity described in this entry has never been implemented. The size of the data, as well as the limited possibilities of applications of handling only two genomes and a single optimal solution, makes the subquadratic version, while a good piece of algorithmics, not really worth for applications.

URL to Code

- There are a few old programs still able to give a sorting sequence of reversals: in San Diego http://grimm.ucsd.edu/GRIMM/, New Mexico www.cs.unm.edu/~moret/GRAPPA/, or Tel Aviv www.math.tau.ac.il/~rshamir/GR/ and more recent ones in Lyon http://doua.prabi.fr/software/luna or Bielefeld http://bibiserv.techfak.uni-bielefeld.de/dcj/welcome.html.
- The standard software for Bayesian sampling in the space of sorting sequences (including nonoptimal ones) is Badger http://bibiserv.techfak.uni-bielefeld.de/dcj/welcome.html, and there is also one biased to optimal solutions called DCJ2HP http://www.renyi.hu/~miklosi/DCJ2HP/ that uses a parallel tempering between DCJ solutions (easier to sample) and reversals solutions.

Cross-References

▸ Sorting Signed Permutations by Reversal (Reversal Distance)

Recommended Reading

1. Ajana Y, Lefebvre J-F, Tillier E, El-Mabrouk N (2002) Exploring the set of all minimal sequences of reversals – an application to test the replication-directed reversal hypothesis. In: Proceedings of the second workshop on algorithms in bioinformatics. Lecture notes in computer science, vol 2452. Springer, Berlin, pp 300–315
2. Attie O, Darling A, Yancopoulos Y (2011) The rise and fall of breakpoint reuse depending on genome resolution. BMC Bioinform 12(supp 9):S1
3. Bader DA, Moret BME, Yan M (2001) A linear-time algorithm for computing inversion distance between

signed permutations with an experimental study. J Comput Biol 8(5):483–491
4. Badr G, Swenson KM, Sankoff D (2011) Listing all parsimonious **reversal** sequences: new algorithms and perspectives. J Comput Biol 18:1201–1210
5. Braga MDV, Sagot MF, Scornavacca C, Tannier E (2008) Exploring the solution space of sorting by reversals with experiments and an application to evolution. IEEE-ACM Trans Comput Biol Bioinform 5:348–356
6. Darling AE, Miklós I, Ragan MA (2008) Dynamics of genome rearrangement in bacterial populations. PLoS Genet (7):e1000128
7. Feijão P, Meidanis J (2011) SCJ: a breakpoint-like distance that simplifies several rearrangement problems. IEEE/ACM Trans Comput Biol Bioinform 8(5):1318–1329
8. Fertin G, Labarre A, Rusu I, Tannier E, Vialette S (2009) Combinatorics of genome rearrangements. MIT, Cambridge
9. Han Y (2006) Improving the efficiency of sorting by reversals. In: Proceedings of the 2006 international conference on bioinformatics and computational biology, Las Vegas
10. Hannenhalli S, Pevzner PA (1999) Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. J ACM (JACM) 46:1–27
11. Kaplan H, Verbin E (2003) Efficient data structures and a new randomized approach for sorting signed permutations by reversals. In: Proceedings of CPM'03. Lecture notes in computer science, vol 2676. Springer, Berlin/Heidelberg, pp 170–185
12. Kaplan H, Shamir R, Tarjan RE (1999) Faster and simpler algorithm for sorting signed permutations by reversals. SIAM J Comput 29:880–892
13. Larget B, Simon DL, Kadane JB (2002) On a Bayesian approach to phylogenetic inference from animal mitochondrial genome arrangements (with discussion). J R Stat Soc B 64:681–693
14. Miklós I, Tannier E (2010) Bayesian sampling of genomic rearrangement scenarios via double cut and join. Bioinformatics 26:3012–3019
15. Ozery-Flato M, Shamir R (2003) Two notes on genome rearrangement. J Bioinform Comput Biol 1:71–94
16. Swenson KM, Rajan V, Lin Y, Moret BME (2010) Sorting signed permutations by inversions in $O(n \log n)$ time. J Comput Biol 17:489–501
17. Tannier E, Sagot M-F (2004) Sorting by reversals in subquadratic time. In: Proceedings of CPM'04. Lecture notes in computer science, vol 3109. Springer, Berlin/Heidelberg, pp 1–13
18. Tannier E, Bergeron A, Sagot M-F (2006) Advances on sorting by reversals. Discret Appl Math 155:881–888
19. Yancopoulos S, Attie O, Friedberg R (2005) Efficient sorting of genomic permutations by translocation, inversion and block interchange. Bioinformatics 21:3340–3346

Spanning Trees with Low Average Stretch

Ittai Abraham[1] and Ofer Neiman[2]
[1]Microsoft Research, Silicon Valley, Palo Alto, CA, USA
[2]Department of Computer Science, Ben-Gurion University of the Negev, Beer Sheva, Israel

Keywords

Embedding; Spanning tree; Stretch

Years and Authors of Summarized Original Work

2012; Abraham, Neiman

Problem Definition

Let $G = (V, E)$ be an undirected graph, with nonnegative weights on the edges $w : E \to \mathbb{R}_+$. Let d_G be the shortest-path metric on G, with respect to the weights. For a spanning (subgraph) tree T of G, define the stretch of an edge $\{u, v\} \in E$ in T as $\text{stretch}_T(u, v) = \frac{d_T(u,v)}{d_G(u,v)}$ and the average stretch as

$$\text{avg} - \text{stretch}_T(G) = \frac{1}{|E|} \sum_{e \in E} \text{stretch}_T(e) .$$

We shall consider the problem of finding a tree T whose average stretch is small. We also study the problem of finding a distribution over spanning trees, such that for all $e \in E$, $\mathbb{E}_T[\text{stretch}_T(e)]$ is small.

Key Results

Low-stretch spanning trees were first studied by [3], who showed that any graph on n vertices has a spanning tree with average stretch $2^{O(\sqrt{\log n \log\log n})}$ and showed a family of graphs

that requires $\Omega(\log n)$ average stretch. Their result was substantially improved by [7], who showed an upper bound of $O(\log^2 n \log\log n)$, and later [1] improved this to a near optimal $\tilde{O}(\log n)$.

The main result discussed here is from [2]:

Theorem 1 *For any graph G with n vertices and m edges, there is a deterministic algorithm that constructs a spanning tree T, such that* $\text{avg} - \text{stretch}_T(G) \leq O(\log n \log\log n)$. *The running time of the algorithm is* $O(m \log n \log\log n)$.

We also show an efficient algorithm to sample from a distribution over spanning trees, such that the expected stretch of any edge is bounded by $O(\log n \log\log n \log\log\log n)$.

Applications

An important problem in algorithm design is obtaining fast algorithms for solving linear systems. For many applications, the matrix is sparse, and while little is known for general sparse matrices, the case of symmetric diagonally dominant (SDD) matrices has received a lot of attention recently. In a seminal sequence of results, Spielman and Teng [12] showed a near-linear time solver for this important case. This solver has proven a powerful algorithmic tool and is used to calculate eigenvalues, obtain spectral graph sparsifiers [11], and approximate maximum flow [6] and many other applications. A basic step in solving these systems $Ax = b$ is combinatorial preconditioning. If one uses the Laplacian matrix corresponding to a spanning tree (and a few extra edges) of the graph whose Laplacian matrix is A, then the condition number depends on the total stretch of the tree. This will improve the run-time of iterative methods, such as conjugate gradient or Chebyshev iterations. See [9, 10] for the latest progress on this direction. In this work we show that one can construct such a spanning tree with both run-time and total stretch bounded by $O(m \log n \log\log n)$.

Probabilistic embedding into trees, introduced by [4], has been a successful paradigm in algorithm design. Many hard optimization problems on graphs can be reduced, via embedding, to a similar problem on a tree, which is often considerably easier. This framework can be applied to approximation algorithms, online algorithms, network design, and other settings. Some of the notable examples are metrical task system, buy-at-bulk network design, the k-server problem, group Steiner tree, etc. An asymptotical optimal result of expected $O(\log n)$ distortion for probabilistic embedding into trees was given by [8]. The trees in the support of the FRT distribution are not subgraphs of the input graph and may contain Steiner nodes and new edges. While this is fine for most applications, there are some that must have trees which are subgraphs, such as minimum cost communication spanning tree: Given a weighted graph $G = (V, E)$ and a requirement matrix $R = (r_{uv})$, the objective is to find a spanning tree T that minimizes $\sum_{u,v \in V} r_{uv} \cdot d_T(u, v)$. Our result implies a $\tilde{O}(\log n)$ approximation.

Petal Decomposition

A basic tool that is often used in constructing tree metrics and spanning trees with low stretch is *sparse graph decomposition*. The idea is to partition the graph into small diameter pieces, such that few edges are cut. Each cluster of the decomposition is partitioned recursively, which yields a hierarchical decomposition. Creating a tree recursively on each cluster of the decomposition, and connecting these in a tree structure, will yield a spanning tree of the graph. The edges cut by the decomposition are potentially stretched by a factor proportional to the diameter of the created tree. The construction has to balance between these two goals: cut a small number of edges and maintain small diameter in the created tree.

One of the main difficulties in such a spanning tree construction is that the radius (The radius of a graph is the maximal distance from a designated center.) may increase by a small factor at every application of the decomposition, which

translates to increased stretch. If we drop the requirement that the tree is a *spanning tree* of the graph and just require a tree metric, then this difficulty does not appear, and indeed, optimal $\Theta(\log n)$ bound is known on the average stretch [5,8]. Our *petal decomposition* allows essentially optimal control on the radius increase of the spanning tree; it increases by at most a factor of 4 over all the recursion levels.

Highways

One of the components in the decomposition scheme is highways. Each cluster $X \subseteq V$ in our decomposition scheme has a designated center $x_0 \in X$ and a "target" $t \in X$. It is guaranteed that the shortest path from x_0 to t will be fully contained in the *final* spanning tree T. This path is called the petal's highway. Intuitively, the highway will provide short paths from the center x_0 to many of the points in the cluster.

Cones and Petals

A cone is a generalization of a ball; the notion of cones was introduced in [7] and was used also in [1] for low-stretch spanning trees. Informally, a cone $C(t,r)$ of radius r centered at t (with respect to the cluster center x_0) contains all the points $z \in X$ such that $d(z,t) + d(t,x_0) \leq d(z,x_0) + r$ (here d is the shortest-path metric on X). In other words, the cone contains all the points for which the path to x_0 through t is not much longer than the direct shortest path to x_0. The parameter r is a bound on the radius increase in the current decomposition.

One way to define a petal is as a *union of cones*. The petal $P(t,r)$ around a target t with radius r is defined as $\bigcup_{0 \leq k \leq r} C(p_k, k/2)$, where p_k is the point of distance $r-k$ from t on the shortest path from t to x_0. The center of the petal is defined as $x = p_0$, and the path from x to t is the petal's highway. The `petal-decomposition` algorithm iteratively picks an arbitrary target of distance at least $3\Delta/4$ (where Δ is the radius of X) away from x_0, generates a petal for it, and removes the petal from the graph. When there are no longer such points, the remaining points will form the central cluster (the stigma). The first petal requires extra care in its target choice, as it may contain the designated target of the cluster, which implies we cannot allow the shortest path to this target to be cut by this or subsequent petals. The radii of the petals are chosen by a region-growing argument that cuts few edges, where the length of the possible range for the radius is $\approx \Delta$. This is in contrast with the previous work, where in order to give an appropriate bound on the radius increase, the range was much smaller than Δ, which immediately translates to a loss in the stretch. The precise method for choosing r is essentially given in [7], and we also give a randomized version similar in spirit to the one in [1].

Fast Petal Construction

The alternative way to define petals and cones is as balls in an appropriately defined *directed* graph created from G. This suggests that we can use a variant of Dijkstra to compute a petal in nearly linear time in the sum of degrees of its vertices. Let $\tilde{G} = (V, A, \bar{w})$ be the weighted directed graph induced by adding the two directed edges $(u,v), (v,u) \in A$ for each $\{u,v\} \in E$ and setting $\bar{w}(u,v) = d(u,v) - (d(v,x_0) - d(u,x_0))$. The cone $C(t,r)$ is simply the ball around t of radius r in $\tilde{G}$. The petal $P(t,r)$ is the ball around t of radius $r/2$ in $\tilde{G}$ with one change: the weight of each edge on the path from t to $x = p_0$ is changed to be $1/2$ of its original weight (i.e., $1/2$ of its weight in G).

Ideas in the Analysis

Informally, the crucial property of a petal and its highway is the following: Assume $z \in P(t,r)$, and $P_{x_0 z}$ is the shortest path from the original center x_0 to z. By forming the petal, we remove all edges between $P(t,r)$ and $X \setminus P(t,r)$ except for the edge from the petal center x toward x_0. Hence, any path from x_0 to z must go through the petal center x. If the new shortest path $P'_{x_0 z}$ (after forming the petal) is (additively) $k/2$ longer than the length of $P_{x_0 z}$, then $z \in C(p_k, k/2)$ and so $P'_{x_0 z}$ will contain part of the new petal's highway of length at least k. Such a property could allow the following wishful thinking: Suppose that in each iteration we increase the distance of a point to the center by at most α but also mark a new

portion of the path of length 2α as edges that are guaranteed to appear in the final tree (part of a highway). In such a case, it is easy to see that the final path will have stretch at most 2: If the original distance was b, once the total increase is b, we have marked $2b$ – all of the path – as a highway that will appear in the tree. Unfortunately, the path from x to z in the final tree may not use the prescribed highway of the parent cluster so the above "wishful thinking" argument does not work.

The key algorithmic idea to alleviate this problem is to decrease the weight of an edge by half when it becomes part of a highway (we ensure that this happens at most once for every edge). This reweighting signals later iterations to use the prescribed highway, as this must remain the shortest path. We maintain the invariant that in every cluster, the highway edges are the only cluster edges which have been reweighted. Now, in every petal (except for maybe the first), we create a *new* petal highway when we form $P(t, r)$. For any $z \in P(t, r)$, *the length of the path from* x_0 *to* z *does not increase at all* (after reweighting the highway): For some $k \leq r$, it increased by at most $k/2$, but a highway length of at least k was reduced by $1/2$.

We have to take care of radius increase generated by the very first petal as well, where it could be that no new highway is created (this petal's highway may be a part of the highway of the original cluster). In this case, we use the fact that the path from x_0 to x_1 (the center of the first petal) must also be on the highway of the original cluster and that its length is at least $\Delta/2$. This implies that even though we may have increased the radius, at least half of the path is guaranteed not to increase ever again. We use a subtle inductive argument to make this intuition precise, and in fact we lose a factor of 2 for each of these cases, so the maximal increase is by a factor of 4.

Recommended Reading

1. Abraham I, Bartal Y, Neiman O (2008) Nearly tight low stretch spanning trees. In: FOCS '08: Proceedings of the 2008 49th annual IEEE symposium on foundations of computer science, Philadelphia. IEEE Computer Society, Washington, DC, pp 781–790
2. Abraham I, Neiman O (2012) Using petal-decompositions to build a low stretch spanning tree. In: Proceedings of the forty-fourth annual ACM symposium on theory of computing, STOC '12, New York. ACM, pp 395–406
3. Alon N, Karp RM, Peleg D, West D (1995) A graph-theoretic game and its application to the k-server problem. SIAM J Comput 24(1):78–100
4. Bartal Y (1996) Probabilistic approximation of metric spaces and its algorithmic applications. In: Proceedings of the 37th annual symposium on foundations of computer science, Burlington. IEEE Computer Society, Washington, DC, pp 184–
5. Bartal Y (2004) Graph decomposition lemmas and their role in metric embedding methods. In: Albers S, Radzik T (eds) Proceedings of the 12th annual European symposium on algorithms – ESA 2004, Bergen, 14–17 Sept 2004. Volume 3221 of Lecture notes in computer science. Springer, pp 89–97
6. Christiano P, Kelner JA, Madry A, Spielman DA, Teng S-H (2011) Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected graphs. In: Proceedings of the 43rd annual ACM symposium on theory of computing, STOC '11, San Jose. ACM, New York, pp 273–282
7. Elkin M, Emek Y, Spielman DA, Teng S-H (2005) Lower-stretch spanning trees. In: Proceedings of the thirty-seventh annual ACM symposium on theory of computing, STOC '05, Baltimore. ACM, New York, pp 494–503
8. Fakcharoenphol J, Rao S, Talwar K (2003) A tight bound on approximating arbitrary metrics by tree metrics. In: Proceedings of the thirty-fifth annual ACM symposium on theory of computing, STOC '03, San Diego. ACM, New York, pp 448–455
9. Koutis I, Miller GL, Peng R (2010) Approaching optimality for solving SDD linear systems. In: 51th annual IEEE symposium on foundations of computer science, 23–26 Oct 2010, Las Vegas, pp 235–244
10. Koutis I, Miller GL, Peng R (2011) A nearly $O(m \log n)$ time solver for SDD linear systems. In: 52th annual IEEE symposium on foundations of computer science, Palm Springs
11. Spielman DA, Srivastava N (2008) Graph sparsification by effective resistances. In: Proceedings of the 40th annual ACM symposium on theory of computing, STOC '08, Victoria. ACM, New York, pp 563–568
12. Spielman DA, Teng S-H (2004) Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In: Proceedings of the thirty-sixth annual ACM symposium on theory of computing, STOC '04, Chicago. ACM, New York, pp 81–90

Sparse Fourier Transform

Eric Price
Department of Computer Science, The University of Texas, Austin, TX, USA

Keywords

Compressed sensing; Sparse recovery

Years and Authors of Summarized Original Work

2012; Hassanieh, Indyk, Katabi, Price

Problem Definition

Suppose that we have access to a vector $x \in \mathbb{C}^n$. How much time does it take to compute its Fourier transform $\hat{x}$? One can do this with the Fast Fourier Transform (FFT) in $O(n \log n)$ time. But can we do better?

We do not know the answer in general, but some classes of algorithms cannot do better [1, 20] and certainly one cannot do better than $O(n)$ time for arbitrary signals x. But the Fourier transform is ubiquitous in signal processing, appearing in compression of audio, images, and video, in manipulation of audio, and in recovery of radio or MRI signals, so we would really like to do better. If we cannot improve on the FFT in general, then perhaps we can for the signals commonly seen in these applications. To do this, we need some notion for how the signals we typically see are "easier" than arbitrary ones.

One such notion is *sparsity*. The main reason to use the Fourier transform in compression is because it concentrates the energy of the signal into a few large (or "heavy") coordinates and many small ones; signals with such concentrated coordinates are called *sparse*. One can then throw out the small coordinates and only store the heavy ones; this is the main principle behind lossy compression such as MP3 or JPEG. In fact, in all the applications discussed in the previous paragraph, the signals typically have an approximately sparse Fourier transform. This brings us to the problem described in this entry: can we speed up the Fourier transform for signals when the result is approximately sparse?

Moreover, as with lossy compression, we often only care about the heavy coordinates and are willing to tolerate an error proportional to the energy in the small coordinates. This relaxation will allow us to compute the sparse Fourier transform in *sublinear* time.

Formal Definition

The discrete Fourier transform $\hat{x} \in \mathbb{C}^n$ of a vector $x \in \mathbb{C}^n$ is given by

$$\hat{x}_j = \sum_{i=1}^{n} \omega^{ij} \text{ for } \omega = e^{2\pi \mathbf{i}/n}$$

We say that $\hat{x}$ is *exactly* k-sparse if it has at most k nonzero coordinates, i.e., $|\text{supp}(x)| \leq k$. We say that $\hat{x}$ is *approximately* k-sparse if most of the energy is contained in the heaviest k coordinates, in particular

$$\text{Err}_k(x) := \min_{k\text{-sparse } \hat{y}} \|\hat{x} - \hat{y}\|_2$$

is small relative to $\|\hat{x}\|_2$. A sparse Fourier transform algorithm can access $x \in \mathbb{C}^n$ in arbitrary positions and outputs a vector $\hat{x}'$ such that

$$\|\hat{x} - \hat{x}'\|_2 \leq C \,\text{Err}_k(x) + \delta \|\hat{x}\|_2 \qquad (1)$$

for some approximation factor $C > 1$ and $\delta \ll 1$. An algorithm for the exactly sparse case would do this for $C = \infty$, while robust algorithms can achieve $C = O(1)$ or even $C = 1 + \varepsilon$. The algorithms we will discuss will feature a logarithmic dependence on $1/\delta$, so one typically sets $\delta = 1/\text{poly}(n)$, and for typical signals, the right-hand side of (1) will be dominated by the $C\,\text{Err}_k(x)$ term; we will assume this for the rest of the entry.

We would like to optimize both the sample complexity – the number of positions of x that are accessed by the algorithm – and the

running time. Optimizing sample complexity is important for applications such as spectrum sensing or MRIs, which do not have the input x in memory but must sample it at some expense.

We also allow the algorithm to be randomized and to fail with some small probability p. For simplicity we set p to a small constant; for any algorithm one can amplify this probability with a $O(\log \frac{1}{p})$ overhead in sample complexity and time. It is an open question whether the algorithms that achieve the best known time and sample complexities can be modified to avoid this overhead.

Related Work

The modern research on sparse Fourier transforms is closely related to work on sparse recovery from general linear measurements. In this problem, one would like to (approximately) recover an (approximately) sparse vector x from linear measurements Ax for some "measurement" matrix A with fewer rows than columns. The sparse Fourier transform is precisely this where A is a subset of rows of the inverse Fourier matrix.

Broadly speaking, there are two conceptual classes of algorithms and results for the general linear measurement setting. The first class, often called *compressed sensing* and first studied in [2, 3, 7], generally (1) involves independent random linear measurements,;(2) shows with high probability, the measurement matrix gives good recovery *for all* vectors x; (3) optimizes the sample complexity but not the running time, which is superlinear or polynomial in n; and (4) give algorithms that work for general classes of measurements and work for both random Gaussian and random Fourier matrices at the same time. These papers often refer to properties like the restricted isometry property that measurement matrices may have and use either convex optimization (e.g., L1 minimization or the LASSO) or iterative greedy methods (e.g., IHT or CoSaMP) to perform the recovery.

The second class, more often called *sparse recovery*, is largely an outgrowth of the streaming algorithms literature [4, 5]. These results generally (1) involve more structured linear measurements that use randomness and also have dependencies among the samples; (2) show *for each* vector x that, with high probability, the measurement matrix gives good recovery; (3) optimize both the sample complexity and the running time, so both may be sublinear in n; and (4) give algorithms that are closely connected to the measurement matrix and would not work for matrices with different structure. These papers often construct the matrix to emulate hash tables and use medians to perform robust recovery.

These statements are generalizations, and not every algorithm matches the trend in all four ways, but they hold more often than not. Our algorithm falls in the second class, which for Fourier measurements can achieve both better sample complexity and better running time than algorithms in the first class.

There's a much older collection of algorithms that can do sparse Fourier transforms in the exact setting when $|\text{supp}(\hat{x})| \leq k$. These include Prony's method from 1795, the matrix pencil method, and Berlekamp-Massey syndrome decoding. These can achieve the optimal sample complexity of $2k$ and recovery time poly(k) (down to $O(k^2 + k \log^c \log n)$ [8]). Additionally, they use a deterministic set of samples and work for all vectors x. However, it is not known how to make the techniques in these algorithms robust to approximately sparse signals, so they do not apply to the signals appearing in typical applications.

Noise-tolerant sparse Fourier transforms were first studied over the Boolean cube, also known as the Hadamard transform. In this setting, Goldreich and Levin [12, 18] showed how to get $O(k \log(n/k))$ samples and $O(k \log^c n)$ time, which is essentially optimal. Mansour [19] extended this to the $\mathbb{C}^n$ setting that we consider in this entry but with more than k^2 sample complexity. Over the next couple decades, a number of subsequent works, including [9, 10, 13, 14, 16], have improved our understanding of the $\mathbb{C}^n$ setting.

Key Results

At present, the two best sparse Fourier transform algorithms are [13], which is fastest at $O(k \log(n/k) \log n)$ time and sample complexity, and [14], which has nearly optimal $O(k \log n)$ sample complexity at the cost of $\tilde{O}(n)$ running time. These works build on [9, 10, 17].

We know that the optimal *nonadaptive* sample complexity – that is, among algorithms that choose the sample set Ω independently of the vector x – is $\Omega(k \log(n/k))$ [6], which matches [14] for $k < n^{0.99}$. One could imagine constructing an algorithm that uses adaptive samples, where one uses the first few samples to decide where to look in future samples. In the general sparse recovery setting, this adaptivity can lead to significant improvements [15], but we know that $\Omega(k \log(n/k)/\log\log n)$ Fourier samples remain necessary in the adaptive setting [13].

Algorithm Overview

At a high level, sparse recovery algorithms are built in three stages: one-sparse recovery, where we solve the problem for $k = 1$; partial k-sparse recovery, where we find and estimate *most* (say, 90 %) of the heavy coordinates or of the energy; and full k-sparse recovery, where we get a good approximation to the entire signal and achieve (1). Each stage uses the previous as (nearly) a black box. This architecture generally holds for the class of "sparse recovery" algorithms; in the sparse Fourier transform setting, the pieces change, but the architecture does not. We will go through each in turn.

One-Sparse Recovery

Let us consider the one-sparse setting for $C = O(1)$. We have access to $x_j = v\omega^{i^* j} + g_j$ for some "signal" $(v, i^*) \in \mathbb{C} \times [n]$ and "noise" $g \in \mathbb{C}^n$ with $\|g\|_2 \le c|v|\sqrt{n}$ for a sufficiently small constant c. To satisfy (1), we would like to find i^* exactly and find v to within $O(\|g\|/\sqrt{n})$.

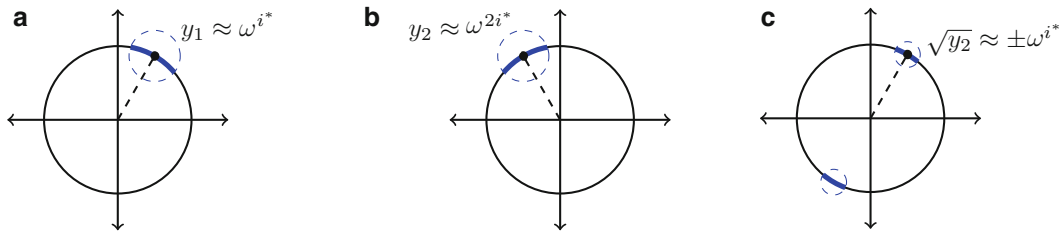
The tricky bit is to find i^*; once we know i^*, then $x_j \omega^{-i^* j}$ is a good estimator of v. In particular, for a random $j \in [n]$, we have $\mathbb{E}_j |x_j \omega^{-i^* j} - v|^2 = \|g\|^2/n$, so taking the median of several such estimates will have $O(\|g\|/\sqrt{n})$ error with large probability. So that just leaves us to find i^*.

As a first step, consider for a fixed $a \in [n]$ looking at the random variable

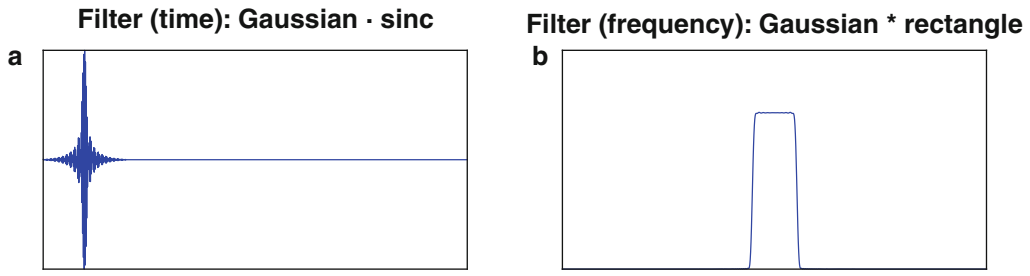
$$y_a := x_{a+j}/x_j \approx \omega^{i^* a}$$

as a distribution over random $j \in [n]$, where addition of indices is taken modulo n. This allows us to remove the influence of v and focus on i^*. We can show that $y_a - \omega^{i^* a} < O(c)$ with large (say, 3/4) probability. Suppose this were instead true with probability 1.

By knowing $\omega^{i^* a}$ to within $O(c)$, we know $i^* a \bmod n$ to with $\pm O(cn)$. For small enough c, this is within $\pm n/4$. Then we could look at y_1 to learn i^* to within $\pm n/4$, y_2 to refine the estimate to $\pm n/8$, and y_4 to refine to $\pm n/16$, until we identify i^* using $\log n$ different y_a. This is illustrated in Fig. 1.



Sparse Fourier Transform, Fig. 1 The first two steps of estimating i^* using y_1 and y_2. Using y_1 we can identify i^* to an $O(cn)$ size region. With y_2 we learn $2i^* \bmod n$ to within $O(cn)$, which tells us that i^* is within one of two antipodal regions of half the size. Based on y_1, we can throw out the spurious region and narrow our estimate of i^* (**a**) Error in y_1. (**b**) Error in y_2. (**c**) Set of ω^{i^*} consistent with y_2



Sparse Fourier Transform, Fig. 2 Filters used in [13]. (**a**) In time domain: $O(k \log n)$ sparse. (**b**) In frequency domain: width $O(n/k)$ rectangle

In reality y_a has a small constant chance of failure at each stage. One could fix this by taking $O(\log \log n)$ different samples of y_a at each stage and using the median, which would give an algorithm with $O(\log n \log \log n)$ sample complexity and time. An alternative, as used in [13], is to learn i^* in chunks of $O(\log \log n)$ bits at a time, which gets the optimal $O(\log n)$ sample complexity using $O(\log^{1.1} n)$ running time.

Partial *k*-Sparse Recovery

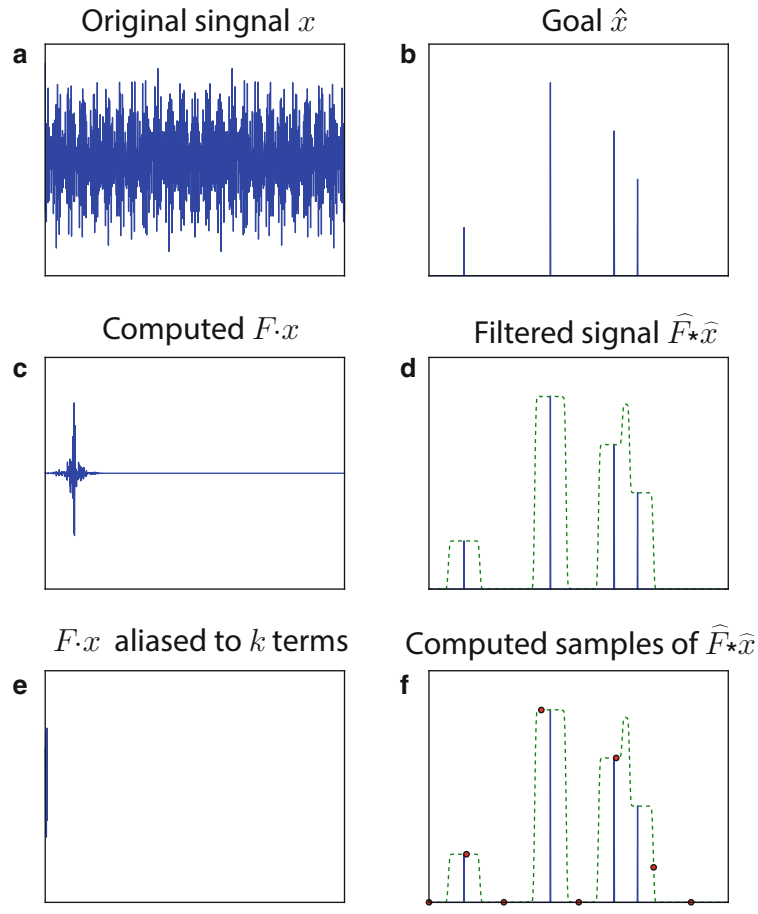
The goal of partial k-sparse recovery is to find *most* of the heavy coordinates of $\hat{x}$. The general idea is to "hash" the coordinates randomly into $B = O(k)$ bins in a way that lets us take measurements of the signal restricted to frequencies within each bin. By taking the measurements corresponding to the one-sparse recovery algorithm, we recover frequencies that are alone in their bin. This will happen with a large constant (say, 90 %) probability for each heavy frequency, so we recover *most* of the heavy frequencies well.

To see how this is done, we start with a deterministic way of hashing the frequencies into bins and then show how to randomize it. Hashing is based on filters that are sparse in both time and frequency domain. The filter F is designed to be as close as possible to a rectangular filter in frequency domain while still being sparse in frequency domain. Figure 2 demonstrates the filter used in [13], where F is a sinc function times a (truncated) Gaussian with support size $O(k \log n)$. In frequency domain, $\hat{F}$ approximates a rectangle of width $O(n/k)$, matching it up to a small transition region between the passband and the stopband and with $1/n^c$ error inside the passband and stopband.

Using these filters, Fig. 3 demonstrates a method for learning information about the signal. Given the signal x, we compute the $O(k \log n)$-size vector $F \cdot x$. We then "alias" it down to $B = O(k)$ elements – adding up terms $1, B+1, 2B+1, \ldots$ – and take the B-dimensional DFT. This lets us compute the red points in Fig. 3f in $O(k \log n + B \log B) = O(k \log n)$ time. The red points are B evenly spaced samples of $\hat{x} * \hat{F}$.

We can think of the ith red point in a different way. The ith red point is the sum of all the entries of $\hat{x} \cdot \text{shift}(\hat{F}, in/B)$, where $\text{shift}(\hat{F}, in/B)$ denotes shifting $\hat{F}$ to the right by in/B. This equals the zeroth time domain coefficient of the vector with Fourier coefficients given by $\hat{x} \cdot \text{shift}(\hat{F}, in/B)$. And if our algorithm looks not at $y_j = F_j x_j$ but $y_j^{(a)} = F_j x_{j+a}$ when computing the red points, then the ith red point will equal the ath time domain coefficient of the vector with Fourier coefficients given by $\hat{x} \cdot \text{shift}(\hat{F}, in/B)$. This lets us sample from the time domain representation of the vectors with Fourier coefficients given by $\hat{x} \cdot \text{shift}(\hat{F}, in/B)$ for $i \in [B]$. It takes $O(k \log n)$ time to get these samples, for $O(\log n)$ overhead (in time and samples) per "effective" sample.

Now, we simply choose our samples a from the distribution requested by the one-sparse recovery algorithm. In every bucket for which $\hat{x} \cdot \text{shift}(\hat{F}, in/B)$ is one-sparse, this procedure will let us recover the heavy frequency. Because the different shifts of $\hat{F}$ give B different buckets,



Sparse Fourier Transform, Fig. 3 The algorithm for hashing used in [13]. For simplicity, the illustrations do not include noise. (**a**) The signal in time domain. (**b**) Corresponds to this signal in frequency domain. (**c**) We observe $\boldsymbol{F} \cdot \boldsymbol{x}$ for a sparse $\boldsymbol{F}$. (**d**) Which has the dashed $\boldsymbol{n}$-dimensional DFT. (**e**) We alias from $\boldsymbol{O(k \log n)}$ terms to $\boldsymbol{O(k)}$. (**f**) And compute the $\boldsymbol{O(k)}$-dimensional DFT (*dots*)

if the frequencies were randomly distributed, this technique would get us partial sparse recovery. The one-sparse recovery algorithm only takes $O(\log(n/k))$ samples because each frequency is known to lie within an $n/B = O(n/k)$ size region; hence the overall method takes $O(k \log n \log(n/k))$ time and samples.

We would like the algorithm to work for arbitrary input signals, so we need a way of randomizing the frequencies. To do this, we further refine the algorithm to choose a random $\sigma, b \in [n]$ with σ relatively prime to n. Then we have the algorithm look at $y_j^{(a)} = F_j x_{\sigma(j+a)} \omega^{-\sigma j b}$. The effect of σ, b is to apply an hash function $j \to \sigma^{-1} j + b$ in frequency domain; this is approximately pairwise independent, so the frequencies become effectively randomly distributed. Each frequency then has a good chance of landing alone in its bucket, so we can recover most frequencies in $O(k \log(n/k) \log n)$ time and samples.

Full *k*-Sparse Recovery

Once we have partial k-sparse recovery, one can naively achieve full k-sparse recovery by repeating the algorithm $O(\log k)$ times. Since each heavy frequency is recovered with 90 % probability in each stage, the median of all the estimations will recover all the heavy frequencies – and in fact achieve (1) – with high probability. This method is simple but loses a $\log k$ factor in running time and sample complexity, which more intricate techniques can avoid.

One such technique, used in [13] and based off [11], is to use smaller and smaller k in successive iterations. Once we have performed partial sparse recovery on $\hat{x}$ to get $\hat{x}^{(1)}$ that contains

90 % of the heavy hitters, we can then perform sparse recovery on the residual $\hat{x} - \hat{x}^{(1)}$. The residual is then roughly $k/10$-sparse, so we run a partial $k/10$-sparse recovery algorithm in the second stage that is much faster than in the first stage. Similar geometric decay happens in later stages, so the total time spent will be dominated by the first stage. This gives $O(k \log(n/k) \log n)$ time and sample complexity for the problem.

Recommended Reading

1. Ailon N (2014) An n\log n lower bound for Fourier transform computation in the well conditioned model. CoRR. abs/1403.1307, http://arxiv.org/abs/1403.1307
2. Candes E, Tao T (2006) Near optimal signal recovery from random projections: universal encoding strategies. IEEE Trans Info Theory
3. Candes E, Romberg J, Tao T (2006) Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information. IEEE Trans Info Theory 52:489–509
4. Charikar M, Chen K, Farach-Colton M (2002) Finding frequent items in data streams. In: ICALP
5. Cormode G, Muthukrishnan S (2004) Improved data stream summaries: the count-min sketch and its applications. In: LATIN
6. Do Ba K, Indyk P, Price E, Woodruff D (2010) Lower bounds for sparse recovery. In: SODA
7. Donoho D (2006) Compressed sensing. IEEE Trans Info Theory 52(4):1289–1306
8. Ghazi B, Hassanieh H, Indyk P, Katabi D, Price E, Shi L (2013) Sample-optimal average-case sparse Fourier transform in two dimensions. In: Allerton
9. Gilbert A, Guha S, Indyk P, Muthukrishnan M, Strauss M (2002) Near-optimal sparse Fourier representations via sampling. In: STOC
10. Gilbert A, Muthukrishnan M, Strauss M (2005) Improved time bounds for near-optimal space Fourier representations. In: SPIE conference on wavelets
11. Gilbert AC, Li Y, Porat E, Strauss MJ (2010) Approximate sparse recovery: optimizing time and measurements. In: STOC, pp 475–484
12. Goldreich O, Levin L (1989) A hard-corepredicate for allone-way functions. In: STOC, pp 25–32
13. Hassanieh H, Indyk P, Katabi D, Price E (2012) Nearly optimal sparse Fourier transform. In: STOC
14. Indyk P, Kapralov M (2014) Sample-optimal Fourier sampling in any constant dimension. In: 2014 IEEE 55th annual symposium on foundations of computer science (FOCS). IEEE, pp 514–523
15. Indyk P, Price E, Woodruff D (2011) On the power of adaptivity in sparse recovery. In: FOCS
16. Iwen MA (2010) Combinatorial sublinear-time Fourier algorithms. Found Comput Math 10:303–338
17. Kushilevitz E, Mansour Y (1991) Learning decision trees using the Fourier spectrum. In: STOC
18. Levin L (1993) Randomness and non-determinism. J Symb Logic 58(3):1102–1103
19. Mansour Y (1992) Randomized interpolation and approximation of sparse polynomials. In: ICALP
20. Morgenstern J (1973) Note on a lower bound on the linear complexity of the fast Fourier transform. J ACM (JACM) 20(2):305–306

Sparse Graph Spanners

Michael Elkin
Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel

Keywords

$(1+\epsilon, \beta)$-spanners; Almost additive spanners

Years and Authors of Summarized Original Work

2004; Elkin, Peleg

Problem Definition

For a pair of numbers α, β, $\alpha \geq 1$, $\beta \geq 0$, a subgraph $G' = (V, H)$ of an unweighted undirected graph $G = (V, E)$, $H \subseteq E$, is an (α, β)-spanner of G if for every pair of vertices $u, w \in V$, $\text{dist}_{G'}(u, w) \leq \alpha \cdot \text{dist}_G(u, w) + \beta$, where $\text{dist}_G(u, w)$ stands for the distance between u and w in G. It is desirable to show that for every n-vertex graph there exists a sparse (α, β)-spanner with as small values of α and β as possible. The problem is to determine asymptotic tradeoffs between α and β on one hand, and the sparsity of the spanner on the other.

Key Results

The main result of Elkin and Peleg [8] establishes the existence and efficient constructibility of

$(1+\epsilon,\beta)$-spanners of size $O(\beta n^{1+1/\kappa})$ for every n-vertex graph G, where $\beta=\beta(\epsilon,\kappa)$ is constant whenever κ and ϵ are. The specific dependence of β on κ and ϵ is $\beta(\kappa,\epsilon)=\kappa^{\log\log\kappa-\log\epsilon}$.

An important ingredient of the construction of [8] is a partition of the graph G into regions of small diameter in such a way that the supergraph induced by these regions is sparse. The study of such partitions was initiated by Awerbuch [3], that used them for network synchronization. Peleg and Schäffer [10] were the first to employ such partitions for constructing spanners. Specifically, they constructed $(O(\kappa),1)$-spanners with $O(n^{1+1/\kappa})$ edges. Althofer et al. [2] provided an alternative proof of the result of Peleg and Schäffer that uses an elegant greedy argument. This argument also enabled Althofer et al. to extend the result to weighted graphs, to improve the constant hidden by the O-notation in the result of Peleg and Schäffer, and to obtain related results for planar graphs.

Applications

Efficient algorithms for computing sparse $(1+\epsilon,\beta)$-spanners were devised in [7] and [13]. The algorithm of [7] was used in [7, 9, 12] for computing almost shortest paths in centralized, distributed, streaming, and dynamic centralized models of computations. The basic approach used in these results is to construct a sparse spanner, and then to compute exact shortest paths on the constructed spanner. The sparsity of the latter guarantees that the computation of shortest paths in the spanner is far more efficient than in the original graph.

Open Problems

The main open question is whether it is possible to achieve similar results with $\epsilon=0$. More formally, the question is: Is it true that for any $\kappa\geq 1$ and any n-vertex graph G there exists $(1,\beta(\kappa))$-spanner of G with $O(n^{1+1/\kappa})$ edges? This question was answered in affirmitive for κ equal to 2, 5/2, and 3 [1, 4-6, 8]. Some lower bounds were recently proved by Woodruff [14].

A less challenging problem is to improve the dependence of β on ϵ and κ. Some progress in this direction was achieved by Thorup and Zwick [13], and very recently by Pettie [11].

Cross-References

▸ Synchronizers, Spanners

Recommended Reading

1. Aingworth D, Chekuri D, Indyk P, Motwani R (1999) Fast estimation of diameter and shortest paths (without matrix multiplication). SIAM J Comput 28(4):1167–1181
2. Althofer I, Das G, Dobkin DP, Joseph D, Soares J (1993) On sparse spanners of weighted graphs. Discret Comput Geom 9:81–100
3. Awerbuch B (1985) Complexity of network synchronization. J ACM 4:804–823
4. Baswana S, Kavitha T, Mehlhorn K, Pettie S (2010) Additive spanners and (alpha, beta)-spanners. ACM Trans Algorithms 7:Article 1
5. Chechik S (2013) New additive spanners. In: Proceedings 24th annual ACM-SIAM symposium on discrete algorithms, New Orleans, Jan 2013, pp 498–512.
6. Dor D, Halperin S, Zwick U (2000) All pairs almost shortest paths. SIAM J Comput 29:1740–1759
7. Elkin M (2005) Computing almost shortest paths. Trans Algorithm 1(2):283–323
8. Elkin M, Peleg D (2004) $(1+\epsilon,\beta)$-spanner constructions for general graphs. SIAM J Comput 33(3):608–631
9. Elkin M, Zhang J (2006) Efficient algorithms for constructing $(1+\epsilon,\beta)$-spanners in the distributed and streaming models. Distrib Comput 18(5):375–385
10. Peleg D, Schäffer A (1989) Graph spanners. J Graph Theory 13:99–116
11. Pettie S (2009) Low-distortion spanners. ACM Trans Algorithms 6:Article 1
12. Roditty L, Zwick U (2004) Dynamic approximate all-pairs shortest paths in undirected graphs. In: Proceedings of symposium on foundations of computer science, Rome, Oct 2004, pp 499–508
13. Thorup M, Zwick U (2006) Spanners and emulators with sublinear distance errors. In: Proceedings of symposium on discrete algorithms, Miami, Jan 2006, pp 802–809

14. Woodruff D (2006) Lower bounds for additive spanners, emulators, and more. In: Proceedings of symposium on foundations of computer science, Berckeley, Oct 2006, pp 389–398

Sparsest Cut

Shuchi Chawla
Department of Computer Science, University of Wisconsin–Madison, Madison, WI, USA

Keywords

Minimum ratio cut

Years and Authors of Summarized Original Work

2004; Arora, Rao, Vazirani

Problem Definition

In the Sparsest Cut problem, informally, the goal is to partition a given graph into two or more large pieces while removing as few edges as possible. Graph partitioning problems such as this one occupy a central place in the theory of network flow, geometric embeddings, and Markov chains, and form a crucial component of divide-and-conquer approaches in applications such as packet routing, VLSI layout, and clustering.

Formally, given a graph $G = (V, E)$, the *sparsity* or *edge expansion* of a non-empty set $S \subset V$, $|S| \leq \frac{1}{2}|V|$, is defined as follows:

$$\alpha(S) = \frac{|E(S, V \setminus S)|}{|S|} .$$

The sparsity of the graph, $\alpha(G)$, is then defined as follows:

$$\alpha(G) = \min_{S \subset V, |S| \leq \frac{1}{2}|V|} \alpha(S) .$$

The goal in the Sparsest Cut problem is to find a subset $S \subset V$ with the minimum sparsity, and to determine the sparsity of the graph.

The first approximation algorithm for the Sparsest Cut problem was developed by Leighton and Rao in 1988 [13]. Employing a linear programming relaxation of the problem, they obtained an $O(\log n)$ approximation, where n is the size of the input graph. Subsequently Arora, Rao and Vazirani [4] obtained an improvement over Leighton and Rao's algorithm using a semi-definite programming relaxation, approximating the problem to within an $O(\sqrt{\log n})$ factor.

In addition to the Sparsest Cut problem, Arora et al. also consider the closely related Balanced Separator problem. A partition $(S, V \setminus S)$ of the graph G is called a c-balanced separator for $0 < c \leq \frac{1}{2}$, if both S and $V \setminus S$ have at least $c|V|$ vertices. The goal in the Balanced Separator problem is to find a c-balanced partition with the minimum sparsity. This sparsity is denoted $\alpha_c(G)$.

Key Results

Arora et al. provide an $O(\sqrt{\log n})$ *pseudo-approximation* to the balanced-separator problem using semi-definite programming. In particular, given a constant $c \in (0, \frac{1}{2}]$, they produce a separator with balance c' that is slightly worse than c (that is, $c' < c$), but sparsity within an $O(\sqrt{\log n})$ factor of the sparsity of the optimal c-balanced separator.

Theorem 1 *Given a graph $G = (V, E)$, let $\alpha_c(G)$ be the minimum edge expansion of a c-balanced separator in this graph. Then for every fixed constant $a < 1$, there exists a polynomial-time algorithm for finding a c'-balanced separator in G, with $c' \geq ac$, that has edge expansion at most $O(\sqrt{\log n}\alpha_c(G))$.*

Extending this theorem to include unbalanced partitions, Arora et al. obtain the following:

Theorem 2 *Let $G = (V, E)$ be a graph with sparsity $\alpha(G)$. Then there exists a polynomial-time*

algorithm for finding a partition $(S, V \setminus S)$, *with* $S \subset V$, $S \neq \emptyset$, *having sparsity at most* $O(\sqrt{\log n}\alpha(G))$.

An important contribution of Arora et al. is a new geometric characterization of vectors in n-dimensional space endowed with the squared-Euclidean metric. This result is of independent significance and has lead to or inspired improved approximation factors for several other partitioning problems (see, for example, [1, 5, 6, 7, 11]).

Informally, the result says that if a set of points in n-dimensional space is randomly projected on to a line, a good separator on the line is, with high probability, a good separator (in terms of squared-Euclidean distance) in the original high-dimensional space. Separation on the line is related to separation in the original space via the following definition of stretch.

Definition 1 (Def. 4 in [4]) Let $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n$ be a set of n points in $\mathcal{R}^n$, equipped with the squared-Euclidean metric $\mathrm{d}(x, y) = \|x - y\|_2^2$. The set of points is said to be (t, γ, β) *-stretched at scale* ℓ, if for at least a γ fraction of all the n-dimensional unit vectors u, there is a partial matching $M_u = \{(x_i, y_i)\}_i$ among these points, with $|M_u| \geq \beta n$, such that for all $(x, y) \in M_u$, $\mathrm{d}(x, y) \leq \ell^2$ and $\langle u, \vec{x} - \vec{y} \rangle \geq t\ell/\sqrt{n}$. Here $\langle \cdot, \cdot \rangle$ denotes the dot product of two vectors.

Theorem 3 *For any* $\gamma, \beta > 0$, *there is a constant* $C = C(\gamma, \beta)$ *such that if* $t > C \log^{1/3} n$, *then no set of n points in* $\mathcal{R}^n$ *can be* (t, γ, β)*-stretched for any scale* ℓ.

In addition to the SDP-rounding algorithm, Arora et al. provide an alternate algorithm for finding approximate sparsest cuts, using the notion of expander flows. This result leads to fast (quadratic time) implementations of their approximation algorithm [3].

Applications

One of the main applications of balanced separators is in improving the performance of divide and conquer algorithms for a variety of optimization problems.

One example is the Minimum Cut Linear Arrangement problem. In this problem, the goal is to order the vertices of a given n vertex graph G from 1 through n in such a way that the capacity of the largest of the cuts $(\{1, 2, \cdots, i\}, \{i + 1, \cdots, n\})$, $i \in [1, n]$, is minimized. Given a ρ-approximation to the balanced separator problem, the following divide and conquer algorithm gives an $O(\rho \log n)$-approximation to the Minimum Cut Linear Arrangement problem: find a balanced separator in the graph, then recursively order the two parts, and concatenate the orderings. The approximation follows by noting that if the graph has a balanced separator with expansion $\alpha_c(G)$, only $O(\rho n \alpha_n(G))$ edges are cut at every level, and given that a balanced separator is found at every step, the number of levels of recursion is at most $O(\log n)$.

Similar approaches can be used for problems such as VLSI layout and Gaussian elimination. (See the survey by Shmoys [14] for more details on these topics.)

The Sparsest Cut problem is also closely related to the problem of embedding squared-Euclidean metrics into the Manhattan (ℓ_1) metric with low distortion. In particular, the integrality gap of Arora et al.'s semi-definite programming relaxation for Sparsest Cut (generalized to include weights on vertices and capacities on edges) is exactly equal to the worst-case distortion for embedding a squared-Euclidean metric into the Manhattan metric. Using the technology introduced by Arora et al., improved embeddings from the squared-Euclidean metric into the Manhattan metric have been obtained [5, 7].

Open Problems

Hardness of approximation results for the Sparsest Cut problem are fairly weak. Recently Chuzhoy and Khanna [9] showed that this problem is APX-hard, that is, there exists a constant $\epsilon > 0$, such that a $(1 + \epsilon)$-approximation

algorithm for Sparsest Cut would imply P = NP. It is conjectured that the weighted version of the problem is NP-hard to approximate better than $O((\log\log n)^c)$ for some constant c, but this is only known to hold true assuming a version of the so-called Unique Games conjecture [8, 12]. On the other hand, the semi-definite programming relaxation of Arora et al. is known to have an integrality gap of $\Omega(\log\log n)$ even in the unweighted case [10]. Proving an unconditional super-constant hardness result for weighted or unweighted Sparsest Cut, or obtaining $o(\sqrt{\log n})$-approximations for these problems remain open.

The directed version of the Sparset Cut problem has also been studied, and is known to be hard to approximate within a $2^{\Omega(\log^{1-\epsilon} n)}$ factor [9]. On the other hand, the best approximation known for this problem only achieves a polynomial factor of approximation–a factor of $O(n^{11/23}\log^{O(1)} n)$ due to Aggarwal, Alon and Charikar [2].

Recommended Reading

1. Agarwal A, Charikar M, Makarychev K, Makarychev Y (2005) Proceedings of the 37th ACM symposium on theory of computing (STOC), Baltimore, May 2005, pp 573–581
2. Aggarwal A, Alon N, Charikar M (2007) Improved approximations for directed cut problems. In: Proceedings of the 39th ACM symposium on theory of computing (STOC), San Diego, June 2007, pp 671–680
3. Arora S, Hazan E, Kale S (2004) Proceedings of the 45th IEEE symposium on foundations of computer science (FOCS), Rome, 17–19 Oct 2004, pp 238–247
4. Arora S, Rao S, Vazirani U (2004) Expander flows, geometric embeddings, and graph partitionings. In: Proceedings of the 36th ACM symposium on theory of computing (STOC), Chicago, June 2004, pp 222–231
5. Arora S, Lee J, Naor A (2005) Euclidean distortion and the sparsest cut. In: Proceedings of the 37th ACM Symposium on Theory of Computing (STOC), Baltimore, May 2005, pp 553–562
6. Arora S, Chlamtac E, Charikar M (2006) New approximation guarantees for chromatic number. In: Proceedings of the 38th ACM symposium on theory of computing (STOC), Seattle, May 2006, pp 215–224
7. Chawla S, Gupta A, Räcke H (2005) Embeddings of negative-type metrics and an improved approximation to generalized sparsest cut. In: Proceedings of the ACM-SIAM symposium on discrete algorithms (SODA), Vancouver, Jan 2005, pp 102–111
8. Chawla S, Krauthgamer R, Kumar R, Rabani Y, Sivakumar D (2005) On the hardness of approximating sparsest cut and multicut. In: Proceedings of the 20th IEEE conference on computational complexity (CCC), San Jose, June 2005, pp 144–153
9. Chuzhoy J, Khanna S (2007) Polynomial flow-cut gaps and hardness of directed cut problems. In: Proceedings of the 39th ACM symposium on theory of computing (STOC), San Diego, June 2007, pp 179–188
10. Devanur N, Khot S, Saket R, Vishnoi N (2006) Integrality gaps for sparsest cut and minimum linear arrangement problems. In: Proceedings of the 38th ACM symposium on theory of computing (STOC), Seattle, May 2006, pp 537–546
11. Feige U, Hajiaghayi M, Lee J (2005) Improved approximation algorithms for minimum-weight vertex separators. In: Proceedings of the 37th ACM symposium on theory of computing (STOC), Baltimore, May 2005, pp 563–572
12. Khot S, Vishnoi N (2005) Proceedings of the 46th IEEE symposium on foundations of computer science (FOCS), Pittsburgh, Oct 2005, pp 53–62
13. Leighton FT, Rao SB (1988) An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In: Proceedings of the 29th IEEE symposium on foundations of computer science (FOCS), White Plains, Oct 1988, pp 422–431
14. Shmoys DB (1997) Cut problems and their application to divide-and-conquer. In: Hochbaum DS (ed) Approximation algorithms for NP-hard problems. PWS Publishing, Boston, pp 192–235

Speed Scaling

Kirk Pruhs
Department of Computer Science, University of Pittsburgh, Pittsburgh, PA, USA

Keywords

Frequency scaling; Speed scaling; Voltage scaling

Years and Authors of Summarized Original Work

1995; Yao, Demers, Shenker

Problem Definition

Speed scaling is a power management technique in modern processor that allows the processor to run at different speeds. There is a power function $P(s)$ that specifies the power, which is energy used per unit of time, as a function of the speed. In CMOS-based processors, the cube-root rule states that $P(s) \approx s^3$. This is usually generalized to assume that $P(s) = s^\alpha$ form some constant α. The goals of power management are to reduce temperature and/or to save energy. Energy is power integrated over time. Theoretical investigations to date have assumed that there is a fixed ambient temperature and that the processor cools according to Newton's law, that is, the rate of cooling is proportional to the temperature difference between the processor and the environment.

In the resulting scheduling problems, the scheduler must not only have a job-selection policy to determine the job to run at each time, but also a speed scaling policy to determine the speed at which to run that job. The resulting problems are generally dual objective optimization problems. One objective is some quality of service measure for the schedule, and the other objective is temperature or energy.

We will consider problems where jobs arrive at the processor over time. Each job i has a release time r_i when it arrives at the processor, and a work requirement w_i. A job i run at speed s takes w_i/s units of time to complete.

Key Results

Yao et al. [5] initiated the theoretical algorithmic investigation of speed scaling problems. Yao et al. [5] assumed that each job i had a deadline d_i, and that the quality of service measure was deadline feasibility (each job completes by its deadline). Yao et al. [5] gives a greedy algorithm YDS to find the minimum energy feasible schedule. The job selection policy for YDS is to run the job with the earliest deadline. To understand the speed scaling policy for YDS, define the intensity of a time interval to be the work that must be completed in this time interval divided by the length of the time interval. YDS then finds the maximum intensity interval, runs the jobs that must be run in this interval at constant speed, eliminates these jobs and this time interval from the instance, and proceeds recursively. Yao et al. [5] gives two online algorithms: OA and AVR. In OA the speed scaling policy is the speed that YDS would run at, given the current state and given that no more jobs will be released in the future. In AVR, the rate at which each job is completed is constant between the time that a job is released and the deadline for that job. Yao et al. [5] showed that AVR is $2^{\alpha-1}\alpha^\alpha$-competitive with respect to energy.

The results in [5] were extended in [2]. Bansal et al. [2] showed that OA is α^α-competitive with respect to energy. Bansal et al. [2] proposed another online algorithm, BKP. BKP runs at the speed of the maximum intensity interval containing the current time, taking into account only the work that has been released by the current time. They show that the competitiveness of BKP with respect to energy is at most $2(\alpha/(\alpha-1))^\alpha \mathrm{e}^\alpha$. They also show that BKP is e-competitive with respect to the maximum speed.

Bansal et al. [2] initiated the theoretical algorithmic investigation of speed scaling to manage temperature. Bansal et al. [2] showed that the deadline feasible schedule that minimizes maximum temperature can in principle be computed in polynomial time. Bansal et al. [2] showed that the competitiveness of BKP with respect to maximum temperature is at most $2^{\alpha+1}\mathrm{e}^\alpha(6(\alpha/(\alpha-1))^\alpha + 1)$.

Pruhs et al. [4] initiated the theoretical algorithmic investigation into speed scaling when the quality-of-service objective is average/total flow time. The flow time of a job is the delay from when a job is released until it is completed. Pruhs et al. [4] give a rather complicated polynomial-time algorithm to find the optimal flow time schedule for unit work jobs, given

a bound on the energy available. It is easy to see that no $O(1)$-competitive algorithm exists for this problem.

Albers and Fujiwara [1] introduce the objective of minimizing a linear combination of energy used and total flow time. This has a natural interpretation if one imagines the user specifying how much energy he is willing to use to increase the flow time of a job by a unit amount. Albers and Fujiwara [1] give an $O(1)$-competitive online algorithm for the case of unit work jobs. Bansal et al. [3] improves upon this result and gives a 4-competitive online algorithm. The speed scaling policies of the online algorithms in [1] and [3] essentially run as power equal to the number of unfinished jobs (in each case modified in a particular way to facilitate analysis of the algorithm). Bansal et al. [3] extend these results to apply to jobs with arbitrary work, and even arbitrary weight. The speed scaling policy is essentially to run at power equal to the weight of the unfinished work. The expression for the resulting competitive ratio is a bit complicated but is approximately 8 when the cube-root rule holds.

The analysis of the online algorithms in [2] and [3] heavily relied on amortized local competitiveness. An online algorithm is locally competitive for a particular objective if for all times the rate of increase of that objective for the online algorithm, plus the rate of change of some potential function, is at most the competitive ratio times the rate of increase of the objective in any other schedule.

Applications

None

Open Problems

The outstanding open problem is probably to determine if there is an efficient algorithm to compute the optimal flow time schedule given a fixed energy bound.

Recommended Reading

1. Albers S, Fujiwara H (2006) Energy-efficient algorithms for flow time minimization. In: STACS. Lecture notes in computer science, vol 3884. Springer, Berlin, pp 621–633
2. Bansal N, Kimbrel T, Pruhs K (2007) Speed scaling to manage energy and temperature. J ACM 54(1)
3. Bansal N, Pruhs K, Stein C (2007) Speed scaling for weighted flow. In: ACM/SIAM symposium on discrete algorithms
4. Pruhs K, Uthaisombut P, Woeginger G (2004) Getting the best response for your ERG. In: Scandanavian workshop on algorithms and theory
5. Yao F, Demers A, Shenker S (1995) A scheduling model for reduced CPU energy. In: IEEE symposium on foundations of computer science, p 374

Sphere Packing Problem

Danny Z. Chen
Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, USA

Keywords

Ball packing; Disk packing

Years and Authors of Summarized Original Work

2001; Chen, Hu, Huang, Li, Xu

Problem Definition

The sphere packing problem seeks to pack spheres into a given geometric domain. The problem is an instance of geometric packing. Geometric packing is a venerable topic in mathematics. Various versions of geometric packing problems have been studied, depending on the shapes of packing domains, the types of packing objects, the position restrictions on the objects, the optimization criteria, the

dimensions, etc. It also arises in numerous applied areas. The sphere packing problem under consideration here finds applications in radiation cancer treatment using Gamma Knife systems. Unfortunately, even very restricted versions of geometric packing problems (e.g., regular-shaped objects and domains in lower dimensional spaces) have been proved to be NP-hard. For example, for *congruent packing* (i.e., packing copies of the same object), it is known that the 2-D cases of packing fixed-sized congruent squares or disks in a simple polygon are NP-hard [7]. Baur and Fekete [2] considered a closely related *dispersion problem* of packing k congruent disks in a polygon of n vertices such that the radius of the disks is maximized; they proved that the dispersion problem cannot be approximated arbitrarily well in polynomial time unless P = NP, and gave a $\frac{2}{3}$-approximation algorithm for the L_∞ disk case with a time bound of $O(n^{38})$.

Chen et al. [4] proposed a practically efficient heuristic scheme, called *pack-and-shake*, for the **congruent sphere packing** problem, based on computational geometry techniques. The problem is defined as follows.

The Congruent Sphere Packing Problem

Given a d-D polyhedral region $R(d = 2, 3)$ of n vertices and a value $r > 0$, find a packing SP of R using spheres of radius r, such that (i) each sphere is contained in R, (ii) no two distinct spheres intersect each other in their interior, and (iii) the ratio (called the packing density) of the covered volume in R by SP over the total volume of R is maximized.

In the above problem, one can view the spheres as "solid" objects. The region R is also called the *domain* or *container*. Without loss of generality, let $r = 1$.

Much work on congruent sphere packing studied the case of packing spheres into an unbounded domain or even the whole space [5]. There are also results on packing congruent spheres into a bounded region. Hochbaum and Maass [8] presented a unified and powerful *shifting technique* for designing pseudo-polynomial time approximation schemes for packing congruent squares into a rectilinear polygon. But, the high time complexities associated with the resulting algorithms restrict their applicability in practice. Another approach is to formulate a packing problem as a non-linear optimization problem, and resort to an available optimization software to generate packings; however, this approach works well only for small problem sizes and regular-shaped domains.

To reduce the running time yet achieve a dense packing, a common idea is to consider objects that form a certain lattice or double-lattice. A number of results were given on lattice packing of congruent objects in the whole (especially high dimensional) space [5]. For a bounded rectangular 2-D domain, Milenkovic [10] adopted a method that first finds the densest translational lattice packing for a set of polygonal objects in the whole plane, and then uses some heuristics to extract the actual bounded packing.

Key Results

The *pack-and-shake* scheme of Chen et al. [4] for packing congruent spheres in an irregular-shaped 2-D or 3-D bounded domain R consists of three phases. In the first phase, the d-D domain R is partitioned into a set of convex subregions (called *cells*). The resulting set of cells defines a dual graph G_D, such that each vertex v of G_D corresponds to a cell $C(v)$ and an edge connects two vertices if and only if their corresponding cells share a $(d-1)$-D face. In the second phase, the algorithm repeats the following *trimming and packing* process until $G_D = \emptyset$: Remove the lowest degree vertex v from G_D and pack the cell $C(v)$. In the third phase, a *shake* procedure is applied to globally adjust the packing to obtain a denser one.

The objective of the trimming and packing procedure is that after each cell is packed, the remaining "packable" subdomain R' of R is always kept as a connected region. The rationale for maintaining the connectivity of R' is as follows. To pack spheres in a bounded domain R, two

typical approaches have been used: (a) packing spheres layer by layer going from the boundary of R towards its interior [9], and (b) packing spheres starting from the "center" of R, such as its medial axis, towards its boundary [3, 13, 14]. Due to the shape irregularity of R, both approaches may fragment the remaining "packable" subdomain R' into more and more disconnected regions; however, at the end of packing each such region, a small "unpackable" area may eventually remain that allows no further packing. It could fit more spheres if the "packable" subdomain R' is lumped together instead of being divided into fragments, which is what the trimming and packing procedure aims to achieve.

Due to the packing of its adjacent cells that have been done by the trimming and packing procedure, the boundary of a cell $C(v)$ that is to be packed may consist of both line segments and arcs (from packed spheres). Hence, a key problem is to pack spheres in a cell bounded by curves of low degrees. Chen et al.'s algorithms [4] for packing each cell are based on certain lattice structures and allow the cell to both translate and rotate. Their algorithms have fairly low time bounds. In certain cases, they even run in nearly linear time.

An interesting feature of the cell packings generated by the trimming and packing procedure is that the resulted spheres cluster together in the middle of the cells of the domain R, leaving some small unpackable areas scattered along the boundary of R. The "shake" procedure in [4] thus seeks to collect these small areas together by "pushing" the spheres towards the boundary of R, in the hope of obtaining some "packable" region in the middle of R.

The approach in [4] is to first obtain a densest lattice unit sphere packing $LSP(C)$ for each cell C of R, and then use a "shake" procedure to globally adjust the resulting packing of R to generate a denser packing SP in R. Suppose the plane P is already packed by infinitely many unit spheres whose center points form a lattice (e.g., the hexagonal lattice). To obtain a densest packing $LSP(C)$ for a cell C from the lattice packing of the plane P, a position and orientation of C on P need to be computed such that C contains the maximum number of spheres from the lattice packing of P. There are two types of algorithms in [4] for computing an optimal placement of C on P: translational algorithms that allow C to be translated only, and translational/rotational algorithms that allow C to be both translated and rotated.

Let $n = |C|$, the number of bounding curves of C, and m be the number of spheres along the boundary of C in a sought optimal packing of C.

Theorem 1 *Given a polygonal region C bounded by n algebraic curves of constant degrees, a densest lattice unit sphere packing of C based only on translational motion can be computed in $O(N \log N + K)$ time, where $N = f(n, m)$ is a function of n and m, and K is the number of intersections between N planar algebraic curves of constant degrees that are derived from the packing instance.*

Note: In the worst case, $N = f(n, m) = n \times m$. But in practice, N may be much smaller. The N planar algebraic curves in Theorem 1 form a structure called *arrangement*. Since all these curves are of a constant degree, any two such curves can intersect each other at most a constant number of times. In the worst case, the number K of intersections between the N algebraic curves, which is also the *size* of the arrangement, is $O(N^2)$. The arrangement of these curves can be computed by the algorithms [1, 6] in $O(N \log N + K)$ time.

Theorem 2 *Given a polygonal region C bounded by n algebraic curves of constant degrees, a densest lattice unit sphere packing of C based on both translational and rotational motions can be computed in $O(T(n) + (N + K') \log N)$ time, where $N = f(n, m)$ is a function of n and m, K' is the size of the arrangement of N*

pseudo-plane surfaces in 3-D that are derived from the packing instance, and T(n) is the time for solving $O(n^2)$ quadratic optimization problem instances associated with the packing instance.

In Theorem 2, $K' = O(N^3)$ in the worst case. In practice, K' can be much smaller.

The results on 2-D sphere packing in [4] can be extended to d-D for any constant integer $d \geq 3$, so long as a good d-D lattice packing of the d-D space is available.

Applications

Recent interest in the considered congruent sphere packing problem was motivated by medical applications in Gamma Knife radiosurgery [4, 11, 12]. Radiosurgery is a minimally invasive surgical procedure that uses radiation to destroy tumors inside human body while sparing the normal tissues. The Gamma Knife is a radiosurgical system that consists of 201 Cobalt-60 sources [3, 14]; the gamma-rays from these sources are all focused on a common center point, thus creating a spherical volume of radiation field. The Gamma Knife treatment normally applies high radiation dose. In this setting, overlapping spheres may result in overdose regions (called *hot spots*) in the target treatment domain, while a low packing density may cause underdose regions (called *cold spots*) and a non-uniform dose distribution. Hence, one may view the spheres used in Gamma Knife packing as "solid" spheres. Therefore, a key geometric problem in Gamma Knife treatment planning is to fit multiple spheres into a 3-D irregular-shaped tumor [3, 13, 14]. The total treatment time crucially depends on the number of spheres used. Subject to a given packing density, the minimum number of spheres used in the packing (i.e., treatment) is desired. The Gamma Knife currently produces spheres of four different radii (4, 8, 14, and 18 mm), and hence the Gamma Knife sphere packing is in general not congruent. In practice, a commonly used approach is to pack larger spheres first, and then fit smaller spheres into the remaining subdomains, in the hope of reducing the total number of spheres involved and thus shortening the treatment time. Therefore, congruent sphere packing can be used as a key subroutine for such a common approach.

Open Problems

An open problem is to analyze the quality bounds of the resulting packing for the algorithms in [4]; such packing quality bounds are currently not yet known. Another open problem is to reduce the running time of the packing algorithms in [4], since these algorithms, especially for sphere packing problems in higher dimensions, are still very time-consuming. In general, it is highly desirable to develop efficient sphere packing algorithms in d-D ($d \geq 2$) with guaranteed good packing quality.

Experimental Results

Some experimental results of the 2-D pack-and-shake sphere packing algorithms were given in [4]. The planar hexagonal lattice was used for the lattice packing. On packings whose sizes are in the hundreds, the C++ programs of the algorithms in [4] based only on translational motion run very fast (a few minutes), while those of the algorithms based on both translation and rotation take much longer time (hours), reflecting their respective theoretical time bounds, as expected. On the other hand, the packing quality of the translation-and-rotation based algorithms is a little better than the translation based algorithms. The packing densities of all the algorithms in the experiments are well above 70 % and some are even close to or above 80 %. Comparing with the nonconvex programming methods, the packing algorithms in [4] seemed to run faster based on the experiments.

Cross-References

▸ Local Approximation of Covering and Packing Problems

Recommended Reading

1. Amato NM, Goodrich MT, Ramos EA (2000) Computing the arrangement of curve segments: divide-and-conquer algorithms via sampling. In: Proceedings of the 11th annual ACM-SIAM symposium on discrete algorithms, pp 705–706
2. Baur C, Fekete SP (2001) Approximation of geometric dispersion problems. Algorithmica 30(3):451–470
3. Bourland JD, Wu QR (1996) Use of shape for automated, optimized 3D radiosurgical treatment planning. In: Proceedings of the SPIE international symposium on medical imaging, pp 553–558
4. Chen DZ, Hu X, Huang Y, Li Y, Xu J (2001) Algorithms for congruent sphere packing and applications. In: Proceedings of the 17th annual ACM symposium on computational geometry, pp 212–221
5. Conway JH, Sloane NJA (1988) Sphere packings, lattices and groups. Springer, New York
6. Edelsbrunner H, Guibas LJ, Pach J, Pollack R, Seidel R, Sharir M (1992) Arrangements of curves in the plane: topology, combinatorics, and algorithms. Theor Comput Sci 92:319–336
7. Fowler RJ, Paterson MS, Tanimoto SL (1981) Optimal packing and covering in the plane are NP-complete. Inf Process Lett 12(3):133–137
8. Hochbaum DS, Maass W (1985) Approximation schemes for covering and packing problems in image processing and VLSI. J ACM 32(1):130–136
9. Li XY, Teng SH, Üngör A (2000) Biting: advancing front meets sphere packing. Int J Numer Methods Eng 49(1–2):61–81
10. Milenkovic VJ (2000) Densest translational lattice packing of nonconvex polygons. In: Proceedings of the 16th ACM annual symposium on computational geometry, pp 280–289
11. Shepard DM, Ferris MC, Ove R, Ma L (2000) Inverse treatment planning for Gamma Knife radiosurgery. Med Phys 27(12):2748–2756
12. Sutou A, Dai Y (2002) Global optimization approach to unequal sphere packing problems in 3D. J Optim Theory Appl 114(3):671–694
13. Wang J (1999) Medial axis and optimal locations for min-max sphere packing. J Comb Optim 3:453–463
14. Wu QR (1996) Treatment planning optimization for Gamma unit radiosurgery. Ph.D. thesis, The Mayo Graduate School

Split Decomposition via Graph-Labelled Trees

Christophe Paul
CNRS, Laboratoire d'Informatique Robotique et Microélectronique de Montpellier, Université Montpellier 2, Montpellier, France

Keywords

Circle graphs; Distance hereditary graphs; LexBFS; Parity graphs; Permutation graphs; Split decomposition

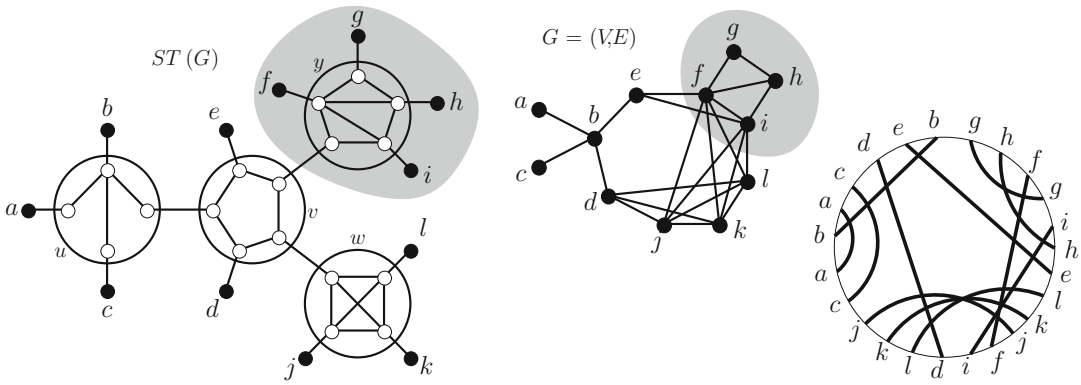
Years and Authors of Summarized Original Work

2012; Gioan, Paul
2014; Gioan, Paul, Tedder, Corneil

Problem Definition

Introduced by Cunningham and Edmonds [11], the *split decomposition*, also known as the *join (or 1-join) decomposition*, ranges among the classical graph decomposition schemes. Given a graph $G = (V, E)$, a bipartition (A, B) of the vertex set V (with $|A| \geq 2$ and $|B| \geq 2$) is a *split* if there are subsets $A' \subseteq A$ and $B' \subseteq B$, called *frontiers*, such that there is an edge between a vertex $u \in A$ and $v \in B$ if and only if $u \in A'$ and $v \in B'$ (see Fig. 1). A graph is *prime* if it does not contain any split. Observe that an induced cycle of length at least 5 is a prime graph. A graph is *degenerate* if every bipartition (A, B) with $|A| \geq 2$ and $|B| \geq 2$ is a split. It can be shown that a degenerate graphs are either *cliques* or *stars*. The split decomposition consists in recursively decompose a graph into a set of disjoint graphs $\{G_1, \ldots G_k\}$, called *split components*, each of which is either prime or degenerate. There are two cases:

1. If G is prime or degenerate, then return the set $\{G\}$;



Split Decomposition via Graph-Labelled Trees, Fig. 1 A circle graph G with a chord diagram on the *right* and its split decomposition tree $ST(G)$ on the *left*. The nodes v and y are prime nodes, whereas u is a star node and w a clique node. The bipartition $(\{f, g, h, i\}, V \setminus \{f, g, h, i\})$ forms a split of G and corresponds to a tree edge of $ST(G)$. The frontiers are $\{f, i\}$ on one side and $\{e, j, k, l\}$ on the other. Observe that $(\{k, l\}, V \setminus \{k, l\})$ is also a split but which is not represented by the tree edge between nodes Y and Z in $ST(G)$. Because G is not a prime graph, it can be represented with several chord diagram. For example, exchanging the chord of y with the chord of z yields an alternative chord diagram

2. If G is neither prime nor degenerate, it contains a split (A, B), with frontiers A' and B'. The split components of G is then the union of the split components of the graphs $G[A] + (a, A')$ and $G[B] + (b, B')$, where a and b are new vertices, called *markers*.

Observe that the split decomposition process naturally defines a decomposition tree whose nodes represent the split components. This decomposition tree can be represented by a *graph-labeled tree* (GLT) (see [16,18]) defined as a pair $(T, \mathcal{F})$, where T is a tree and $\mathcal{F}$ a set of graphs, such that each node u of T is *labeled* by the graph $G(u) \in \mathcal{F}$, and there exists a bijection ρ_u between the edges of T incident to u and the vertices of $G(u)$, called *marker vertices*. We say that two leaves ℓ_a and ℓ_b of T are *accessible* if for every pair of consecutive tree edges uv and vw on the path from ℓ_a and ℓ_b in T, $\rho_v(uv)$ and $\rho_v(vw)$ are adjacent in $G(v)$. From a GLT $(T, \mathcal{F})$, we define an *accessibility graph* $\mathcal{G}(T, \mathcal{F})$ whose vertex set is the leaf set of T and two vertices a and b are adjacent if the corresponding leaves ℓ_a and ℓ_b are accessible. It is easy to observe that every tree edge e of a GLT $(T, \mathcal{F})$ defines a split (A, B) of $\mathcal{G}(T, \mathcal{F})$ where A and B respectively contain the vertices corresponding to the leaves of the two connected components of $T - e$. Cunningham and Edmonds [11] formalized the family of splits as an example of *partite family of bipartitions* thereby implying that every graph admits a canonical split decomposition tree (see Fig. 1). In terms of GLTs, this translates as follows:

Theorem 1 ([11, 16, 18]) *Let G be a connected graph. There exists a unique GLT $(T, \mathcal{F})$ whose labels are either prime or degenerate, having a minimal number of nodes and such that $G = \mathcal{G}(T, \mathcal{F})$. This GLT is called the* split tree *of G and denoted $ST(G)$.*

The problem we are interested in is to efficiently compute the split tree $ST(G)$ of a graph $G = (V, E)$. The first polynomial-time algorithm was and runs in time $O(nm)$, where $n = |V|$ and $m = |E|$. Ma and Spinrad [23] later developed an $O(n^2)$ algorithm. Finally Dahlhaus [12] designed the first linear-time algorithm which was recently revisited by Charbit et al. [5].

Key Results

As mentioned above, the split tree of a graph can be computed in linear time. The algorithm we

describe here is nearly optimal, that is, runs in time $O(n+m)\cdot\alpha(n+m)$, where α is the inverse Ackermann function. The fact that this algorithm incrementally builds the split tree is responsible of the small additional complexity cost. More precisely, updating the tree structure of the GLT representing the split tree relies on the union-find data structure [15]. But having an incremental split decomposition algorithm allows an extension of the algorithm, within the same time complexity, to the circle graph recognition [17], a problem for which computing the split decomposition is a corner step. But so far, a subquadratic time complexity cannot be reached using the previous linear (or quadratic) split decomposition algorithms.

Theorem 2 ([18]) *The split tree $ST(G)$ of a graph $G = (V, E)$, with $|V| = n$ and $|E| = m$, can be built incrementally according to an LBFS ordering in time $O(n + m) \cdot \alpha(n + m)$, where α is the inverse Ackermann function.*

It is important to observe that to reach the expected complexity, the algorithm inserts the vertices according to a *LexBFS ordering* [25]. These orderings, resulting from a *lexicographic breadth-first search*, appear in a number of recognition algorithms, such as *chordal graphs* [25], *comparability graphs* [20], *interval graphs* [22], and *cographs* [3]. The idea is that structural properties can be shown on the last vertex visited by a LexBFS. For example, in chordal graphs the last vertex is simplicial; in comparability graphs it is a source of some transitive orientation. LexBFS, introduced in [25], works as follows: it numbers the vertices decreasingly from $n = |V|$ down to 1; initially every vertex receives an empty label; then iteratively, an arbitrary unnumbered vertex x with lexicographically largest label is selected and numbered i, and i is appended to the label of every unnumbered neighbor of x. On the graph of Fig. 1, $\sigma = b, a, e, d, c, f, i, j, k, l, h, g$ is a LexBFS ordering.

Applications

Many graph classes can be characterized by means of the split decomposition. Below, we review the most important of these classes. Finally, we discuss the links between split decomposition and other decomposition approaches.

Graph Classes

Distance Hereditary Graphs

The family of graphs for which the split tree does not contain any prime node is called *totally decomposable* (or *totally separable*). This terminology follows from the observation that for every subgraph of size at least 4, every nontrivial bipartition of the vertex set forms a split. A graph G is *distance hereditary* [1] if for every induced connected subgraph H of G and every pair of vertices x and y of H, the distance between x and y is the same in H and G. It turns out that a graph G is totally decomposable if and only if it is *distance hereditary* [1]. In other words, a graph G is distance hereditary if and only if every node of $ST(G)$ is either a star or a clique node. The first linear-time recognition algorithm of distance hereditary graphs, due to [21], relies on a breadth-first search characterization (see also [13]). More recently, a linear-time algorithm has been designed to update the split tree of a distance hereditary graph under vertex and edge insertion, leading to an alternative (vertex incremental) linear-time recognition algorithm for distance hereditary graphs.

Theorem 3 ([16]) *Let $ST(G)$ be the split tree of a distance hereditary graph $G = (V, E)$, $S \subseteq V$ be a subset of vertices of G and $e = (x, y) \notin E$ be a non-edge of G. Then:*

- *In $O(1)$-time, we can compute $ST(G + e)$ where $G+e = (V, E\cup\{e\})$ if $G+e$ is distance hereditary;*
- *In $O(|S|)$-time, we can compute $ST(G + (x, S)$ where $G + (x, S) = (V \cup \{x\}, E \cup \{(x, y) \mid y \in S\}$ if $G + (x, S)$ is distance hereditary.*

Subclasses of Totally Decomposable Graphs

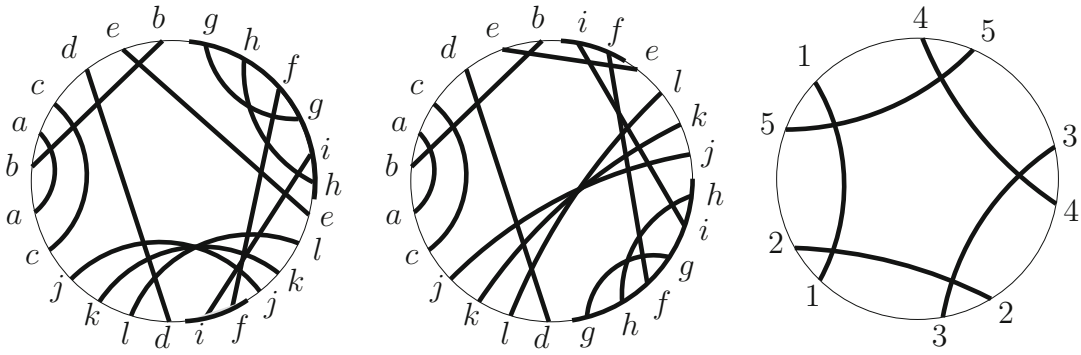
A GLT is called *clique-star tree* if its nodes are labeled either with cliques or stars. As a consequence of the discussion of the previous paragraph, distance hereditary graphs are the graphs corresponding the clique-star trees. Imposing any constraint on a clique-star tree thereby immediately defines a subclass of distance hereditary graphs. It turns out that many important graph subclasses of distance hereditary graphs can be characterized with the split decomposition.

The *cographs*, also known as *complement-reducible graphs* [8] or P_4*-free graphs*, are probably the most studied subclass of distance hereditary graphs. Cographs are also known as the class of graphs totally decomposable with respect to the modular decomposition [19], and their combinatorial structure is captured by the so-called cotree. As noticed in [16], it is easy to observe that a graph G is a cograph if and only if its split tree $ST(G)$ is a clique-star tree that can be rooted either at a node or at a tree edge such that every star node is "oriented" toward that root (that is the marker vertex corresponding the center of every star node is oriented toward the root).

The class of *ptolemaic graphs* or *3-leaf power* are also interesting. The class of ptolemaic graphs is defined as the intersection of distance hereditary graphs and chordal graphs. Chordal graphs are the graphs without induced chordless cycles of length four or more. It follows that a graph G is ptolemaic if and only if $ST(G)$ is a clique-star tree such that for every pair of star nodes u and v, not both extremities of the path from u to v in $ST(G)$ are attached to the center marker vertex of u and v (otherwise this would generate a chordless 4-cycle). As a subclass of chordal graph, 3-leaf powers inherit the restrictions of ptolemaic graphs on the split tree with the additive constraint that no clique node lies on the path between two star nodes (see [16] for details).

Circle Graphs

The split decomposition plays an important role in the context of *circle graphs* defined as intersection graphs of a set of chords in a circle. The main reason is that a graph G is a circle graph if and only if every split component of G is a circle graph. In other words, as clique and stars are circle graphs, G is a circle graph if and only if the prime nodes of $ST(G)$ are labeled with circle graphs. Observe that this characterization shows that distance hereditary graphs form a subclass of circle graphs. By the way the first quadratic time circle graph recognition algorithm was obtained by computing the split decomposition of the input graph and reducing the problem to the recognition of prime circle graphs [23, 26]. The key property is that a prime circle graph has a unique (up to mirror) *chord diagram* [2, 14] (see Fig. 2).



Split Decomposition via Graph-Labelled Trees, Fig. 2 On the *left*, two distinct chord diagrams of the graph G depicted in Fig. 1 results from symmetric insertion of the chords representing the vertices $\{f, g, h, i\}$ (remind that $(\{f, g, h, i\}, V \setminus \{f, g, h, i\})$ form a split). On the *right*, the chord diagram on $\{1, 2, 3, 4, 5\}$ is the unique (up to rotation and mirror) chord diagram of the 5-cycle, which is a prime graph

The linear-time split decomposition algorithm [12], proposed in the mid-1990s, did not lead to a linear-time circle graph recognition algorithm. For almost two decades, the quadratic time complexity [27] remained the best known complexity. The quadratic barrier has been broken using the almost linear-time split decomposition algorithm of [17]. The key ingredient was to insert the vertices according to a LexBFS ordering. Indeed, in the unique chord diagram of a prime circle graph G, the neighborhood of the last vertex x of a LexBFS ordering satisfies a sort of consecutiveness property. More precisely, the chord diagram of G contains a set of consecutive chord extremities starting and ending with the extremities of x's chord and containing one and only one chord extremity per neighbor of x and no chord extremity of non-neighbors of x. This property is used to incrementally build the split tree of a circle graph using chord diagrams to represent prime nodes. It is worth to observe that the split tree of a circle graph G together with the chord diagrams of each of its prime nodes provides a canonical (linear space) representation of the set of (exponentially many) chord diagrams of G.

Theorem 4 ([17]) *Let $G = (V, E)$ be a graph such that $|V| = n$ and $|E| = m$. There exists a $O(n + m) \cdot \alpha(n + m)$-time algorithm, where α is the inverse Ackermann function, deciding whether G is a circle graph. Moreover, if G is a circle graph, the algorithm outputs a split-tree representation G from which any chord diagram of G can be extracted in linear time.*

Perfect Graphs

The recent proof [6] of the famous conjecture of Berge on perfect graphs states that a graph is *perfect* if and only if it does not contain an odd cycle of length at least 5 nor its complement as induced subgraph. It is easy to observe that a graph is perfect if and only if its prime components are perfect graphs. The split decomposition does not formally appear in the structural decomposition theorem of perfect graphs [6, 28] as it is subsumed by the so-called balanced skew partition. In the context of perfect graphs, *parity graphs* [4] form a nice example of class of graphs simply characterized through their split decomposition. A graph is a parity graph if for every pair x, y of vertices, the length of every chordless path between x and y is of the same parity. This constraint can be translated into a condition on odd cycles or into a condition on their split tree. Indeed it can be proved that a graph is a parity graph if and only if its prime nodes are labeled with bipartite graphs [7].

Related Graph Decompositions

Modular Decomposition

The split decomposition is often introduced as a generalization of the *modular decomposition* (also known as homogeneous decomposition) [19]. A *module* in a graph $G = (V, E)$ is a subset M of vertices such that every vertex not in M is either fully adjacent or fully nonadjacent to the vertices of M. Clearly, if M is a module of size at least 2, then $(M, V \setminus M)$ defines a split. Indeed the split decomposition is sometimes used to further decompose graphs that are primes with respect to the modular decomposition.

Width Parameters

Rank-width [24] and *clique-width* [10] are two important width parameters both sharing some connections with the split decomposition. As the rank-width of a graph is small if its clique-width is small and vice versa, we only briefly describe the former parameter. A rank-decomposition of a graph G is defined as a ternary tree whose leaves are in one-to-one correspondence with the vertices of G. It follows that every internal tree edge defines a bipartition, say (A, B) of the vertices of G. The rank-width of a bipartition (A, B) is defined as the rank of the incidence matrix between A and B, and the width of a rank-decomposition is the maximum width over its bipartitions. The rank-width of a graph G is then the minimum width over its rank-decompositions. Observe that the every split is a rank-width 1 bipartition. It follows that the rank-width of a graph is the maximum rank-width of its prime components. As a consequence rank-width one graphs are exactly distance hereditary graphs. To conclude, let us

mention that computing the split decomposition of a graph is a key step in the polynomial-time recognition algorithm of clique-width three graphs [9].

Recommended Reading

1. Bandelt H-J, Mulder HM (1986) Distance hereditary graphs. J Comb Theory Ser B 41:182–208
2. Bouchet A (1987) Reducing prime graphs and recognizing circle graphs. Combinatorica 7:243–254
3. Bretscher A, Corneil D, Habib M, Paul C (2008) A simple linear time lexbfs cograph recognition algorithm. SIAM J Discret Math 22(4):1277–1296
4. Burlet M, Uhry JP (1984) Parity graphs. Ann Discret Math 21:253–277
5. Charbit P, de Montgolfier F, Raffinot M (2012) Linear time split decomposition revisited. SIAM J Discret Math 26(2):499–514
6. Chudnovsky M, Robertson N, Seymour P, Thomas R (2006) The strong perfect graph theorem. Ann Math 161:51–229
7. Cicerone S, Di Stefano G (1999) On the extension of bipartite to parity graphs. Discret Appl Math 95:181–195
8. Corneil D, Lerchs H, Stewart-Burlingham LK (1981) Complement reducible graphs. Discret Appl Math 3(1):163–174
9. Corneil D, Habib M, Lanlignel JM, Reed B, Rotics U (2012) Polynomial-time recognition of clique-width $\leqslant 3$ graphs. Discret Appl Math 160(6):834–865
10. Courcelle B, Engelfriet J, Rozenberg G (1993) Handle rewriting hypergraph grammars. J Comput Syst Sci 46:218–270
11. Cunningham WH, Edmonds J (1980) A combinatorial decomposition theory. Can J Math 32(3):734–765
12. Dahlhaus E (1994) Efficient parallel and linear time sequential split decomposition (extended abstract). In: Foundations of software technology and theoretical computer science – FSTTCS, Madras. Volume 880 of lecture notes in computer science, pp 171–180
13. Damiand G, Habib M, Paul C (2001) A simple paradigm for graph recognition: application to cographs and distance hereditary graphs. Theor Comput Sci 263:99–111
14. Gabor CP, Hsu WL, Suppovit KJ (1989) Recognizing circle graphs in polynomial time. J ACM 36:435–473
15. Gabow H, Tarjan R (1983) A linear-time algorithm for a special case of disjoint set union. In: Annual ACM symposium on theory of computing (STOC), Boston, pp 246–251
16. Gioan E, Paul C (2012) Split decomposition and graph-labelled trees: characterizations and fully dynamic algorithms for totally decomposable graphs. Discret Appl Math 160(6):708–733
17. Gioan E, Paul C, Tedder M, Corneil D (2013) Circle graph recognition in time $O(n+m)\alpha(n+m)$. Algorithmica 69(4): 759–788 (2014)
18. Gioan E, Paul C, Tedder M, Corneil D (2013) Practical split-decomposition via graph-labelled trees. Algorithmica 69(4): 789–843 (2014)
19. Habib M, Paul C (2010) A survey on algorithmic aspects of modular decomposition. Comput Sci Rev 4:41–59
20. Habib M, McConnell RM, Paul C, Viennot L (2000) Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing. Theor Comput Sci 234:59–84
21. Hammer P, Maffray F (1990) Completely separable graphs. Discret Appl Math 27:85–99
22. Korte N, Möhring R (1989) An incremental linear-time algorithm for recongizing interval graphs. SIAM J Comput 18(1):68–81
23. Ma T-H, Spinrad J (1994) An $O(n^2)$ algorithm for undirected split decomposition. J Algorithms 16:145–160
24. Oum SI (2005) Graphs of bounded rank-width. PhD thesis, Princeton University
25. Rose DJ, Tarjan RE, Lueker GS (1976) Algorithmic aspects of vertex elimination on graphs. SIAM J Comput 5(2):266–283
26. Spinrad J (1989) Prime testing for the split decomposition of a graph. SIAM J Discret Math 2(4):590–599
27. Spinrad J (1994) Recognition of circle graphs. J Algorithms 16:264–282
28. Trotignon N (2013) Perfect graphs: a survey. Technical report 1301.5149, arxiv

Squares and Repetitions

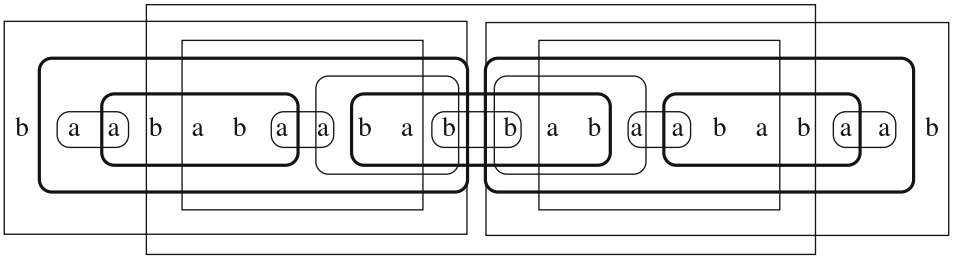
Maxime Crochemore[1,2,4] and Wojciech Rytter[3]
[1]Department of Computer Science, King's College London, London, UK
[2]Laboratory of Computer Science, University of Paris-East, Paris, France
[3]Institute of Informatics, Warsaw University, Warsaw, Poland
[4]Université de Marne-la-Vallée, Champs-sur-Marne, France

Keywords

Powers; Runs; Tandem repeats

Years and Authors of Summarized Original Work

1999; Kolpakov, Kucherov



Squares and Repetitions, Fig. 1 The structure of $RUNS(x)$ where $x = \texttt{baababaababbabaababaab} = \texttt{b}z^2(z^R)^2\texttt{b}$, for $z = \texttt{aabab}$. The operation $\cdot^R$ is reversing the string

Problem Definition

Periodicities and repetitions in strings have been extensively studied and are important both in theory and practice (combinatorics of words, pattern-matching, computational biology). The words of the type ww and www, where w is a nonempty primitive (not of the form u^k for an integer $k > 1$) word, are called squares and cubes, respectively. They are well-investigated objects in combinatorics on words [16] and in string-matching with small memory [5].

A string w is said to be periodic iff $period(w) \leq |w|/2$, where $period(w)$ is the smallest positive integer p for which $w[i] = w[i + p]$ whenever both sides of the equality are defined. In particular each square and cube is periodic.

A repetition in a string $x = x_1 x_2 \ldots x_n$ is an interval $[i \,..\, j] \subseteq [1 \,..\, n]$ for which the associated factor $x[i \,..\, j]$ is periodic. It is an occurrence of a periodic word $x[i \,..\, j]$, also called a positioned repetition. A word can be associated with several repetitions, see Fig. 1.

Initially people investigated mostly positioned squares, but their number is $\Omega(n \log n)$ [2], hence algorithms computing all of them cannot run in linear time, due to the potential size of the output. The optimal algorithms reporting all positioned squares or just a single square were designed in [1, 2, 3, 19]. Unlike this, it is known that only $O(n)$ (un-positioned) squares can appear in a string of length n [8].

The concept of maximal repetitions, called runs (equivalent terminology) in [14], has been introduced to represent all repetitions in a succinct manner. The crucial property of runs is that there are only $O(n)$ runs in a word of length n [15, 21].

A *run* in a string x is an interval $[i \,..\, j]$ such that both the associated string $x[i \,..\, j]$ has period $p \leq (j - i + 1)/2$, and the periodicity cannot be extended to the right nor to the left: $x[i - 1] \neq x[x + p - 1]$ and $x[j - p + 1] \neq x[j + 1]$ when the elements are defined. The set of runs of x is denoted by $RUNS(x)$. An example is displayed in Fig. 1.

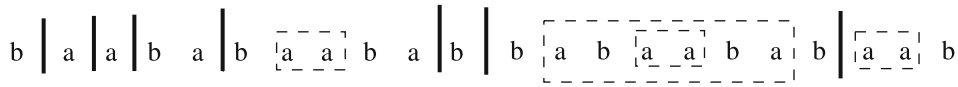
Key Results

The main results concern fast algorithms for computing positioned squares and runs, as well as combinatorial estimation on the number of corresponding objects.

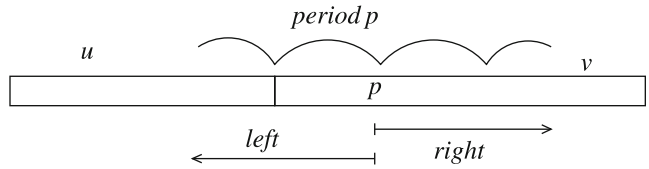
Theorem 1 (Crochemore [1], Apostolico-Preparata [2], Main-Lorentz [19]) *There exists an $O(n \log n)$ worst-case time algorithm for computing all the occurrences of squares in a string of length n.*

Techniques used to design the algorithms are based on partitioning, suffix trees, and naming segments. A similar result has been obtained by Franek, Smyth, and Tang using suffix arrays [11]. The key component in the next algorithm is the function described in the following lemma.

Lemma 2 (Main-Lorentz [19]) *Given two square-free strings u and v, reporting if uv contains a square centered in u can be done in worst-case time $O(|u|)$.*



Squares and Repetitions, Fig. 2 The f-factorization of the example string $x = \texttt{baababaababbabaababaab}$ and the set of its internal runs; all other runs overlap factorization points



Squares and Repetitions, Fig. 3 If an overlapping run with period p starts in u, ends in v, and its part in v is of size at least p then it is easily detectable by computing continuations of the periodicity p in two directions: left and right

Using suffix trees or suffix automata together with the function derived from the lemma, the following fact has been shown.

Theorem 3 (Crochemore [3], Main-Lorentz [19]) *Testing the square-freeness of a string of length n can be done in worst-case time $O(n \log a)$, where a is the size of the alphabet of the string.*

As a consequence of the algorithms and of the estimation on the number of squares, the most important result related to repetitions can be formulated as follows.

Theorem 4 (Kolpakov-Kucherov [15], Rytter [21], Crochemore-Ilie [4])

(1) *All runs in a string can be computed in linear time (on a fixed-size alphabet).*
(2) *The number of all runs is linear in the length of the string.*

The point (2) is very intricate, it is of purely combinatorial nature and has nothing to do with the algorithm. We sketch shortly the basic components in the constructive proof of the point (1). The main idea is to use, as for the previous theorem, the f-factorization (see [3]): a string x is decomposed into factors $u_1, u_2, \ldots, u_k$, where u_i is the longest segment which appears before (possibly with overlap) or is a single letter if the segment is empty.

The runs which fit in a single factor are called internal runs, other runs are called here overlapping runs. There are three crucial facts:

- all overlapping runs can be computed in linear time,
- each internal run is a copy of an earlier overlapping run,
- the f-factorization can be computed in linear time (on a fixed-size alphabet) if we have the suffix tree or suffix automaton of the string. Figure 2 shows f-factorization and internal runs of an example string.

It follows easily from the definition of the f-factorization that if a run overlaps two (consecutive) factors u_{k-1} and u_k then its size is at most twice the total size of these two s factors.

Figure 3 shows the basic idea for computing runs that overlap $u\,v$ in time $O(|u| + |v|)$. Using similar tables as in the Morris–Pratt algorithm (border and prefix tables), see [6], we can test the continuation of a period p from position p in v to the left and to the right. The corresponding tables can be constructed in linear time in a preprocessing phase. After computing all overlapping runs the internal runs can be copied from their earlier occurrences by processing the string from left to right.

Another interesting result concerning periodicities is the following lemma and its fairly immediate corollary.

Lemma 5 (Three Prefix Squares, Crochemore-Rytter [5]) *If u, v, and w are three primitive words satisfying: $|u| < |v| < |w|$, uu is a prefix of vv, and vv is a prefix of ww, then $|u| + |v| \leq |w|$*

Corollary 1 *Any nonempty string x possesses less than $\log_\Phi |y|$ prefixes that are squares.*

In the configuration of the lemma, a second consequence is that uu is a prefix of w. Therefore, a position in a string x cannot be the largest position of more than two squares, which yields the next corollary. A simple direct proof of it is by Ilie [13], see also [17].

Corollary 2 (Fraenkel and Simpson [8]) *Any string x contains at most $2|x|$ (different) squares, that is: card$\{u \mid u$ primitive and u^2 factor of $y\} \leq 2|x|$.*

The structure of all squares and of un-positioned runs has been also computed within the same time complexities as above in [18] and [12].

Applications

Detecting repetitions in strings is an important element of several questions: pattern matching, text compression, and computational biology to quote a few. Pattern-matching algorithms have to cope with repetitions to be efficient as these are likely to slow down the process; the large family of dictionary-based text compression methods use a weaker notion of repeats (like the software gzip); repetitions in genomes, called satellites, are intensively studied because, for example, some over-repeated short segments are related to genetic diseases; some satellites are also used in forensic crime investigations.

Open Problems

The most intriguing question remains the asymptotically tight bound for the maximum number $\rho(n)$ of runs in a string of size n. The first proof (by painful induction) was quite difficult and has not produced any *concrete* constant coefficient in the $O(n)$ notation. This subject has been studied in [9, 10, 22, 23]. The best-known lower bound of approximately $0.927\,n$ is from [10]. The exact number of runs has been considered for special strings: *Fibonacci words* and (more generally) *Sturmian words* [7, 14, 20]. It is proved in a structural and intricate manner in the full version of [21] that $\rho(n) \leq 3.44\,n$, by introducing a *sparse-neighbors technique*. The neighbors are runs for which both the distance between their starting positions is small and the difference between their periods is also proportionally small (according to some fixed coefficient of proportionality). The occurrences of neighbors satisfy certain *sparsity* properties which imply the linear upper bound. Several variations for the definitions of neighbors and sparsity are possible. Considering runs having close centers the bound has been lowered to $1.6\,n$ in [4].

As a conclusion, we believe that the following fact is valid.

Conjecture: A string of length n contains less than n runs, i.e., $|\mathrm{RUNS}|(n) < n$.

Cross-References

Elements of the present entry are of main importance for run-length compression as well as for ▸ Multidimensional Compressed Pattern Matching. They are also related to the ▸ Approximate Tandem Repeats entries because "tandem repeat" is a synonym of repetition and "power."

Recommended Reading

1. Apostolico A, Preparata FP (1983) Optimal off-line detection of repetitions in a string. Theor Comput Sci 22(3):297–315
2. Crochemore M (1981) An optimal algorithm for computing the repetitions in a word. Inf Process Lett 12(5):244–250

3. Crochemore M (1986) Transducers and repetitions. Theor Comput Sci 45(1):63–86
4. Crochemore M, Ilie L (2007) Analysis of maximal repetitions in strings. J Comput Sci
5. Crochemore M, Rytter W (1995) Squares, cubes, and time-space efficient string searching. Algorithmica 13(5):405–425
6. Crochemore M, Rytter W (2003) Jewels of stringology. World Scientific, Singapore
7. Franek F, Karaman A, Smyth WF (2000) Repetitions in Sturmian strings. Theor Comput Sci 249(2):289–303
8. Fraenkel AS, Simpson RJ (1998) How many squares can a string contain? J Comb Theory Ser A 82:112–120
9. Fraenkel AS, Simpson RJ (1999) The exact number of squares in fibonacci words. Theor Comput Sci 218(1):95–106
10. Franek F, Simpson RJ, Smyth WF (2003) The maximum number of runs in a string. In: Proceedings of the 14-th Australian workshop on combinatorial algorithms. Curtin University Press, Perth, pp 26–35
11. Franek F, Smyth WF, Tang Y (2003) Computing all repeats using suffix arrays. J Autom Lang Comb 8(4):579–591
12. Gusfield D, Stoye J (2004) Linear time algorithms for finding and representing all the tandem repeats in a string. J Comput Syst Sci 69(4):525–546
13. Ilie L (2005) A simple proof that a word of length n has at most 2n distinct squares. J Comb Theory Ser A 112(1):163–164
14. Iliopoulos C, Moore D, Smyth WF (1997) A characterization of the squares in a Fibonacci string. Theor Comput Sci 172:281–291
15. Kolpakov R, Kucherov G (1999) Finding maximal repetitions in a word in linear time. In: Proceedings of the 40th symposium on foundations of computer science. IEEE Computer Society, Los Alamitos, pp 596–604
16. Lothaire M (ed) (2002) Algebraic combinatorics on words. Cambridge University Press, Cambridge
17. Lothaire M (ed) (2005) Applied combinatorics on words. Cambridge University Press, Cambridge
18. Main MG (1989) Detecting leftmost maximal periodicities. Discret Appl Math 25:145–153
19. Main MG, Lorentz RJ (1984) An O(n log n) algorithm for finding all repetitions in a string. J Algorithms 5(3):422–432
20. Rytter W (2006) The structure of subword graphs and suffix trees of Fibonacci words. In: Implementation and application of automata, CIAA 2005. Lecture notes in computer science, vol 3845. Springer, Berlin, pp 250–261
21. Rytter W (2006) The number of runs in a string: improved analysis of the linear upper bound. In: Proceedings of the 23rd annual symposium on theoretical aspects of computer science. Lecture notes in computer science, vol 3884. Springer, Berlin, pp 184–195
22. Smyth WF (2000) Repetitive perhaps, but certainly not boring. Theor Comput Sci 249(2):343–355
23. Smyth WF (2003) Computing patterns in strings. Addison-Wesley, Boston

Stable Marriage

Robert W. Irving
School of Computing Science, University of Glasgow, Glasgow, UK

Keywords

Stable matching

Years and Authors of Summarized Original Work

1962; Gale, Shapley

Problem Definition

The objective in *stable matching problems* is to match together pairs of elements of a set of participants, taking into account the preferences of those involved and focusing on a stability requirement. The stability property ensures that no pair of participants would both prefer to be matched together rather than to accept their allocation in the matching. Such problems have widespread application, for example, in the allocation of medical students to hospital posts, students to schools or colleges, etc.

An instance of the classical *stable marriage problem* (SM), introduced by Gale and Shapley [2], involves a set of $2n$ participants comprising n men $\{m_1, \ldots, m_n\}$ and n women $\{w_1, \ldots, w_n\}$. Associated with each participant is a *preference list*, which is a total order over the participants of the opposite sex. A man m_i *prefers* woman w_j to woman w_k if w_j precedes w_k on the preference list of m_i and similarly for the women.

A *matching* M is a bijection between the sets of men and women, in other words a set of man-woman pairs so that each man and each woman belongs to exactly one pair of M. For a man m_i, $M(m_i)$ denotes the *partner* of m_i in M, i.e., the unique woman w_j such that (m_i, w_j) is in M. Similarly, $M(w_j)$ denotes the partner of woman w_j in M. A matching M is *stable* if there is no *blocking pair*, namely, a pair (m_i, w_j) such that m_i prefers w_j to $M(m_i)$ and w_j prefers m_i to $M(w_j)$.

Relaxing the requirements that the numbers of men and women are equal and that each participant should rank *all* of the members of the opposite sex gives the *stable marriage problem with incomplete lists* (SMI). So an instance of SMI comprises a set of n_1 men $\{m_1, \ldots, m_{n1}\}$ and a set of n_2 women $\{w_1, \ldots, w_{n2}\}$, and each participant's preference list is a total order over a *subset* of the participants of the opposite sex. The implication is that if woman w_j does not appear on the list of man m_i, then she is not an acceptable partner for m_i and vice versa. A man-woman pair is *acceptable* if each member of the pair is on the preference list of the other, and a matching M is now a set of acceptable pairs such that each man and each woman is in *at most* one pair of M. In this context, a blocking pair for matching M is an acceptable pair (m_i, w_j) such that m_i either is unmatched in M or prefers w_j to $M(m_i)$ and, likewise, w_j either is unmatched or prefers m_i to $M(w_j)$. A matching is stable if it has no blocking pair. So in an instance of SMI, a stable matching need not match all of the participants.

Gale and Shapley also introduced a many-one version of stable marriage, which they called the *college admissions problem*, but which is now more usually referred to as the ▸ Hospitals/Residents Problem (HR) because of its well-known applications in the medical employment field. This problem is covered in detail in Entry 150 of this volume.

A comprehensive treatment of many aspects of the stable marriage problem, as of 1989, appears in the monograph of Gusfield and Irving [5]. A more recent detailed exposition is given by Manlove [14].

Key Results

Theorem 1 *For every instance of SM or SMI, there is at least one stable matching.*

Theorem 1 was proved constructively by Gale and Shapley [2] as a consequence of the algorithm that they gave to find a stable matching.

Theorem 2 *1. For a given instance of SM involving n men and n women, there is a $O(n^2)$ time algorithm that finds a stable matching.*
2. For a given instance of SMI in which the combined length of all the preference lists is a, there is a $O(a)$ time algorithm that finds a stable matching.

The algorithm for SMI is a simple extension of that for SM. Each can be formulated in a variety of ways, but is most usually expressed in terms of a sequence of "proposals" from the members of one sex to the members of the other. A pseudocode version of the SMI algorithm appears in Fig. 1, in which the traditional approach of allowing men to make proposals is adopted.

The complexity bound of Theorem 2(1) first appeared in Knuth's monograph on stable marriage [12]. The fact that this algorithm is asymptotically optimal was subsequently established by Ng and Hirschberg [17] via an adversary argument. On the other hand, Wilson [21] proved that the average running time, taken over all possible instances of SM, is $O(n \log n)$.

The algorithm of Fig. 1, in its various guises, has come to be known as the Gale-Shapley algorithm. The variant of the algorithm given here is called *man oriented*, because men have the advantage of proposing. Reversing the roles of men and women gives the *woman-oriented* variant. The "advantage" of proposing is remarkable, as spelled out in the next theorem.

Theorem 3 *The man-oriented version of the Gale-Shapley algorithm for SM or SMI yields the man-optimal stable matching in which each man has the best partner that he can have in any stable matching, but in which each woman has her worst possible partner. The woman-oriented version yields the woman-optimal stable matching, which has analogous properties favoring the women.*

```
M = ∅;
assign each person to be free;      /* i. e., not a member of a pair in M */
while (some man m is free and has not proposed to every woman on his list)
    m proposes to w, the first woman on his list to whom he has not proposed;
    if (w is free)
        add (m, w) to M;            /* w accepts m */
    else if (w prefers m to her current partner m′)
        remove (m′, w) from M;      /* w rejects m′, setting m′ free */
        add (m, w) to M;            /* w accepts m */
    else
        M remains unchanged;        /* w rejects m */
return M;
```

Stable Marriage, Fig. 1 The Gale-Shapley algorithm

The optimality property of Theorem 3 was established by Gale and Shapley [2], and the corresponding "pessimality" property was first observed by McVitie and Wilson [16].

As observed earlier, a stable matching for an instance of SMI need not match all of the participants. But the following striking result was established by Gale and Sotomayor [3] and Roth [19] (in the context of the more general HR problem).

Theorem 4 *In an instance of SMI, all stable matchings have the same size and match exactly the same subsets of the men and women.*

For a given instance of SM or SMI, there may be many different stable matchings. Indeed Knuth [12] showed that the maximum possible number of stable matchings grows exponentially with the number of participants. He also pointed out that the set of stable matchings forms a distributive lattice under a natural dominance relation, a result attributed to Conway. This powerful algebraic structure that underlies the set of stable matchings can be exploited algorithmically in a number of ways. For example, Gusfield [4] showed how all k stable matchings for an instance of SM can be generated in $O(n^2 + kn)$ time (▸ Optimal Stable Marriage).

Extensions of these problems that are important in practice, so-called SMT and SMTI (extensions of SM and SMI, respectively), allow the presence of *ties* in the preference lists. In this context, three different notions of stability have been defined [7] – *weak*, *strong*, and *super-stability*, depending on whether the definition of a blocking pair requires that both members should improve, or at least one member improves and the other is no worse off, or merely that neither member is worse off. The following theorem summarizes the basic algorithmic results for these three varieties of stable matchings.

Theorem 5 *For a given instance of SMT or SMTI:*

1. *A weakly stable matching is guaranteed to exist and can be found in $O(n^2)$ or $O(a)$ time, respectively.*
2. *A super-stable matching may or may not exist; if one does exist, it can be found in $O(n^2)$ or $O(a)$ time, respectively.*
3. *A strongly stable matching may or may not exist; if one does exist, it can be found in $O(n^3)$ or $O(na)$ time, respectively.*

Theorem 5 parts (1) and (2) are due to Irving [7] (for SMT) and Manlove [13] (for SMTI). Part (3) is due to Kavitha et al. [11], who improved earlier algorithms of Irving and Manlove.

It turns out that, in contrast to the situation described by Theorem 4, weakly stable matchings in SMTI can have different sizes. The natural problem of finding a maximum cardinality

weakly stable matching, even under severe restrictions on the ties, is NP-hard [15]. ▸ Stable Marriage with Ties and Incomplete Lists explores this problem further.

Interesting special cases of SM and its variants arise when the preference lists on one or both sides are derived from a "master" list that ranks participants (e.g.,, according to some objective criterion). Such problems are explored by Irving et al. [10].

The stable marriage problem is an example of a *bipartite* matching problem. The extension in which the bipartite requirement is dropped is the so-called *stable roommates* (SR) problem.

Gale and Shapley had observed that, unlike the case of SM, an instance of SR may or may not admit a stable matching, and Knuth [12] posed the problem of finding an efficient algorithm for SR or proving it NP-complete. Irving [6] established the following theorem via a nontrivial extension of the Gale-Shapley algorithm.

Theorem 6 *For a given instance of SR, there exists a $O(n^2)$ time algorithm to determine whether a stable matching exists and if so to find such a matching.*

Variants of SR may be defined, as for SM, in which preference lists may be incomplete and/or contain ties – these are denoted by SRI, SRT, and SRTI – and in the presence of ties, the three flavors of stability, weak, strong, and super, are again relevant.

Theorem 7 *For a given instance of SRT or SRTI:*

1. *A weakly stable matching may or may not exist, and it is an NP-complete problem to determine whether such a matching exists.*
2. *A super-stable matching may or may not exist; if one does exist, it can be found in $O(n^2)$ or $O(a)$ time, respectively.*
3. *A strongly stable matching may or may not exist; if one does exist, it can be found in $O(n^4)$ or $O(a^2)$ time, respectively.*

Theorem 7 part (1) is due to Ronn [18], part (2) is due to Irving and Manlove [9], and part (3) is due to Scott [20].

Applications

Undoubtedly the best known and most important applications of stable matching algorithms are in centralized matching schemes in the medical and educational domains. ▸ Hospitals/Residents Problem includes a summary of some of these applications.

Open Problems

The parallel complexity of stable marriage remains open. The best known parallel algorithm for SMI is due to Feder et al. [1] and has $O(\sqrt{a}\log^3 a)$ running time using a polynomially bounded number of processors. It is not known whether the problem is in NC, but nor is there a proof of P-completeness.

One of the open problems posed by Knuth in his early monograph on stable marriage [12] was that of determining the maximum possible number x_n of stable matchings for any SM instance involving n men and n women. This problem remains open, although Knuth himself showed that x_n grows exponentially with n. Irving and Leather [8] conjecture that, when n is a power of 2, this function satisfies the recurrence

$$x_n = 3x_{n/2}^2 - 2x_{x/4}^4.$$

Many open problems remain in the setting of weak stability, such as finding a good approximation algorithm for a maximum cardinality weakly stable matching – see ▸ Stable Marriage with Ties and Incomplete Lists – and enumerating all weakly stable matchings efficiently.

Cross-References

▸ Hospitals/Residents Problem
▸ Optimal Stable Marriage

S

▸ Ranked Matching
▸ Stable Marriage and Discrete Convex Analysis
▸ Stable Marriage with Ties and Incomplete Lists
▸ Stable Partition Problem

Recommended Reading

1. Feder T, Megiddo N, Plotkin SA (2000) A sublinear parallel algorithm for stable matching. Theor Comput Sci 233(1–2):297–308
2. Gale D, Shapley LS (1962) College admissions and the stability of marriage. Am Math Mon 69:9–15
3. Gale D, Sotomayor M (1985) Some remarks on the stable matching problem. Discret Appl Math 11:223–232
4. Gusfield D (1987) Three fast algorithms for four problems in stable marriage. SIAM J Comput 16(1):111–128
5. Gusfield D, Irving RW (1989) The stable marriage problem: structure and algorithms. MIT, Cambridge
6. Irving RW (1985) An efficient algorithm for the stable roommates problem. J Algorithms 6:577–595
7. Irving RW (1994) Stable marriage and indifference. Discret Appl Math 48:261–272
8. Irving RW, Leather P (1986) The complexity of counting stable marriages. SIAM J Comput 15(3):655–667
9. Irving RW, Manlove DF (2002) The stable roommates problem with ties. J Algorithms 43:85–105
10. Irving RW, Manlove DF, Scott S (2008) The stable marriage problem with master preference lists. Discret Appl Math 156:2959–2977
11. Kavitha T, Mehlhorn K, Michail D, Paluch K (2004) Strongly stable matchings in time O(nm), and extension to the H/R problem. In: Proceedings of the 21st symposium on theoretical aspects of computer science (STACS 2004). Lecture notes in computer science, vol 2996, pp 222–233. Springer, Berlin
12. Knuth DE (1976) Mariages stables. Les Presses de L'Université de Montréal, Montréal
13. Manlove DF (1999) Stable marriage with ties and unacceptable partners. Technical report TR-1999-29, Department of Computing Science, University of Glasgow
14. Manlove DF (2013) Algorithmics of matching under preferences. World Scientific, Singapore
15. Manlove DF, Irving RW, Iwama K, Miyazaki S, Morita Y (2002) Hard variants of stable marriage. Theor Comput Sci 276(1–2):261–279
16. McVitie D, Wilson LB (1971) The stable marriage problem. Commun ACM 14:486–490
17. Ng C, Hirschberg DS (1990) Lower bounds for the stable marriage problem and its variants. SIAM J Comput 19:71–77
18. Ronn E (1990) NP-complete stable matching problems. J Algorithms 11:285–304
19. Roth AE (1984) The evolution of the labor market for medical interns and residents: a case study in game theory. J Polit Econ 92(6):991–1016
20. Scott S (2005) A study of stable marriage problems with ties. Ph.D. thesis, Department of Computing Science, University of Glasgow
21. Wilson LB (1972) An analysis of the stable marriage assignment algorithm. BIT 12:569–575

Stable Marriage and Discrete Convex Analysis

Akihisa Tamura
Department of Mathematics, Keio University, Yokohama, Japan

Keywords

Stable matching

Years and Authors of Summarized Original Work

2000; Eguchi, Fujishige, Tamura, Fleiner

Problem Definition

In the stable marriage problem first defined by Gale and Shapley [7], there is one set each of men and women having the same size, and each person has a strict preference order on persons of the opposite gender. The problem is to find a matching such that there is no pair of a man and a woman who prefer each other to their partners in the matching. Such a matching is called a *stable marriage* (or *stable matching*). Gale and Shapley showed the existence of a stable marriage and gave an algorithm for finding one. Fleiner [4] extended the stable marriage problem to the framework of matroids, and Eguchi, Fujishige, and Tamura [3] extended this formulation to a more general one in terms of discrete convex analysis, which was developed by Murota [8, 9]. Their formulation is described as follows.

Let M and W be sets of men and women who attend a dance party at which each person dances a waltz T times and the number of times that

he/she can dance with the same person of the opposite gender is unlimited. The problem is to find an "agreeable" allocation of dance partners, in which each person is assigned at most T persons of the opposite gender with possible repetition. Let $E = M \times W$, i.e., the set of all man-woman pairs. Also define $E_{(i)} = \{i\} \times W$ for all $i \in M$ and $E_{(j)} = M \times \{j\}$ for all $j \in W$. Denoting by $x(i, j)$ the number of dances between man i and woman j, an allocation of dance partners can be described by a vector $x = (x(i, j) : i \in M, j \in W) \in \mathbf{Z}^E$, where $\mathbf{Z}$ denotes the set of all integers. For each $y \in \mathbf{Z}^E$ and $k \in M \cup W$, denote by $y_{(k)}$ the restriction of y on $E_{(k)}$. For example, for an allocation $x \in \mathbf{Z}^E$, $x_{(k)}$ represents the allocation of person k with respect to x. Each person k describes his/her preferences on allocations by using a value function $f_k : \mathbf{Z}^{E_{(k)}} \to \mathbf{R} \cup (-\infty)$, where $\mathbf{R}$ denotes the set of all reals and $f_k(y) = -\infty$ means that allocation $y \in \mathbf{Z}^{E_{(k)}}$ is unacceptable for k. Note that the valuation of each person on allocations is determined only by his/her allocations. Let dom $f_k = \{y | f_k(y) \in \mathbf{R}\}$. Assume that each value function f_k satisfies the following assumption:

(A) dom f_k is bounded and hereditary and has $\mathbf{0}$ as the minimum point, where $\mathbf{0}$ is the vector of all zeros and heredity means that for any $y, y' \in \mathbf{Z}^{E_{(k)}}$, $\mathbf{0} \le y' \le y \in$ dom f_k implies $y' \in$ dom f_k.

For example, the following value functions with $M = \{1\}$ and $W = \{2, 3\}$

$$f_1(x(1,2), x(1,3)) =$$
$$\begin{cases} 10(x(1,2) + x(1,3)) - x(1,2)^2 - x(1,3)^2 & \text{if } x(1,2), x(1,3) \ge 0 \\ & \text{and } x(1,2) + x(1,3) \le 3 \\ -\infty & \text{otherwise,} \end{cases}$$

$$f_j(x(1,j)) = \begin{cases} x(1,j) & \text{if } x(1,j) \in \{0,1,2,3\} (j = 2,3) \\ -\infty & \text{otherwise} \end{cases}$$

represent the case where (1) everyone wants to dance as many times, up to three, as possible and (2) man 1 wants to divide his dances between women 2 and 3 as equally as possible. Allocations $(x(1,2), x(1,3)) = (1,2)$ and (2,1) are stable in the sense below.

A vector $x \in \mathbf{Z}^E$ is called a *feasible allocation* if $x_{(k)} \in$ dom f_k for all $k \in M \cup W$. An allocation x is said to satisfy *incentive constraints* if each person has no incentive to unilaterally decrease the current units of x, that is, if it satisfies

$$f_k(x_{(k)}) = \max\{f_k(y) | y \le x_{(k)}\} \ (\forall k \in M \cup W). \quad (1)$$

An allocation x is called *unstable* if it does not satisfy incentive constraints or there exist $i \in M$, $j \in W$, $y' \in \mathbf{Z}^{E_{(i)}}$ and $y'' \in \mathbf{Z}^{E_{(j)}}$ such that

$$f_i(x_{(i)}) < f_i(y'), \quad (2)$$

$$y'(i, j') \le x(i, j') \quad (\forall j' \in W \backslash \{j\}), \quad (3)$$

$$f_j(x_{(j)}) < f_j(y''), \quad (4)$$

$$y''(i', j) \le x(i', j) \quad (\forall i' \in M \backslash \{i\}), \quad (5)$$

$$y'(i, j) = y''(i, j). \quad (6)$$

Conditions (2) and (3) say that man i can strictly increase his valuation by changing the current number of dances with j without increasing the numbers of dances with other women, and (4) and (5) describe a similar situation for women. Condition (6) requires that i and j agree on the number of dances between them. An allocation x is called *stable* if it is not unstable.

Problem 1 Given disjoint sets M and W and value functions $f_k : \mathbf{Z}^{E_{(k)}} \to \mathbf{R} \cup \{-\infty\}$ for $k \in M \cup W$ satisfying assumption (A), find a stable allocation x.

S

Remark 1 A time schedule for a given feasible allocation can be given by a famous result on graph coloring, namely, "any bipartite graph can be edge-colorable with the maximum degree colors."

Key Results

The work of Eguchi, Fujishige, and Tamura [3] gave a solution to Problem 1 in the case where each value function f_k is $\mathrm{M}^\natural$-concave.

Discrete Convex Analysis: $\mathrm{M}^\natural$-Concave Functions

Let V be a finite set. For each $S \subseteq V$, e_S denotes the characteristic vector of S defined by $e_S(v) = 1$ if $v \in S$ and $e_S(v) = 0$ otherwise. Also define e_0 as the zero vector in $\mathbf{Z}^V$. For a vector $x \in \mathbf{Z}^V$, its positive support $\mathrm{supp}^+(x)$ and negative support $\mathrm{supp}^-(x)$ are defined by $\mathrm{supp}^+(x) = \{u \in V | x(u) > 0\}$ and $\mathrm{supp}^-(x) = \{u \in V | x(u) < 0\}$. A function $f : \mathbf{Z}^V \to \mathbf{R} \cup \{-\infty\}$ is called *$M^\natural$-concave* if it satisfies the following condition $\forall\, x, y \in \mathrm{dom}\, f$, $\forall\, u \in \mathrm{supp}^+(x-y)$, $\exists v \in \mathrm{supp}^-(x-y) \cup \{0\}$:

$$f(x) + f(y) \leq f(x - e_u + e_v) + f(y + e_u - e_v).$$

The above condition says that the sum of the function values at two points does not decrease as the points symmetrically move one or two steps closer to each other on the set of integral lattice points of $\mathbf{Z}^V$. This is a discrete analogue of the fact that for an ordinary concave function, the sum of the function values at two points does not decrease as the points symmetrically move closer to each other on the straight line segment between the two points.

Example 1 A nonempty family $\mathcal{T}$ of subsets of V is called a *laminar family* if $X \cap Y = \emptyset$, $X \subseteq Y$ or $Y \subseteq X$ holds for every $X, Y \in \mathcal{T}$. For a laminar family $\mathcal{T}$ and a family of univariate concave functions $f_Y : \mathbf{R} \to \mathbf{R} \cup \{-\infty\}$ indexed by $Y \in \mathcal{T}$, the function $f : \mathbf{Z}^V \to \mathbf{R} \cup \{-\infty\}$ defined by

$$f(x) = \sum_{Y \in \mathcal{T}} f_Y \left(\sum_{v \in Y} x(v) \right) \quad (\forall\, x \in \mathbf{Z}^V)$$

is $\mathrm{M}^\natural$-concave. The stable marriage problem can be formulated as Problem 1 by using value functions of this type.

Example 2 For the independence family $\mathcal{I} \subseteq 2^V$ of a matroid on V and $w \in \mathbf{R}^V$, the function $f : \mathbf{Z}^V \to \mathbf{R} \cup \{-\infty\}$ defined by

$$f(x) = \begin{cases} \sum_{u \in X} w(u) & \text{if } x = e_X \text{ for some } X \in \mathcal{I} \\ -\infty & \text{otherwise} \end{cases} \quad (\forall\, x \in \mathbf{Z}^V)$$

is $\mathrm{M}^\natural$-concave. Fleiner [4] showed that there always exists a stable allocation for value functions of this type.

Theorem 1 ([6]) *Assume that the value functions $f_k (k \in M \cup W)$ are $\mathrm{M}^\natural$-concave satisfying (A). Then, a feasible allocation x is stable if and only if there exist $Z_M = (z_{(i)} | i \in M) \in (\mathbf{Z} \cup \{+\infty\})^E$ and $z_W = (z_{(j)} | j \in W) \in (\mathbf{Z} \cup \{+\infty\})^E$ such that*

$$x_{(i)} \in \arg\max\{f_i(y) | y \leq z_{(i)}\} \quad (\forall\, i \in M), \tag{7}$$

$$x_{(j)} \in \arg\max\{f_j(y) | y \leq z_{(j)}\} \quad (\forall\, j \in W), \tag{8}$$

$$z_M(e) = +\infty \text{ or } z_W(e) = +\infty \quad (\forall\, e \in E), \tag{9}$$

where $\arg\max\{f_i(y) | y \leq z_{(i)}\}$ denotes the set of all maximizers of f_i under the constraints $y \leq z_{(i)}$.

Theorem 2 ([3]) *Assume that the value functions $f_k (k \in M \cup W)$ are $\mathrm{M}^\natural$-concave satisfying (A). Then, there always exists a stable allocation.*

Eguchi, Fujishige, and Tamura [3] proved Theorem 2 by showing that the following algorithm finds a feasible allocation x, and z_M, z_W satisfying (7), (8), and (9).

Here, $z_W \vee x_M$ is defined by $(z_W \vee x_M)(e) = \max\{z_W(e), x_M(e)\}$ for all $e \in E$.

Algorithm EXTENDED-GS

Input: $M^\natural$-concave functions f_M, f_W with $f_M(x) = \sum_{i \in M} f_i(x_{(i)})$ and $f_W(x) = \sum_{j \in W} f_j(x_{(j)})$;
Output: (x, z_M, z_W) satisfying (7), (8), and (9);
$z_M := (+\infty, \cdots, +\infty)$, $z_W := x_W := \mathbf{0}$;
repeat{
let x_M be any element in
$\arg\max\{f_M(y) | x_W \le y \le z_M\}$;
let x_W be any element in
$\arg\max\{f_W(y) | y \le x_M\}$;
for each $e \in E$ with $x_M(e) > x_W(e)$ {
$z_M(e) := x_W(e)$;
$z_W(e) := +\infty$;
};
} **until** $x_M = x_W$;
return $(x_M, z_M, z_W \vee x_M)$.

Applications

Abraham, Irving, and Manlove [1] dealt with a student-project allocation problem which is a concrete example of models in [4] and [3] and discussed the structure of stable allocations.

Fleiner [5] generalized the stable marriage problem and its extension in [4] to a wide framework and showed the existence of a stable allocation by using a fixed point theorem.

Fujishige and Tamura [6] proposed a common generalization of the stable marriage problem and the assignment game defined by Shapley and Shubik [10] by utilizing $M^\natural$-concave functions and gave a constructive proof of the existence of a stable allocation.

Open Problems

Algorithm EXTENDED-GS solves the maximization problem of an $M^\natural$-concave function in each iteration. A maximization problem of an $M^\natural$-concave function f on E can be solved in polynomial time in $|E|$ and $\log L$, where $L = \max\{||x - y||_\infty | x, y \in \text{dom } f\}$, provided that the function value $f(x)$ can be calculated in constant time for each x [11, 12]. Eguchi, Fujishige, and Tamura [3] showed that EXTENDED-GS terminates after at most L iterations, where L is defined by $\{||x||_\infty | x \in \text{dom } f_M\}$ in this case, and there exist a series of instances in which EXTENDED-GS requires numbers of iterations proportional to L. On the other hand, Baïou and Balinski [2] gave a polynomial time algorithm in $|E|$ for the special case where f_M and f_W are linear on rectangular domains. Whether a stable allocation for the general case can be found in polynomial time in $|E|$ and $\log L$ or not is open.

Cross-References

- Assignment Problem
- Hospitals/Residents Problem
- Optimal Stable Marriage
- Stable Marriage
- Stable Marriage with Ties and Incomplete Lists

Recommended Reading

1. Abraham DJ, Irving RW, Manlove DF (2007) Two algorithms for the student-project allocation problem. J Discret Algorithms 5:73–90
2. Baïou M, Balinski, M (2002) Erratum: the stable allocation (or ordinal transportation) problem. Math Oper Res 27:662–680
3. Eguchi A, Fujishige S, Tamura A (2003) A generalized Gale-Shapley algorithm for a discrete-concave stable-marriage model. In: Ibaraki T, Katoh N, Ono H (eds) Algorithms and computation: 14th international symposium (ISAAC 2003), Kyoto. LNCS, vol 2906. Springer, Berlin, pp 495–504
4. Fleiner T (2001) A matroid generalization of the stable matching polytope. In: Gerards B, Aardal K (eds) Integer programming and combinatorial optimization: 8th international IPCO conference, Utrecht. LNCS, vol 2081. Springer, Berlin, pp 105–114
5. Fleiner T (2003) A fixed point approach to stable matchings and some applications. Math Oper Res 28:103–126
6. Fujishige S, Tamura A (2007) A two-sided discrete-concave market with bounded side payments: an approach by discrete convex analysis. Math Oper Res 32:136–155
7. Gale D, Shapley SL (1962) College admissions and the stability of marriage. Am Math Mon 69:9–15
8. Murota K (1998) Discrete convex analysis. Math Program 83:313–371
9. Murota K (2003) Discrete convex analysis. Society for Industrial and Applied Mathematics, Philadelphia
10. Shapley SL, Shubik M (1971) The assignment game I: the core. Int J Game Theor 1:111–130

11. Shioura A (2004) Fast scaling algorithms for M-convex function minimization with application to the resource allocation problem. Discret Appl Math 134:303–316
12. Tamura A (2005) Coordinatewise domain scaling algorithm for M-convex function minimization. Math Program 102:339–354

Stable Marriage with One-Sided Ties

Hiroki Yanagisawa
IBM Research – Tokyo, Tokyo, Japan

Keywords

Approximation algorithms; Incomplete lists; Integer programming; Linear programming relaxation; One-sided ties; Stable marriage problem

Years and Authors of Summarized Original Work

2007; Halldórsson, Iwama, Miyazaki, Yanagisawa
2014; Huang, Iwama, Miyazaki, Yanagisawa

Problem Definition

Over the last 50 years, the stable marriage problem has been extensively studied for many problem settings (see, e.g., [11]), and one of the most intensively studied problem settings is MAX SMTI (MAXimum Stable Marriage with Ties and Incomplete lists). An input for the stable marriage problem consists of n men, n women, and each person's preference list for the people of the opposite sex. In MAX SMTI, the preference list of each person can be incomplete, which means that each person is allowed to exclude unacceptable people from the preference list, and the preference list of each person is allowed to include *ties* to show indifference between two or more people.

The objective of MAX SMTI is to find the largest matching that satisfies a stability condition. Before describing the stability condition, we review some notation. A matching M is defined as a set of pairs of man m and woman w such that m and w are acceptable to each other. The *size* of a matching M is defined as the number of pairs in M. We say that a person p is *single* if p is not matched in M. When man m and woman w are matched in M, we write $M(m) = w$ and $M(w) = m$. We say that matching M is *stable* if it does not contain any pair of man and woman, each of whom prefers the other to the partner in M (if any). More precisely, a matching M is stable if there is no pair of man m' and woman w' that satisfy all three conditions (i)–(iii): (i) m' and w' are acceptable to each other but not matched in M, (ii) m' is single in M or m' strictly prefers w' to $M(m')$, and (iii) w' is single in M or w' strictly prefers m' to $M(w')$. MAX SMTI asks us to find a stable matching of the largest size, and this problem is known to be NP-hard [12]. Therefore, the approximability of this problem has been intensively studied.

In this entry, we show recent results for two major variants of MAX SMTI. One of the variants is MAX SMOTI (MAXimum Stable Marriage with One-Sided Ties and Incomplete lists), in which only women are allowed to include ties in their preference lists and the preference lists of men are strictly ordered. The other variant is MAX SSMTI (Special SMTI), which is an even more restricted variant of MAX SMOTI where the ties are only allowed at the ends of the women's preference lists. Note that these two variants are still known to be NP-hard [12].

Problem 1 (MAX SMOTI)
INPUT: n men, n women, and each person's preference list, where only women have ties
OUTPUT: A stable matching of maximum size

Problem 2 (MAX SSMTI)
INPUT: n men, n women, and each person's preference list, where ties are at the ends of the women's preference lists
OUTPUT: A stable matching of maximum size

Stable Marriage with One-Sided Ties, Table 1 Examples of instances for MAX SMOTI and MAX SSMTI

MAX SMOTI		MAX SSMTI	
$m_1 : w_2\ w_1$	$w_1 : m_1\ m_2$	$m_1 : w_1\ w_3$	$w_1 : (m_1\ m_2\ m_3)$
$m_2 : w_2\ w_3\ w_1$	$w_2 : (m_1\ m_2)\ m_3$	$m_2 : w_2\ w_3\ w_1$	$w_2 : m_2$
$m_3 : w_3\ w_2$	$w_3 : m_2\ m_3$	$m_3 : w_3\ w_1$	$w_3 : m_2\ (m_1\ m_3)$

Examples

Table 1 shows examples of instances for MAX SMOTI and MAX SSMTI. The instance for MAX SMOTI contains a set of men $\{m_1, m_2, m_3\}$ and a set of women $\{w_1, w_2, w_3\}$. The preference list of each person is described in decreasing order of preference, and tied people are enclosed in a pair of parenthesis. For example, woman w_2 is indifferent between m_1 and m_2 but prefers m_1 or m_2 over m_3. A matching $M = \{(m_2, w_1), (m_3, w_2)\}$ is not stable for this MAX SMOTI instance, because m_2 strictly prefers w_2 to w_1 $(= M(m_2))$ and w_2 strictly prefers m_2 to m_3 $(= M(w_2))$. An example of a stable matching for this instance is $M' = \{(m_1, w_2), (m_2, w_3)\}$, and we can find another larger stable matching $M^* = \{(m_1, w_1), (m_2, w_2), (m_3, w_3)\}$ of size 3.

Key Results

Here we review past research on MAX SMOTI and MAX SSMTI. We start by describing a simple proposal-based algorithm (often referred to as the Gale-Shapley algorithm or the deferred acceptance algorithm), which is guaranteed to find a *stable* matching. In this algorithm, all of the men and women are initially set to be single. We pick an arbitrary man m who is single, and let man m propose to woman w at the top of his preference list. When man m proposes to w, he deletes woman w from his preference list. Woman w always accepts any proposal if she is single, which makes a matching pair of m and w. We repeat this proposal procedure to find more and more matching pairs. When a woman w, who is already matched to a man m, receives another proposal from man m', woman w chooses the more highly ranked man based on her preference list. (That is, the matching partner of w is unchanged if w prefers m to m', and the matching partner of w is changed from m to m' and m becomes unmatched if w prefers m' to m.) If m and m' are tied in w's preference list, then w chooses an arbitrary man. The proposal procedure continues until we cannot find any man who can propose. (That is, this algorithm terminates when all of the men become matched or the preference lists of all single men become empty.) Any matching obtained by this algorithm can be proven to be stable. The size of the obtained stable matching depends mostly on the decisions by women when a woman receives two proposals from men who are tied in her preference list. In the worst case, the size of an obtained matching can be half of the optimum matching, and hence, the approximation ratio of this algorithm is 2. It was an open problem whether or not there exists an approximation algorithm whose approximation ratio is strictly better than 2. Iwama, Miyazaki, and Yamauchi [8] provided an affirmative answer for this open problem with a 1.875-approximation algorithm.

After this breakthrough, Király [10] developed a new simple 1.5-approximation algorithm for MAX SMOTI (which also applies to MAX SSMTI) by improving the decision strategy of the proposal-based algorithm when women receive multiple proposals from tied men. His algorithm proceeds in the same way as the proposal-based algorithm until one of the men's preference lists become empty. When the preference list of a man becomes empty, he enters into his second round. Specifically, he recovers his original preference list so that he can propose to the women in his original preference list again, but his status is changed to "promoted." A promoted man is not allowed to recover his original preference list when his preference list becomes empty the second time, and hence, no man can enter a

third round in Kiráry's algorithm. The decision strategy of the women is changed so that a woman is forced to choose a promoted man (if one exists) when she receives two proposals from men who are tied in her preference list. This improvement of the decision strategy is the key to achieve the 1.5-approximation.

Iwama, Miyazaki, and Yanagisawa [9] further improved the approximation ratio to 25/17 (< 1.4706) for MAX SMOTI with a new algorithm GSA-LP, which uses a more complex proposal sequence of the men and a more sophisticated decision strategy for the women. In GSA-LP, we compute an optimum solution for a linear programming relaxation of a natural integer programming formulation of the problem in advance and use it for the decision strategy of the women. In addition, the proposal sequence is changed so that a man can propose to a woman many times, and a man is allowed to recover his original preference list at most twice (in other words, a man is allowed to go into a third round). These changes yield an improved approximation ratio for MAX SMOTI. Very recently, GSA-LP was shown to achieve a 1.25-approximation for MAX SSMTI [5].

For MAX SMOTI, there are successive improvements over GSA-LP. Huang and Kavitha [4] developed another new algorithm that achieves a 22/15 (<1.4667)-approximation by using a maximum matching algorithm. Radnai [14] showed 41/28 (<1.4643)-approximation by using a more detailed analysis of this new algorithm and also showed that a lower bound of the approximation ratio of this algorithm is at least 13/9 (>1.4444). Dean and Jalasutram improved the analysis of GSA-LP and showed that the approximation ratio of GSA-LP is at most 19/13 (<1.4616) if we increase the number of rounds from three to four [1]. We also note that if the lengths of ties are restricted to two, then the approximation ratio of this restricted MAX SMOTI variant can be further improved. A randomized algorithm [2] achieves 10/7 (<1.4286)-approximation and Huang and Kavitha devised another deterministic algorithm [4] with the same approximation ratio.

For the negative side, both MAX SMOTI and MAX SSMTI are NP-hard to approximate within any constant factor better than 21/19 (>1.1052) and hard to approximate within any constant factor better than 5/4 (= 1.25) under the unique games conjecture [3, 15]. These lower bounds hold even if we restrict the lengths of the ties to two. Note that the approximation ratio of the GSA-LP algorithm for MAX SSMTI is 1.25, which matches the lower bound under the unique games conjecture.

Applications

MAX SSMTI was introduced by Irving and Manlove [6] based on an actual application of the Scottish Foundation Allocation Scheme, which allocates residents (medical students) to hospitals. In this scheme, each resident submits a strictly ordered preference list, while each hospital submits a preference list that may contain one tie of arbitrary length at the end of the list. The objective of this allocation scheme is to maximize the number of allocated residents, and it is easy to reformulate this many-to-one allocation scheme as a one-to-one matching problem (MAX SSMTI) using a cloning technique [11].

Open Problems

An obvious future goal is to narrow the gap between the upper and lower bounds of the approximability of MAX SMOTI. Assuming the unique games conjecture is true, we now know that the best possible approximation ratio is between 1.4616 and 1.25. Even if we restrict the lengths of ties to two, all we can do now is reduce the upper bound slightly down to 1.4286. Thus, there is still much room for improvement.

As for MAX SSMTI, the 1.25-approximation of the GSA-LP algorithm is the best possible if the unique games conjecture is true. A future project could investigate if we can construct a faster approximation algorithm, because the

GSA-LP algorithm uses a linear programming relaxation technique, which takes superlinear time in the worst case.

Experimental Results

Irving and Manlove [7] reported on experimental evaluations of some heuristic algorithms including the Király's algorithm on real-world and random instances for MAX SMOTI. Subsequently, Podhradský [13] conducted experimental evaluations on random instances for MAX SMOTI and MAX SSMTI using some other heuristic algorithms including GSA-LP.

Cross-References

- ▸ Simpler Approximation for Stable Marriage
- ▸ Stable Marriage
- ▸ Stable Marriage with Ties and Incomplete Lists

Recommended Reading

1. Dean BC, Jalasutram R (2015) Factor revealing LPs and stable matching with ties and incomplete lists. In Proceedings of the 3rd international workshop on matching under preferences, 2015 (to appear)
2. Halldórsson MM, Iwama K, Miyazaki S, Yanagisawa H (2004) Randomized approximation of the stable marriage problem. Theor Comput Sci 325(3):439–465
3. Halldórsson MM, Iwama K, Miyazaki S, Yanagisawa H (2007) Improved approximation results for the stable marriage problem. ACM Trans Algorithms 3(3):Article No. 30
4. Huang CC, Kavitha T (2014) An improved approximation algorithm for the stable marriage problem with one-sided ties. In: Proceedings of IPCO 2014, Bonn, pp 297–308
5. Huang CC, Iwama K, Miyazaki S, Yanagisawa H (2015) Approximability of finding largest stable matchings. Manuscript under submission
6. Irving RW, Manlove DF (2008) Approximation algorithms for hard variants of the stable marriage and hospital/residents problems. J Comb Optim 16(3):279–292
7. Irving RW, Manlove DF (2009) Finding large stable matchings. J Exp Algorithmics 14:Article No. 2
8. Iwama K, Miyazaki S, Yamauchi N (2007) A 1.875: approximation algorithm for the stable marriage problem. In: Proceedings of SODA 2007, New Orleans, pp 288–297
9. Iwama K, Miyazaki S, Yanagisawa H (2014) A 25/17-approximation algorithm for the stable marriage problem with one-sided ties. Algorithmica 68(3):758–775
10. Király Z (2011) Better and simpler approximation algorithms for the stable marriage problem. Algorithmica 60(1):3–20
11. Manlove DF (2013) Algorithmics of matching under preferences. World Scientific, Hackensack
12. Manlove DF, Irving RW, Iwama K, Miyazaki S, Morita Y (2002) Hard variants of stable marriage. Theor Comput Sci 276(1–2):261–279
13. Podhradský A (2010) Stable marriage problem algorithms. Master's thesis, Faculty of Informatics, Masaryk University
14. Radnai A (2014) Approximation algorithms for the stable matching problem. Master's thesis, Eötvös Loránd University
15. Yanagisawa H (2007) Approximation algorithms for stable marriage problems. Ph.D. thesis, Kyoto University

Stable Marriage with Ties and Incomplete Lists

Kazuo Iwama[1,2] and Shuichi Miyazaki[3]
[1]Computer Engineering, Kyoto University, Sakyo, Kyoto, Japan
[2]School of Informatics, Kyoto University, Sakyo, Kyoto, Japan
[3]Academic Center for Computing and Media Studies, Kyoto University, Kyoto, Japan

Synonyms

Stable matching problem

Keywords

Approximation algorithms; Incomplete lists; Stable marriage problem; Ties

Years and Authors of Summarized Original Work

2007; Iwama, Miyazaki, Yamauchi

Problem Definition

In the original setting of the stable marriage problem introduced by Gale and Shapley [2], each preference list has to include all members of the other party, and furthermore, each preference list must be totally ordered (see entry ▸ Stable Marriage also).

One natural extension of the problem is then to allow persons to include ties in preference lists. In this extension, there are three variants of the stability definition, super-stability, strong stability, and weak stability (see below for definitions). In the first two stability definitions, there are instances that admit no stable matching, but there is a polynomial-time algorithm in each case that determines if a given instance admits a stable matching and finds one if one exists [9]. On the other hand, in the case of weak stability, there always exists a stable matching, and one can be found in polynomial time.

Another possible extension is to allow persons to declare unacceptable partners, so that preference lists may be incomplete. In this case, every instance admits at least one stable matching, but a stable matching may not be a perfect matching. However, if there are two or more stable matchings for one instance, then all of them have the same size [3].

The problem treated in this entry allows both extensions simultaneously, which is denoted as SMTI (stable marriage with ties and incomplete lists).

Notations

An instance I of SMTI comprises n men, n women, and each person's preference list that may be incomplete and may include ties. If a man m includes a woman w in his list, w is *acceptable* to m. $w_i \succ_m w_j$ means that m strictly prefers w_i to w_j in I. $w_i =_m w_j$ means that w_i and w_j are tied in m's list (including the case $w_i = w_j$). The statement $w_i \succeq_m w_j$ is true if and only if $w_i \succ_m w_j$ or $w_i =_m w_j$. Similar notations are used for women's preference lists. A matching M is a set of pairs (m, w) such that m is acceptable to w, and vice versa, and each person appears at most once in M. If a man m is matched with a woman w in M, it is written as $M(m) = w$ and $M(w) = m$.

A man m and a woman w are said to form a *blocking pair for weak stability* for M if they are not matched together in M, but by matching them, both become better off, namely, (i) $M(m) \neq w$ but m and w are acceptable to each other, (ii) $w \succ_m M(m)$ or m is single in M, and (iii) $m \succ_w M(w)$ or w is single in M.

Two persons x and y are said to form a *blocking pair for strong stability* for M if they are not matched together in M, but by matching them, one becomes better off, and the other does not become worse off, namely, (i) $M(x) \neq y$ but x and y are acceptable to each other, (ii) $y \succ_x M(x)$ or x is single in M, and (iii) $x \succeq_y M(y)$ or y is single in M.

A man m and a woman w are said to form a *blocking pair for super-stability* for M if they are not matched together in M, but by matching them, neither becomes worse off, namely, (i) $M(m) \neq w$ but m and w are acceptable to each other, (ii) $w \succeq_m M(m)$ or m is single in M, and (iii) $m \succeq_w M(w)$ or w is single in M.

A matching M is called *weakly stable* (*strongly stable* and *super-stable*, respectively) if there is no blocking pair for weak (strong and super, respectively) stability for M.

Problem 1 (SMTI)

INPUT: n men, n women, and each person's preference list
OUTPUT: A stable matching

Problem 2 (MAX SMTI)

INPUT: n men, n women, and each person's preference list
OUTPUT: A stable matching of maximum size

The following problem is a restriction of MAX SMTI in terms of the length of preference lists:

Problem 3 ((p,q)-MAX SMTI)

INPUT: n men, n women, and each person's preference list, where each man's preference list includes at most p women and each woman's preference list includes at most q men
OUTPUT: A stable matching of maximum size

Definition of the Approximation Ratio

A goodness measure of an approximation algorithm T for a maximization problem is defined as follows: the *approximation ratio* of T is $\max\{opt(x)/T(x)\}$ over all instances x of size N, where $opt(x)$ and $T(x)$ are the sizes of the optimal and the algorithm's solutions, respectively.

Key Results

SMTI and MAX SMTI in Super-Stability and Strong Stability

Theorem 1 ([20]) *There is an $O(n^2)$-time algorithm that determines if a given SMTI instance admits a super-stable matching and finds one if one exists.*

Theorem 2 ([17]) *There is an $O(n^3)$-time algorithm that determines if a given SMTI instance admits a strongly stable matching and finds one if one exists.*

It is shown that all stable matchings for a fixed instance are of the same size [20]. Therefore, the above theorems imply that MAX SMTI can also be solved in the same time complexity.

SMTI and MAX SMTI in Weak Stability

In the case of weak stability, every instance admits at least one stable matching, but one instance can have stable matchings of different sizes. If the size is not important, a stable matching can be found in polynomial time by breaking ties arbitrarily and applying the Gale-Shapley algorithm.

Theorem 3 *There is an $O(n^2)$-time algorithm that finds a weakly stable matching for a given SMTI instance.*

However, if larger stable matchings are required, the problem becomes hard.

Theorem 4 ([5, 13, 21, 24]) *MAX SMTI is NP-hard and cannot be approximated within $33/29 - \epsilon$ for any positive constant ϵ unless P=NP. ($33/29 > 1.137$)*

The following approximation ratio is achieved by a local search type algorithm.

Theorem 5 ([14]) *There is a polynomial-time approximation algorithm for MAX SMTI whose approximation ratio is at most 15/8 (=1.875).*

There are a couple of approximation algorithms for restricted inputs.

Theorem 6 ([6]) *There is a polynomial-time randomized approximation algorithm for MAX SMTI whose expected approximation ratio is at most $10/7(\simeq 1.429)$ if, in a given instance, ties appear in one side only and the length of each tie is two.*

Theorem 7 ([6]) *There is a polynomial-time randomized approximation algorithm for MAX SMTI whose expected approximation ratio is at most $7/4(= 1.75)$ if, in a given instance, the length of each tie is two.*

Theorem 8 ([7]) *There is a polynomial-time approximation algorithm for MAX SMTI whose approximation ratio is at most $2/(1 + L^{-2})$ if, in a given instance, ties appear in one side only and the length of each tie is at most L.*

Theorem 9 ([7]) *There is a polynomial-time approximation algorithm for MAX SMTI whose approximation ratio is at most $13/7(\simeq 1.858)$ if, in a given instance, the length of each tie is two.*

(p, q)-MAX SMTI in Weak Stability

Irving et al. [12] show the boundary between P and NP-hardness in terms of the length of preference lists.

Theorem 10 ([12]) *$(2,\infty)$-MAX SMTI is solvable in time $O(n^{\frac{3}{2}} \log n)$.*

Theorem 11 ([12]) *$(3,3)$-MAX SMTI is NP-hard.*

Theorem 12 ([12]) *$(3,4)$-MAX SMTI is NP-hard and cannot be approximated within some constant $\delta(> 1)$ unless P=NP.*

Applications

One of the most famous applications of the stable marriage problem is a centralized assignment system between medical students (residents) and

hospitals. This is an extension of the stable marriage problem to a many-one variant: Each hospital declares the number of residents it can accept, which may be more than one, while each resident has to be assigned to at most one hospital. Actually, there are several applications in the world, known as NRMP in the USA [4], CaRMS in Canada [1], SFAS (previously known as SPA) in Scotland [10, 11], and JRMP in Japan [16]. One of the optimization criteria is clearly the number of matched residents. In a real-world application such as the above hospitals-residents matching, hospitals and residents tend to submit short preference lists that may include ties, in which case, the problem can be naturally considered as MAX SMTI.

Open Problems

An apparent open problem is to narrow the gap of approximability and inapproximability of MAX SMTI in weak stability.

Since the publication of the key result of this chapter (Theorem 5), there have been a lot of improvement. Király [18] presented a linear time 5/3-approximation algorithm (see ▸ Simpler Approximation for Stable Marriage). McDermid [22] then presented a 1.5-approximation algorithm (see ▸ Simpler Approximation for Stable Marriage), and Király [19] and Paluch [23] presented simpler algorithms with the same approximation ratio, which is the current best upper bound. The lower bound was improved by Yanagisawa [24], who showed that MAX SMTI is inapproximable to within a ratio smaller than $33/29(>1.137)$ unless P = NP. He also showed that MAX SMTI is inapproximable within a ratio smaller than $4/3(>1.333)$ under the Unique Games Conjecture (UGC).

As for the special case where ties can appear in one side only (see ▸ Stable Marriage with One-Sided Ties), Király [18] presented a 1.5-approximation algorithm. It was then improved to $25/17(<1.471)$ [15] and to $22/15(<1.467)$ [8], which is the current best upper bound. The current best lower bounds are $21/19(\simeq 1.105)$ under P$\neq$NP and 1.25 under UGC [7].

Cross-References

- ▸ Simpler Approximation for Stable Marriage
- ▸ Assignment Problem
- ▸ Hospitals/Residents Problem
- ▸ Hospitals/Residents Problems with Quota Lower Bounds
- ▸ Optimal Stable Marriage
- ▸ Ranked Matching
- ▸ Stable Marriage
- ▸ Stable Marriage and Discrete Convex Analysis
- ▸ Stable Partition Problem
- ▸ Simpler Approximation for Stable Marriage
- ▸ Stable Marriage with One-Sided Ties

Recommended Reading

1. Canadian Resident Matching Service (CaRMS), http://www.carms.ca/
2. Gale D, Shapley LS (1962) College admissions and the stability of marriage. Am Math Mon 69:9–15
3. Gale D, Sotomayor M (1985) Some remarks on the stable matching problem. Discret Appl Math 11:223–232
4. Gusfield D, Irving RW (1989) The stable marriage problem: structure and algorithms. MIT Press, Boston
5. Halldórsson MM, Irving RW, Iwama K, Manlove DF, Miyazaki S, Morita Y, Scott S (2003) Approximability results for stable marriage problems with ties. Theor Comput Sci 306:431–447
6. Halldórsson MM, Iwama Ka, Miyazaki S, Yanagisawa H (2004) Randomized approximation of the stable marriage problem. Theor Comput Sci 325(3):439–465
7. Halldórsson MM, Iwama Ka, Miyazaki S, Yanagisawa H (2007) Improved approximation results of the stable marriage problem. ACM Trans Algorithms 3(3):Article No. 30
8. Huang C-C, Kavitha T (2014) An improved approximation algorithm for the stable marriage problem with one-sided ties. In: Proceedings of the IPCO 2014, Bonn. LNCS, vol 8494, pp 297–308
9. Irving RW (1994) Stable marriage and indifference. Discret Appl Math 48:261–272
10. Irving RW (1998) Matching medical students to pairs of hospitals: a new variation on a well-known theme. In: Proceedings of the ESA 1998, Venice. LNCS, vol 1461, pp 381–392
11. Irving RW, Manlove DF, Scott S (2000) The hospitals/residents problem with ties. In: Proceedings of the SWAT 2000, Bergen. LNCS, vol 1851, pp 259–271

12. Irving RW, Manlove DF, O'Malley G (2009) Stable marriage with ties and bounded length preference lists. J Discret Algorithms 7(2):213–219
13. Iwama K, Manlove DF, Miyazaki S, Morita Y (1999) Stable marriage with incomplete lists and ties. In: Proceedings of the ICALP 1999, Prague. LNCS, vol 1644, pp 443–452
14. Iwama K, Miyazaki S, Yamauchi N (2007) A 1.875-approximation algorithm for the stable marriage problem. In: Proceedings of the SODA 2007, New Orleans, pp 288–297
15. K. Iwama, Miyazaki S, Yanagisawa H (2014) A 25/17-approximation algorithm for the stable marriage problem with one-sided ties. Algorithmica 68:758–775
16. Japanese Resident Matching Program (JRMP), http://www.jrmp.jp/
17. Kavitha T, Mehlhorn K, Michail D, Paluch KE (2007) Strongly stable matchings in time $O(nm)$ and extension to the hospitals-residents problem. ACM Trans Algorithms 3(2):Article No. 15
18. Király Z (2011) Better and simpler approximation algorithms for the stable marriage problem. Algorithmica 60(1):3–20
19. Király Z (2013) Linear time local approximation algorithm for maximum stable marriage. MDPI Algorithms 6(3):471–484
20. Manlove DF (1999) Stable marriage with ties and unacceptable partners. Technical Report no. TR-1999-29 of the Computing Science Department of Glasgow University
21. Manlove DF, Irving RW, Iwama K, Miyazaki S, Morita Y (2002) Hard variants of stable marriage. Theor Comput Sci 276(1–2):261–279
22. McDermid EJ (2009) A 3/2-approximation algorithm for general stable marriage. In: Proceedings of the ICALP, Rhodes. LNCS, vol 5555, pp 689–700
23. Paluch KE (2014) Faster and simpler approximation of stable matchings. Algorithms 7(2):189–202
24. Yanagisawa H (2007) Approximation algorithms for stable marriage problems. Ph.D. Thesis, Kyoto University

Stable Partition Problem

Katarína Cechlárová
Faculty of Science, Institute of Mathematics, P. J. Šafárik University, Košice, Slovakia

Keywords

Coalition formation; Hedonic games; Stability

Years and Authors of Summarized Original Work

2002; Cechlárová, Hajduková
2004; Ballester

Problem Definition

Let N be a finite set of players; a nonempty subset of N is called a coalition. Each player $i \in N$ has a preference relation $\succeq_i$ (complete, reflexive, and transitive) over all the coalitions that contain i. Notation $S \succeq_i T$ means that player i *weakly prefers* coalition S to coalition T; if $S \succeq_i T$ and not $T \succeq_i S$, then player i *strictly prefers* S to T, denoted by $S \succ_i T$. If $S \succeq_i T$ and $T \succeq_i S$, then player i *is indifferent between* coalitions S and T (there is a *tie* in his preference list). Player i has strict preferences if her preference list contains no ties. There are several possible ways of representing preferences, but it is usually supposed that preference relations can be evaluated in polynomial time.

An instance I of the stable partition problem (or coalition formation game, or hedonic game) is given by the set of players and their preferences.

A partition Π is a collection of disjoint coalitions whose union equals N. It is supposed that each participant's appreciation of a coalition structure only depends on the coalition $\Pi(i)$ she is a member and not on the composition of other coalitions. Of interest are partitions that fulfill some kind of stability requirements.

We say that a coalition $S \subseteq N$ *strongly blocks* a partition Π, if each player $i \in S$ strictly prefers S to $\Pi(i)$, and a coalition $S \subseteq N$ *weakly blocks* a partition Π, if each player $i \in S$ weakly prefers S to $\Pi(i)$ and there exists at least one player $j \in S$ who strictly prefers S to $\Pi(j)$. Partition Π is:

- *Individually stable* if each player i weakly prefers $\Pi(i)$ to $\{i\}$;
- *Nash stable (NS)* if each player i weakly prefers $\Pi(i)$ to $X \cup \{i\}$ for each $X \in \Pi \cup \emptyset$;

- *Individually stable (IS)* if whenever a player i strictly prefers $X \cup \{i\}$ to $\Pi(i)$ for some $X \in \Pi$, then $X \succ_j X \cup \{i\}$ for at least one player $j \in X$;
- *Contractually individually stable (CIS)* if whenever a player i strictly prefers $X \cup \{i\}$ to $\Pi(i)$ for some $X \in \Pi$, then $X \succ_j X \cup \{i\}$ for at least one player $j \in X$ or $\Pi(i) \succ_j \Pi(i) \backslash \{i\}$ for at least one player $i \in \Pi(i)$;
- *Core stable* if it admits no blocking coalition;
- *Strictly core stable* if it admits no weakly blocking coalition;

Most of these definitions were introduced in [3] and [4] where also some sufficient conditions for the existence of stable partitions were formulated. An overview of the implications between these definitions can be found in [1]. The following problems have been studied algorithmically for various stability notions $\mathbb{S}$:

- $\mathbb{S}$-STABILITY-VERIFICATION: Given I and a partition Π, is Π a $\mathbb{S}$-stable partition?
- $\mathbb{S}$-STABILITY-EXISTENCE: Given I, does a $\mathbb{S}$-stable partition exist?
- $\mathbb{S}$-STABILITY-CONSTRUCTION: Given I, construct a $\mathbb{S}$-stable partition.
- $\mathbb{S}$-STABILITY-STRUCTURE: Describe the structure of $\mathbb{S}$-stable partitions for a given I.

The computational complexity of these problems depends on the specification of the preference relation in the input.

An Important special case of the stable partition problem arises when each coalition can contain at most two players. This is known under the name the *Stable Matching Problem* and is treated in detail in [14]; see also references in the entry Stable Marriage.

Key Results

Trivial Encoding

In the *trivial encoding*, each player lists all her individually rational coalitions (i.e., those that player i weakly prefers to coalition $\{i\}$).

Theorem 1 *Under the trivial encoding, the* STABILITY-VERIFICATION *problem is polynomially solvable for any stability definition.* STABILITY-EXISTENCE *is NP-complete for IR, NS, core, and strict core [2].* CORE-STABILITY-EXISTENCE *is NP-complete [2], even in the case when each player i has her preference list of the form $C_1(i) \succ_i C_2(i) \succ_i \{i\}$ and all acceptable coalitions have size three [11].*

As the trivial encoding may be of exponential size in the number of players, more succinct preference representations have been studied.

Anonymous Preferences

Players have anonymous preferences if all coalitions of the same size are tied, i.e., players do not care about the actual content of the coalitions, only about their sizes.

Theorem 2 *Under anonymous preferences, the* CORE-STABILITY-VERIFICATION *problem is polynomially solvable and* CORE-STABILITY-EXISTENCE *is NP-complete [2].*

Additive Preferences

In an additive hedonic game, each player i has a real-valued function $v_i : N \to \mathbb{R}$ and $S \succ_i T$ if and only if $\sum_{j \in S} v_i(j) > \sum_{j \in T} v_i(j)$.

Theorem 3 *In additive hedonic games,* STABILITY-VERIFICATION *is co-NP-complete in the strong sense for core and strict core [1, 17].* CORE-STABILITY-EXISTENCE *and* STRICT-CORE-STABILITY-EXISTENCE *are strongly NP-hard [18] even in the symmetric case [1].* INDIVIDUAL-STABILITY-EXISTENCE *and* NASH-STABILITY-EXISTENCE *are strongly NP-complete [1, 18]. Moreover,* CORE-STABILITY-EXISTENCE *is $\sum_2^p$-complete [19].*

Special cases of additive preferences arise if $v_i(j) \in \{-1, |N|\}$ for each $i, j \in N$ (friend-oriented case) or $v_i(j) \in \{1, -|N|\}$ for each $i, j \in N$ (enemy-oriented case). Under friend-oriented as well as under enemy-oriented preferences, a core-stable partition always exists [12], however, the following assertion holds.

Theorem 4 ([12]) *Under enemy-oriented preferences,* CORE-STABILITY-VERIFICATION *and* CORE-STABILITY-CONSTRUCTION *are strongly NP-complete and NP-hard, respectively.*

Preferences Derived from the Best and/or Worst Player

Suppose that each player i linearly orders only individual players or, more precisely, a subset of them – these are *acceptable* for i.

Preferences over players are extended to preferences over coalitions on the basis of the best or the worst player in the coalition as follows:

B-preferences – a player orders coalitions first on the basis of the most preferred member of the coalition, and if those are equal or tied, the coalition with smaller cardinality is preferred;

W-preferences – a player orders coalitions on the basis of the least preferred member of the coalition;

BW-preferences – a player orders coalitions first on the basis of the best member of the coalition, and if those are equal or tied, the coalition with a more preferred worst member is preferred.

In this case, preferences are considered *strict*, if the preferences over individuals are strict, and they are called *dichotomous* if all acceptable participants are tied in each preference list.

Theorem 5 *Under B-preferences,* STABILITY-VERIFICATION *is polynomial for core and strict core. A strict core and a core stable partition always exist if preferences over players are strict [9]. However, if preferences over players contain ties,* STABILITY-EXISTENCE *for core and strict core is NP-complete [6]. In the dichotomous case, a core stable partition can be constructed in polynomial time, but* STRICT-CORE-STABILITY-EXISTENCE *is NP-complete [5].*

Let us remark here that in the case of strict preferences, a strict core stable partition can be found by the famous Top Trading Cycles algorithm [9, 20].

The stable partition problem under W-preferences was studied in [7] and many features similar to the Stable Roommates Problem [14] were described. First, if a blocking coalition exists, then there is a blocking coalition of size at most 2. Hence, CORE-STABILITY-VERIFICATION is polynomial. CORE-STABILITY-EXISTENCE and CORE-STABILITY-CONSTRUCTION are polynomial in the strict preferences case, which can be shown using an extension of Irving's Stable Roommates Algorithm (discussed in detail in [14]). This algorithm can also be used to derive some results for CORE-STABILITY-STRUCTURE. In the case of ties, CORE-STABILITY-EXISTENCE is NP-complete.

Under BW preferences, in the strict preferences case, a core partition always exists and one can be obtained by the Top Trading Cycles algorithm, but STRICT-CORE-STABILITY-EXISTENCE is NP-hard. If preferences contain ties, CORE-STABILITY-EXISTENCE is NP-hard too [8]. CORE-STABILITY-VERIFICATION remains open.

Applications

Stable partitions arise in various economic and game theoretical models. They appear in the study of countries formation [10] and in multi-agent coordination scenarios and social networking services [13]. Stability is also desired in barter exchange economies with discrete commodities [20, 21], including exchange of kidneys for transplantations [5, 16]. Notice that in case when the cooperation of players consists in the exchange of some items within one partition set, the exchange cycle has also to be specified.

Open Problems

Due to the great number of variants, a lot of open problems exists. In almost all cases, STABILITY-STRUCTURE is not satisfactorily solved. For instances with no stable partition, one may seek one that minimizes the number of players who have an incentive to deviate. Parallel algorithms were also not studied.

Experimental Results

Stochastic local search algorithms for CORE-STABILITY-VERIFICATION in the additive preferences case were reported in [15].

Cross-References

▸ Stable Marriage

Recommended Reading

1. Aziz H, Brandt F, Seeding HG (2013) Computing desirable partitions in additively separable hedonic games. Artif Intell 195:316–334
2. Ballester C (2004) NP-completeness in hedonic games. Games Econ Behav 49(1):1–30
3. Banerjee S, Konishi H, Sönmez T (2001) Core in a simple coalition formation game. Soc Choice Welf 18:135–153
4. Bogomolnaia A, Jackson MO (2002) The stability of hedonic coalition structures. Games Econ Behav 38(2):201–230
5. Cechlárová K, Fleiner T, Manlove D (2005) The kidney exchange game. In: Zadnik-Stirn L, Drobne S (eds) Proceedings of SOR'05, Slovenia, pp 77–83
6. Cechlárová K, Hajduková J (2002) Computational complexity of stable partitions with B-preferences. Int J Game Theory 31(3):353–364
7. Cechlárová K, Hajduková J (2004) Stable partitions with W-preferences. Discret Appl Math 138(3):333–347
8. Cechlárová K, Hajduková J (2004) Stability of partitions under WB-preferences and BW-preferences. Int J Inf Technol Decis Making 3(4):605–614. Special Issue on Computational Finance and Economics
9. Cechlárová K, Romero-Medina A (2001) Stability in coalition formation games. Int J Game Theory 29:487–494
10. Cechlárová K, Dahm M, Lacko V (2001) Efficiency and stability in a discrete model of country formation. J Glob Optim 20(3–4):239–256
11. Deineko VG, Woeginger GJ (2013) Two hardness results for core stability in additive hedonic coalition formation games. Discret Appl Math 161:1837–1842
12. Dimitrov D, Borm P, Hendrickx R, Sung SCh (2006) Simple priorities and core stability in hedonic games. Soc Choice Welf 26(2):421–433
13. Elkind E, Wooldridge M (2009) Hedonic coalition nets. In: Proceedings of the 8th international conference on autonomous agents and multiagent systems (AAMAS 2009), Budapest, pp 417–424
14. Gusfield D, Irving RW (1989) The stable marriage problem. Structure and algorithms. MIT, Cambridge
15. Keinänen H (2010) Stochastic local search for core membership checking in hedonic games. LNCS 6220:56–70
16. Roth A, Sönmez T, Ünver U (2004) Kidney exchange. Q J Econ 119:457–488
17. Sung SCh, Dimitrov D (2007) On core membership testing for hedonic coalition formation games. Oper Res Lett 35:155–158
18. Sung SCh, Dimitrov D (2010) Computational complexity in additive hedonic games. Eur J Oper Res 203:635–639
19. Woeginger GJ (2013) A hardness result for core stability in additive hedonic games. Math Soc Sci 65:101-104
20. Shapley L, Scarf H (1974) On cores and indivisibility. J Math Econ 1:23–37
21. Yuan Y (1996) Residence exchange wanted: a stable residence exchange problem. Eur J Oper Res 90:536–546

Stackelberg Games: The Price of Optimum

Alexis Kaporis[1] and Paul (Pavlos) Spirakis[2,3,4]
[1]Department of Information and Communication Systems Engineering, University of the Aegean, Karlovasi, Samos, Greece
[2]Computer Engineering and Informatics, Research and Academic Computer Technology Institute, Patras University, Patras, Greece
[3]Computer Science, University of Liverpool, Liverpool, UK
[4]Computer Technology Institute (CTI), Patras, Greece

Keywords

Coordination ratio; Cournot game

Years and Authors of Summarized Original Work

2006; Kaporis, Spirakis

Problem Definition

Stackelberg games [15] may model the interplay among an authority and rational individuals that selfishly demand resources on a large-scale

network. In such a game, the authority *(Leader)* of the network is modeled by a distinguished player. The selfish users *(Followers)* are modeled by the remaining players.

It is well known that selfish behavior may yield a *Nash Equilibrium* with cost arbitrarily higher than the optimum one, yielding unbounded *Coordination Ratio* or *Price of Anarchy (PoA)* [7, 13]. Leader plays his strategy first assigning a portion of the total demand to some resources of the network. Followers observe and react selfishly assigning their demand to the most appealing resources. Leader aims to drive the system to an a posteriori Nash equilibrium with cost close to the overall optimum one [4, 6, 8, 10]. Leader may also be eager for his own rather than system's performance [2, 3].

A Stackelberg game can be seen as a special, and easy [6] to implement, case of *Mechanism Design*. It avoids the complexities of either computing taxes or assigning prices, or even designing the network at hand [9]. However, a central authority capable to control the overall demand on the resources of a network may be unrealistic in networks which evolute and operate under the effect of many and diversing economic entities. A realistic way [4] to act centrally even in large nets could be via *Virtual Private Networks (VPNs)* [1]. Another flexible way is to combine such strategies with *Tolls* [5, 14].

A dictator controlling the entire demand optimally on the resources surely yields PoA = 1. On the other hand, rational users do prefer a liberal world to live. Thus, it is important to compute the optimal Leader strategy which controls the *minimum* of the resources *(Price of Optimum)* and yields PoA $=$ 1. What is the complexity of computing the Price of Optimum? This is not trivial to answer, since the Price of Optimum depends crucially on computing an optimal Leader strategy. In particular, [6] proved that computing the optimal Leader strategy is hard.

The central result of this lemma is Theorem 5. It says that on nonatomic flows and arbitrary $s-t$ networks and latencies, computing the minimum portion of flow and Leader's optimal strategy sufficient to induce PoA $=$ 1 is easy [10].

Problem $(G(V,E),s,t \in V,r)$ INPUT: Graph $G, \forall e \in E$ latency ℓ_e, flow r, a source-destination pair (s,t) of vertices in V.
OUTPUT: (i) The minimum portion α_G of the total flow r sufficient for an optimal Stackelberg strategy to induce the optimum on G. (ii) The optimal Stackelberg strategy.

Models and Notations

Consider a graph $G(V,E)$ with parallel edges allowed. A number of rational and selfish users wish to route from a given source s to a destination node t an amount of flow r. Alternatively, consider a partition of users in k commodities, where user(s) in commodity i wish to route flow r_i through a source-destination pair (s_i,t_i), for each $i = 1,\ldots,k$. Each edge $e \in E$ is associated to a latency function $\ell_e()$, positive, differentiable, and strictly increasing on the flow traversing it.

Nonatomic Flows

There are infinitely many users, each routing his/her infinitesimally small amount of the total flow r_i from a given source s_i to a destination vertex t_i in graph $G(V,E)$. A flow f is an assignment of jobs f_e on each edge $e \in E$. The cost of the injected flow f_e (satisfying the standard constraints of the corresponding network-flow problem) that traverses edge $e \in E$ equals; $c_e(f_e) = f_e \times \ell_e(f_e)$. It is assumed that on each edge e the cost is convex with respect to the injected flow f_e. The overall system's cost is the sum $\sum_{e\in E} f_e \times \ell_e(f_e)$ of all edge costs in G. Let $f_{\mathcal{P}}$ the amount of flow traversing the $s_i - t_i$ path $\mathcal{P}$. The latency $\ell_{\mathcal{P}}(f)$ of $s_i - t_i$ path $\mathcal{P}$ is the sum $\sum_{e\in\mathcal{P}} \ell_e(f_e)$ of latencies per edge $e \in \mathcal{P}$. The cost $C_{\mathcal{P}}(f)$ of $s_i - t_i$ path $\mathcal{P}$ equals the flow $f_{\mathcal{P}}$ traversing it multiplied by path latency $\ell_{\mathcal{P}}(f)$. That is, $C_{\mathcal{P}}(f) = f_{\mathcal{P}} \times \sum_{e\in\mathcal{P}} \ell_e(f_e)$. In a Nash equilibrium, all $s_i - t_i$ paths traversed by nonatomic users in part i have a common latency, which is at most the latency of any untraversed

$s_i - t_i$ path. More formally, for any part i and any pair $\mathcal{P}_1$, $\mathcal{P}_2$ of $s_i - t_i$ paths, if $f_{\mathcal{P}_1} > 0$ then $\ell_{\mathcal{P}_1}(f) \leq \ell_{\mathcal{P}_2}(f)$. By the convexity of edge costs, the Nash equilibrium is unique and computable in polynomial time given a floating-point precision. Also computable is the unique *Optimum* assignment O of flow, assigning flow o_e on each $e \in E$ and minimizing the overall cost $\sum_{e \in E} o_e \ell_e(o_e)$. However, not all optimally traversed $s_i - t_i$ paths experience the same latency. In particular, users traversing paths with high latency have incentive to reroute toward more speedy paths. Therefore, the optimal assignment is unstable on selfish behavior.

A Leader dictates a *weak* Stackelberg strategy if on each commodity $i = 1, \ldots, k$ controls a fixed α portion of flow $r_i, \alpha \in [0, 1]$. A *strong* Stackelberg strategy is more flexible, since Leader may control $\alpha_i r_i$ flow in commodity i such that $\sum_{i=1}^{k} \alpha_i = \alpha$. Let a Leader dictating flow s_e on edge $e \in E$. The a posteriori latency $\tilde{\ell}_e(n_e)$ of edge e, with respect to the induced flow n_e by the selfish users, equals $\tilde{\ell}_e(n_e) = \ell_e(n_e + s_e)$. In the a posteriori Nash equilibrium, all $s_i - t_i$ paths traversed by the free selfish users in commodity i have a common latency, which is at most the latency of any selfishly untraversed path, and its cost is $\sum_{e \in E} (n_e + s_e) \times \tilde{\ell}_e(n_e)$.

Atomic Splittable Flows

There is a finite number of atomic users $1, \ldots, k$. Each user i is responsible for routing a non-negligible flow-amount r_i from a given source s_i to a destination vertex t_i in graph G. In turn, each flow-amount r_i consists of infinitesimally small jobs.

Let flow f assigning jobs f_e on each edge $e \in E$. Each edge flow f_e is the sum of partial flows $f_e^1, \ldots, f_e^k$ injected by the corresponding users $1, \ldots, k$. That is, $f_e = f_e^1 + \cdots + f_e^k$. As in the model above, the latency on a given $s_i - t_i$ path $\mathcal{P}$ is the sum $\sum_{e \in \mathcal{P}} \ell_e(f_e)$ of latencies per edge $e \in \mathcal{P}$. Let $f_{\mathcal{P}}^i$ be the flow that user i ships through an $s_i - t_i$ path $\mathcal{P}$. The cost of user i on a given $s_i - t_i$ path $\mathcal{P}$ is analogous to her path flow $f_{\mathcal{P}}^i$ routed via $\mathcal{P}$ times the total path latency $\sum_{e \in \mathcal{P}} \ell_e(f_e)$. That is, the path cost equals $f_{\mathcal{P}}^i \times \sum_{e \in \mathcal{P}} \ell_e(f_e)$. The overall cost $C_i(f)$ of user i is the sum of the corresponding path costs of all $s_i - t_i$ paths.

In a Nash equilibrium no user i can improve his cost $C_i(f)$ by rerouting, given that any user $j \neq i$ keeps his routing fixed. Since each atomic user minimizes its cost, if the game consists of only one user, then the cost of the Nash equilibrium coincides to the optimal one.

In a Stackelberg game, a distinguished atomic Leader player controls flow r_0 and plays first assigning flow s_e on edge $e \in E$. The a posteriori latency $\tilde{\ell}_e(x)$ of edge e on induced flow x equals $\tilde{\ell}_e(x) = \ell_e(x + s_e)$. Intuitively, after Leader's move, the induced selfish play of the k atomic users is equivalent to atomic splittable flows on a graph where each initial edge latency ℓ_e has been mapped to $\tilde{\ell}_e$. In game parlance, each atomic user $i \in \{1, \ldots, k\}$, having *fixed* Leader's strategy, computes his *best reply* against all other atomic users $\{1, \ldots, k\} \backslash \{i\}$. If n_e is the induced Nash flow on edge e, this yields total cost $\sum_{e \in E} (n_e + s_e) \times \tilde{\ell}_e(n_e)$.

Atomic Unsplittable Flows

The users are finite $1, \ldots, k$ and user i is allowed to send his non-negligible job r_i only on a *single* path. Despite this restriction, all definitions given in atomic splittable model remain the same.

Key Results

Let us see first the case of atomic splittable flows, on parallel M/M/1 links with different speeds connecting a given source-destination pair of vertices.

Theorem 1 (Korilis, Lazar, Orda [6]) *The Leader can enforce in polynomial time the network optimum if his/her controls flow r_0 exceeding a critical value $\underline{r}^0$.*

In the sequel, we focus on nonatomic flows on $s-t$ graphs with parallel links. In [6] primarily were studied cases that Leader's flow cannot induce network's optimum and was shown that an optimal Stackelberg strategy is easy to compute. In this vain, if $s-t$ parallel link instances are restricted to ones with linear latencies of equal slope, then an optimal strategy is easy [4].

Theorem 2 (Kaporis, Spirakis [4]) *The optimal Leader strategy can be computed in polynomial time on any instance (G, r, α) where G is an $s-t$ graph with parallel links and linear latencies of equal slope.*

Another positive result is that the optimal strategy can be approximated within $(1+\epsilon)$ in polynomial time, given that link latencies are polynomials with nonnegative coefficients.

Theorem 3 (Kumar, Marathe [8]) *There is a fully polynomial approximate Stackelberg scheme that runs in poly $\left(m, \frac{1}{\epsilon}\right)$ time and outputs a strategy with cost $(1+\epsilon)$ within the optimum strategy.*

For parallel link $s-t$ graphs with arbitrary latencies more can be achieved: in polynomial time a "threshold" value α_G is computed, sufficient for the Leader's portion to induce the optimum. The complexity of computing optimal strategies changes in a dramatic way around the critical value α_G from "hard" to "easy" (G, r, α) Stackelberg scheduling instances. Call α_G as the *Price of Optimum* for graph G.

Theorem 4 (Kaporis, Spirakis [4]) *On an input $s-t$ parallel link graph G with arbitrary strictly increasing latencies, the minimum portion α_G sufficient for a Leader to induce the optimum, as well as his/her optimal strategy, can be computed in polynomial time.*

As a conclusion, the Price of Optimum α_G essentially captures the hardness of instances (G, r, α). Since, for Stackelberg scheduling instances $(G, r, \alpha \geq \alpha_G)$, the optimal Leader strategy yields PoA $= 1$ and it is computed as hard as in P, while for $(G, r, \alpha < \alpha_G)$ the optimal strategy yields PoA < 1 and it is as easy as *NP* [10].

The results above are limited to parallel links connecting a given $s-t$ pair of vertices. Is it possible to efficiently compute the Price of Optimum for nonatomic flows on arbitrary graphs? This is not trivial to settle. Not only because it relies on computing an optimal Stackelberg strategy, which is hard to tackle [10], but also because Proposition B.3.1 in [11] ruled out previously known performance guarantees for Stackelberg strategies on general nets.

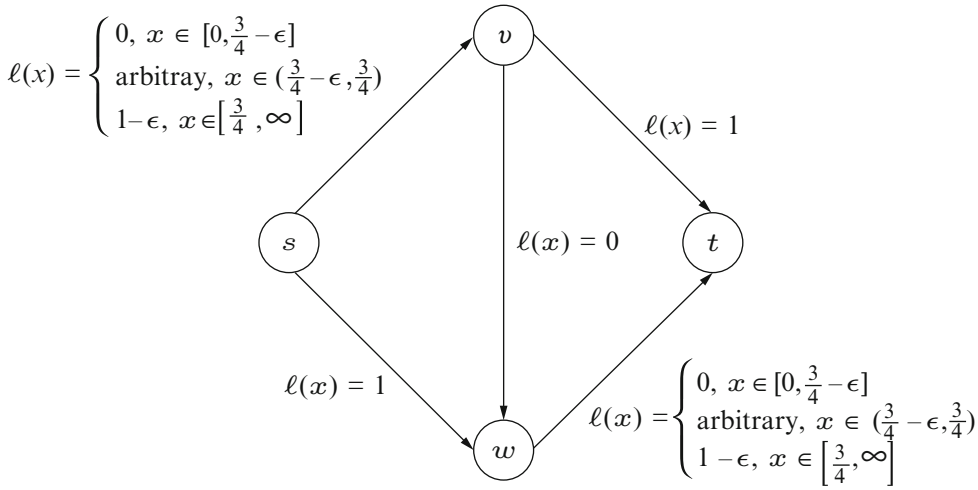
The central result of this lemma is presented below and completely resolves this question (extending Theorem 4).

Theorem 5 (Kaporis, Spirakis [4]) *On arbitrary $s-t$ graphs G with arbitrary latencies, the minimum portion α_G sufficient for a Leader to induce the optimum, as well as her optimal strategy, can be computed in polynomial time.*

Example

Consider the optimum assignment O of flow r that wishes to travel from source vertex s to sink t. O assigns flow o_e incurring latency $\ell_e(o_e)$ per edge $e \in G$. Let $\mathcal{P}_{s\to t}$ the set of all $s-t$ paths. The *shortest paths* in $\mathcal{P}_{s\to t}$ with respect to costs $\ell_e(o_e)$ per edge $e \in G$ can be computed in polynomial time. That is, the paths that given flow assignment O achieve path latency: $\min_{P\in\mathcal{P}_{s\to t}} \left(\sum_{e\in P} \ell_e(o_e)\right)$, i.e., minimize their path latency. It is crucial to observe that if we want the *induced* Nash assignment by the Stackelberg strategy to attain the optimum cost, then these shortest paths are *the only choice* for selfish users that are eager to travel from s to t. Furthermore, the uniqueness of the optimum assignment O determines the minimum part of flow which can be selfishly scheduled on these shortest paths. Observe that any flow assigned by O on a non-shortest $s-t$ path has incentive to opt for a shortest one. Then a Stackelberg strategy *must* freeze the flow on all non-shortest $s-t$ paths.

In particular, the idea sketched above achieves coordination ratio 1 on the graph in Fig. 1. On this graph Roughgarden proved that $\frac{1}{\alpha}\times$ (optimum



Stackelberg Games: The Price of Optimum, Fig. 1 A bad example for Stackelberg routing

cost) guarantee is *not* possible for general (s,t)-networks, Appendix B.3 in [11]. The optimal edge flows are ($r = 1$):

$$o_{s\to v} = \tfrac{3}{4} - \epsilon, o_{s\to w} = \tfrac{1}{4} + \epsilon, o_{v\to w} = \tfrac{1}{2} - 2\epsilon,$$
$$o_{v\to t} = \tfrac{1}{4} + \epsilon, o_{w\to t} = \tfrac{3}{4} - \epsilon$$

The shortest path $P_0 \in \mathcal{P}$ with respect to the optimum O is $P_0 = s \to v \to w \to t$ (see [11] pp. 143, 5th-3th lines before the end) and its flow is $f_{P_0} = \frac{1}{2} - 2\epsilon$. The non-shortest paths are $P_1 = s \to v \to t$ and $P_2 = s \to w \to t$ with corresponding optimal flows: $f_{P_1} = \frac{1}{4} + \epsilon$ and $f_{P_2} = \frac{1}{4} + \epsilon$. Thus, the Price of Optimum is

$$f_{P_1} + f_{P_2} = \frac{1}{2} + 2\epsilon = r - f_{P_0}$$

Applications

Stackelberg strategies are widely applicable in networking [6], see also Section 6.7 in [12].

Open Problems

It is important to extend the above results on atomic unsplittable flows.

Cross-References

- Algorithmic Mechanism Design
- Best Response Algorithms for Selfish Routing
- Facility Location
- Non-approximability of Bimatrix Nash Equilibria
- Price of Anarchy
- Selfish Unsplittable Flows: Algorithms for Pure Equilibria

Recommended Reading

1. Birman K (1997) Building secure and reliable network applications. Manning, Greenwich
2. Douligeris C, Mazumdar R (2006) Multilevel flow control of queues. In: Johns Hopkins conference on information sciences, Baltimore, 22–24 Mar 1989
3. Economides A, Silvester, J (1990) Priority load sharing: an approach using Stackelberg games. In: 28th annual Allerton conference on communications, control and computing, Monticello
4. Kaporis AC, Spirakis PG (2009) The price of optimum in Stackelberg games on arbitrary single commodity networks and latency functions. Theor Comput Sci **410**(8–10):745–755
5. Karakostas G, Kolliopoulos SG (2009) Stackelberg strategies for selfish routing in general multicommodity networks. Algorithmica **53**(1):132–153

6. Korilis YA, Lazar AA, Orda A (1997) Achieving network optima using stackelberg routing strategies. IEEE/ACM Trans Netw 5(1):161–173
7. Koutsoupias E, Papadimitriou CH (2009) Worst-case equilibria. Comput Sci Rev **3**(2):65–69
8. Kumar VSA, Marathe MV (2002) Improved results for Stackelberg scheduling strategies. In: 29th international colloquium, automata, languages and programming, Málaga. LNCS. Springer, pp 776–787
9. Roughgarden T (2001) Designing networks for selfish users is hard. In: 42nd IEEE annual symposium of foundations of computer science, Las Vegas, pp 472–481
10. Roughgarden T (2004) Stackelberg scheduling strategies. SIAM J Comput **33**(2):332–350
11. Roughgarden T 2002 Selfish routing. Dissertation, Cornell University. http://theory.stanford.edu/~tim/
12. Roughgarden T (2005) Selfish routing and the price of anarchy. MIT, Cambridge (2005)
13. Roughgarden T, Tardos É (2002) How bad is selfish routing? J ACM **49**(2):236–259
14. Swamy C (2007) The effectiveness of Stackelberg strategies and tolls for network congestion games. In: ACM-SIAM symposium on discrete algorithms, Philadelphia
15. von Stackelberg H (1934) Marktform und Gleichgewicht. Springer, Vienna

Staged Assembly

Andrew Winslow
Department of Computer Science, Tufts University, Medford, MA, USA

Keywords

Context-free grammars; DNA computing; Jigsaw; Natural computing; Polyomino; Shape decomposition

Years and Authors of Summarized Original Work

2008; Demaine, Demaine, Fekete, Ishaque, Rafalin, Schweller, Souvaine
2013; Demaine, Eisenstat, Ishaque, Winslow
2013; Winslow

Problem Definition

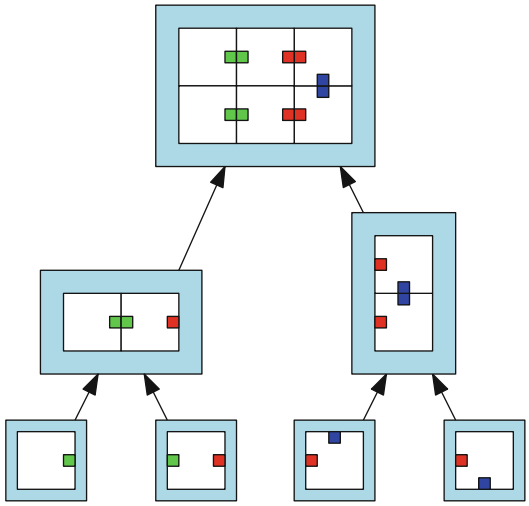
Algorithmic self-assembly is concerned with hands-off assembly of complex structures by mixing collections of simple particles that aggregate according to local rules. Staged self-assembly utilizes sequences of mixings to reduce the number of particle types used. The standard model of staged self-assembly builds on the abstract Tile Assembly Model (aTAM) of Winfree [7], where each particle is a non-rotatable unit square *tile* with a labeled *glue* on each side. Tiles attach to other tiles edgewise via glues of the same label, forming polyomino-shaped aggregates called *assemblies*.

In the simplest model, a pair of assemblies (of which single tiles are a special case) can attach via a single matching glue. In a more general model, a pair of assemblies can attach if they share a total of $\tau \in \mathbb{N}$ glues. The parameter τ is called the *temperature* of the system.

The self-assembly process is carried out by combining an infinite number of copies of a collection of *reagent assemblies* in a *bin*, where they attach in every possible way. The subset of the resulting assemblies that cannot attach to any other assemblies define the *product assemblies* of the mixing, i.e., the set of assemblies that remain once the assembly process is complete. A system consisting of a single bin with single-tile reagent assemblies is a *hierarchical* [2], *two-handed* [1], or *polyomino* [5] self-assembly system.

In a *staged self-assembly system* [3], the products of one bin can be used as the reagents of other bin (see Fig. 1). The directed acyclic graph describing the relationship mixings is called the *mix graph* of the system. An initial set of mixings each have a single tile as the only product assembly and no reagent assemblies.

Objectives In general, the goal is to design a system with a mixing containing a single product assembly of a desired polyomino shape while minimizing the size of some aspect of the system. Several aspects are considered, including the number of distinct tiles (*tile complexity*), number of edges of the mix graph (*mix graph complexity*),



Staged Assembly, Fig. 1 A staged self-assembly system. Each bin (*blue box*) contains the product assemblies of the bin. The reagent assemblies of a bin are the products of other bins (*incoming arrows*)

width of the mix graph (*bin complexity*), height of the mix graph (*stage complexity*), and temperature of the system. The computational complexity of finding an optimal system for an input shape under some measure of system complexity is also considered. In some cases, the desired polyomino shape also has each cell labeled, and the goal is to construct a given labeled shape using labeled tiles.

Problem 1 (Smallest Staged Self-Assembly System)

INPUT*: A labeled polyomino P.*
OUTPUT*: A staged self-assembly system containing a bin with a single product assembly with labeled shape P that is minimum in some measure.*

Key Results

We describe the results by increasing generality of the shapes assembled.

Lines

In the most constrained case, the input polyomino is an unlabeled $1 \times n$ polyomino (a *line*). Lines can be assembled by $\tau = 1$ systems using $O(1)$ tile types, $O(1)$ bins, and $O(\log n)$ stages and mix graph edges. The idea is to repeatedly double the length of a line assembly as proved by Demaine et al. [3].

Demaine, Eisenstat, Ishaque, and Winslow [4] prove that the case of labeled lines is roughly equivalent to the problem of finding the smallest context-free grammar that is a single string consisting of the labels of the line read from left to right. In particular, any context-free grammar $\mathcal{G}$ with $|\mathcal{G}|$ rules and deriving a single string σ can be converted into a $\tau = 1$ staged self-assembly system $\mathcal{S}$ assembling a line with left-to-right label string σ, where the number of edges in the mix graph of $\mathcal{S}$ is $O(|\mathcal{G}|)$. The complexity of the smallest staged self-assembly system where the input polyomino is a labeled line and the system has an upper limit on the number of glue types appearing on the tiles was proven to be NP-hard [4].

Squares

Demaine et al. [3] prove that unlabeled $n \times n$ squares are possible with a $\tau = 1$ staged systems containing $O(1)$ tile types and bins, $O(\log n)$ stages, and thus a mix graph with $O(\log n)$ stages. The system uses an idea similar to that for assembling lines but in two steps: first assemble $n \times 1$ columns and then combine them to form $n \times 2$, then $n \times 4$, etc., rectangles. This construction uses a *jigsaw* technique to ensure attaching rectangles cannot assemble askew.

Demaine et al. also prove that unlabeled $n \times n$ squares can be assembled using $\tau = 2$ staged systems using $O(1)$ tile types, $O(\sqrt{\log n})$ bins, and $O(\log \log n)$ stages. The approach is to *simulate* known $\tau = 2$ single-bin systems that efficiently assemble squares by constructing *macrotiles*: large assemblies that simulate the behavior of distinct tile types by encoding glue types in geometry on their surfaces. Such macrotiles allow staged systems to trade off tile types for stages by replacing many distinct tile types with an initial sequence of stages that assemble macrotile versions of the tiles.

General Shapes

For general shapes, several different results highlight the trade-offs in complexity enabled by staged assembly. Demaine et al. [3] prove that any unlabeled shape can be assembled by a $\tau = 1$ staged system using $O(1)$ tile types, $O(\log n)$ bins, and a number of stages proportional to the diameter of the dual grid graph of the shape. If the shape is monotone, then a similar system with $O(n)$ bins and $O(\log n)$ stages (increased bins but decreased stages) exists.

If the system is permitted to assemble a scaled version of the input shape, then the system of Soloveichik and Winfree [6] can be simulated with macrotiles, resulting in a $\tau = 2$ staged system with $O(1)$ tile types, $O(K/\log K)$ bins, and $O(\log \log K)$ stages, where K is the Kolmogorov complexity of the shape. For labeled shapes, Winslow [8] proves that any polyomino context-free grammar $\mathcal{G}$ (a generalization of context-free grammars to two dimensions) with $|\mathcal{G}|$ rules deriving a single labeled polyomino P can be converted into a staged system $\mathcal{S}$ assembling a scaled version of P consisting of labeled macrotiles where the number of edges in the mix graph of $\mathcal{S}$ is $O(|\mathcal{G}|)$.

Applications

The theory of algorithmic self-assembly is rooted in the design of nanoscale particle systems, particularly DNA-based systems. For staged self-assembly in particular, the capability of assembling complex shapes using only $O(1)$ tile types is highly desirable in practice, as engineering many tile types with desired glues is often far more challenging than carrying out a sequence of mixings.

Open Problems

The complexity of the smallest staged self-assembly problem where the number of glue types used is unconstrained remains open, both for the case of lines and general shapes. For lines, the problem is known to be in NP and when the number of glues is constrained is NP-complete (both proved in [4]). For general shapes, the problem is only known to be in PSPACE (proved in [9]) and NP-hard when the number of glues is constrained, following from the special case of lines. The complexity of verifying that a staged assembly system produces a given shape also remains open and is only known to lie in PSPACE.

Cross-References

- Combinatorial Optimization and Verification in Self-Assembly
- Experimental Implementation of Tile Assembly
- Patterned Self-Assembly Tile Set Synthesis
- Self-Assembly at Temperature 1
- Self-Assembly of Squares and Scaled Shapes

Recommended Reading

1. Cannon S, Demaine ED, Demaine ML, Eisenstat S, Patitz MJ, Schweller RT, Summers SM, Winslow A (2013) Two hands are better than one (up to constant factors): self-assembly in the 2HAM vs. aTAM. In: STACS 2013, Kiel, LIPIcs, vol 20. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp 172–184
2. Chen H, Doty D (2012) Parallelism and time in hierarchical self-assembly. In: Proceedings of the 23rd annual ACM-SIAM symposium on discrete algorithms (SODA), Kyoto, pp 1163–1182
3. Demaine ED, Demaine ML, Fekete SP, Ishaque M, Rafalin E, Schweller RT, Souvaine DL (2008) Staged self-assembly: nanomanufacture of arbitrary shapes with $O(1)$ glues. Nat Comput 7(3):347–370
4. Demaine ED, Eisenstat S, Ishaque M, Winslow A (2013) One-dimensional staged self-assembly. Nat Comput 12(2):247–258
5. Luhrs C (2010) Polyomino-safe DNA self-assembly via block replacement. Nat Comput 9(1):97–109
6. Soloveichik D, Winfree E (2007) Complexity of self-assembled shapes. SIAM J Comput 36(6):1544–1569
7. Winfree E (1998) Algorithmic self-assembly of DNA. PhD thesis, Caltech
8. Winslow A (2013a) Staged self-assembly and polyomino context-free grammars. In: Soloveichik D, Yurke B (eds) DNA 19, Tempe
9. Winslow A (2013b) Staged self-assembly and polyomino context-free grammars. PhD thesis, Tufts University

Statistical Multiple Alignment

István Miklós
Department of Plant Taxonomy and Ecology, Eötvös Loránd University, Budapest, Hungary

Keywords

Multiple HMM; Statistical alignment; Stochastic modeling of insertions and deletions; Time-continuous Markov models

Years and Authors of Summarized Original Work

2003; Hein, Jensen, Pedersen

Problem Definition

The three main types of mutations modifying biological sequences are insertions, deletions, and substitutions. The simplest model involving these three types of mutations is the so-called Thorne-Kishino-Felsenstein model [16]. In this model, the characters of a sequence evolve independently. Each character in the sequence can be substituted with another character according to a prescribed reversible time-continuous Markov model on the possible characters. Insertion-deletions are modeled as a birth-death process. Insertions can happen at the beginning of the sequence, at the end of the sequence, and between any two characters. It is possible to insert a character into the empty sequence. The time span between two insertions is exponentially distributed with parameter λ, and this parameter does not depend on the context of the position. The newborn character is drawn from the equilibrium distribution of the substitution process. Each character is deleted after an exponentially distributed waiting time with parameter μ, and its two positions where insertions can happen are joined.

The multiple statistical alignment problem is to calculate the likelihood of a set of sequences, namely, what is the probability of observing a set of sequences, given all the necessary parameters that describe the evolution of sequences. Hein, Jensen, and Pedersen were the first who gave an algorithm to calculate this probability [5]. Their algorithm has $O(5^n L^n)$ running time, where n is the number of sequences, and L is the geometric mean of the sequences. The running time has been improved to $O(2^n L^n)$ by Lunter et al. [9].

Notations

Substitutions

A time-continuous Markov model for a substitution process on an alphabet Σ is given by a $k \times k$ rate matrix Q, with constraints

$$q_{i,j} \geq 0 \qquad \forall i \neq æ \tag{1}$$

$$\sum_i q_{i,j} = 0 \qquad \forall i \tag{2}$$

where k is the size of the alphabet. The probability that a character a_i will be character a_j after time t can be calculated with the exponentiation of the rate matrix:

$$P_t(a_j|a_i) = p_{i,j} \qquad \text{where} \tag{3}$$

$$P = e^{Qt} \tag{4}$$

The exponentiated matrix can be easily calculated if the rate matrix is diagonalized, namely, if $Q = W\Lambda W^{-1}$, where Λ is a diagonal matrix, then

$$e^{Qt} = We^{\Lambda t}W^{-1} \tag{5}$$

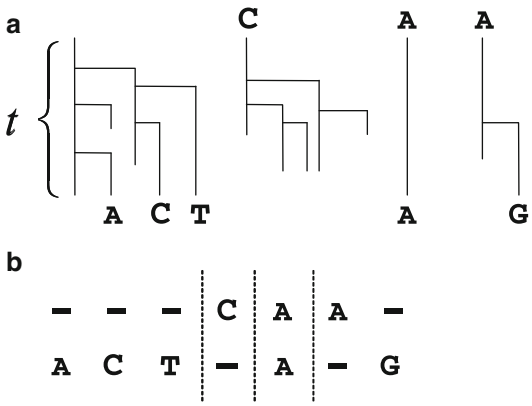
$e^{\Lambda t}$ can be easily calculated, since it is a diagonal matrix containing $e^{\lambda_i t}$ in the ith position of the diagonal.

Insertions and Deletions

A Galton-Watson tree is a rooted, edge-weighted binary tree that describes a birth-death process for a time span t. The process starts at the root of the tree, and a split represents a birth. Edge weights represent times, and leaves having a distance from the root smaller than t represent death events. Leaves being t time fare from the root are the individuals that live at time point t.

Insertion-deletion events transforming one sequence into another can be described with Galton-Watson forests: births represent insertions, and deaths represent deletions. Each character of the ancestral sequence has a tree, and there is an additional tree at the beginning of the sequence associated to an imaginary character. This imaginary character cannot die. Roots of the trees are the characters of the ancestral sequence, and each character of the descendant sequence is a leaf of one of the trees, being t time fare from the root. There might be additional leaves that are not associated with characters of the descendant sequences; these are the died out lineages. The forest is aligned such that edges do not cross each other while the characters of the two sequences keep their original order. Each Galton-Watson forest indicates an alignment of the two sequences; see Fig. 1. Given a birth and death process, the probability density of a Galton-Watson tree can be calculated easily. Assuming independence, the probability of a Galton-Watson forest is the product of the probabilities of its trees. The probability of an alignment is the integral of the probabilities of the forests that represent it. Due to independence, it is enough to tell the probability of alignment patterns that might arise as an image of a Galton-Watson tree (see Fig. 1b); the probability of an alignment is the product of the probabilities of its patterns.

In the Thorne-Kishino-Felsenstein model (TKF91 model) [16], both the birth and the death processes are Poisson processes with parameters λ and μ, respectively. The probability of the possible patterns can be found on Fig. 2.



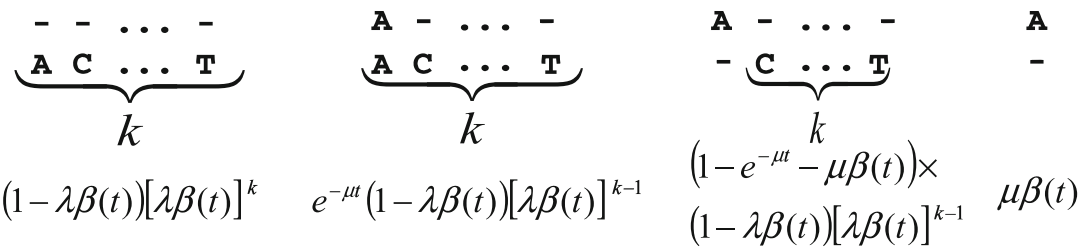
Statistical Multiple Alignment, Fig. 1 (**a**) A Galton-Watson forest representing insertion-deletion events. The first tree starts with an immortal element that is responsible to the insertions at the beginning of the sequence. (**b**) The alignment indicated by the Galton-Watson forest above. Each tree makes a pattern of the alignment; patterns are separated with *dashed lines*

Evolutionary Trees

An evolutionary tree is a leaf-labeled, edge-weighted, rooted binary tree. Labels are the species related by the evolutionary tree, and weights are evolutionary distances. It might happen that the evolutionary changes had different speed at different lineages, and hence the tree is not necessary ultrametric, namely, the root not necessary has the same distance to all leaves. The nodes of an evolutionary tree can be partially ordered such that two nodes are comparable if there is a path from the root to any of the leaves containing the two nodes in question, and in this case the smaller node is the one that is closer to the root on the path. Each node v of an evolutionary tree indicates a subtree that contains v and all the nodes that are greater than v. Hereafter we consider only these subtrees.

Given a set S of l-long sequences over alphabet Σ, a substitution model M on Σ and an evolutionary tree T are labeled by the sequences. The likelihood of the tree is the probability of observing the sequences at the leaves of the tree, given that the substitution process starts at the root of the tree with the equilibrium distribution. This likelihood is denoted by $P(S|T, M)$. The substitution likelihood problem is to calculate the likelihood of the tree.

Let Σ be a finite alphabet and let $S_1 = s_{1,1}s_{1,2}\dots s_{1,L_1}$, $S_2 = s_{2,1}s_{2,2}\dots s_{2,L_2}$, $\dots$ $S_n = s_{n,1}s_{n,2}\dots s_{n,L_n}$ be sequences over this alphabet. Let a TKF91 model $TKF91$ be given with its parameters: substitution model M, insertion rate λ, and deletion rate μ. Let T be an evolutionary tree labeled by S_1, S_2, $\dots$ S_n. The multiple statistical alignment problem is to calculate the likelihood of the tree, $P(S_1, S_2, \dots S_n|T, TKF91)$, given that



Statistical Multiple Alignment, Fig. 2 The probabilities of alignment patterns. From *left* to *right*: k insertions at the beginning of the alignment, a match followed by $k-1$ insertions, a deletion followed by k insertions, and a deletion not followed by insertions. $\beta = \frac{1-e^{(\lambda-\mu)t}}{\mu-\lambda e^{(\lambda-\mu)t}}$

the TKF91 process starts at the root with the equilibrium distribution.

Multiple Hidden Markov Models

It will turn out that the TKF91 model can be transformed to a multiple Hidden Markov Model; therefore we formally define it here. A multiple Hidden Markov Model (multiple HMM) is a directed graph with distinguished start and end states, (the in degree of the start and the out degree of the end state are both 0), together with the following described transition and emission distributions. Each vertex has a transition distribution over its out edges. The vertexes can be divided into two classes, the emitting and silent states. Each emitting state emits one-one random character to a prescribed set of sequences; it is possible that a state emits only one character to one sequence. For each state, an emission distribution over the alphabet and the set of sequences gives the probabilities which characters will be emitted to which sequences. The Markov process is a random walk from the start to the end, following the transition distribution on the out edges. When the walk is in an emitting state, characters are emitted according to the emission distribution of the state. The process is hidden since the observer sees only the emitted sequences, and the observer does not observe which character is emitted by which state, even the observer does not see which characters are co-emitted. The multiple HMM problem is to calculate the emission probability of a set of sequences for a multiple HMM. This probability can be calculated with the forward algorithm that has $O(V^2L^n)$ running time, where V is the number of emitting states in the multiple HMM, L is the geometric mean of the sequences, and n is the number of sequences [3].

Key Results

Substitutions have been modeled with time-continuous Markov models since the late 1960s [8], and an efficient algorithm for likelihood calculations was published in 1980 [4]. The running time of this efficient algorithm grows linearly both with the number of sequences and with the length of the sequences being analyzed, and it grows squarely with the size of the alphabet. The algorithm belongs to the class of dynamic programming algorithms. For each character, subtree, and position x, the algorithm calculates what would be the likelihood of the characters in position x in the sequences belonging to the subtree if the substitution process started in the root of the subtree with the given character. These probabilities are called conditional likelihoods. It is easy to show that

$$L_p(\alpha, x) = \left(\sum_{\alpha_1} P_{t_1}(\alpha_1|\alpha) L_{d_1}(\alpha_1, x)\right) \left(\sum_{\alpha_2} P_{t_2}(\alpha_2|\alpha) L_{d_2}(\alpha_2, x)\right) \quad (6)$$

where d_1 and d_2 are the descendant nodes of the parent node p and t_1 and t_2 are the length of the edges connecting p with d_1 and d_2, respectively. The likelihood of the tree can be calculated from the conditional likelihoods of the tree. Recall that

$P(S|T, M)$ is the likelihood of observing a set of sequences S on the leaves of an evolutionary tree T under the substitution model M:

$$P(S|T, M) = \prod_x \sum_\alpha L_{root}(\alpha, x)\pi_\alpha \quad (7)$$

Thorne, Kishino, and Felsenstein gave an $O(nm)$ running time algorithm for calculating the likelihood of an n-long and an m-long sequence under their model [16]. It was not clear for long time how to extend this algorithm to more than two sequences. In 2001, several researchers [7,12] realized that the TKF91 model for two sequences is equivalent with a pair Hidden Markov Model (pair HMM) in the sense that the transition and emission probabilities of the pair HMM can be parameterized with λ, μ and the transition and equilibrium probabilities of the substitution model; moreover there is a bijection between the paths emitting the two sequences and alignments such that the probability of a path in the pair HMM equals to the probability of the corresponding alignment of the two sequences. Hence the likelihood of two sequences can be calculated with the forward algorithm of the pair HMM.

After this discovery, it was relatively easy to develop an algorithm for multiple statistical alignment [5]. The key observation is that a multiple HMM can be created as a composition of pair HMMs along the evolutionary tree. This technique was already known in the speech recognition literature [14], and was also rediscovered by Ian Holmes [6], who named this technique as transducer composition. The number of states in the so-created multiple HMM is $O(5^{\frac{n}{2}})$, where n is the number of leaves of the tree. The emission probabilities are the substitution likelihoods on the tree, which can be efficiently calculated as shown above. The running time of the forward algorithm is $5^n L^n$, where L is the geometric mean of the sequence lengths.

Lunter et al. [9] introduced an algorithm that does not need a multiple HMM description of the TKF91 model to calculate the likelihood of a tree. Using a logical sieve algorithm, they were able to reduce the running time to $O(2^n L^n)$. They called their algorithm the "one-state recursion" since their dynamic programming algorithm does not need different state of a multiple HMM to calculate the likelihood correctly.

Applications

Since the running time of the best known algorithm for multiple statistical alignment grows exponentially with the number of sequences, on its own it is not useful in practice. However, Lunter et al. also showed that there is a one-state recursion to calculate the likelihood of the tree given an alignment [10]. The running time of this algorithm grows only linearly with both the alignment length and the number of sequences. Since the number of states in a multiple HMM that can emit the same multiple alignment column might grow exponentially, this version of the one-state recursion is a significant improvement. The one-state recursion for multiple alignments is used in a Bayesian Markov chain Monte Carlo where the state space is the Descartes product of the possible multiple alignments and evolutionary trees. The one-state recursion provides an efficient likelihood calculation for a point in the state space [11].

Csűrös and Miklós introduced a model for gene content evolution that is equivalent with the multiple statistical alignment problem for alphabet size 1 [2]. They gave a polynomial running time algorithm that calculates the likelihood of the tree. The running time is $O(n + hL^2)$, where n is the number of sequences, h is the height of the evolutionary tree, and L is the sum of the sequence lengths.

Thorne, Kishino, and Felsenstein also introduced a fragment model, also called the TKF92 model, in which multiple insertions and deletions are allowed [17]. The birth process is still a Poisson process, but instead of single characters, fragments of characters are inserted with a geometrically distributed length. The fragments are unbreakable, and the death process is going on the fragments. The TKF92 model for a pair of sequences also can be described into a pair HMM and the TKF92 model on a tree can be

transformed to a multiple HMM. Such multiple HMM is used in the StatAlign software package [13]. The software package has been extended to predict the common structure of sequences (e.g., slowly quickly evolving regions, RNA secondary structures) by combining this multiple HMM with other stochastic models describing the structure of sequences [1, 15].

Open Problems

It is conjectured that the multiple statistical alignment problem cannot be solved in polynomial time for any nontrivial alphabet size. One also can ask what the most likely multiple alignment is or, equivalently, what the most probable path in the multiple HMM is that emits the given sequences. For a set of sequences, a TKF91 model, and an evolutionary tree, the decision problem "Is there a multiple alignment that is more probable than p" is conjectured to be NP-complete.

It is conjectured that there is no one-state recursion for the TKF92 model.

Recommended Reading

1. Arunapuram P, Edvardsson I, Golden M, Anderson JW, Novák Á, Sükösd Z, Hein J (2013) StatAlign 2.0: combining statistical alignment with RNA secondary structure prediction. Bioinformatics 29(5):654–655
2. Csűrös M, Miklós I (2006) A probabilistic model for gene content evolution with duplication, loss, and horizontal transfer. In: Proceedings of RECOMB2006. Lecture notes in bioinformatics, Springer Verlag, vol 3909, pp 206–220
3. Durbin R, Eddy S, Krogh A, Mitchison G (1998). Biological sequence analysis. Cambridge University Press, Cambridge
4. Felsenstein J (1981) Evolutionary trees from DNA sequences: a maximum likelihood approach. J Mol Evol 17:368–376
5. Hein JJ, Jensen JL, Pedersen CNS (2003) Recursions for statistical multiple alignment. PNAS 100:14960–14965
6. Holmes I (2003) Using guide trees to construct multiple-sequence evolutionary hmms. Bioinformatics 19:i147–i157
7. Holmes I, Bruno WJ (2001) Evolutionary HMMs: a Bayesian approach to multiple alignment. Bioinformatics 17(9):803–820
8. Jukes TH, Cantor CR (1969) Evolution of protein molecules. In: Munro HN (ed) Mammalian protein metabolism. Academic, New York, pp 21–132
9. Lunter GA, Miklós I, Song YS, Hein J (2003) An efficient algorithm for statistical multiple alignment on arbitrary phylogenetic trees. J Comput Biol 10(6):869–889
10. Lunter GA, Miklós I, Drummond AJ, Jensen JL, Hein JJ (2003) Bayesian phylogenetic inference under a statistical indel model. In: Proceedings of WABI2003. Lecture notes in bioinformatics, Springer Verlag, vol 2812, pp 228–244
11. Lunter GA, Miklós I, Drummond AJ, Jensen JL, Hein JJ (2005) Bayesian coestimation of phylogeny and sequence alignment. BMC Bioinformatics 6:83
12. Metzler D, Fleißner R, Wakolbringer A, von Haeseler A (2001) Assessing variability by joint sampling of alignments and mutation rates. J Mol Evol 53:660–669
13. Novák A, Miklós I, Lyngsø R, Hein J (2008) StatAlign: an extendable software package for joint bayesian estimation of alignments and evolutionary trees. Bioinformatics 24(20):2403–2404
14. Pereira F, Riley M (1997) Speech recognition by composition of weighted finite automata. In: Finite-state language processing. MIT, Cambridge, pp 149–173
15. Satija R, Novák A, Miklós I, Lyngsø R, Hein J (2009) BigFoot: Bayesian alignment and phylogenetic footprinting with MCMC. BMC Evol Biol 9:217
16. Thorne JL, Kishino H, Felsenstein J (1991) An evolutionary model for maximum likelihood alignment of DNA sequences. J Mol Evol 33:114–124
17. Thorne JL, Kishino H, Felsenstein J (1992) Inching toward reality: an improved likelihood model of sequence evolution. J Mol Evol 34:3–16

Statistical Query Learning

Vitaly Feldman
IBM Research – Almaden, San Jose, CA, USA

Keywords

Classification noise; Noise-tolerant learning; PAC learning; SQ dimension; Statistical query

Years and Authors of Summarized Original Work

1998; Kearns

Problem Definition

The problem deals with learning to classify from random labeled examples in Valiant's PAC model [30]. In the *random classification noise* model of Angluin and Laird [1], the label of each example given to the learning algorithm is flipped randomly and independently with some fixed probability η called the *noise rate*. Robustness to such benign form of noise is an important goal in the design of learning algorithms. Kearns defined a powerful and convenient framework for constructing noise-tolerant algorithms based on *statistical queries*. Statistical query (SQ) learning is a natural restriction of PAC learning that models algorithms that use statistical properties of a data set, rather than individual examples. Kearns demonstrated that any learning algorithm that is based on statistical queries can be automatically converted to a learning algorithm in the presence of random classification noise of arbitrary rate smaller than the information-theoretic barrier of $1/2$. This result was used to give the first noise-tolerant algorithm for a number of important learning problems. In fact, virtually all known noise-tolerant PAC algorithms were either obtained from SQ algorithms or can be easily cast into the SQ model.

In subsequent work, the model of Kearns has been extended to other settings and found a number of additional applications in machine learning and theoretical computer science.

Definitions and Notation

Let $\mathcal{C}$ be a class of $\{-1, +1\}$-valued functions (also called *concepts*) over an input space X. In the basic PAC model, a learning algorithm is given examples of an unknown function f from $\mathcal{C}$ on points randomly chosen from some unknown distribution $\mathcal{D}$ over X and should produce a hypothesis h that approximates f. More formally, an *example oracle* $\mathrm{EX}(f, \mathcal{D})$ is an oracle that upon being invoked returns an example $\langle x, f(x)\rangle$, where x is chosen randomly with respect to $\mathcal{D}$, independently of any previous examples. A learning algorithm for $\mathcal{C}$ is an algorithm that for every $\epsilon > 0$, $\delta > 0$, $f \in \mathcal{C}$, and distribution $\mathcal{D}$ over X, given ϵ, δ, and access to $\mathrm{EX}(f, \mathcal{D})$ outputs, with probability at least $1 - \delta$, a hypothesis h that ϵ-approximates f with respect to $\mathcal{D}$ (i.e., $\mathbf{Pr}_{\mathcal{D}}[f(x) \neq h(x)] \leq \epsilon$). Efficient learning algorithms are algorithms that run in time polynomial in $1/\epsilon$, $1/\delta$ and the size of the learning problem s. The size of a learning problem is determined by the description length of f under some fixed representation scheme for functions in $\mathcal{C}$ and the description length of an element in X (often proportional to the dimension n of the input space).

A number of variants of this basic framework are commonly considered. The basic PAC model is also referred to as *distribution-independent* learning to distinguish it from *distribution-specific* PAC learning in which the learning algorithm is required to learn with respect to a single distribution $\mathcal{D}$ known in advance. A *weak* learning algorithm is a learning algorithm that can produce a hypothesis whose error on the target concept is noticeably less than $1/2$ (and not necessarily any $\epsilon > 0$). More precisely, a weak learning algorithm produces a hypothesis h such that $\mathbf{Pr}_{\mathcal{D}}[f(x) \neq h(x)] \leq 1/2 - 1/p(s)$ for some fixed polynomial p. The basic PAC model is often referred to as *strong* learning in this context.

In the random classification noise model $\mathrm{EX}(f, \mathcal{D})$ is replaced by a faulty oracle $\mathrm{EX}^{\eta}(f, \mathcal{D})$, where η is the noise rate. When queried, this oracle returns a noisy example $\langle x, b\rangle$ where $b = f(x)$ with probability $1 - \eta$ and $\neg f(x)$ with probability η independently of previous examples. When η approaches $1/2$ the label of the corrupted example approaches the result of a random coin flip, and therefore, the running time of learning algorithms in this model is allowed to depend on $\frac{1}{1-2\eta}$ (the dependence must be polynomial for the algorithm to be considered efficient). For simplicity, one usually assumes that η is known to the learning algorithm. This assumption can be removed using a simple technique due to Laird [26].

To formalize the idea of learning from statistical properties of a large number of examples, Kearns introduced a new oracle $\mathrm{STAT}(f, \mathcal{D})$ that replaces $\mathrm{EX}(f, \mathcal{D})$. The oracle $\mathrm{STAT}(f, \mathcal{D})$ takes

as input a *statistical query* (SQ) of the form (χ, τ), where χ is a $\{-1, +1\}$-valued function on labeled examples and $\tau \in [0, 1]$ is the *tolerance* parameter. Given such a query, the oracle responds with an estimate v of $\mathbf{Pr}_{\mathcal{D}}[\chi(x, f(x)) = 1]$ that is accurate to within an additive $\pm\tau$.

Note that the oracle does not guarantee anything else on the value v beyond $|v - \mathbf{Pr}_{\mathcal{D}}[\chi(x, f(x)) = 1]| \leq \tau$ and an SQ learning algorithm needs to work with any possible implementation of the oracle. Yang proposed a stronger, *honest* version of the oracle which to a call with function χ returns the value of $\chi(x, f(x))$, where x is chosen randomly and independently according to $\mathcal{D}$ [32]. This version was shown to be equivalent to the original model up to polynomial factors [17].

Chernoff bounds easily imply that $\mathrm{STAT}(f, \mathcal{D})$ can, with high probability, be simulated using $\mathrm{EX}(f, \mathcal{D})$ by estimating $\mathbf{Pr}_{\mathcal{D}}[\chi(x, f(x)) = 1]$ on $O(\tau^{-2})$ examples. Therefore, the SQ model is a restriction of the PAC model. Efficient SQ algorithms allow only efficiently evaluatable χ's and impose an inverse polynomial lower bound on the tolerance parameter over all oracle calls. Kearns also observes that in order to simulate all the statistical queries used by an algorithm, one does not necessarily need new examples for each estimation. Instead, assuming that the set of possible queries of the algorithm has Vapnik-Chervonenkis dimension d, all its statistical queries can be simulated using $\tilde{O}(d\,\tau^{-2}(1-2\eta)^{-2}\log(1/\delta))$ examples [24].

Key Results

Statistical Queries and Noise-Tolerance

The main result given by Kearns is a way to simulate statistical queries using noisy examples.

Lemma 1 ([24]) *Let (χ, τ) be a statistical query such that χ can be evaluated on any input in time T and let $\mathrm{EX}^{\eta}(f, \mathcal{D})$ be a noisy oracle. The value $\mathbf{Pr}_{\mathcal{D}}[\chi(x, f(x)) = 1]$ can, with probability at least $1-\delta$, be estimated within τ using $O(\tau^{-2}(1-2\eta)^{-2}\log(1/\delta))$ examples from $\mathrm{EX}^{\eta}(f, \mathcal{D})$ and time $O(\tau^{-2}(1-2\eta)^{-2}\log(1/\delta)\cdot T)$.*

This simulation is based on estimating several probabilities using examples from the noisy oracle and then offsetting the effect of noise. The lemma implies that any efficient SQ algorithm for a concept class $\mathcal{C}$ can be converted to an efficient learning algorithm for $\mathcal{C}$ tolerating random classification noise of any rate $\eta < 1/2$.

Theorem 1 ([24]) *Let $\mathcal{C}$ be a concept class efficiently* PAC *learnable from statistical queries. Then $\mathcal{C}$ is efficiently* PAC *learnable in the presence of random classification noise of rate η for any $\eta < 1/2$.*

Balcan and Feldman describe more general conditions on noise under which a specific SQ algorithm can be simulated in the presence of noise [3].

Statistical Query Algorithms

Kearns showed that, despite the major restriction on the way an SQ algorithm accesses the examples, many PAC learning algorithms known at the time can be modified to use statistical queries instead of random examples [24]. Examples of learning algorithms for which he described an SQ analogue and thereby obtained a noise-tolerant learning algorithm include:

- Learning decision trees of constant rank.
- *Attribute-efficient* algorithms for learning conjunctions.
- Learning axis-aligned rectangles over $\mathbb{R}^n$.
- Learning AC^0 (constant-depth unbounded fan-in) Boolean circuits over $\{0,1\}^n$ with respect to the uniform distribution in quasipolynomial time.

Subsequent works have provided numerous additional examples of algorithms used in theory and practice of machine learning that can either be implemented using statistical queries or can be replaced by an alternative SQ-based algorithm of similar complexity. For example, the Perceptron algorithm and learning of linear threshold functions [6, 12], boosting [2], attribute-efficient learning via the Winnow algorithm (cf. [16]), k-means clustering [5] and convex optimization-

based methods [20]. We note that many learning algorithms rely only on evaluations of functions on random examples and therefore can be seen as using access to the honest statistical query oracle. In such cases the SQ implementation follows immediately from the equivalence of the Kearns' SQ oracle and the honest one [17].

The only known example of a technique for which there is no SQ analogue is Gaussian elimination for solving linear equations over a finite field. This technique can be used to learn parity functions that are not learnable using SQs (as we discuss below). As a result, with the exception of the parity learning problem, known bounds on the complexity of learning from random examples are, up to polynomial factors, the same as known bound for learning with statistical queries.

Statistical Query Dimension

The restricted way in which SQ algorithms use examples makes it simpler to understand the limitations of efficient learning in this model. A long-standing open problem in learning theory is learning of the concept class of all parity functions over $\{0,1\}^n$ with noise (a parity function is a XOR of some subset of n Boolean inputs). Kearns has demonstrated that parities cannot be efficiently learned using statistical queries even under the uniform distribution over $\{0,1\}^n$ [24]. This hardness result is unconditional in the sense that it does not rely on any unproven complexity assumptions.

The technique of Kearns was generalized by Blum et al. who proved that efficient SQ learnability of a concept class $\mathcal{C}$ is characterized by a relatively simple combinatorial parameter of $\mathcal{C}$ called the *statistical query dimension* [7]. The quantity they defined, measures the maximum number of "nearly uncorrelated" functions in a concept class. (The definition and the results were simplified and strengthened in subsequent works [17, 29] and we use the improved statements here.) More formally,

Definition 1 For a concept class $\mathcal{C}$ and distribution $\mathcal{D}$, the *statistical query dimension* of $\mathcal{C}$ with respect to $\mathcal{D}$, denoted SQ-DIM$(\mathcal{C},\mathcal{D})$, is the largest number d such that $\mathcal{C}$ contains d functions $f_1, f_2, \ldots, f_d$ such that for all $i \neq j$, $|\mathbf{E}_{\mathcal{D}}[f_i f_j]| \leq \frac{1}{d}$.

Blum et al. relate the SQ dimension to learning in the SQ model as follows.

Theorem 2 ([7, 17]) *Let $\mathcal{C}$ be a concept class and $\mathcal{D}$ be a distribution such that* SQ-DIM$(\mathcal{C},\mathcal{D}) = d$.

- *If all queries are made with tolerance of at least $1/d^{1/3}$, then at least $d^{1/3}-2$ queries are required to learn $\mathcal{C}$ with error $1/2 - 1/(2d^3)$ in the SQ model.*
- *There exists an algorithm for learning $\mathcal{C}$ with respect to $\mathcal{D}$ that makes d fixed queries, each of tolerance $1/(4d)$, and finds a hypothesis with error at most $1/2 - 1/(2d)$.*

Thus SQ-DIM characterizes weak SQ learnability relative to a fixed distribution $\mathcal{D}$ up to a polynomial factor. Parity functions are uncorrelated with respect to the uniform distribution and therefore, any concept class that contains a superpolynomial number of parity functions cannot be learned by statistical queries with respect to the uniform distribution. This, for example, includes such important concept classes as *k-juntas* over $\{0,1\}^n$ (or functions that depend on at most k input variables) for $k = \omega(1)$ and *decision trees* of superconstant size.

Simon showed that (strong) PAC learning relative to a fixed distribution $\mathcal{D}$ using SQs can also be characterized by a more general and involved dimension [28]. Simpler and tighter characterizations of distribution-specific PAC learning using SQs have been demonstrated by Feldman [15] and Szörényi [29]. Feldman also extended the characterization to the agnostic learning model.

Despite characterizing the number of queries of certain tolerance, the SQ-DIM and its generalizations capture surprisingly well the computational complexity of SQ learning of most concept classes. One reason for this is that if a concept class has polynomial SQ-DIM then it can be learned by a polynomial-time algorithm with advice also referred to as a "non-uniform"

algorithm (cf. [18]). However it was shown by Feldman and Kanade that for strong PAC learning there exist artificial problems whose computational complexity is larger than their statistical query complexity [18].

Applications of these characterizations to proving lower bounds on SQ algorithms can be found in [11, 15, 19, 25]. Relationships of SQ-DIM to other notions of complexity of concept classes were investigated in [22, 27].

Applications

The ideas behind the use of statistical queries to produce noise-tolerant algorithms were adapted to learning using *membership queries* (or ability to ask for the value of the unknown function at any point) and used to give a noise-tolerant algorithm for learning DNF with respect to the uniform distribution [9, 21]. The SQ model of learning was generalized to active learning (or learning where labels are requested only for some of the points) and used to obtain new efficient noise-tolerant active learning algorithms [3].

The restricted way in which an SQ algorithm uses data implies it can be used to obtain learning algorithms with additional useful properties. Blum et al. [5] show that an SQ algorithm can be used to obtain a *differentially-private* [13] algorithm for the problem. In fact, SQ algorithms are equivalent to *local* (or *randomized-response*) differentially-private algorithms [23]. Chu et al. [10] show that SQ algorithms can be automatically parallelized on multicore architectures and give many examples of popular machine learning algorithms that can be sped up using this approach.

The SQ learning model has also been instrumental in understanding Valiant's model of evolution as learning [31]. Feldman showed that the model is equivalent to learning with a restricted form of SQs referred to as correlational SQs [14]. A correlational SQ is a query of the form $\chi(x, \ell) = g(x) \cdot \ell$ for some $g : X \to [-1, 1]$. Such queries were first studied by Ben-David et al. [4] (remarkably, before the introduction of the SQ model itself) and distribution-specific learning with such queries is equivalent to learning with (unrestricted) SQs.

Statistical query-based access can naturally be defined for any problem where the input is a set of i.i.d. samples from a distribution. Feldman et al. show that lower bounds based on SQ-DIM can be extended to this more general setting and give examples of applications [17, 20].

Open Problems

The main questions related to learning with random classification noise are still open. Is every concept class efficiently learnable in the PAC model also learnable in the presence of random classification noise? Is every concept class efficiently learnable in the presence of random classification noise of arbitrarily high rate (less than $1/2$) also efficiently learnable using statistical queries? A partial answer to this question was provided by Blum et al. who show that Gaussian elimination can be used in low dimension to obtain a class learnable with random classification noise of constant rate $\eta < 1/2$ but not learnable using SQs [8]. For both questions a central issue seems to be obtaining a better understanding of the complexity of learning parities with noise.

The complexity of learning from statistical queries remains an active area of research with many open problems. For example, there is currently an exponential gap between known lower and upper bounds on the complexity of distribution-independent SQ learning of polynomial-size DNF formulae and AC^0 circuits (cf. [27]). Several additional open problems on complexity of SQ learning can be found in [16, 19, 22].

Cross-References

- Attribute-Efficient Learning
- Learning Constant-Depth Circuits
- Learning DNF Formulas
- Learning Heavy Fourier Coefficients of Boolean Functions
- Learning with Malicious Noise
- PAC Learning

Recommended Reading

1. Angluin D, Laird P (1988) Learning from noisy examples. Mach Learn 2:343–370
2. Aslam J, Decatur S (1998) General bounds on statistical query learning and pac learning with noise via hypothesis boosting. Inf Comput 141(2):85–118
3. Balcan M-F, Feldman V (2013) Statistical active learning algorithms. In: NIPS, Lake Tahoe, pp 1295–1303
4. Ben-David S, Itai A, Kushilevitz E (1990) Learning by distances. In: Proceedings of COLT, Rochester, pp 232–245
5. Blum A, Dwork C, McSherry F, Nissim K (2005) Practical privacy: the SuLQ framework. In: Proceedings of PODS, Baltimore, pp 128–138
6. Blum A, Frieze A, Kannan R, Vempala S (1997) A polynomial time algorithm for learning noisy linear threshold functions. Algorithmica 22(1/2):35–52
7. Blum A, Furst M, Jackson J, Kearns M, Mansour Y, Rudich S (1994) Weakly learning DNF and characterizing statistical query learning using Fourier analysis. In: Proceedings of STOC, Montréal, pp 253–262
8. Blum A, Kalai A, Wasserman H (2003) Noise-tolerant learning, the parity problem, and the statistical query model. J ACM 50(4):506–519
9. Bshouty N, Feldman V (2002) On using extended statistical queries to avoid membership queries. J Mach Learn Res 2:359–395
10. Chu C, Kim S, Lin Y, Yu Y, Bradski G, Ng A, Olukotun K (2006) Map-reduce for machine learning on multicore. In: Proceedings of NIPS, Vancouver, pp 281–288
11. Dachman-Soled D, Feldman V, Tan L-Y, Wan A, Wimmer K (2014) Approximate resilience, monotonicity, and the complexity of agnostic learning. arXiv, CoRR, abs/1405.5268
12. Dunagan J, Vempala S (2004) A simple polynomial-time rescaling algorithm for solving linear programs. In: Proceedings of STOC, Chicago, pp 315–320
13. Dwork C, McSherry F, Nissim K, Smith A (2006) Calibrating noise to sensitivity in private data analysis. In: TCC, New York, pp 265–284
14. Feldman V (2008) Evolvability from learning algorithms. In: Proceedings of STOC, Victoria, pp 619–628
15. Feldman V (2012) A complete characterization of statistical query learning with applications to evolvability. J Comput Syst Sci 78(5):1444–1459
16. Feldman V (2014) Open problem: the statistical query complexity of learning sparse halfspaces. In: COLT, Barcelona, pp 1283–1289
17. Feldman V, Grigorescu E, Reyzin L, Vempala S, Xiao Y (2013) Statistical algorithms and a lower bound for planted clique. In: STOC, Palo Alto. ACM, pp 655–664
18. Feldman V, Kanade V (2012) Computational bounds on statistical query learning. In: COLT, Edinburgh, pp 16.1–16.22
19. Feldman V, Lee H, Servedio R (2011) Lower bounds and hardness amplification for learning shallow monotone formulas. In: COLT, Budapest, vol 19, pp 273–292
20. Feldman V, Perkins W, Vempala S (2013) On the complexity of random satisfiability problems with planted solutions. In: CoRR, abs/1311.4821
21. Jackson J, Shamir E, Shwartzman C (1997) Learning with queries corrupted by classification noise. In: Proceedings of the fifth Israel symposium on the theory of computing systems, Ramat-Gan, pp 45–53
22. Kallweit M, Simon H (2011) A close look to margin complexity and related parameters. In: COLT, Budapest, pp 437–456
23. Kasiviswanathan SP, Lee HK, Nissim K, Raskhodnikova S, Smith A (2011) What can we learn privately? SIAM J Comput 40(3):793–826
24. Kearns M (1998) Efficient noise-tolerant learning from statistical queries. J ACM 45(6): 983–1006
25. Klivans A, Sherstov A (2007) Unconditional lower bounds for learning intersections of halfspaces. Mach Learn 69(2–3):97–114
26. Laird P (1988) Learning from good and bad data. Kluwer Academic, Boston
27. Sherstov AA (2008) Halfspace matrices. Comput Complex 17(2):149–178
28. Simon H (2007) A characterization of strong learnability in the statistical query model. In: Proceedings of symposium on theoretical aspects of computer science, Aachen, pp 393–404
29. Szörényi B (2009) Characterizing statistical query learning: simplified notions and proofs. In: Proceedings of ALT, Porto, pp 186–200
30. Valiant LG (1984) A theory of the learnable. Commun ACM 27(11):1134–1142
31. Valiant LG (2009) Evolvability. J ACM 56(1):3.1–3.21. Earlier version in ECCC, 2006
32. Yang K (2005) New lower bounds for statistical query learning. J Comput Syst Sci 70(4):485–509

Statistical Timing Analysis

Sachin S. Sapatnekar
Department of Electrical and Computer Engineering, University of Minnesota, Minneapolis, MN, USA

Keywords

Delay uncertainty; Electronic design automation; Principal component analysis; Process variations; Static timing analysis; Timing closure

Years and Authors of Summarized Original Work

2003; Chang, Sapatnekar
2005; Chang, Sapatnekar

Problem Definition

The timing behavior of integrated systems is strongly affected by the characteristics of transistors and wires in the system. Variations in the manufacturing process can cause drifts in these characteristics from one manufactured part to another. The traditional approach to addressing these variations was to choose a worst-case value for each process parameter, but this has become unsustainable in the face of current-day variations. Statistical timing analysis provides a computationally efficient way to translate the probability density function of the underlying process parameter spread to the distribution of circuit timing.

A key underlying structure for timing analysis is a graph $G(V, E)$ of a combinational circuit, where the vertex set V corresponds to the gates, primary inputs, and primary outputs of the circuit, and each connection between these gates corresponds to an edge in E. The delay of each gate corresponds to a probability distribution that is a function of the distributions of the underlying (possibly correlated) process parameters, and the task of combinational statistical timing analysis is to obtain the distribution of the maximum (or minimum) delay of the circuit, over all primary outputs. The extension of this problem to general edge-triggered sequential circuits is straightforward. Such circuits can be decomposed into independent combinational blocks, and the maximum (or minimum) operator acts on the delay distribution at all primary outputs of all combinational blocks of the sequential circuit.

Key Results

The framework that is used for statistical timing analysis is based on graph-based topological traversals that maintain a closed-form structure for the delay from the primary inputs of the circuit to the output of each vertex (referred to as the *arrival time*). The computation under this paradigm scales linearly with $|E|$. While it is certainly possible to perform statistical timing analysis through Monte Carlo simulations based on samples of the process parameter space, such an approach is uncompetitive compared to graph traversal algorithms. The traversal approach consists of three key steps [1, 2]:

- Translating the underlying process parameter variations to an orthogonal set of random variables
- Representing gate delay variations in terms of this orthogonal set
- Performing a topological traversal of G and computing the arrival time at each node and maximum delay of the circuit

Orthogonalizing Process Parameter Distributions

A common assumption is that the underlying process parameters, such as the transistor width W and effective length L_{eff} of devices, gate oxide thickness (T_{ox}), and device threshold voltage (V_{t}) due to random dopant fluctuations, show a Gaussian distribution. Each individual device is separately represented by such a parameter. The distributions of T_{ox} and V_{t} are largely uncorrelated across devices. In contrast, the dimension-based parameters, W and L_{eff}, show strong spatial correlations, whereby the distributions of nearby devices are strongly correlated, and this correlation falls off as a function of distance.

The existence of correlations can significantly complicate the task of statistical timing analysis, since all pairwise combinations of random variables must be considered during the optimization, potentially leading to quadratic complexity in $|V|$. To overcome this, an initial principal component analysis (PCA) [7] step is carried out that orthogonalizes the underlying Gaussians, enabling linear-time analysis. PCA is a one-time operation for a given process (which is used for numerous designs). Therefore, although its worst-case com-

plexity is cubic in $|V|$, the expense is practically manageable as it is amortized over numerous designs. Furthermore, sparsity properties of the correlation matrix realistically imply that in practice, the cost of PCA scales considerably slower than this cubic rate.

For cases where the underlying process parameters may be a mix of Gaussians or non-Gaussians, it is possible to orthogonalize the Gaussian parameters using PCA and non-Gaussian parameters using independent component analysis (ICA) [4]. The approach in [8] extends the graph-based approach presented here and shows how statistical timing analysis can be performed for case where some or all process parameters are non-Gaussian.

Gate Delay Distribution

To build a model for the gate delay that captures the underlying variations in process parameters, we observe that the delay function $d = f(\mathbf{P})$, where $\mathbf{P}$ is a set of process parameters, can be approximated d linearly using a first-order Taylor expansion:

$$d = d_0 + \sum_{\forall \text{ parameters } p_i} \left[\frac{\partial f}{\partial p_i}\right]_0 \Delta p_i \quad (1)$$

where d_0 is the nominal value of d, calculated at the nominal values of parameters in the set $\mathbf{P}$; $\left[\frac{\partial f}{\partial p_i}\right]_0$ is computed at the nominal values of p_i; $\Delta p_i = p_i - \mu_{p_i}$ is a normally distributed random variable; and $\Delta p_i \sim N(0, \sigma_{p_i})$. The delay function here can be arbitrarily complex.

If all parameters in $\mathbf{P}$ can be modeled by Gaussian distributions, this approximation implies that d is a linear combination of Gaussians, which is therefore Gaussian. Its mean μ_d and variance σ_d^2 are

$$\mu_d = d_0 \quad (2)$$

$$\sigma_d^2 = \sum_{\forall i} \left[\frac{\partial f}{\partial p_i}\right]_0^2 \sigma_{p_i}^2 + 2 \sum_{\forall i \neq j} \left[\frac{\partial f}{\partial p_i}\right]_0 \left[\frac{\partial f}{\partial p_j}\right]_0 \text{cov}\left(p_i, p_j\right) \quad (3)$$

where $\text{cov}\left(p_i, p_j\right)$ is the covariance of p_i and p_j.

This approximation is valid when Δp_i has relatively small variations, in which domain the first-order Taylor expansion is adequate and the approximation is acceptable with little loss of accuracy. This is generally true of the impact of within-die variations on delay, where the process parameter variations are relatively small in comparison with the nominal values, and the function changes by a small amount under this perturbation. Hence, the delays, as functions of the process parameters, can be approximated as normal distributions when the parameter variations are assumed to be normal. Higher-order expansions based on quadratics have also be explored to cover cases where the variations are larger [6, 11].

Circuit Delay Distribution

A PCA-based approach maintains the invariant that the output arrival time at each gate is a Gaussian variable represented as

$$a_i(p_1, \ldots, p_n) = a_i^0 + \sum_{i=1}^{n} k_i p_i' + k_{n+1} p_{n+1}' \quad (4)$$

Here, the primed variables correspond to the principal components of the unprimed variables and maintain the form of the arrival time after each sum and max operation. Gate delays, as represented in Eq. 1, can be translated into a similar representation based on principal components as a one-time step during gate library characterization. Under orthogonalization, many operations become much simpler since the covariance terms disappear: for example, Eq. 3 can

be evaluated in linear time instead of quadratic time.

The task of statistical timing analysis is to translate these gate delay distributions to circuit delay probabilities while performing a topological traversal. The operations performed at each node encountered during this traversal in STA are of two types [5]:

- A gate (vertex) is being processed in STA when the arrival times of all inputs are known, at which time the candidate delay values at the output are computed using the "sum" operation that adds the delay at each input with the input-to-output pin delay.
- Once these candidate delays have been found, the "max" operation is applied to determine the maximum arrival time at the output.

Since the gate delays are Gaussian, the "sum" operation is merely an addition of Gaussians, which is well known to be a Gaussian. The computation of the max function, however, poses greater problems. The set of candidate delays are all Gaussian, so that this function must find the maximum of Gaussians. Such a maximum may be reasonably approximated using a Gaussian [3]. A detailed description of how the invariant representation is maintained under the max operation is presented in [1, 2].

The cost of this method corresponds to running a bounded number of deterministic STAs, and it is demonstrated to be accurate, given the statistics of **P**.

Applications

Statistical timing analysis has been extensively used in industry [10] and has seeded a large amount of academic research. Integrated circuit manufacturing foundries have promoted the use of statistical timing analysis by providing PCA-like information with their process parameter models, thus enabling design flows that are statistically based.

The ideas of statistical analysis have also motivated simpler and more approximate methods, used in industry today, based on on-chip variation (OCV) derating factors. In its most elementary form, OCV adds margins to each timing path to account for possible variation. More involved versions of OCV, such as advanced OCV (AOCV), capture the essence of spatial correlation by using derating factors that depend on factors such as spatial distance and logical depth of a path [9].

Experimental Results

Statistical timing analysis based on orthogonalization brings down the computational cost from quadratic to linear in the number of variables and can be applied to large circuit instances. The method is capable of considering both spatial correlations and structural correlations, i.e., correlations between paths that share gates, since such correlations are embedded into the invariant representation. This makes the approach accurate and computationally practical, as described in [1, 2, 10] and the large body of follow-on work.

URLs to Code and Data Sets

The MinnSSTA statistical static timing analyzer is available at http://www.ece.umn.edu/~sachin/software/MinnSSTA/index.html.

Recommended Reading

1. Chang H, Sapatnekar SS (2003) Statistical timing analysis considering spatial correlations using a single PERT-like traversal. In: Proceedings of the IEEE/ACM international conference on computer-aided design, San Jose, pp 621–625

2. Chang H, Sapatnekar SS (2005) Statistical timing analysis under spatial correlations. IEEE Trans Comput-Aided Des Integr Circuits Syst 24(9):1467–1482
3. Clark CE (1961) The greatest of a finite set of random variables. Oper Res 9:85–91
4. Hyvärinen A, Oja E (2000) Independent component analysis: algorithms and applications. Neural Netw 13:411–430
5. Jacobs E, Berkelaar MRCM (2000) Gate sizing using a statistical delay model. In: Proceedings of design and test in Europe, Paris, pp 283–290
6. Li X, Le J, Gopalakrishnan P, Pileggi LT (2007) Asymptotic probability extraction for nonnormal performance distributions. IEEE Trans Comput-Aided Des Integr Circuits Syst 26(1):16–37
7. Morrison DF (1976) Multivariate statistical methods. McGraw-Hill, New York
8. Singh J, Sapatnekar SS (2008) A scalable statistical static timing analyzer incorporating correlated non-Gaussian and Gaussian parameter variations. IEEE Trans Comput-Aided Des Integr Circuits Syst 27(1):160–173
9. Synopsys Inc (2009) PrimeTime® Advanced OCV Technology. www.synopsys.com/Tools/Implementation/SignOff/CapsuleModule/PrimeTime_AdvancedOCV_WP.pdf
10. Visweswariah C, Ravindran K, Kalafala K, Walker SG, Narayan S, Beece DK, Piaget J, Venkateswaran N, Hemmett JG (2006) First-order incremental block-based statistical timing analysis. IEEE Trans Comput-Aided Des Integr Circuits Syst 25(10):2170–2180
11. Zhan Y, Strojwas AJ, Li X, Pileggi LT, Newmark D, Sharma M (2005) Correlation-aware statistical timing analysis with non-Gaussian delay distributions. In: Proceedings of the ACM/IEEE design automation conference, San Jose, pp 77–82

Steiner Forest

Guido Schäfer
Institute for Mathematics and Computer Science, Technical University of Berlin, Berlin, Germany

Keywords

Requirement join; *R*-join

Years and Authors of Summarized Original Work

1995; Agrawal, Klein, Ravi

Problem Definition

The *Steiner forest problem* is a fundamental problem in network design. Informally, the goal is to establish connections between pairs of vertices in a given network at minimum cost. The problem generalizes the well-known *Steiner tree problem*. As an example, assume that a telecommunication company receives communication requests from their customers. Each customer asks for a connection between two vertices in a given network. The company's goal is to build a minimum cost network infrastructure such that all communication requests are satisfied.

Formal Definition and Notation

More formally, an instance $I = (G, c, R)$ of the Steiner forest problem is given by an undirected graph $G = (V, E)$ with vertex set V and edge set E, a non-negative cost function $c: E \to \mathbb{Q}^+$, and a set of vertex pairs $R = \{(s_1, t_1), \ldots, (s_k, t_k)\} \subseteq V \times V$. The pairs in R are called *terminal pairs*. A feasible solution is a subset $F \subseteq E$ of the edges of G such that for every terminal pair $(s_i, t_i) \in R$ there is a path between s_i and t_i in the subgraph $G[F]$ induced by F. Let the cost $c(F)$ of F be defined as the total cost of all edges in F, i.e., $c(F) = \sum_{e \in F} c(e)$. The goal is to find a feasible solution F of minimum cost $c(F)$. It is easy to see that there exists an optimum solution which is a forest.

The Steiner forest problem may alternatively be defined by a set of *terminal groups* $R = \{g_1, \ldots, g_k\}$ with $g_i \subseteq V$ instead of terminal pairs. The objective is to compute a minimum cost subgraph such that all terminals belonging to the same group are connected. This definition is equivalent to the one given above.

Related Problems

A special case of the Steiner forest problem is the *Steiner tree problem* (see also the entry ▸ Steiner Trees). Here, all terminal pairs share a common root vertex $r \in V$, i.e., $r \in \{s_i, t_i\}$ for all terminal pairs $(s_i, t_i) \in R$. In other words, the problem consists of a set of terminal vertices $R \subseteq V$ and a root vertex $r \in V$ and the goal is to connect the

terminals in R to r in the cheapest possible way. A minimum cost solution is a tree.

The *generalized Steiner network problem* (see the entry ▸ Generalized Steiner Network), also known as the *survivable network design problem*, is a generalization of the Steiner forest problem. Here, a *connectivity requirement* function $r: V \times V \to \mathbb{N}$ specifies the number of edge disjoint paths that need to be established between every pair of vertices. That is, the goal is to find a minimum cost multi-subset H of the edges of G (H may contain the same edge several times) such that for every pair of vertices $(x, y) \in V$ there are $r(x, y)$ edge disjoint paths from x to y in $G[H]$. The goal is to find a set H of minimum cost. Clearly, if $r(x, y) \in \{0, 1\}$ for all $(x, y) \in V \times V$, this problem reduces to the Steiner forest problem.

Key Results

Agrawal, Klein and Ravi [1, 2] give an approximation algorithm for the Steiner forest problem that achieves an approximation ratio of 2. More precisely, the authors prove the following theorem.

Theorem 1 *There exists an approximation algorithm that for every instance $I = (G, c, R)$ of the Steiner forest problem, computes a feasible forest F such that*

$$c(F) \leq \left(2 - \frac{1}{k}\right) \cdot \mathsf{OPT}(I),$$

where k is the number of terminal pairs in R and $\mathsf{OPT}(I)$ is the cost of an optimal Steiner forest for I.

Related Work

The Steiner tree problem is NP-hard [10] and APX-complete [4, 8]. The current best lower bound on the achievable approximation ratio for the Steiner tree problem is 1.0074 [21]. Goemans and Williamson [11] generalized the results obtained by Agrawal, Klein and Ravi to a larger class of connectivity problems, which they term *constrained forest problems*. For the Steiner forest problem, their algorithm achieves the same approximation ratio of $(2 - 1/k)$. The algorithms of Agrawal, Klein and Ravi [2] and Goemans and Williamson [11] are both based on the classical *undirected cut formulation* for the Steiner forest problem [3]. The integrality gap of this relaxation is known to be $(2 - 1/k)$ and the results in [2, 11] are therefore tight. Jain [15] presents a 2-approximation algorithm for the generalized Steiner network problem.

Primal-Dual Algorithm

The main ideas of the algorithm by Agrawal, Klein and Ravi [2] are sketched below; subsequently, AKR is used to refer to this algorithm. The description given here differs from the one in [2]; the interested reader is referred to [2] for more details.

The algorithm is based on the following integer programming formulation for the Steiner forest problem. Let $I = (G, c, R)$ be an instance of the Steiner forest problem. Associate an indicator variable $x_e \in \{0, 1\}$ with every edge $e \in E$. The value of x_e is 1 if e is part of the forest F and 0 otherwise. A subset $S \subseteq V$ of the vertices is called a *Steiner cut* if there exists at least one terminal pair $(s_i, t_i) \in R$ such that $|\{s_i, t_i\} \cap S| = 1$; S is said to *separate* terminal pair (s_i, t_i). Let $\mathcal{S}$ be the set of all Steiner cuts. For a subset $S \subseteq V$, define $\delta(S)$ as the the set of all edges in E that have exactly one endpoint in S. Given a Steiner cut $S \in \mathcal{S}$, any feasible solution F of I must contain at least one edge that *crosses* the cut S, i.e., $\sum_{e \in \delta(S)} x_e \geq 1$. This gives rise to the following *undirected cut formulation*:

$$\text{minimize} \quad \sum_{e \in E} c(e) x_e \qquad \text{(IP)}$$

$$\text{subject to} \quad \sum_{e \in \delta(S)} x_e \geq 1 \quad \forall S \in \mathcal{S} \qquad (1)$$

$$x_e \in \{0, 1\} \quad \forall e \in E. \qquad (2)$$

The dual of the linear programming relaxation of (IP) has a variable y_S for every Steiner cut $S \in \mathcal{S}$. There is a constraint for every edge $e \in E$ that requires that the total dual assigned to sets $S \in \mathcal{S}$ that contain exactly one endpoint of e is at most the cost $c(e)$ of the edge:

$$\text{maximize} \quad \sum_{S \in \mathcal{S}} y_S \qquad \text{(D)}$$

$$\text{subject to} \quad \sum_{S \in \mathcal{S}: e \in \delta(S)} y_S \leq c(e) \quad \forall e \in E \quad (3)$$

$$y_S \geq 0 \quad \forall S \in \mathcal{S}. \quad (4)$$

Algorithm AKR is based on the *primal-dual schema* (see, e.g., [22]). That is, the algorithm constructs both a feasible primal solution for (IP) and a feasible dual solution for (D). The algorithm starts with an infeasible primal solution and reduces its degree of infeasibility as it progresses. At the same time, it creates a feasible dual packing of subsets of large total value by raising dual variables of Steiner cuts.

One can think of an execution of AKR as a process over time. Let x^τ and y^τ, respectively, be the primal incidence vector and feasible dual solution at time τ. Initially, let $x_e^0 = 0$ for all $e \in E$ and $y_S^0 = 0$ for all $S \in \mathcal{S}$. Let F^τ denote the forest corresponding to the set of edges with $x_e^\tau = 1$. A tree T in F^τ is called *active* at time τ if it contains a terminal that is separated from its mate; otherwise it is *inactive*. Intuitively, AKR grows trees in F^τ that are active. At the same time, the algorithm raises dual values of Steiner cuts that correspond to active trees. If two active trees collide, they are merged. The process terminates if all trees are inactive and thus there are no unconnected terminal pairs. The interplay of the primal (growing trees) and the dual process (raising duals) is somewhat subtle and outlined next.

An edge $e \in E$ is *tight* if the corresponding constraint (3) holds with equality; a path is tight if all its edges are tight. Let H^τ be the subgraph of G that is induced by the tight edges for dual y^τ. The connected components of H^τ induce a partition $\mathcal{C}^\tau$ on the vertex set V. Let $\mathcal{S}^\tau$ be the set of all Steiner cuts contained in $\mathcal{C}^\tau$, i.e., $\mathcal{S}^\tau = \mathcal{C}^\tau \cap \mathcal{S}$. AKR raises the dual values y_S for all sets $S \in \mathcal{S}^\tau$ uniformly at all times $\tau \geq 0$. Note that y^τ is dual feasible. The algorithm maintains the invariant that F^τ is a subgraph of H^τ at all times. Consider the event that a path P between two trees T_1 and T_2 of F^τ becomes tight. The missing edges of P are then added to F^τ and the process continues. Eventually, all trees in F^τ are inactive and the process halts.

Applications

The computation of (approximate) solutions for the Steiner forest problem has various applications both in theory and practice; only a few recent developments are mentioned here.

Algorithms for more complex network design problems often rely on good approximation algorithms for the Steiner forest problem. For example, the recent approximation algorithms [6, 9, 12] for the *multi-commodity rent-or-buy problem* (MRoB) are based on the random sampling framework by Gupta et al. [12, 13]. The framework uses a Steiner forest approximation algorithm that satisfies a certain *strictness* property as a subroutine. Fleischer et al. [9] show that AKR meets this strictness requirement, which leads to the current best 5-approximation algorithm for MRoB. The strictness property also plays a crucial role in the boosted sampling framework by Gupta et al. [14] for two-stage stochastic optimization problems with recourse.

Online versions of Steiner tree and forest problems have been studied by by Awerbuch et al. [5] and Berman and Coulston [7]. In the area of algorithmic game theory, the development of *group-strategyproof cost sharing mechanisms* for network design problems such as the Steiner tree problem has recently received a lot of attention; see e.g., [16, 17, 19, 20]. An adaptation of AKR yields such a cost sharing mechanism for the Steiner forest problem [18].

Cross-References

- Generalized Steiner Network
- Steiner Trees

Recommended Reading

The interested reader is referred in particular to the articles [2, 11] for a more detailed description of primal-dual approximation algorithms for general network design problems.

1. Agrawal A, Klein P, Ravi R (1991) When trees collide: an approximation algorithm for the generalized Steiner problem on networks. In: Proceedings of the 23rd annual ACM symposium on theory of computing. Association for Computing Machinery, New York, pp 134–144
2. Agrawal A, Klein P, Ravi R (1995) When trees collide: an approximation algorithm for the generalized Steiner problem in networks. SIAM J Comput 24(3):445–456
3. Aneja YP (1980) An integer linear programming approach to the Steiner problem in graphs. Networks 10(2):167–178
4. Arora S, Lund C, Motwani R, Sudan M, Szegedy M (1998) Proof verification and the hardness of approximation problems. J ACM 45(3):501–555
5. Awerbuch B, Azar Y, Bartal Y (1996) On-line generalized Steiner problem. In: Proceedings of the 7th annual ACM-SIAM symposium on discrete algorithms, 2005. Society for Industrial and Applied Mathematics, Philadelphia, pp 68–74
6. Becchetti L, Könemann J, Leonardi S, Pál M (2005) Sharing the cost more efficiently: improved approximation for multicommodity rent-or-buy. In: Proceedings of the 16th annual ACM-SIAM symposium on discrete algorithms. Society for Industrial and Applied Mathematics, Philadelphia, pp 375–384
7. Berman P, Coulston C (1997) On-line algorithms for Steiner tree problems. In: Proceedings of the 29th annual ACM symposium on theory of computing. Association for Computing Machinery, New York, pp 344–353
8. Bern M, Plassmann P (1989) The Steiner problem with edge lengths 1 and 2. Inf Process Lett 32(4):171–176
9. Fleischer L, Könemann J, Leonardi S, Schäfer G (2006) Simple cost sharing schemes for multicommodity rent-or-buy and stochastic Steiner tree. In: Proceedings of the 38th annual ACM symposium on theory of computing. Association for Computing Machinery, New York, pp 663–670
10. Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, San Francisco
11. Goemans MX, Williamson DP (1995) A general approximation technique for constrained forest problems. SIAM J Comput 24(2):296–317
12. Gupta A, Kumar A, Pál M, Roughgarden T (2003) Approximation via cost-sharing: a simple approximation algorithm for the multicommodity rent-or-buy problem. In: Proceedings of the 44th annual IEEE symposium on foundations of computer science. IEEE Computer Society, Washington, pp 606–617
13. Gupta A, Kumar A, Pál M, Roughgarden T (2007) Approximation via cost-sharing: simpler and better approximation algorithms for network design. J ACM 54(3):Article 11
14. Gupta A, Pál M, Ravi R, Sinha A (2004) Boosted sampling: approximation algorithms for stochastic optimization. In: Proceedings of the 36th annual ACM symposium on theory of computing. Association for Computing Machinery, New York, pp 417–426
15. Jain K (2001) A factor 2 approximation for the generalized Steiner network problem. Combinatorica 21(1):39–60
16. Jain K, Vazirani VV (2001) Applications of approximation algorithms to cooperative games. In: Proceedings of the 33rd annual ACM symposium on theory of computing. Association for Computing Machinery, New York, pp 364–372
17. Kent K, Skorin-Kapov D (1996) Population monotonic cost allocation on mst's. In: Proceedings of the 6th international conference on operational research. Croatian Operational Research Society, Zagreb, pp 43–48
18. Könemann J, Leonardi S, Schäfer G (2005) A group-strategyproof mechanism for Steiner forests. In: Proceedings of the 16th annual ACM-SIAM symposium on discrete algorithms. Society for Industrial and Applied Mathematics, Philadelphia, pp 612–619
19. Megiddo N (1978) Cost allocation for Steiner trees. Networks 8(1):1–6
20. Moulin H, Shenker S (2001) Strategyproof sharing of submodular costs: budget balance versus efficiency. Econ Theory 18(3):511–533
21. Thimm M (2003) On the approximability of the Steiner tree problem. Theor Comput Sci 295(1–3):387–402
22. Vazirani VV (2001) Approximation algorithms. Springer, Berlin

Steiner Trees

Weili Wu[1,2,3] and Yaocun Huang[3]
[1]College of Computer Science and Technology, Taiyuan University of Technology, Taiyuan, Shanxi Province, China
[2]Department of Computer Science, California State University, Los Angeles, CA, USA
[3]Department of Computer Science, The University of Texas at Dallas, Richardson, TX, USA

Keywords

Approximation algorithm design

Years and Authors of Summarized Original Work

2006; Du, Graham, Pardalos, Wan, Wu, Zhao

Definition

Given a set of points, called *terminals*, in a metric space, the problem is to find the shortest tree interconnecting all terminals. There are three important metric spaces for Steiner trees, the Euclidean plane, the rectilinear plane, and the edge-weighted network. The Steiner tree problems in those metric spaces are called the *Euclidean Steiner tree (EST)*, the *rectilinear Steiner tree (RST)*, and the *network Steiner tree (NST)*, respectively. EST and RST have been found to have polynomial-time approximation schemes (PTAS) by using adaptive partition. However, for NST, there exists a positive number r such that computing r-approximation is NP-hard. So far, the best performance ratio of polynomial-time approximation for NST is achieved by k-restricted Steiner trees. However, in practice, the iterated 1-Steiner tree is used very often. Actually, the iterated 1-Steiner was proposed as a candidate of good approximation for Steiner minimum trees a long time ago. It has a very good record in computer experiments, but no correct analysis was given showing the iterated 1-Steiner tree having a performance ratio better than that of the minimum spanning tree until the recent work by Du et al. [9]. There is minimal difference in construction of the 3-restricted Steiner tree and the iterated 1-Steiner tree, which makes a big difference in analysis of those two types of trees. Why does the difficulty of analysis make so much difference? This will be explained in this article.

History and Background

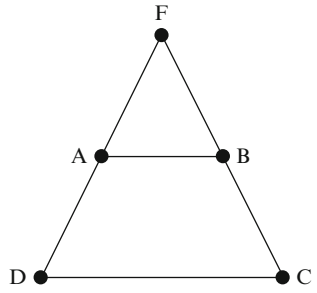
The Steiner tree problem was proposed by Gauss in 1835 as a generalization of the Fermat problem. Given three points A, B, and C in the Euclidean plane, Fermat studied the problem of finding a point S to minimize $|SA| + |SB| + |SC|$. He determined that when all three inner angles of triangle ABC are less than 120°, the optimal S should be at the position that $\angle ASB = \angle BSC = \angle CSA = 120°$.

The generalization of the Fermat problem has two directions:

1. Given n points in the Euclidean plane, find a point S to minimize the total distance from S to n given points. This is still called the Fermat problem.
2. Given n points in the Euclidean plane, find the shortest network interconnecting all given points.

Gauss found the second generalization through communication with Schumacher. On March 19, 1836, Schumacher wrote a letter to Gauss and mentioned a paradox about Fermat's problem: Consider a convex quadrilateral $ABCD$. It is known that the solution of Fermat's problem for four points A, B, C, and D is the intersection E of diagonals AC and BD. Suppose extending DA and CB can obtain an intersection F. Now, move A and B to F. Then E will also be moved to F. However, when the angle at F is less than 120°, the point F cannot be the solution of Fermat's problem for three given points F, D, and C. What happens? (Fig. 1.)

On March 21, 1836, Gauss wrote a letter replying to Schumacher in which he explained that the mistake of Schumacher's paradox occurs at the place where Fermat's problem for four points A, B, C, and D is changed to Fermat's problem for three points F, C, and D. When A and B are identical to F, the total distance from E to four points A, B, C, and D equals $2|EF| + |EC| + |ED|$, not $|EF| + |EC| + |ED|$. Thus,



Steiner Trees, Fig. 1 Convex quadrilateral $ABCD$, Fermat's problem

the point E may not be the solution of Fermat's problem for F, C, and D. More importantly, Gauss proposed a new problem. He said that it is more interesting to find the shortest network rather than a point. Gauss also presented several possible connections of the shortest network for four given points.

It was unfortunate that Gauss' letter was not seen by researchers of Steiner trees at an earlier stage. Especially, R. Courant and H. Robbins who in their popular book *What is mathematics?* (published in 1941) [6] called Gauss' problem the Steiner tree so that "Steiner tree" became a popular name for the problem.

The Steiner tree became an important research topic in mathematics and computer science due to its applications in telecommunication and computer networks. Starting with Gilbert and Pollak's work published in 1968, many publications on Steiner trees have been generated to solve various problems concerning it.

One well-known problem is the *Gilbert-Pollak conjecture* on the Steiner ratio, which is the least ratio of lengths between the Steiner minimum tree and the minimum spanning tree on the same set of given points. Gilbert and Pollak in 1968 conjectured that the Steiner ratio in the Euclidean plane is $\sqrt{3}/2$ which is achieved by three vertices of an equilateral triangle. A great deal of research effort has been put into the conjecture and it was finally proved by Du and Hwang [7].

Another important problem is called the *better approximation*. For a long time no approximation could be proved to have a performance ratio smaller than the inverse of the Steiner ratio. Zelikovsky [14] made the first breakthrough. He found a polynomial-time 11/6-approximation for NST which beats 1/2, the inverse of the Steiner ratio in the edge-weighted network. Later, Berman and Ramaiye [2] gave a polynomial-time 92/72-approximation for RST, and Du, Zhang, and Feng [8] closed the story by showing that in any metric space, there exists a polynomial-time approximation with a performance ratio better than the inverse of the Steiner ratio provided that for any set of a fixed number of points, the Steiner minimum tree is polynomial-time computable.

All the above better approximations came from the family of k-restricted Steiner trees. By improving some detail of construction, the constant performance ratio was decreasing, but the improvements were also becoming smaller. In 1996, Arora [1] made significant progress for EST and RST. He showed the existence of PTAS for EST and RST. Therefore, the theoretical researchers now pay more attention to NST. Bern and [3] showed that NST is MAX SNP-complete. This means that there exists a positive number r; computing the r-approximation for NST is NP-hard. The best-known performance for NST was given by Robin and Zelikovsky [12]. They also gave a very simple analysis to a well-known heuristic, the iterated 1-Steiner tree for pseudo-bipartite graphs.

Analysis of the iterated 1-Steiner tree is another long-standing open problem. Since Chang [4,5] proposed that the iterated 1-Steiner tree approximates the Steiner minimum tree in 1972, its performance has been claimed to be very good through computer experiments [10, 13], but no theoretical analysis supported this claim. Actually, both the k-restricted Steiner tree and the iterated 1-Steiner tree are obtained by greedy algorithms, but with different types of potential functions. For the iterated 1-Steiner tree, the potential function is non-submodular, but for the k-restricted Steiner tree, it is submodular; a property that holds for k-restricted Steiner trees may not hold for iterated 1-Steiner trees. Actually, the submodularity of potential function is very important in analysis of greedy approximations [11]. Du et al. [9] gave a correct analysis for the iterated 1-Steiner tree with a general technique to deal with non-submodular potential function.

Key Results

Consider input edge-weighted graph $G = (V, E)$ of NST. Assume that G is a complete graph and the edge weight satisfies the triangular inequality; otherwise, consider the complete graph on V with each edge (u, v) having a weight equal to the length of the shortest path between u and v in G. Given a set P of terminals, a *Steiner tree* is a

tree interconnecting all given terminals such that every leaf is a terminal.

In a Steiner tree, a terminal may have degree more than one. The Steiner tree can be decomposed, at those terminals with degree more than one, into smaller trees in which every terminal is a leaf. In such a decomposition, each resulting small tree is called a *full component*. The *size* of a full component is the number of terminals in it. A Steiner tree is *k-restricted* if every full component of it has a size at most k. The shortest k-restricted Steiner tree is also called the *k-restricted Steiner minimum tree*. Its length is denoted by $smt_k(P)$. Clearly, $smt_2(P)$ is the length of the minimum spanning tree on P, which is also denoted by $mst(P)$. Let $smt(P)$ denote the length of the Steiner minimum tree on P. If $smt_3(P)$ can be computed in polynomial time, then it is better than $mst(P)$ for an approximation of $smt(P)$. However, so far no polynomial-time approximation has been found for $smt_3(P)$. Therefore, Zelikovsky [14] used a greedy approximation of $smt_3(P)$ to approximate $smt(P)$. Actually, Chang [4, 5] used a similar greedy algorithm to compute an iterated 1-Steiner tree. Let $\mathcal{F}$ be a family of subgraphs of input edge-weighted graph G. For any connected subgraph H, denote by $mst(H)$ the length of the minimum spanning tree of H, and for any subgraph H, denote by $mst(H)$ the sum of $mst(H')$ for H' over all connected components of H. Define

$$gain(H) = mst(P) - mst(P : H) - mst(H),$$

where $mst(P : H)$ is the length of the minimum spanning tree interconnecting all unconnected terminals in P after every edge of H shrinks into a point.

Greedy Algorithm $H \leftarrow \emptyset$;
while P has not been interconnected by H **do** choose $F \in \mathcal{F}$ to maximize $gain(H \cup F)$;
output $mst(H)$.

When $\mathcal{F}$ consists of all full components of size at most three, this greedy algorithm gives the 3-restricted Steiner tree of Zelikovsky [14]. When $\mathcal{F}$ consists of all 3-stars and all edges where a 3-star is a tree with three leaves and a central vertex, this greedy algorithm produces the iterated 1-Steiner tree. An interesting fact pointed out by Du et al. [9] is that the function $gain(\cdot)$ is submodular over all full components of size at most three, but not submodular over all 3-stars and edges.

Let us consider a base set E and a function f from all subsets of E to real numbers. f is *submodular* if for any two subsets A, B of E,

$$f(A) + f(B) \geq f(A \cup B) + f(A \cap B).$$

For $x \in E$ and $A \subseteq E$, denote $\Delta_x f(A) = f(A \cup \{x\}) - f(A)$.

Lemma 1 *f is submodular if and only if for any $A \subset E$ and distinct $x, y \in E - A$,*

$$\Delta_x \Delta_y f(A) \leq 0. \tag{1}$$

Proof Suppose f is submodular. Set $B = A \cup \{x\}$ and $C = A \cup \{y\}$. Then $B \cup C = A \cup A \cup \{x, y\}$ and $B \cap C = A$. Therefore, one has

$$f(A \cup \{x, y\}) - f(A \cup \{x\}) - f(A \cup \{y\}) + f(A) \leq 0,$$

that is, (1) holds.

Conversely, suppose (1) holds for any $A \subset E$ and distinct $x, y \in E - A$. Consider two subsets A, B of E. If $A \subseteq B$ or $B \subseteq A$, it is trivial to have

$$f(A) + f(B) \geq f(A \cup B) + f(A \cap B).$$

Therefore, one may assume that $A \backslash B \neq \emptyset$ and $B \backslash A \neq \emptyset$. Write $A \backslash B = \{x_1, \ldots, x_k\}$ and $B \backslash A = \{y_1, \ldots, y_h\}$. Then

$$\begin{aligned} & f(A \cup B) - f(A) - f(B) + f(A \cap B) \\ &= \sum_{i=1}^{k} \sum_{j=1}^{h} \Delta_{x_i} \Delta_{y_j} f(A \cup \{x_1, \ldots, x_{i-1}\} \cup \{y_1, \ldots, y_{j-1}\}) \\ &\leq 0, \end{aligned}$$

where $\{x_1, \ldots x_{i-1}\} = \emptyset$ for $i = 1$ and $\{y_1, \ldots, y_{j-1}\} = \emptyset$ for $j = 1$. □

Lemma 2 *Define $f(H) = -mst(P : H)$. Then f is submodular over edge set E.*

Proof Note that for any two distinct edges x and y not in subgraph H,

$$\begin{aligned}
&\Delta_x \Delta f(H) \\
&\quad = -mst(P : H \cup x \cup y) + mst(P : H \cup x) \\
&\qquad + mst(P : H \cup y) - mst(P : H) \\
&\quad = (mst(P : H) - mst(P : H \cup x \cup y)) \\
&\qquad - (mst(P : H) - mst(P : H \cup x)) \\
&\qquad + (mst(P : H) - mst(P : H \cup y)).
\end{aligned}$$

Let T be a minimum spanning tree for unconnected terminals after every edge of H shrinks into a point. T contains a path P_x connecting two endpoints of x and also a path P_y connecting two endpoints of y. Let e_x (e_y) be a longest edge in P_x (P_y). Then

$$\begin{aligned}
mst(P : H) - mst(P : H \cup x) &= length(e_x), \\
mst(P : H) - mst(P : H \cup y) &= length(e_y).
\end{aligned}$$

$mst(P : H) - mst(P : H \cup x \cup y)$ can be computed as follows: Choose a longest edge $e\prime$ from $P_x \cup P_y$. Note that $T \cup x \cup y - e'$ contains a unique cycle Q. Choose a longest edge $e\prime\prime$ from $(P_x \cup P_y) \cap Q$. Then

$$mst(P : H) - mst(P : H \cup x \cup y) = length(e'')$$

Now, to show the submodularity of f, it suffices to prove

$$length(e_x) + length(e_y) \geq length(e'') \quad (2)$$

Case 1. $e_x P_x \cap P_y$ and $e_y P_x \cap P_y$. Without loss of generality, assume $length(e_x) \geq length(e_y)$. Then one may choose $e' = e_x$ so that $(P_x \cup P_y) \cap Q = P_y$. Hence, one can choose $e'' = e_y$. Therefore, the equality holds for (2).

Case 2. $e_x P_x \cap P_y$ and $e_y \in P_x \cap P_y$. Clearly, $length(e_x) \geq length(e_y)$. Hence, one may choose $e' = e_x$ so that $(P_x \cup P_y) \cap Q = P_y$. Hence, one can choose $e'' = e_y$. Therefore, the equality holds for (2).

Case 3. $e_x \in P_x \cap P_y$ and $e_y P_x \cap P_y$. Similar to Case 2.

Case 4. $e_x \in P_x \cap P_y$ and $e_y \in P_x \cap P_y$. In this case, $length(e_x) = length(e_y) = length(e')$. Hence, (2) holds. □

The following explains that the submodularity of $gain(\cdot)$ holds for a k-restricted Steiner tree.

Theorem 1 *Let ε be the set of all full components of a Steiner tree. Then $gain(\cdot)$ as a function on the power set of ε is submodular.*

Proof Note that for any $\mathcal{H} \subset \mathcal{E}$ and $x, y \in \mathcal{E} - \mathcal{H}$,

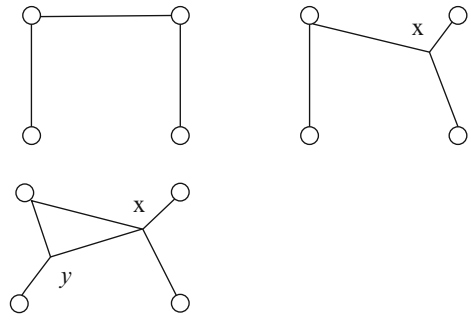
$$\Delta_x \Delta_y mst(H) = 0,$$

where $H = \cup_{z \in \mathcal{H}} z$. Thus, this theorem follows from Lemma 2.

Let $\mathcal{F}$ be the set of 3-stars and edges chosen in the greedy algorithm to produce an iterated 1-Steiner tree. Then $gain(\cdot)$ may not be submodular on $\mathcal{F}$. To see this fact, consider two 3-stars x and y in Fig. 2. Note that $gain(x \cup y) > gain(x), gain(y) \leq 0$, and $gain(\emptyset) = 0$. One has

$$gain(x \cup y) - gain(x) - gain(y) + gain(\emptyset) > 0.$$

□



Steiner Trees, Fig. 2 An example for the proof of Theorem 1

Applications

The Steiner tree problem is a classic NP-hard problem with many applications in the design of computer circuits, long-distance telephone lines, multicast routing in communication networks, etc. There exist many heuristics of the greedy type for Steiner trees in the literature. Most of them have a good performance in computer experiments, without support from theoretical analysis. The approach given in this work may apply to them.

Open Problems

It is still open whether computing the 3-restricted Steiner minimum tree is NP-hard or not. For $k \geq 4$, it is known that computing the k-restricted Steiner minimum tree is NP-hard.

Cross-References

- ▸ Greedy Approximation Algorithms
- ▸ Minimum Spanning Trees
- ▸ Rectilinear Steiner Tree

Recommended Reading

1. Arora S (1996) Polynomial-time approximation schemes for Euclidean TSP and other geometric problems. In: Proceedings of the 37th IEEE symposium on foundations of computer science, pp 2–12
2. Berman P, Ramaiyer V (1994) Improved approximations for the Steiner tree problem. J Algorithms 17:381–408
3. Bern M, Plassmann P (1989) The Steiner problem with edge lengths 1 and 2. Inf Proc Lett 32:171–176
4. Chang SK (1972) The generation of minimal trees with a Steiner topology. J ACM 19:699–711
5. Chang SK (1972) The design of network configurations with linear or piecewise linear cost functions. In: Symposium on computer-communications, networks, and teletraffic. IEEE Computer Society Press, Los Alamitos, pp 363–369
6. Crourant R, Robbins H (1941) What is mathematics? Oxford University Press, New York
7. Du DZ, Hwang FK (1990) The Steiner ratio conjecture of Gilbert-Pollak is true. Proc Natl Acad Sci USA 87:9464–9466
8. Du DZ, Zhang Y, Feng Q (1991) On better heuristic for Euclidean Steiner minimum trees. In: Proceedings of the 32nd FOCS. IEEE Computer Society Press, Los Alamitos
9. Du DZ, Graham RL, Pardalos PM, Wan PJ, Wu W, Zhao W (2008) Analysis of greedy approximations with nonsubmodular potential functions. In: Proceedings of 19th ACM-SIAM symposium on discrete algorithms (SODA). ACM, New York, pp 167–175
10. Kahng A, Robins G (1990) A new family of Steiner tree heuristics with good performance: the iterated 1-Steiner approach. In: Proceedings of IEEE international conference on computer-aided design, Santa Clara, pp 428–431
11. Wolsey LA (1982) An analysis of the greedy algorithm for the submodular set covering problem. Combinatorica 2:385–393
12. Robin G, Zelikovsky A (2000) Improved Steiner trees approximation in graphs. In: SIAM-ACM symposium on discrete algorithms (SODA), San Francisco, pp 770–779
13. Smith JM, Lee DT, Liebman JS (1981) An $O(N \log N)$ heuristic for Steiner minimal tree problems in the Euclidean metric. Networks 11:23–39
14. Zelikovsky AZ (1993) The 11/6-approximation algorithm for the Steiner problem on networks. Algorithmica 9:463–470

Stochastic Knapsack

Viswanath Nagarajan
University of Michigan, Ann Arbor, MI, USA

Keywords

Adaptive; Packing constraints; Stochastic optimization

Years and Authors of Summarized Original Work

2004; Dean, Goemans, Vondrák
2011; Bhalgat, Goel, Khanna

Problem Definition

This problem deals with packing a maximum reward set of items into a knapsack of given capacity, when the item sizes are random. The input is a collection of n items, where each item $i \in [n] := \{1, \ldots, n\}$ has reward $r_i \geq 0$ and

size $S_i \geq 0$, and a knapsack capacity $B \geq 0$. In the *stochastic knapsack* problem, all rewards are deterministic but the sizes are random. The random variables S_is are independent with known, arbitrary distributions. The actual size of an item is known only when it is placed into the knapsack. The objective is to add items sequentially (one by one) into the knapsack so as to maximize the expected reward of the items that fit into the knapsack. As usual, a subset T of items is said to fit into the knapsack if the total size $\sum_{i \in T} S_i$ is at most the knapsack capacity B.

A feasible solution (or policy) to the stochastic knapsack problem is represented by a decision tree. Nodes in this decision tree denote the current "state" of the solution (i.e., previously added items and the residual knapsack capacity) as well as the new item to place into the knapsack at this state. Branches in the decision tree denote the random size instantiations of items placed into the knapsack. Such solutions are called *adaptive policies*, to emphasize the fact that the items being placed may depend on previously observed outcomes. More formally, an adaptive policy is given by a mapping $\pi : 2^{[n]} \times [0, B] \to [n]$, where $\pi(T, C)$ denotes the next item to place into the knapsack when some subset $T \subseteq [n]$ of items has already been added, and $C = B - \sum_{i \in T} S_i$ is the residual knapsack capacity. The policy ends when the knapsack overflows (i.e., the total size of items added exceeds the knapsack capacity); we use the convention that no reward is obtained from the last overflowing item.

Notice that an arbitrary adaptive policy may require exponential space to even store. This motivates a special class of solutions, called *nonadaptive policies*. A nonadaptive policy is just specified by a fixed ordering of the n items, and the solution adds items into the knapsack in that order (irrespective of the actual size instantiations) until the knapsack overflows. Again, there is no reward obtained from the last overflowing item. While it may be easier to obtain a good nonadaptive policy, the obvious drawback is that nonadaptive policies may perform much worse than adaptive policies. The benefit of being adaptive is quantified by a measure called the *adaptivity gap*, which is the maximum ratio (over all instances) of the expected reward of an optimal adaptive policy to the expected reward of an optimal nonadaptive policy.

In both the adaptive and nonadaptive settings, the stochastic knapsack problem is at least NP-hard, since it generalizes the deterministic knapsack problem. Moreover, certain questions regarding adaptive policies are PSPACE-hard [4].

Notation We assume that the item size distributions are given explicitly. For any item $i \in [n]$ define its effective reward $w_i = r_i \cdot \Pr[S_i \leq B]$ and its mean truncated size $\mu_i = \mathbb{E}[\min\{S_i, B\}]$. Note that the expected reward obtained by placing the single item i into the knapsack is exactly w_i.

Key Results

Dean, Goemans, and Vondrák introduced the stochastic knapsack problem and the notion of adaptivity gaps. They proved the following.

Theorem 1 ([4]) *There is a polynomial time algorithm for the stochastic knapsack problem that computes a nonadaptive policy having expected reward at least $\frac{1}{4}$ that of an optimal adaptive policy.*

As a consequence, the adaptivity gap of the stochastic knapsack problem is also upper bounded by four. Dean, Goemans, and Vondrák [4] also showed an instance of stochastic knapsack that lower bounds the adaptivity gap by $\frac{5}{4}$.

The algorithm in Theorem 1 uses a natural greedy approach. It outputs the better of the following two nonadaptive policies:

- Place the single item $i^* = \arg\max_{i \in [n]} w_i$.
- Place items in nonincreasing order of w_i / μ_i.

In terms of adaptive policies, Bhalgat, Goel, and Khanna proved the following.

Theorem 2 ([2, 3]) *For any constant $\epsilon > 0$, there is polynomial time algorithm for the stochastic knapsack problem that computes an*

adaptive policy having expected reward at least $\frac{1}{2+\epsilon}$ *that of an optimal adaptive policy.*

The algorithm in Theorem 2 relies on an intricate transformation of general size distributions to certain canonical distributions and an algorithm for computing a near-optimal adaptive policy under canonical size distributions.

Extensions

Several variants of the stochastic knapsack problem have been studied, and good algorithms have been obtained for them.

Correlated Stochastic Knapsack

This is a generalization of the stochastic knapsack problem, where each item's reward is also random and possibly correlated with its size. The distributions across items are still independent: so the correlations are only between the size and reward of a single item. Gupta, Krishnaswamy, Molinaro, and Ravi [6] gave an algorithm for this problem that computes a nonadaptive policy having expected reward within factor 8 of the optimal adaptive policy. Recently, Ma [8] gave an algorithm that for any constant $\epsilon > 0$ computes an adaptive policy having expected reward within factor $2 + \epsilon$ of the optimal adaptive policy; this algorithm requires item sizes and the capacity B to be specified in unary.

Budgeted Multi-armed Bandit

The input to this problem consists of a bound B and n "arms" (each arm is a Markov chain with rewards at its states and a specified starting state). A feasible policy consists of B steps. In each step, the policy can select one arm $i \in [n]$: upon selecting arm i, it gets the reward at the current state of arm i and the arm transitions to its next state according to its Markov chain. The objective is to maximize the expected total reward over B steps of the policy. Again, we are interested in adaptive policies, whose actions may depend on past outcomes. Guha and Munagala [5] introduced this problem and gave a $(2 + \epsilon)$-approximation algorithm, under the assumption that the rewards of each arm satisfy a "Martingale" condition (which is natural in many settings). Gupta, Krishnaswamy, Molinaro, and Ravi [6] gave the first constant-factor approximation algorithm for this problem without the Martingale reward assumption. The constant factor in the latter result was improved to 6.75 by Ma [8].

Stochastic Orienteering

This problem is defined on a finite metric space (V, d) with vertex set V and distance function $d : V \times V \to \mathbb{R}_+$ that satisfies (i) symmetry $d(u, v) = d(v, u)$ for all $u, v \in V$ and (ii) triangle inequality $d(u, w) \leq d(u, v) + d(v, w)$ for all $u, v, w \in V$. The distances between vertices denote travel times. Each vertex $i \in V$ corresponds to a job having deterministic reward $r_i \geq 0$ and random processing time $S_i \geq 0$. The random variables S_is are independent with known, arbitrary distributions. Given a start-vertex $\rho \in V$ and bound B, the goal is to compute a policy, which describes a (possibly adaptive) path originating from ρ that visits vertices and runs the respective jobs. The actual processing time of a job is known only when it completes. The policy ends when the total time (for travel plus processing) exceeds B. The objective is to maximize the expected total reward; there is no reward obtained from a partially completed job (which may occur at the end of the policy). As before, an optimal policy may be adaptive and choose the next job to run based on previously observed outcomes. Gupta, Krishnaswamy, Nagarajan, and Ravi [7] gave an $O(\log \log B)$-approximation algorithm for the stochastic orienteering problem; this result requires the bound B, distances, and processing times to be integer valued. As a corollary, [7] also upper bounded the adaptivity gap by $O(\log \log B)$. Recently, Bansal and Nagarajan [1] gave an $\Omega\left(\sqrt{\log \log B}\right)$ lower bound on the adaptivity gap.

Applications

The stochastic knapsack problem and its variants model various applications in advertising, logistics, medical diagnosis, and robotics.

Open Problems

It is not known if the stochastic knapsack problem is any harder to approximate than the usual (deterministic) knapsack problem. In particular, is there a PTAS for stochastic knapsack? Determining a tight bound on its adaptivity gap is also an interesting open question.

Recommended Reading

1. Bansal N, Nagarajan V (2014) On the adaptivity gap of stochastic orienteering. In: IPCO, Bonn, pp 114–125
2. Bhalgat A (2011) A $(2+\epsilon)$-approximation algorithm for the stochastic knapsack problem. Unpublished manuscript
3. Bhalgat A, Goel A, Khanna S (2011) Improved approximation results for stochastic knapsack problems. In: SODA, San Francisco, pp 1647–1665
4. Dean BC, Goemans, MX, Vondrák J (2008) Approximating the stochastic knapsack problem: the benefit of adaptivity. Math Oper Res 33(4):945–964
5. Guha S, Munagala K (2013) Approximation algorithms for Bayesian multi-armed bandit problems. CoRR abs/1306.3525
6. Gupta A, Krishnaswamy R, Molinaro M, Ravi R (2011) Approximation algorithms for correlated knapsacks and non-martingale bandits. In: FOCS, Palm Springs, pp 827–836
7. Gupta A, Krishnaswamy R, Nagarajan V, Ravi R (2012) Approximation algorithms for stochastic orienteering. In: SODA, Kyoto, pp 1522–1538
8. Ma W (2014) Improvements and generalizations of stochastic knapsack and multi-armed bandit approximation algorithms: extended abstract. In: SODA, Portland, pp 1154–1163

Stochastic Scheduling

Jay Sethuraman
Industrial Engineering and Operations Research, Columbia University, New York, NY, USA

Keywords

Queueing; Sequencing

Years and Authors of Summarized Original Work

2001; Glazebrook, Nino-Mora

Problem Definition

Scheduling is concerned with the allocation of scarce resources (such as machines or servers) to competing activities (such as jobs or customers) over time. The distinguishing feature of a *stochastic* scheduling problem is that some of the relevant data are modeled as *random variables*, whose distributions are known, but whose actual realizations are not. Stochastic scheduling problems inherit several characteristics of their deterministic counterparts. In particular, there are virtually an unlimited number of problem types depending on the machine environment (single machine, parallel machines, job shops, flow shops), processing characteristics (preemptive versus nonpreemptive, batch scheduling versus allowing jobs to arrive "over time," due dates, deadlines), and objectives (makespan, weighted completion time, weighted flow time, weighted tardiness). Furthermore, stochastic scheduling models have some new, interesting features (or difficulties!):

- The scheduler may be able to make inferences about the remaining processing time of a job by using information about its elapsed processing time; whether the scheduler is allowed to make use of this information or not is a question for the modeler.
- Many scheduling algorithms make decisions by comparing the processing times of jobs. If jobs have deterministic processing times, this poses no problems as there is only one way to compare them. If the processing times are random variables, comparing processing times is a subtle issue. There are many ways to compare pairs of random variables, and some are only *partial* orders. Thus, any algorithm that operates by comparing processing times must now specify the particular ordering used to

compare random variables (and to determine what to do if two random variables are not comparable under the specified ordering).

These considerations lead to the notion of a scheduling *policy*, which specifies how the scarce resources have to be allocated to the competing activities as a function of the *state* of the system at any point in time. The state of the system includes information such as prior job completions, the elapsed time of jobs currently in service, the realizations of the random release dates and due dates (if any), and any other information that can be inferred based on the history observed so far. A policy that is allowed to make use of all this information is said to be *dynamic*, whereas a policy that is not allowed to use any state information is *static*.

Given any policy, the objective function for a stochastic scheduling model operating under that policy is typically a random variable. Thus, comparison of two policies entails the comparison of the associated random variables, so the *sense* in which these random variables are compared must be specified. A common approach is to find a solution that optimizes the *expected value* of the objective function (which has the advantage that it is a total ordering); less commonly, other orderings such as the stochastic ordering or the likelihood ratio ordering are used.

Key Results

Consider a single machine that processes n jobs, with the (random) processing time of job i given by a distribution $F_i(\cdot)$ whose mean is p_i. The Weighted Shortest Expected Processing Time first (WSEPT) rule sequences the jobs in decreasing order of w_i / p_i. Smith [13] proved that the WSEPT rule minimizes the sum of weighted completion times when the processing times are deterministic. Rothkopf [11] generalized this result and proved the following:

Theorem 1 *The WSEPT rule minimizes the expected sum of the weighted completion times in the class of all nonpreemptive dynamic policies (and hence also in the class of all nonpreemptive static policies).*

If preemption is allowed, the WSEPT rule is not optimal. Nevertheless, Sevcik [12] showed how to assign an "index" to each job at each point in time such that scheduling a job with the largest index at each point in time is optimal. Such policies are called index policies and have been investigated extensively because they are (relatively) simple to implement and analyze. Often, the optimality of index policies can be proved under some assumptions on the processing time distributions. For instance, Weber, Varaiya, and Walrand [14] proved the following result for scheduling n jobs on m identical parallel machines:

Theorem 2 *The SEPT rule minimizes the expected sum of completion times in the class of all nonpreemptive dynamic polices, if the processing time distributions of the jobs are stochastically ordered.*

For the same problem but with the makespan objective, Bruno, Downey, and Frederickson [3] proved the optimality of the Longest Expected Processing Time first rule provided all the jobs have exponentially distributed processing times.

One of the most significant achievements in stochastic scheduling is the proof of optimality of index policies for the multiarmed bandit problem and its many variants, due originally to Gittins and Jones [5, 6]. In an instance of the bandit problem, there are N projects, each of which is in any one of a possibly finite number of states. At each (discrete) time, any one of the projects can be attempted, resulting in a random reward; the attempted project undergoes a (Markovian) state transition, whereas the other projects remain frozen and do not change state. The goal of the decision maker is to determine an optimal way to attempt the projects so as to maximize the total discounted reward. Of course one can solve this problem as a large, stochastic dynamic program, but such an approach does not reveal any structure and is moreover computationally impractical

except for very small problems. (Also, if the state space of any project is countable or infinite, it is not clear how one can solve the resulting DP exactly!) The remarkable result of Gittins and Jones [6] is the optimality of index policies: to each state of each project, one can associate an index so that attempting a project with the largest index at any point in time is optimal. The original proof of Gittins and Jones [6] has subsequently been simplified by many authors; moreover, several alternative proofs based on different techniques have appeared, leading to a much better understanding of the class of problems for which index policies are optimal [2, 4, 5, 10, 17].

While index policies are easy to implement and analyze, they are often not optimal in many problems. It is therefore natural to investigate the gap between an optimal index policy (or a natural heuristic) and an optimal policy. For example, the WSEPT rule is a natural heuristic for the problem of scheduling jobs on identical parallel machines to minimize the expected sum of the weighted completion times. However, the WSEPT rule is not necessarily optimal. Weiss [16] showed that, under mild and reasonable assumptions, the expected number of times that the WSEPT rule differs from the optimal decision is bounded above by a *constant*, independent of the number of jobs. Thus, the WSEPT rule is asymptotically optimal. As another example of a similar result, Whittle [18] generalized the multiarmed bandit model to allow for state transitions in projects that are *not* activated, giving rise to the "restless bandit" model. For this model, Whittle [18] proposed an index policy whose asymptotic optimality was established by Weber and Weiss [15].

A number of stochastic scheduling models allow for jobs to arrive over time according to a stochastic process. A commonly used model in this setting is that of a multiclass queueing network. Multiclass queueing networks serve as useful models for problems in which several types of *activities* compete for a limited number of shared *resources*. They generalize deterministic job-shop problems in two ways: jobs arrive *over time* and each job has a *random* processing time at each stage. The optimal control problem in a multiclass queueing network is to find an optimal allocation of the available resources to activities over time. Not surprisingly, index policies are optimal only for restricted versions of this general model. An important example is scheduling a multiclass single-server system with feedback: there are N types of jobs; type i jobs arrive according to a Poisson process with rate λ_i, require service according to a service-time distribution $F_i(\cdot)$ with mean processing time s_i, and incur holding costs at rate c_i per unit time. A type i job after undergoing processing becomes a type j job with probability p_{ij} or exits the system with probability $1-\sum_j p_{ij}$ isn't in document. The objective is to find a scheduling policy that minimizes the expected holding cost rate in steady state. Klimov [9] proved the optimality of index policies for this model, as well as for the objective in which the total discounted holding cost is to be minimized. While the optimality result does not hold when there are many parallel machines, Glazebrook and Niño-Mora [7] showed that this rule is asymptotically optimal. For more general models, the prevailing approach is to use approximations such as fluid approximations [1] or diffusion approximations [8].

Applications

Stochastic scheduling models are applicable in many settings, most prominently in computer and communication networks, call centers, logistics and transportation, and manufacturing systems [4, 10].

Cross-References

- List Scheduling
- Minimum Weighted Completion Time

Recommended Reading

1. Avram F, Bertsimas D, Ricard M (1995) Fluid models of sequencing problems in open queueing networks: an optimal control approach. In: Kelly FP, Williams RJ (eds) Stochastic networks. Proceedings

of the international mathematics association, vol 71. Springer, New York, pp 199–234
2. Bertsimas D, Niño-Mora J (1996) Conservation laws, extended polymatroids and multiarmed bandit problems: polyhedral approaches to indexable systems. Math Oper Res 21(2):257–306
3. Bruno J, Downey P, Frederickson GN (1981) Sequencing tasks with exponential service times to minimize the expected flow time or makespan. J ACM 28:100–113
4. Dacre M, Glazebrook K, Nino-Mora J (1999) The achievable region approach to the optimal control of stochastic systems. J R Stat Soc Ser B 61(4):747–791
5. Gittins JC (1979) Bandit processes and dynamic allocation indices. J R Stat Soc Ser B 41(2):148–177
6. Gittins JC, Jones DM (1974) A dynamic allocation index for the sequential design experiments. In: Gani J, Sarkadu K, Vince I (eds) Progress in statistics. European meeting of statisticians I. North Holland, Amsterdam, pp 241–266
7. Glazebrook K, Niño-Mora J (2001) Parallel scheduling of multiclass M/M/m queues: approximate and heavy-traffic optimization of achievable performance. Oper Res 49(4):609–623
8. Harrison JM (1988) Brownian models of queueing networks with heterogenous customer populations. In: Fleming W, Lions PL (eds) Stochastic differential systems, stochastic control theory and applications. Proceedings of the international mathematics association. Springer, New York, pp 147–186
9. Klimov GP (1974) Time-sharing service systems I. Theory Probab Appl 19:532–551
10. Pinedo M (2002) Scheduling: theory, algorithms and systems, 2nd edn. Prentice Hall, Englewood Cliffs
11. Rothkopf M (1966) Scheduling with random service times. Manag Sci 12:707–713
12. Sevcik KC (1974) Scheduling for minimum total loss using service time distributions. J ACM 21:66–75
13. Smith WE (1956) Various optimizers for single-stage production. Nav Res Logist Q 3:59–66
14. Weber RR, Varaiya P, Walrand J (1986) Scheduling jobs with stochastically ordered processing times on parallel machines to minimize expected flow time. J Appl Probab 23:841–847
15. Weber RR, Weiss G (1990) On an index policy for restless bandits. J Appl Probab 27:637–648
16. Weiss G (1992) Turnpike optimality of Smith's rule in parallel machine stochastic scheduling. Math Oper Res 17:255–270
17. Whittle P (1980) Multiarmed bandit and the Gittins index. J R Stat Soc Ser B 42:143–149
18. Whittle P (1988) Restless bandits: activity allocation in a changing world. In: Gani J (ed) A celebration of applied probability. Journal of applied probability, vol 25A. Applied Probability Trust, Sheffield, pp 287–298

String Matching

Maxime Crochemore[1,2,3] and Thierry Lecroq[4]
[1]Department of Computer Science, King's College London, London, UK
[2]Laboratory of Computer Science, University of Paris-East, Paris, France
[3]Université de Marne-la-Vallée, Champs-sur-Marne, France
[4]Computer Science Department and LITIS Faculty of Science, Université de Rouen, Rouen, France

Keywords

Border; Pattern matching; Period; Prefix; Shift function; String matching; Suffix

Years and Authors of Summarized Original Work

1977; Knuth, Morris, Pratt
1977; Boyer, Moore
1994; Crochemore, Czumaj, Gąsieniec, Jarominek, Lecroq, Plandowski, Rytter

Problem Definition

Given a *pattern string* $P = p_1 p_2 \dots p_m$ and a *text string* $T = t_1 t_2 \dots t_n$, both being sequences over an alphabet Σ of size σ, the *exact string-matching (ESM)* problem is to find one or, more generally, all the text positions where P occurs in T, that is, compute the set $\{j \mid 1 \le j \le n-m+1 \text{ and } P = t_j t_{j+1} \dots t_{j+m-1}\}$.

Both worst- and average-case complexities are considered. For the latter one assumes that pattern and text are randomly generated by choosing each character uniformly and independently from Σ. For simplicity and practicality the assumption $m = o(n)$ is made in this entry.

Key Results

Most algorithms that solve the ESM problem proceed in two steps: a preprocessing phase of the pattern P followed by a searching phase over the

text T. The preprocessing phase serves to collect information on the pattern in order to speed up the searching phase.

The searching phase of string-matching algorithms works as follows: they first align the left ends of the pattern and the text, then compare the aligned symbols of the text and the pattern – this specific work is called an attempt or a scan – and after a whole match of the pattern or after a mismatch, they shift the pattern to the right. They repeat the same procedure again until the right end of the pattern goes beyond the right end of the text. The scanning part can be viewed as operating on the text through a window, which size is most often the length of the pattern. This processing manner is called the scan and shift mechanism. Different scanning strategies of the window lead to algorithms having specific properties and advantages.

The brute force algorithm for the ESM problem consists in checking if P occurs at each position j on T, with $1 \leq j \leq n - m + 1$. It does not need any preprocessing phase. It runs in quadratic time $O(mn)$ with constant extra space and performs $O(n)$ character comparisons on average. This is to be compared with the following bounds.

Theorem 1 (Cole et al. [6]) *The minimum number of character comparisons to solve the ESM problem in the worst case is* $n + \frac{9}{4m}(n - m)$, *and there exists an algorithm performing at most* $n + \frac{8}{3(m+1)}(n-m)$ *character comparisons in the worst case.*

Theorem 2 (Yao [26]) *The ESM problem needs* $\Omega\left(\frac{\log_\sigma m}{m} \times n\right)$ *time in expectation.*

Online Text Parsing

The first linear ESM algorithm appears in the 1970s. The preprocessing phase consists in computing the periods of the pattern prefixes, or equivalently the length of the longest border for all the prefixes of the pattern. A border of a string is both a prefix and a suffix of it distinct from the string itself. Let $next[i]$ be the length of the longest border of $p_1 \dots p_{i-1}$. Consider an attempt at position j, when the pattern $p_1 \dots p_m$ is aligned with the segment $t_j \dots t_{j+m-1}$ of the text. Assume that the first mismatch (during a left to right scan) occurs between symbols p_i and t_{i+j} for $1 \leq i \leq m$. Then, $p_1 \dots p_{i-1} = t_j \dots t_{i+j-1} = u$ and $a = p_i \neq t_{i+j} = b$. A prefix v of the pattern may match a suffix of the portion u of the text. By the definition of table *next*, a shift that aligns $p_{next[i]}$ with t_{i+j} cannot miss any occurrence of P in T, and thus backtracking in the text is not necessary. There exist two variants [18, 19], depending on whether $p_{next[i]}$ has to be different from p_i or not. The second is slightly more efficient.

Theorem 3 (Knuth, Morris, and Pratt [18]) *The text searching can be done in time* $O(n)$ *and space* $O(m)$. *Preprocessing the pattern can be done in time* $O(m)$.

The search can also be realized using an implementation with successor by default of the deterministic automaton $\mathcal{D}(P)$ recognizing the language $\Sigma^* P$. The size of the implementation is $O(m)$ independent of the alphabet size, due to the fact that $\mathcal{D}(P)$ possesses $m + 1$ states, m forward arcs, and at most m backward arcs. Using the automaton for searching a text leads to an algorithm having an efficient delay (maximum time for processing a character of the text).

Theorem 4 (Hancart [15]) *Searching for the pattern* P *can be done with a delay of* $O(min\{\sigma, \log_2 m)\})$ *letter comparisons.*

Note that for most algorithms the pattern preprocessing is not necessarily done before the text parsing, as it can be performed on the fly during the parsing.

Algorithms Sublinear on the Average

The Boyer-Moore algorithm [3] is among the most efficient ESM algorithms. A simplified version of it, or the entire algorithm, is often implemented in text editors for the search and substitute commands.

The algorithm scans the characters of the window from right to left beginning with its rightmost symbol. In case of a mismatch (or a complete match of the pattern), it uses two precomputed functions to shift the pattern to the right.

These two shift functions are called the *bad-character shift* and the *good-suffix shift*. They are based on the following observations. Assume that a mismatch occurs between character $p_i = a$ of the pattern and character $t_{i+j} = b$ of the text during an attempt at position j. Then, $p_{i+1} \dots p_m = t_{i+j+1} \dots t_{j+m} = u$ and $p_i \neq t_{i+j}$. The good-suffix shift consists in aligning the segment $t_{i+j+1} \dots t_{j+m}$ with its rightmost occurrence in P that is preceded by a character different from p_i. Another variant called the *best-suffix shift* consists in aligning the segment $t_{i+j} \dots t_{j+m}$ with its rightmost occurrence in P. Both variants can be computed in time and space $O(m)$ independent of the alphabet size. If there exists no such segment, the shift consists in aligning the longest suffix v of $t_{i+j+1} \dots t_{j+m}$ with a matching prefix of x. The bad-character shift consists in aligning the text character t_{i+j} with its rightmost occurrence in $p_1 \dots p_{m-1}$. If t_{i+j} does not appear in the pattern, no occurrence of P in T can overlap the symbol t_{i+j}, then the left end of the pattern is aligned with the character at position $i + j + 1$. The search can then be done in $O(n/m)$ in the best case.

Theorem 5 (Cole [5]) *During the search for a nonperiodic pattern P of length m (such that the length of the longest border of P is less than $m/2$) in a text T of length n, the Boyer-Moore algorithm performs at most $3n$ comparisons between letters of P and of T.*

In practice, when scanning the window from right to left during an attempt, it is sometimes more efficient to only use the bad-character shift. This was first done by the Horspool algorithm [16]. Other practical efficient algorithms are the Quick Search by Sunday [24] and the Tuned Boyer-Moore by Hume and Sunday [17].

Yao's bound can be reached using an indexing structure giving access to all the factors of the reverse pattern. This is done by the Reverse Factor algorithm also called BDM (for Backward Dawg Matching).

Theorem 6 (Crochemore et al. [9]) *The search can be done in optimal expected time $O\left(\frac{\log_\sigma m}{m} \times n\right)$ using the suffix automaton or the suffix tree of the reverse pattern.*

A factor oracle can be used instead of an index structure. A factor oracle is an automaton simpler than the suffix automaton that may recognize some additional strings of length smaller than m. The only string of length m accepted by the factor oracle of a string w of length m is w itself. Then it can be used for solving the ESM problem. This is done by the Backward Oracle Matching (BOM) algorithm of Allauzen, Crochemore, and Raffinot [1]. Its behavior in practice is similar to the one of the BDM algorithm.

Time-Space Optimal Algorithms

Algorithms of this type run in linear time (for both preprocessing and searching) and need only constant space in addition to the inputs.

Theorem 7 (Galil and Seiferas [13]) *The search can be done optimally in time $O(n)$ and constant extra space.*

After Galil and Seiferas' first solution, other solutions are by Crochemore-Perrin [8] and Rytter [22]. These algorithms rely on a partition of the pattern in two parts; they first search for the right part of the pattern from left to right, and then, if no mismatch occurs, they search for the left part. The partition can be the perfect factorization [13], the critical factorization [8], or based on the lexicographically maximum suffix of the pattern [22]. Another solution by Crochemore [7] is a variant of KMP [18]: it computes lower bounds of pattern prefixes periods on the fly and requires no preprocessing.

Bit-Parallel Solution

It is possible to use the bit-parallelism technique for ESM.

Theorem 8 (Baeza-Yates and Gonnet [2]; Wu and Manber [25]) *If the length m of the string P is smaller than the number of bits of a machine word, the preprocessing phase can be done in time and space $O(\sigma)$ and the searching phase in time $O(n)$.*

It is even possible to use this bit-parallelism technique to simulate the BDM algorithm. This is realized by the BNDM (Backward Nondeterministic Dawg Matching) algorithm [20].

There exists another method that uses the bit-parallelism technique that is optimal on the average. It considers sparse q-grams and thus avoids to scan a lot of text positions. It is due to Fredriksson and Grabowski [12].

Applications

The methods that are described here apply to the treatment of the natural language, of genetic and musical sequences, the problems of safety related to data flows like virus detection, and the management of the textual databases, to quote only some immediate applications.

Open Problems

There remain only a few open problems on this question. It is still unknown if it is possible to design an average optimal time constant space string-matching algorithm. The exact size of the Boyer-Moore automaton is still unknown [3]. The Boyer-Moore automaton was first introduced by Knuth [18]. Its states encode all the possible situations when searching the pattern with the Boyer-Moore algorithm and remember every text character already matched in the window.

Experimental Results

The book of G. Navarro and M. Raffinot [21] is a good introduction and presents an experimental map of ESM algorithms for different alphabet sizes and pattern lengths. Basically, the Shift-Or algorithm is efficient for small alphabets and short patterns, the BNDM algorithm is efficient for medium-sized alphabets and medium-length patterns, the Horspool algorithm is efficient for large alphabets, and the BOM algorithm is efficient for long patterns. The article of S. Faro and T. Lecroq [11] updates the experimental map with the most recent results.

URLs to Code and Data Sets

The site monge.univ-mlv.fr/~lecroq/string presents a large number of ESM algorithms (see also [4]). Each algorithm is implemented in C code and a Java applet is given. The site www.dmi.unict.it/~faro/smart presents SMART, a string-matching research tool, which contains the C code of a great number of exact string-matching algorithms and some corpora (natural language, musical, biological, and random texts). The user can easily plug its own algorithm to compare it against some selected algorithms.

Cross-References

- Approximate String Matching is the version where errors are permitted;
- Multiple String Matching is the version where a finite set of patterns is searched for in a text;
- Regular Expression Matching is the more complex case where P can be a regular expression;
- Suffix Trees and Arrays refers to the case where the text is preprocessed.

Further information can be found in the three following books: [10, 14] and [23].

Recommended Reading

1. Allauzen C, Crochemore M, Raffinot M (1999) Factor oracle: a new structure for pattern matching. In: SOFSEM'99, Milovy. LNCS, vol 1725, pp 291–306
2. Baeza-Yates RA, Gonnet GH (1992) A new approach to text searching. C ACM 35(10):74–82
3. Boyer RS, Moore JS (1977) A fast string searching algorithm. C ACM 20(10):762–772
4. Charras C, Lecroq T (2004) Handbook of exact string matching algorithms. King's College, London
5. Cole R (1994) Tight bounds on the complexity of the Boyer-Moore string matching algorithm. SIAM J Comput 23(5):1075–1091
6. Cole R, Hariharan R, Paterson M, Zwick U (1995) Tighter lower bounds on the exact complexity of string matching. SIAM J Comput 24(1):30–45

7. Crochemore M (1992) String-matching on ordered alphabets. Theor Comput Sci 92(1):33–47
8. Crochemore M, Perrin D (1991) Two-way string matching. J ACM 38(3):651–675
9. Crochemore M, Czumaj A, Gąsieniec L, Jarominek S, Lecroq T, Plandowski W, Rytter W (1994) Speeding up two string matching algorithms. Algorithmica 12(4/5):247–267
10. Crochemore M, Hancart C, Lecroq T (2007) Algorithms on strings. Cambridge University Press, New York
11. Faro S, Lecroq T (2013) The exact online string matching problem: a review of the most recent results. C ACM, Harlow, 45(2):13
12. Fredriksson K, Grabowski S (2005) Practical and optimal string matching. In: Proceedings of SPIRE'2005, Buenos Aires. LNCS, vol 3772, pp 374–385
13. Galil Z, Seiferas J (1983) Time-space optimal string matching. J Comput Syst Sci 26(3):280–294
14. Gusfield D (1997) Algorithms on strings, trees and sequences. Cambridge University Press, New York
15. Hancart C (1993) On Simon's string searching algorithm. Inf Process Lett 47(2):95–99
16. Horspool RN (1980) Practical fast searching in strings. Softw Pract Exp 10(6):501–506
17. Hume A, Sunday DM (1991) Fast string searching. Softw Pract Exp 21(11):1221–1248
18. Knuth DE, Morris JH Jr, Pratt VR (1977) Fast pattern matching in strings. SIAM J Comput 6(1):323–350
19. Morris JH Jr, Pratt VR (1970) A linear pattern-matching algorithm. Report 40, University of California, Berkeley
20. Navarro G, Raffinot M (1998) A bit-parallel approach to suffix automata: fast extended string matching. In: Farach-Colton M (ed) Proceedings of the 9th annual symposium on combinatorial pattern matching, Piscataway. Lecture notes in computer science, vol 1448. Springer, Berlin, Piscataway, New Jersey, USA, pp 14–33
21. Navarro G, Raffinot M (2002) Flexible pattern matching in strings – practical on-line search algorithms for texts and biological sequences. Cambridge University Press, Cambridge
22. Rytter W (2003) On maximal suffixes and constant-space linear-time versions of KMP algorithm. Theor Comput Sci 299(1–3):763–774
23. Smyth WF (2002) Computing patterns in strings. Addison Wesley Longman, Harlow
24. Sunday DM (1990) A very fast substring search algorithm. C ACM 33(8):132–142
25. Wu S, Manber U (1992) Fast text searching allowing errors. C ACM 35(10):83–91
26. Yao A (1979) The complexity of pattern matching for a random string. SIAM J Comput 8:368–387

String Sorting

Rolf Fagerberg
Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark

Keywords

Sorting of multidimensional keys; Vector sorting

Years and Authors of Summarized Original Work

1997; Bentley, Sedgewick

Problem Definition

The problem is to sort a set of strings into lexicographical order. More formally: A *string* over an *alphabet* Σ is a finite sequence $x_1x_2x_3\ldots x_k$ where $x_i \in \Sigma$ for $i = 1,\ldots,k$. The x_is are called the *characters* of the string, and k is the *length* of the string. If the alphabet Σ is ordered, the *lexicographical order* on the set of strings over Σ is defined by declaring a string $x = x_1x_2x_3\ldots x_k$ smaller than a string $y = y_1y_2y_3\ldots y_l$ if either there exists a $j \geq 1$ such that $x_i = y_i$ for $1 \leq i < j$ and $x_j < y_j$ or if $k < l$ and $x_i = y_i$ for $1 \leq i \leq k$. Given a set S of strings over some ordered alphabet, the problem is to sort S according to lexicographical order.

The input to the string sorting problem consists of an array of pointers to the strings to be sorted. The output is a permutation of the array of pointers, such that traversing the array will point to the strings in nondecreasing lexicographical order.

The complexity of string sorting depends on the alphabet as well as the machine model. The main solution [15] described in this entry works for alphabets of unbounded size (i. e., comparisons are the only operations on characters of Σ) and can be implemented on a pointer

machine. See below for more information on the asymptotic complexity of string sorting in various settings.

Key Results

This section is structured as follows: first, the key result appearing in the title of this entry [15] is described; then an overview of other relevant results in the area of string sorting is given.

The string sorting algorithm proposed by Bentley and Sedgewick in 1997 [15] is called three-way radix quicksort [5]. It works for unbounded alphabets, for which it achieves optimal performance.

Theorem 1 *The algorithm three-way radix quicksort sorts K strings of total length N in time $O(K \log K + N)$.*

This time complexity is optimal, which follows by considering strings of the form `bbb...bx`, where all xs are different: Sorting the strings can be no faster than sorting the xs, and all `b`s must be read (else an adversary could change one unread `b` to `a` or `c`, making the returned order incorrect). A more precise version of the bounds above (upper as well as lower) is $K \log K + D$, where D is the sum of the lengths of the *distinguishing prefixes* of the strings. The distinguishing prefix d_s of a string s in a set S is the shortest prefix of s which is not a prefix of another string in S (or is s itself, if s is a prefix of another string). Clearly, $K \leq D \leq N$.

The three-way radix quicksort of Bentley and Sedgewick is not the first algorithm to achieve this complexity; however, it is a very simple and elegant way of doing it. As demonstrated in [3, 15], it is also very fast in practice. Although various elements of the algorithm had been noted earlier, their practical usefulness for string sorting was overlooked until the work in [15].

Three-way radix quicksort is shown in pseudocode in Fig. 1 (adapted from [5]), where S is a list of strings to be sorted and d is an integer. To sort S, an initial call SORT(S, 1) is made. The value s_d denotes the dth character of the string s, and + denotes concatenation. The presentation in Fig. 1 assumes that all strings end in a special end-of-string (EOS) character (such as the null character in C). In an actual implementation, S will be an array of pointers to strings, and the sort will be in-place (using an in-place method from standard quicksort for three-way partitioning of the array into segments holding $S_<$, $S_=$, and $S_>$), rendering concatenation superfluous.

```
SORT(S, d)
  IF |S| ≤ 1:
    RETURN
  Choose a partitioning character v ∈ {s_d | s ∈ S}
  S< = {s ∈ S | s_d < v}
  S= = {s ∈ S | s_d = v}
  S> = {s ∈ S | s_d > v}
  SORT(S<, d)
  IF v ≠ EOS:
    SORT(S=, d + 1)
  SORT(S>, d)
  S = S< + S= + S>
```

String Sorting, Fig. 1 Three-way radix quicksort (assuming each string ends in a special EOS character)

Correctness follows from the following invariant being maintained by the algorithm: At the start of a call SORT(S, d), all strings in S agree on the first $d - 1$ characters.

Time complexity depends on how the partitioning character v is chosen. One particular choice is the median of all the dth characters (including doublets) of the strings in S. Partitioning and median finding can be done in time $O(|S|)$, which is $O(1)$ time per string partitioned. Hence, the total running time of the algorithm is the sum over all strings of the number of partitionings they take part in. For each string, let a partitioning be of type I if the string ends up in $S_<$ or $S_>$ and of type II if it ends up in $S_=$. For a string s, type II can only occur $|d_s|$ times and type I can only occur $\log K$ times. Hence, the running time is $O(K \log K + D)$.

Like for standard quicksort, median finding impairs the constant factors of the algorithm, and more practical choices of partitioning character include selecting a random element among all the dth characters of the strings in S and selecting the median of three elements in this set. The worst-case bound is lost, but the result is a fast, randomized algorithm.

Note that the ternary recursion tree of three-way radix quicksort is equivalent to a trie over the input strings where each trie node is implemented by a binary search tree whose node elements are the child edges (in the trie) of the trie node. In more detail, a node in a binary tree contains the character of a trie edge and a pointer to the root of the binary tree implementing the corresponding trie child. The search keys in a binary tree are the characters in its nodes. This trie implementation is named ternary search trees in [15]. In the recursion tree of three-way radix quicksort, an edge representing a recursive call on $S_<$ or $S_>$ corresponds to a tree edge inside a binary tree implementing a trie node, and an edge representing a recursive call on $S_=$ corresponds to a trie edge.

For the version of the algorithm where the partitioning character v is chosen as the median of all the dth characters, it is not hard to see that the binary trees representing the trie nodes become weighted trees. These are binary trees in which each element x has an associated weight w_x, and searches for x take $O(\log W/w_x)$, where $W = \Sigma_x w_x$ is the sum of all weights in the binary tree. Here, the weight of a binary tree node storing character x is the number of strings which in the trie reside below the corresponding trie edge. As shown in [13], in such a trie implementation, searching for a string P among K stored strings takes time $O(\log K + |P|)$, which is optimal for unbounded (i.e., comparison-based) alphabets. Hence, by the correspondence between the recursion trees of three-way radix quicksort and ternary search trees, three-way radix quicksort may additionally be viewed as a construction algorithm for an efficient dictionary structure for strings.

Other key results in the area of string sorting are now described. The classic string sorting algorithm is radixsort, which assumes a constant-sized alphabet. The least-significant-digit-first variant is easy to implement and runs in $O(N + l|\Sigma|)$ time, where l is the length of the longest string. The most-significant-digit-first variant is more complicated to implement but has a better running time of $O(D + d|\Sigma|)$, where D is the sum of the lengths of the distinguishing prefixes and d is the longest distinguishing prefix. McIlroy et al. [12] discusses in depth efficient implementations of radixsort.

If the alphabet consists of integers, then on a word-RAM the complexity of string sorting is essentially determined by the complexity of integer sorting. More precisely, the time (when allowing randomization) for sorting strings is $\Theta(\text{Sort}_{\text{Int}}(K) + N)$, where $\text{Sort}_{\text{Int}}(K)$ is the time to sort K integers [2], which currently is known to be $O(K\sqrt{\log\log K})$ [11].

Returning to comparison-based model, the papers [8, 10] give generic methods for turning any data structure over one-dimensional keys into a data structure over strings. Using finger search trees, this gives an adaptive sorting method for strings which uses $O(N + K\log(F/K))$ time, where F is the number of inversions among the strings to be sorted.

Concerning space complexity, it has been shown [9] that string sorting can still be done in $O(K\log K + N)$ time using only $O(1)$ space besides the strings themselves. However, this assumes that all strings have equal lengths.

All algorithms so far are designed to work in internal memory, where CPU time is assumed to be the dominating factor. For external memory computation, a more relevant cost measure is the number of I/Os performed, as captured by the I/O model [1], which models a two-level memory hierarchy with an infinite outer memory, an inner memory of size M, and transfer (I/Os) between the two levels taking place in blocks of size B. For external memory, upper bounds

were first given in [4], along with matching lower bounds in restricted I/O models. For a comparison-based model where strings may only be moved in blocks of size B (hence, characters may not be moved individually), it is shown in [4] that string sorting takes $\Theta(N_1/B \log_{M/B}(N_1/B) + K_2 \log_{M/B} K_2 + N/B)$ I/Os, where N_1 is the total length of strings shorter than B characters, K_2 is the number of strings of at least B characters, and N is the total number of characters. This bound is equal to the sum of the I/O costs of sorting the characters of the short strings, sorting B characters from each of the long strings, and scanning all strings. In the same paper, slightly better bounds in a model where characters may be moved individually in internal memory are given, as well as some upper bounds for non-comparison-based string sorting. Further bounds (using randomization) for non-comparison-based string sorting have been given, with I/O bounds of $O(K/B \log_{M/B}(K/M) \log\log_{M/B}(K/M) + N/B)$ [7] and $O(K/B(\log_{M/B}(N/M))^2 \log_2 K + N/B)$ (Ferragina, personal communication).

Returning to internal memory, it may also there be the case that memory hierarchy effects are the determining factor for the running time of algorithms but now due to cache faults rather than disk I/Os. Heuristic algorithms (i.e., algorithms without good worst-case bounds), aiming at minimizing cache faults for internal memory string sorting, have been developed. Of these, the burstsort line of algorithms [16] performs particularly well in experiments.

Applications

Data sets consisting partly or entirely of string data are very common: Most database applications have strings as one of the data types used, and in some areas, such as bioinformatics, Web retrieval, and word processing, string data is predominant. Additionally, strings form a general and fundamental data model, containing, e.g., integers and multidimensional data as special cases. Since sorting is arguably among the most important data processing tasks in any domain, string sorting is a general and important problem with wide practical applications.

Open Problems

As appears from the bounds discussed above, the asymptotic complexity of the string sorting problem is known for comparison-based alphabets. For integer alphabets on the word-RAM, the problem is almost closed in the sense that it is equivalent to integer sorting, for which the gap left between the known bounds and the trivial linear lower bound is small.

In external memory, the situation is less settled. As noted in [4], a natural upper bound to hope for in a comparison-based setting is to meet the lower bound of $\Theta(K/B \log_{M/B} K/M + N/B)$ I/Os, which is the sorting bound for K single characters plus the complexity of scanning the input. The currently known upper bounds only get close to this when leaving the comparison-based setting and allowing randomization.

Experimental Results

In [15], experimental comparison of two implementations (one simple and one tuned) of three-way radix quicksort with a tuned quicksort [6] and a tuned radixsort [12] showed the simple implementation to always outperform the quicksort implementation and the tuned implementation to be competitive with the radixsort implementation.

In [3], experimental comparison among existing and new radixsort implementations (including the one used in [15]), as well as tuned quicksort and tuned three-way radix quicksort, was performed. This study confirms the picture of three-way radix quicksort as very competitive, always being one of the fastest algorithms, and arguably the most robust across various input distributions.

Data Sets

The data sets used in [15]: http://www.cs.princeton.edu/~rs/strings/. The data sets used in [3]: http://dl.acm.org/citation.cfm?id=297136.

URL to Code

Code in C from [15]: http://www.cs.princeton.edu/~rs/strings/.
Code in C from [3]: http://dl.acm.org/citation.cfm?id=297136.
Code in Java from [14]: http://www.cs.princeton.edu/~rs/Algs3.java1-4/code.txt.

Cross-References

- Suffix Array Construction
- Suffix Tree Construction
- Suffix Tree Construction in Hierarchical Memory

Acknowledgments Research supported by Danish Council for Independent Research, Natural Sciences.

Recommended Reading

1. Aggarwal A, Vitter JS (1988) The input/output complexity of sorting and related problems. Commun ACM 31:1116–1127
2. Andersson A, Nilsson S (1994) A new efficient radix sort. In: Proceedings of the 35th annual symposium on foundations of computer science (FOCS'94), Santa Fe. IEEE Computer Society Press, pp 714–721
3. Andersson A, Nilsson S (1998) Implementing radixsort. ACM J Exp Algorithmics 3:7
4. Arge L, Ferragina P, Grossi R, Vitter JS (1997) On sorting strings in external memory (extended abstract). In: Proceedings of the 29th annual ACM symposium on theory of computing (STOC'97), El Paso. ACM, pp 540–548
5. Bentley J, Sedgewick R (1998) Algorithm alley: sorting strings with three-way radix quicksort. Dr Dobb's J Softw Tools 23:133–134, 136–138
6. Bentley JL, McIlroy MD (1993) Engineering a sort function. Softw Pract Exp 23:1249–1265
7. Fagerberg R, Pagh A, Pagh R (2006) External string sorting: faster and cache-oblivious. In: Proceedings of the 23rd annual symposium on theoretical aspects of computer science (STACS'06), Marseille. LNCS, vol 3884. Springer, pp 68–79
8. Franceschini G, Grossi R (2004) A general technique for managing strings in comparison-driven data structures. In: Proceedings of the 31st international colloquium on automata, languages and programming (ICALP'04), Turku. LNCS, vol 3142. Springer, pp 606–617
9. Franceschini G, Grossi R (2005) Optimal in-place sorting of vectors and records. In: Proceedings of the 32nd international colloquium on automata, languages and programming (ICALP'05), Lisbon. LNCS, vol 3580. Springer, pp 90–102
10. Grossi R, Italiano GF (1999) Efficient techniques for maintaining multidimensional keys in linked data structures. In: Proceedings of the 26th international colloquium on automata, languages and programming (ICALP'99), Prague. LNCS, vol 1644. Springer, pp 372–381
11. Han Y, Thorup M (2002) Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In: Proceedings of the 43rd annual symposium on foundations of computer science (FOCS'02), Vancouver. IEEE Computer Society Press, pp 135–144
12. McIlroy PM, Bostic K, McIlroy MD (1993) Engineering radix sort. Comput Syst 6:5–27
13. Mehlhorn K (1979) Dynamic binary search. SIAM J Comput 8:175–198
14. Sedgewick, R (2003) Algorithms in Java, Parts 1–4, 3rd edn. Addison-Wesley, Boston
15. Sedgewick R, Bentley J (1997) Fast algorithms for sorting and searching strings. In: Proceedings of the 8th annual ACM-SIAM symposium on discrete algorithms (SODA'97), New Orleans. ACM, pp 360–369
16. Sinha R, Zobel J, Ring D (2006) Cache-efficient string sorting using copying. ACM J Exp Algorithmics 11: Article No. 1.2

Strongly Connected Dominating Set

Zhang Zhao
College of Mathematics Physics and Information Engineering, Zhejiang Normal University, Zhejiang, Jinhua, China

Keywords

Absorbing set; Approximation algorithm; Dominating set; Strongly connected

Years and Authors of Summarized Original Work

2006; Du, Thai, Li, Liu, Zhu
2007: Park, Willson, Wang, Thai, Wu, Du
2009; Li, Du, Wan, Gao, Zhang, Wu
2012; Xu, Li
2014; Zhang, Wu, Wu, Li, Chen

Problem Definition

Let $G = (V, E)$ be a directed graph. For an arc $(u, v) \in E$, u is said to *dominate* v, and v is said to *absorb* u. Vertex u is also called a *dominator* of v, and vertex v is called an *absorber* of u. A vertex set $D \subseteq V$ is a *dominating set* (DS) of G if every vertex in $V \setminus D$ has a dominator in D; it is an *absorbing set* (AS) of G if every vertex in $V \setminus D$ has an absorber in D. A directed graph G is *strongly connected* if for any pair of ordered vertices $u, v \in V$, there is a directed path in G from u to v. The "Minimum Strongly Connected Dominating and Absorbing Set" problem (MSCDAS) is to find a vertex set D such that D is both a dominating set and an absorbing set of G and the subgraph of G induced by D is strongly connected.

Disk graph is a geometric graph which is of particular interest in the study of MSCDAS, since disk graph is a model of heterogeneous wireless sensor network, and as one can see in the application part, MSCDAS plays an important role in wireless sensor network. In a *disk graph*, every vertex u corresponds to a sensor on the plane equipped with an omnidirectional antenna of transmission radius $r(u)$. Another sensor v can correctly decode the message sent by u if and only if v is in the disk centered at u with radius $r(u)$. Hence, there is an arc (u, v) in the disk graph if and only if $\|uv\| \leq r(u)$, where $\|\cdot\|$ is the Euclidean distance between u and v. In particular, if all sensors are equipped with the same transmission radius, then the disk graph degenerates to an undirected graph called *unit disk graph*.

Key Results

Hardness Results

In a general digraph, the MSCDAS problem cannot be approximated within a factor of $(1-\varepsilon)\ln n$ for any real number $\varepsilon > 0$, where n is the number of vertices in the digraph. Even in disk graph, MSCDAS is still NP-hard. These hardness results follow from the fact that their undirected counterparts have these hardness results [1,5].

MSCDAS in General Digraph

Li et al. [8] gave a $(3H(n-1)-1)$-approximation for MSCDAS, where $H(\gamma) = \sum_{i=1}^{\gamma} 1/i$ is the harmonic number.

The algorithm is based on the following observation. For a vertex u in a digraph G, a spanning *in-arborescence* (resp. *out-arborescence*) rooted at u is a spanning sub-digraph of G in which every vertex except u has in-degree (resp. out-degree) exactly one and vertex u has in-degree (resp. out-degree) zero. For a spanning arborescence T of G, denote by $\text{int}(T)$ the set of internal vertices of T. For any vertex u, suppose T^{in} and T^{out} are spanning in-arborescence and spanning out-arborescence of G rooted at u, respectively. Then $\text{int}(T^{\text{in}}) \cup \text{int}(T^{\text{out}})$ is an SCDAS of G.

Define the problem "Spanning Arborescence with Fewest Internal Vertices" (SAFIV) as follows: given a digraph G and a vertex u, find a spanning arborescence T rooted at u such that $|\text{int}(T)|$ is as small as possible. By the above observation, if SAFIV has a ρ-approximation, then MSCDAS has a 2ρ-approximation. Li et al. gave a $(1.5H(n-1)-0.5)$-approximation for SAFIV, and thus the approximation ratio $(3H(n-1)-1)$ for MSCDAS follows.

The approximation algorithm for SAFIV uses the idea in [6, 7] which study the problem of "Minimum Node-Weighted Steiner Tree" (MNWST). The idea is to iteratively merge smaller arborescences greedily (a vertex is a trivial arborescence) until finally one gets one arborescence including all vertices which is rooted at the given vertex. It was pointed out in [8] that using the method in [6], the approximation ratio for SAFIV can be further reduced to $1.35\ln n$. Since SAFIV is at least as hard as the minimum

connected dominating set problem, it cannot be approximated within factor $(1-\varepsilon)\ln n$. Any progress narrowing the gap between $\ln n$ and $1.35\ln n$ would be interesting.

MSCDAS in Disk Graph

Making use of geometric properties, can the approximation ratio for MSCDAS be better in a disk graph? The answer is yes. Du et al. [2] were the first to give a constant approximation in this setting. Their idea was further explored by Park et al. [11] to output an SCDAS with size at most $9.6(k+1/2)^2\text{opt}+14.8(k+1/2)^2$, where opt is the size of an optimal solution and $k = r_{\max}/r_{\min}$, the ratio between the maximum radius and the minimum radius. The core in their work is an algorithm for SAFIV, which first colors all vertices white and then, by growing a search tree step by step, turns the colors to either black, blue, or gray. The set of black vertices forms a dominating set, and the set of blue vertices connects these black vertices into an out-arborescence. In fact, black vertices are mutually independent, where two vertices u and v are said to be *independent* if either uv or vu is not an arc. Two independent vertices have distance greater than $r_{\min}$. Such a property guarantees an upper bound for the number of black vertices. Furthermore, the structure of a search tree guarantees that the number of blues vertices is no larger than that of black vertices. Then, the desired approximation ratio follows. It should be noted that if $r_{\max}/r_{\min}$ is unbounded, then the approximation ratio is not a constant.

Without a bounded assumption on $r_{\max}/r_{\min}$, Xu and Li [12] showed that a $(2+\varepsilon)$-approximation exists for MDAS, which is a combination of a PTAS for MDS and a PTAS for MAS. In fact, the PTAS for MAS is a special case of the "Geometric Hitting Set" problem studied in [10], and the PTAS for MDS is a variation for the MDS problem in an undirected graph studied in [4]. Both PTASs are obtained through a local search method. The analysis is based on the separator theorem for planar graphs [3, 9]. Zhang et al. [13] also obtained approximation ratio $(2+\varepsilon)$ using the same method. Based on such a DAS, adding Steiner nodes to connect, Zhang et al. showed that a $(4+3\ln(2+\varepsilon)\text{opt}+\varepsilon)$-approximation exists for MSCDAS. When the optimal value opt is substantially smaller than n, this is an improvement on ratio $3H(n-1)-1$ for disk graphs.

Applications

One application of MSCDAS is the communication in wireless sensor network (WSN). In a WSN, information is distributed among sensors by multi-hop transmissions. If all sensors transmit messages in a flooding manner, then a lot of energy is wasted, and large amount of interference is created. To alleviate such problems, it is desirable that only a small fraction of sensors participate in the transmission, while information can still be successfully shared. An SCDAS can serve for this purpose. Suppose D is an SCDAS of directed graph G (the topology of the WSN). If there is a message at source sensor u to be sent to destination sensor v, then the message can be first sent from u to its absorber; since $G[D]$ is strongly connected, it can be successfully relayed to the dominator of v and then sent to v.

Open Problems

It is still open whether there exists a constant approximation algorithm for MSCDAS in disk graph.

Recommended Reading

1. Clark BN, Colbourn CJ, Johnson DS (1991) Unit disk graphs. Ann Discret Math 48:165–177
2. Du D-Z, Thai M, Li Y, Liu D, Zhu S (2006) Strongly connected dominating sets in wireless sensor networks with unidirectional links. In: Proceedings of APWEB, Harbin. LNCS, vol 3841, pp 13–24
3. Frederickson GN (1987) Fast algorithms for shortest paths in planar graphs, with applications. SIAM J Comput 16:1004–1022
4. Gibson M, Pirwani I (2010) Algorithms for dominating set in disk graphs: breaking the $\log n$ barrier. In: Algorithms – ESA, Liverpool. LNCS Vol. 6346, pp 243–254

5. Guha S, Khuller S (1998) Approximation algorithms for connected dominating sets. Algorithmica 20(4):374–387
6. Guha S, Khuller S (1999) Improved methods for approximating node weighted Steiner trees and connected dominating sets. Inf Comput 150:57–74
7. Klein P, Ravi R (1995) A nearly best-possible approximation algorithm for node-weighted Steiner trees. J Algorithms 19:104–114
8. Li D, Du H, Wan P-J, Gao X, Zhang Z, Wu W (2009) Construction of strongly connected dominating sets in asymmetric multihop wireless networks. Theor Comput Sci 410:661–669
9. Lipton RJ, Tarjan RE (1979) A separator theorem for planar graphs. SIAM J Appl Math 36:177–189
10. Mustafa NH, Ray S (2009) PTAS for Geometric hitting set problems via local search. In: SCG'09, Aarhus, 8–10 June 2009
11. Park MA, Willson J, Wang C, Thai M, Wu W, Du D-Z (2007) A dominating and absorbent set in a wireless ad-hoc network with different transmission ranges. In: MobiHoc'07, Montréal
12. Xu X, Li X (2012) Efficient construction of dominating set in wireless networks. http://arxiv.org/abs/1208.5738
13. Zhang Z, Wu W, Wu L, Li Y, Chen Z (accepted) Strongly connected dominating and absorbing set in directed disk graph. To be published in Int J Sens Netw

Subexponential Parameterized Algorithms

Fedor V. Fomin
Department of Informatics, University of Bergen, Bergen, Norway

Keywords

Chordal graph; Exponential time hypothesis; Graph editing; Interval graph; Minimum fill-in; Parameterized complexity

Years and Authors of Summarized Original Work

2009; Alon, Lokshtanov, Saurabh
2013; Fomin, Villanger
2014; Drange, Fomin, Pilipczuk, Villanger

Problem Definition

A *parameterized problem* is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed, finite alphabet. The second component is called the *parameter* of the problem. The central notion in parameterized complexity is the notion of *fixed-parameter tractability (FPT)*. A parameterized problem L is called FPT if it can be determined in time $f(k) \cdot n^c$ whether or not $(x,k) \in L$, where $n = |(x,k)|$, f is a computable function depending only on k, and c is a constant independent of n and k. The complexity class containing all fixed-parameter tractable problems is called FPT.

While in the definition of class FPT, we are happy with any computable function f, from application perspective it is often desirable to have the asymptotic growth of f as slow as possible. Take as an example an FPT problem VERTEX COVER which has been subjected to intense scrutiny with progressively faster algorithms designed for it. Let us remind that in the VERTEX COVER problem, we are asked if an n vertex graph G contains a vertex cover of size k or in other words a set of vertices S such that every edge of G has at least one endpoint in S. Starting from a k^k algorithm of Buss and Goldsmith in 1993, there have been algorithms with $f(k) \in \{2^k, 1.324718^k, 1.29175^k, 1.2906^k, 1.271^k, 1.2738^k\}$. The current fastest algorithm for VERTEX COVER runs in time $1.2738^k n^{\mathcal{O}(1)}$ (see the entry ▸ Vertex Cover Search Trees from this book). The ever-decreasing running time leads to the following natural question: can VERTEX COVER admit a *subexponential* time algorithm? That is, can it have an algorithm with running time $2^{o(k)n^{\mathcal{O}(1)}}$? The negative answer to this question would imply that $P \neq NP$. However, using a stronger assumption in complexity theory, namely, exponential time hypothesis (ETH) (see the entry ▸ Exponential Lower Bounds for k-SAT Algorithms in this book), one can show that if ETH holds, then the answer to our question is NO. Moreover, subject to ETH, there are no subexponential algorithms for many other natural *NP*-complete problems. Thus, another natural question arises:

is it true that every *NP*-complete problem cannot be solved in subexponential time? Interestingly, the answer to this question is again NO, and there are examples in the literature of such problems. Coming back to our example of VERTEX COVER problem, if we restrict the input graph to be planar, the problem remains *NP*-complete, but the brute-force algorithm problem can be sped up even more. That is, VERTEX COVER on planar graphs can be solved in time $2^{O(\sqrt{k})} \cdot n^{\mathcal{O}(1)}$, by *a subexponential algorithm*. We refer to more parameterized subexponential algorithms on planar graphs to the ▸ Bidimensionality in this book.

Until recently, the only subexponential algorithms were known for "geometric" graph problems, that is, problems on planar graphs or graphs excluding some fixed graph as minors. In 2009, Alon, Lokshtanov, and Saurabh [1] obtained the first parameterized subexponential algorithm for a natural "nongeometric" problem. This result has acted as catalyst for the discovery of new subexponential time algorithms. In this article, we give a short overview of these algorithms.

Key Results

FAST

In the FEEDBACK ARC SET IN TOURNAMENTS (FAST) problem, we are given an n-vertex tournament T and a positive integer k; the question is whether one can make T into a directed acyclic graph by deleting at most k arcs.

FAST
Input: A tournament $T = (V, E)$ and a nonnegative integer k.
Parameter: k.
Question: Is there $F \subseteq E$, $|F| \leq k$, such that diraph $H = (V, E \setminus F)$ is acyclic?

Alon, Lokshtanov, and Saurabh in [1] obtained a parameterized subexponential algorithm for FAST.

Theorem 1 ([1]) *FAST is solvable in time* $2^{\sqrt{k}\log k} n^{\mathcal{O}(1)}$.

The theorem is proved by making use of a novel randomized technique called *Chromatic Coding*. It appeared that subexponential algorithms exist for several other problems on tournaments (see entry ▸ Computing Cutwidth and Pathwidth of Semi-complete Digraphs in this book).

Fill-In

The next "nongeometric" problem for which a subexponential algorithm was found happened to be the classical MINIMUM FILL-IN problem.

A graph is *chordal* (or triangulated) if every cycle of length at least four contains a chord, i.e., an edge between nonadjacent vertices of the cycle. The MINIMUM FILL-IN problem (also known as MINIMUM TRIANGULATION and CHORDAL GRAPH COMPLETION) is to decide if a given graph G can be transformed into a chordal graph by adding at most k edges.

MINIMUM FILL-IN
Input: A graph $G = (V, E)$ and a nonnegative integer k.
Parameter: k.
Question: Is there $F \subseteq [V]^2$, $|F| \leq k$, such that graph $H = (V, E \cup F)$ is chordal?

Theorem 2 ([6]) MINIMUM FILL-IN *is solvable in time* $2^{\sqrt{k}\log k} n^{\mathcal{O}(1)}$.

The proof of the theorem is based on a combinatorial bound estimating the number of specific objects in the graph, namely, potential maximal cliques.

Completion to Graph Classes

Since discoveries of subexponential algorithms for FAST and MINIMUM FILL-IN, it appeared that several other graph modification problems admit subexponential algorithms. In particular, it was shown that problems of completion to a certain subclass of chordal graphs like trivially perfect, threshold [4], split [7], proper interval [2], and interval graphs [3] admit parameterized subexponential algorithms.

On the other hand, it has been shown that for a number of other graph classes, like cographs, completion to these classes of graphs cannot be done in parameterized subexponential time unless the exponential time hypothesis (ETH) fails [4].

Open Problems

The most natural open question about the given subexponential algorithms is the question about lower bounds. As a concrete example, an algorithm for FAST with running time bound $2^{o(\sqrt{k})}n^{\mathcal{O}(1)}$ would actually be a $2^{o(n)}$ time algorithm which inclines us to suspect that $2^{\mathcal{O}(\sqrt{k})}$ is the best possible dependency on k in the running time for this problem. Unfortunately, there is a big gap here between what we suspect and what we can prove, even assuming ETH. The only tight bound on parameterized subexponential algorithms for graph modification problems we are aware of is the p-CLUSTERING problem [5].

Cross-References

- Bidimensionality
- Computing Cutwidth and Pathwidth of Semi–complete Digraphs
- Exact Algorithms for Treewidth
- Exponential Lower Bounds for k-SAT Algorithms

Recommended Reading

1. Alon N, Lokshtanov D, Saurabh S (2009) Fast FAST. In: Proceedings of the 36th international colloquium of automata, languages and programming (ICALP). Lecture notes in computer science, vol 5555. Springer, Berlin/New York, pp 49–58
2. Bliznets I, Fomin FV, Pilipczuk M, Pilipczuk M (2014) A subexponential parameterized algorithm for proper interval completion. In: Proceedings of the 22nd annual European symposium on algorithms (ESA 2014). Lecture notes in computer science, vol 8737. Springer, Heidelberg, pp 173–183
3. Bliznets I, Fomin FV, Pilipczuk M, Pilipczuk M (2014) A subexponential parameterized algorithm for interval completion. CoRR. abs/1402.3473
4. Drange PG, Fomin FV, Pilipczuk M, Villanger Y (2014) Exploring subexponential parameterized complexity of completion problems. In: Proceedings of the 31st international symposium on theoretical aspects of computer science (STACS). Leibniz international proceedings in informatics (LIPIcs), vol 25. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Dagstuhl, pp 288–299
5. Fomin FV, Kratsch S, Pilipczuk M, Pilipczuk M, Villanger Y (2013) Tight bounds for parameterized complexity of cluster editing. In: Proceedings of the 30th international symposium on theoretical aspects of computer science (STACS). Leibniz international proceedings in informatics (LIPIcs), vol 20. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Dagstuhl, pp 32–43
6. Fomin FV, Villanger Y (2013) Subexponential parameterized algorithm for minimum fill-in. SIAM J Comput 42(6):2197–2216
7. Ghosh E, Kolay S, Kumar M, Misra P, Panolan F, Rai A, Ramanujan MS (2012) Faster parameterized algorithms for deletion to split graphs. In: Proceedings of the 13th Scandinavian symposium and workshops on algorithm theory (SWAT). Lecture notes in computer science, vol 7357. Springer, Berlin/New York, pp 107–118

Subset Sum Algorithm for Bin Packing

Julián Mestre
Department of Computer Science, University of Maryland, College Park, MD, USA
School of Information Technologies, The University of Sydney, Sydney, NSW, Australia

Keywords

Approximation algorithm; Bin packing; Greedy; Knapsack; Subset sum

Years and Authors of Summarized Original Work

1972; Graham
1999; Gupta, Ho

2009; Epstein, Kleiman, Mestre 2011; Epstein, Kleiman

Problem Definition

Bin packing is a classical problem in combinatorial optimization. Given a collection of n items with different sizes, the objective is to pack the items into a minimum number of uniform capacity bins. More formally, the input of the bin packing problem is described by a set of n items $I = \{1, \ldots, n\}$ and a size function $s : I \to [0, 1]$. The output is a packing of the items into bins $B_1, \ldots, B_k \subseteq I$ such that $s(B_j) \leq 1$ for $j = 1, \ldots, k$, where the notation $s(B)$ denotes $\sum_{i \in B} s_i$ for any $B \subseteq I$. The objective is to minimize the number bins used in the packing.

The SUBSET-SUM algorithm is an intuitively appealing greedy heuristic for the bin packing problem: Starting from the empty packing, the algorithm repeatedly finds a subset B of yet-unpacked items maximizing $s(B)$ subject to $s(B) \leq 1$, adds B to the packing, and iterates. Each iteration requires that we solve an instance of the *knapsack* problem. In practice, instead of finding the optimal solution, one can use an fully polynomial time approximation scheme (FPRAS) to compute a $(1 - \epsilon)$-approximate solution [6].

This note is concerned with the worst-case asymptotic performance of the SUBSET-SUM algorithm. For a given instance $s : I \to [0, 1]$, we use $\mathrm{OPT}(s)$ to denote the number of bins used in an optimal packing of s and $\mathrm{SS}(s)$ to denote the number of bins used by the SUBSET-SUM algorithm. Then for a given class $\mathcal{C}$ of instances, we define the worst-case asymptotic approximation ratio of SUBSET-SUM as

$$R_{\mathrm{SS}}^{\infty}(\mathcal{C}) = \lim_{k \to \infty} \sup_{\substack{s \in \mathcal{C} \\ \mathrm{OPT}(s) = k}} \frac{\mathrm{SS}(s)}{\mathrm{OPT}(s)}. \qquad (1)$$

Finally, we use R_{SS}^{∞} to denote the ratio for general instances of the problem.

Key Results

Lower Bound on R_{SS}^{∞}

Graham [4] provided a family of instance exhibiting an approximation ratio that tends to $\sum_{i=1}^{\infty} \frac{1}{2^i - 1} \approx 1.6067$.

Theorem 1 (Graham [4]) $R_{\mathrm{SS}}^{\infty} \geq \sum_{i=1}^{\infty} \frac{1}{2^i - 1} \approx 1.6067$.

Proof Consider the following instance parameterized by two positive integers r and N. For each $j = 1, \ldots, r$, we create N items of size $2^{-i} + \delta$, where $\delta = 2^{-2r}$. Let us denote this instance with s. Provided that $2^i - 1$ divides N for all $i = 1, \ldots, r$, it is not hard to see that SUBSET-SUM first packs the smallest items into $N/(2^r - 1)$ bins, then it packs the second-smallest items into $N/(2^{r-1} - 1)$ bins, and so on, until it packs the largest items into N bins. On the other hand, the optimal solution uses just N bins by packing one item of each size class per bin. Therefore,

$$\frac{\mathrm{SS}(s)}{\mathrm{OPT}(s)} = \sum_{i=1}^{r} \frac{1}{2^i - 1}, \qquad (2)$$

which quickly approaches 1.6067 as r grows. □

Upper Bound on R_{SS}^{∞}

A trivial upper bound on R_{SS}^{∞} is 2. This follows from the fact only the last bin can be less than half full. Caprara and Pferschy [1] gave the first nontrivial upper bound, by showing that R_{SS}^{∞} is at most $4/3 + \ln 4 \approx 1.6210$. Interestingly, Graham [4] had conjectured that the true value of R_{SS}^{∞} should match his lower bound. This conjecture was finally proven by Epstein et al. [2].

Theorem 2 (Epstein et al. [3]) $R_{\mathrm{SS}}^{\infty} \leq \sum_{i=1}^{\infty} \frac{1}{2^i - 1} \approx 1.6067$.

The proof of this result uses *weighting functions* and a *factor revealing mathematical program*. Here we only sketch the high level idea of the approach. Let B be one of the bins opened by SUBSET-SUM. For every item $i \in B$ we define

$$w_i = \begin{cases} \frac{s_i}{s(B)} & \text{if } 1 - s_{\min} \leq s(B), \\ s_i & \text{otherwise,} \end{cases} \qquad (3)$$

S

where $s_{\min}$ is the size of the smallest yet-unpacked item just before opening B.

The weights are used to charge the cost of the packing computed by SUBSET-SUM to an optimal packing. The following lemma allows us to bound the performance of the algorithm provided we can show that the sum of the weights is comparable to the cost of the SUBSET-SUM packing and that no bin in the optimal solution is charged too much.

Lemma 1 *Let $\mathcal{O}$ be an optimal solution and $\mathcal{B}$ be the solution computed* SUBSET-SUM. *If there is a weighting function w such that $w(O) \leq \rho$ for all $O \in \mathcal{O}$ and $|\mathcal{B}| \leq w(I) + \delta$, then $|\mathcal{B}| \leq \rho|\mathcal{O}| + \delta$.*

Proof Because $\mathcal{O}$ is a packing $\sum_{O \in \mathcal{O}} w(O) = w(I)$, therefore,

$$|\mathcal{B}| \leq w(I) + \delta = \sum_{O \in \mathcal{O}} w(O) + \delta \leq \rho|\mathcal{O}| + \delta. \quad \square$$

The key contribution of Epstein et al. [3] was bounding the parameters ρ and δ associated with the weighting function (3). Bounding δ is a relatively straightforward exercise. Bounding ρ is more involved and requires analytically solving a mathematical program. Here we only state their bounds.

Lemma 2 (Epstein et al. [3]) *Let $\mathcal{B}$ be the* SUBSET-SUM *packing and let w be the weighting function* (3) *for $\mathcal{B}$. Then*

1. $|\mathcal{B}| \leq w(I) + 1$,
2. $w(B) \leq \sum_{i=1}^{\infty} \frac{1}{2^i - 1}$ *for all* $B \subseteq I$ *such that* $s(B) \leq 1$.

Theorem 2 follows immediately from Lemmas 1 and 2.

Parametric Case

As it is the case with most bin packing heuristics, the performance of SUBSET-SUM improves when the items are small relative to the capacity of the bin. In a *parametric analysis* of a heuristic, we restrict our attention to instances where the maximum item size is bounded. More formally, for every real $\alpha \in (0, 1]$, we define $\mathcal{C}_\alpha$ to be the class of instances s such that $\max_{i \in I} s_i \leq \alpha$.

Theorem 3 (Epstein et al. [3]) *For every integer $t \geq 1$ and $\alpha \in (\frac{1}{t+1}, \frac{1}{t}]$, we have $R_{\mathrm{SS}}^{\infty}(\mathcal{C}_\alpha) = 1 + \sum_{i=1}^{\infty} \frac{1}{(t+1)2^i - 1}$.*

Notice that this is a strict generalization of Theorems 1 and 2, which only cover the case $\alpha = 1$.

Applications

There is an interesting connection between the performance of the SUBSET-SUM algorithm and the quality of equilibria of a game-theoretic version of bin packing. Let us associate a game with each instance $s : I \to [0, 1]$ of the bin packing problem. The set of players in this game is I, the set of items. Each player can decide in which bin it wants to be packed; this is the player's strategy space. For each bin B chosen in this uncoordinated fashion, if $s(B) > 1$ then the players in B are charged ∞; otherwise, player $i \in B$ is charged $\frac{s_i}{s(B)}$. These payments enforce that a strategy profile is a valid packing if and only if the payments are finite. Furthermore, if the payments are finite, the sum of these payments equals the number of bins in the packing.

A strategy profile is said to be a Nash Equilibrium (NE) if there is no player that can switch bins to decrease its payment. The *price of anarchy* of the bin packing game is the asymptotic worst-case ratio between the number of bins used by an NE and the number of bins in an optimal packing. A packing is said to be a Strong Nash Equilibrium (SNE) if no coalition of players can switch bins to decrease the sum of their payments. The *strong price of anarchy* of the bin packing game is the asymptotic worst-case ratio between the number of bins used by an SNE and the number of bins by an optimal packing.

Theorem 4 (Epstein and Kleiman [2]) *The strong price of anarchy for the bin packing game is exactly R_{SS}^{∞}.*

Notice that every SNE is an NE, since we can think of an NE as requiring that there are no "coalitions" of size 1. Therefore, Theorem 4 establishes a lower bound on the price of anarchy for the bin packing game. However, not every NE is an SNE. In fact, it is known that the price of anarchy for the bin packing game is strictly worse than its strong price of anarchy [2, 3].

Experimental Results

Gupta and Ho [5] performed an experimental evaluation of SUBSET-SUM. (Gupta and Ho call the algorithm *minimum bin slack* because they formulate each iteration as trying to minimize the slack (unused space) of the bin, which is equivalent to maximizing the bin's usage.) The instances used in the evaluation were randomly generated by selecting the item sizes uniformly at random from different numerical ranges. They compared the performance of SUBSET-SUM to two well-known heuristics: FIRST-FIT-DECREASING and BEST-FIT-DECREASING. They observed that SUBSET-SUM performed better on average without incurring a significant computational overhead.

Cross-References

- ▸ Bin Packing
- ▸ Knapsack
- ▸ Price of Anarchy

Recommended Reading

1. Caprara A, Pferschy U (2004) Worst-case analysis of the subset sum algorithm for bin packing. Oper Res Lett 32(2):159–166
2. Epstein L, Kleiman E (2011) Selfish bin packing. Algorithmica 60(2):368–394
3. Epstein L, Kleiman E, Mestre J (2009) Parametric packing of selfish items and the subset sum algorithm. In: Proceedings of the 5th workshop on internet and network economics, Rome, Italy pp 67–78
4. Graham RL (1972) Bounds on multiprocessing anomalies and related packing algorithms. In: Proceedings of the 1972 spring joint computer conference, Atlantic City, New Jersey, USA pp 205–217
5. Gupta J, Ho J (1999) A new heuristic algorithm for the one-dimensional bin-packing problem. Prod Plan Control 10(6):598–603
6. Kellerer H, Pferschy U, Pisinger D (2004) Knapsack problems. Springer, Berlin/New York

Substring Parsimony

Mathieu Blanchette
Department of Computer Science, McGill University, Montreal, QC, Canada

Years and Authors of Summarized Original Work

2001; Blanchette, Schwikowski, Tompa

Problem Definition

The Substring Parsimony Problem, introduced by Blanchette et al. [1] in the context of motif discovery in biological sequences, can be described in a more general framework:
Input:

- A discrete space $\mathcal{S}$ on which an integral distance d is defined (i.e., $d(x, y) \in \mathbb{N}\ \forall x, y \in \mathcal{S}$).
- A rooted binary tree $T = (V, E)$ with n leaves. Vertices are labeled $\{1, 2, \ldots, n, \ldots, |V|\}$, where the leaves are vertices $\{1, 2, \ldots, n\}$.
- Finite sets $S_1, S_2, \ldots, S_n$, where set $S_i \subseteq \mathcal{S}$ is assigned to leaf i, for all $i = 1 \ldots n$.
- A non-negative integer t

Output: All solutions of the form $(x_1, x_2, \ldots, x_n, \ldots, x_{|V|})$ such that:

- $x_i \in \mathcal{S}$ for all $i = 1 \ldots |V|$

- $x_i \in S_i$ for all $i = 1 \ldots n$
- $\sum_{(u,v)\in E} d(x_u, x_v) \leq t$

The problem thus consists of choosing one element x_i from each set S_i such that the Steiner distance of the set of points is at most t. This is done on a Steiner tree T of fixed topology. The case where $|S_i| = 1$ for all $i = 1 \ldots n$ is a standard Steiner tree problem on a fixed tree topology (see [11]). It is known as the Maximum Parsimony Problem and its complexity depends on the space $\mathcal{S}$.

Key Results

The substring parsimony problem can be solved using a dynamic programming algorithm. Let $u \in V$ and $s \in \mathcal{S}$. Let $W_u[s]$ be the score of the best solution that can be obtained for the subtree rooted at node u, under the constraint that node u is labeled with s, i.e.,

$$W_u[s] = \min_{\substack{x_1,\ldots,x_{|V|}\in\mathcal{S} \\ x_u = s}} \sum_{\substack{(i,j)\in E \\ i,j\in \text{subtree}(u)}} d(x_i, x_j).$$

Let v be a child of u, and let $X_{(u,v)}[s]$ be the score of the best solution that can be obtained for the subtree consisting of node u together with the subtree rooted at its child v, under the constraint that node u is labeled with s:

$$X_{(u,v)}[s] = \min_{\substack{x_1,\ldots,x_{|V|}\in\mathcal{S} \\ x_u = s}} \sum_{\substack{(i,j)\in E \\ i,j\in \text{subtree}(v)\cup\{(u,v)\}}} d(x_i, x_j).$$

Then, we have:

$$W_u[s] = \begin{cases} 0 & \text{if } u \text{ is a leaf and } s \in S_u \\ +\infty & \text{if } u \text{ is a leaf and } s \notin S_u \\ \sum_{v\in\text{children}(u)} X_{(u,v)}[s] & \text{if } u \text{ is not a leaf} \end{cases}$$

and

$$X_{(u,v)}[s] = \min_{y'\in\mathcal{S}} W_u[s'] + d(s, s').$$

Tables W and X can thus be computed using a dynamic programming algorithm, proceeding in a post-order traversal of the tree. Solutions can then be recovered by tracing the computation back for all s such that $W_{\text{root}}[s] \leq t$. Note that the same solution may be recovered more than once in this process.

A straight-forward implementation of this dynamic programming algorithm would run in time $O(n \cdot |\mathcal{S}|^2 \cdot \gamma(\mathcal{S}))$, where $\gamma(\mathcal{S})$ is the time needed to compute the distance between any two points in $\mathcal{S}$. Let $N_a(\mathcal{S})$ be the maximum number of a-neighbors a point in $\mathcal{S}$ can have, i.e., $N_a(\mathcal{S}) = \max_{x\in\mathcal{S}} |\{y \in \mathcal{S} : d(x, y) = a\}|$. Blanchette et al. [3] showed how to use a modified breadth-first search of the space $\mathcal{S}$ to compute each table $X_{(u,v)}$ in time $O(|\mathcal{S}| \cdot N_1(\mathcal{S}))$, thus reducing the total time complexity to $O(n \cdot |\mathcal{S}| \cdot N_1(\mathcal{S}))$. Since only solutions with a score of at most t are of interest, the complexity can be further reduced by only computing those table entries which will yield a score of at most t. This results in an algorithm whose running time is $O(n \cdot M \cdot N_{\lfloor t/2 \rfloor}(\mathcal{S}) \cdot N_1(\mathcal{S}))$ where $M = \max_{i=1\ldots n} |S_i|$.

The problem has been mostly studied in the context of biological sequence analysis, where $\mathcal{S} = \{A, C, G, T\}^k$, for some small k ($k = 5, \ldots, 20$ are typical values). The distance d is the Hamming distance, and a phylogenetic tree T is given. The case where $|S_i| = 1$ for all $i = 1 \ldots n$ is known as the Maximum Parsimony Problem and can be solved in time $O(n \cdot k)$ using Fitch's algorithm [9] or Sankoff's algorithm [12]. In the more general version, a long DNA sequence P_u of length L is assigned to each leaf u. The set S_u is defined as the set of all k-substrings of P_u. In this case, $M = L - k + 1 \in O(L)$, and $N_a \in O(\min(4^k, (3k)^a))$, resulting in a complexity of $O(n \cdot L \cdot 3k \cdot \min(4^k, (3k)^{\lfloor d/2 \rfloor}))$. Notice that for a fixed k and d, the algorithm is linear over the whole sequence. The problem was independently shown to be NP-hard by Blanchette et al. [3] and by Elias [7].

Applications

Most applications are found in computational biology, although the algorithm can be applied to a wide variety of domains. The algorithm

for the substring parsimony problem has been implemented in a software package called FootPrinter [5] and applied to the detection of transcription factor binding sites in orthologous DNA regulatory sequences through a method called phylogenetic footprinting [4]. Other applications include the search for conserved RNA secondary structure motifs in orthologous RNA sequences [2]. Variants of the problem have been defined to identify motifs regulating alternative splicing [13]. Blanchette et al. [3] study a relaxation of the problem where one does not require that a substring be chosen from each of the input sequences, but instead asks that substrings be chosen from a sufficiently large subset of the input sequence. Fang and Blanchette [8] formulate another variant of the problem where substring choices are constrained to respect a partial order relation defined by a set of local multiple sequence alignments.

Open Problems

Optimizations taking advantage of the specific structure of the space $\mathcal{S}$ may yield more efficient algorithms in certain cases. Many important variations could be considered. First, the case where the tree topology is not given needs to be considered, although the resulting problems would usually be NP-hard even when $|S_i| = 1$. Another important variation is one where the phylogenetic relationships between trees is not given by a tree but rather by a phylogenetic network [10]. Finally, randomized algorithms similar to those proposed by Buhler et al. [6] may yield important and practical improvements.

URL to Code

http://bio.cs.washington.edu/software.html

Cross-References

- ▸ Closest Substring
- ▸ Efficient Methods for Multiple Sequence Alignment with Guaranteed Error Bounds
- ▸ Local Alignment (with Affine Gap Weights)
- ▸ Local Alignment (with Concave Gap Weights)
- ▸ Statistical Multiple Alignment
- ▸ Steiner Trees

Recommended Reading

1. Blanchette M (2001) Algorithms for phylogenetic footprinting. In: RECOMB01: proceedings of the fifth annual international conference on computational molecular biology, Montreal. ACM, pp 49–58
2. Blanchette M (2002) Algorithms for phylogenetic footprinting. PhD thesis, University of Washington
3. Blanchette M, Schwikowski B, Tompa M (2002) Algorithms for phylogenetic footprinting. J Comput Biol 9(2):211–223
4. Blanchette M, Tompa M (2002) Discovery of regulatory elements by a computational method for phylogenetic footprinting. Genome Res 12:739–748
5. Blanchette M, Tompa M (2003) Footprinter: a program designed for phylogenetic footprinting. Nucleic Acids Res 31(13):3840–3842
6. Buhler J, Tompa M (2001) Finding motifs using random projections. In: RECOMB01: proceedings of the fifth annual international conference on computational molecular biology, pp 69–76
7. Elias I (2006) Settling the intractability of multiple alignment. J Comput Biol 13:1323–1339
8. Fang F, Blanchette M (2006) Footprinter3: phylogenetic footprinting in partially alignable sequences. Nucleic Acids Res 34(2):617–620
9. Fitch WM (1971) Toward defining the course of evolution: minimum change for a specified tree topology. Syst Zool 20:406–416
10. Huson DH, Bryant D (2006) Application of phylogenetic networks in evolutionary studies. Mol Biol Evol 23(2):254–267
11. Sankoff D, Rousseau P (1975) Locating the vertices of a Steiner tree in arbitrary metric space. Math Program 9:240–246
12. Sankoff DD (1975) Minimal mutation trees of sequences. SIAM J Appl Math 28:35–42
13. Shigemizu D, Maruyama O (2004) Searching for regulatory elements of alternative splicing events using phylogenetic footprinting. In: Proceedings of the fourth workshop on algorithms for bioinformatics. Lecture notes in computer science. Springer, Berlin, pp 147–158

Succinct and Compressed Data Structures for Permutations and Integer Functions

Jérémy Barbay
Department of Computer Science (DCC),
University of Chile, Santiago, Chile

Keywords

Adaptive; Compression; Functions; Permutation

Years and Authors of Summarized Original Work

2012; Munro, Raman, Raman, Rao
2012; Barbay, Fischer, Navarro
2013; Barbay, Navarro
2013; Barbay

Problem Definition

A basic building block for compressed data structures for texts and functions is the representation of a permutation of the integers $\{1,\ldots,n\}$, denoted by $[1\ldots n]$. A permutation π is trivially representable in $n\lceil \lg n\rceil$ bits which is within $O(n)$ bits of the information theoretic bound of $\lg(n!)$, but instances from restricted classes of permutations can be represented using much less space.

We are interested in encodings of permutations that can efficiently access them. Given a permutation π over $[1\ldots n]$, an integer k and an integer $i \in [1\ldots n]$, data structures on permutations aim to support the following operators as fast as possible, using as little additional space as possible:

- $\pi(i)$: application of the permutation to i,
- $\pi^{-1}(i)$: application of the inverse permutation to i,
- $\pi^{(k)}(i)$: $\pi()$ iteratively applied k times starting with value i (e.g., $\pi^{(2)}(i) = \pi(\pi(i))$).

Key Results

We distinguish between two types of solutions: the succinct index and two succinct data structures for permutations introduced by Munro et al. [1], and the various compressed data structures proposed later [2–4].

Succinct Data Structures

Munro et al. [1] studied the problem of succinctly representing a permutation to support operators on it quickly. They give several solutions, described below.

"Shortcut" Index Supporting $\pi()$ and $\pi^{-1}()$

Given an integer parameter t, the operators $\pi()$ and $\pi^{-1}()$ can be supported by simply writing down π in an array of n words of $\lceil \lg n\rceil$ bits each, plus an auxiliary array S of at most n/t back pointers called shortcuts: in each cycle of length at least t, every t-th element has a pointer t steps back. Then, $\pi(i)$ is simply the i-th value in the primary structure, and $\pi^{-1}(i)$ is found by moving forward until a back pointer is found and then continuing to follow the cycle to the location that contains the value i.

The trick is in the encoding of the locations of the back pointers: this is done with a simple bit vector B of length n, in which a 1 indicates that a back pointer is associated with a given location. B is augmented using $o(n)$ additional bits so that the number of 1's up to a given position and the position of the r-th 1 can be found in constant time (i.e., using the rank and select operators on binary strings [5]). This gives the location of the appropriate back pointer in the auxiliary array S. As there are back pointers every t elements in the cycle, finding the predecessor requires $O(t)$ memory accesses.

Theorem 1 *For any strictly positive integer n and any permutation π on $[1\ldots n]$ which can be decomposed into δ cycles of respective sizes $c_1,\ldots,c_\delta$, there is a representation of π using within $(\sum_{i\in[1\ldots\delta]}\lfloor \frac{c_i}{t}\rfloor)\lg n + 2n + o(n) \subseteq \frac{n\lg n}{t} + 2n + o(n)$ bits to support the operator $\pi()$ in constant time and the operator $\pi^{-1}()$ in time within $O(t)$.*

Interestingly enough, Munro et al. [1] did not notice that their construction is actually an index and that the raw encoding can be replaced by any data structure supporting the operator $\pi()$, including the compressed ones later described [4].

"Cycle" Data Structure Supporting $\pi^k()$

For arbitrary i and k, $\pi^k()$ is supported by writing the cycles of π together with a bit vector B marking the beginning of each cycle. Observe that the cycle representation itself is a permutation in "standard form"; call it σ. The first task is to find i in the representation: it is in position $\sigma^{-1}(i)$. The segment of the representation containing i is found through the rank and select operators on B. Then $\pi^k(i)$ is determined by taking k modulo the cycle length, moving that number of steps around the cycle starting at the position of i, and applying $\sigma()$ to obtain the value to return.

Other than the support of the operators on σ, all operators are performed in constant time; hence the asymptotic supporting time of $\pi^k()$ depends on the supporting time in which the data structure chosen to represent σ supports the operators $\sigma()$ and $\sigma^{-1}()$. Munro et al. [1] proposed the following, using a raw encoding of σ with a shortcut index to support $\sigma^{-1}()$:

Theorem 2 *For any strictly positive integer n and any permutation π on $[1 \dots n]$, there is a representation of π using at most $(1+\varepsilon)n \lg n + O(n)$ bits to support the operator $\pi^k()$ in time within $O(1/\varepsilon)$, for any ε less than 1 and for any arbitrary value of k.*

Under a restricted model of pointer machine, this technique is optimal: using $O(n)$ extra bits (i.e., $O(n/\log n)$ extra words), time within $\Omega(\log n)$ is necessary to support both $\pi()$ and $\pi^{-1}()$.

"Benes Network" Data Structure Supporting $\pi^k()$

Any permutation can be implemented by a communication network composed of switches: this is called a Benes Network and uses even less space under the RAM model than the solutions described in the previous sections. Sparsely adding pointers accelerates the support of $\pi^k()$ to time within $O(\frac{\log n}{\log \log n})$.

Theorem 3 *For any strictly positive integer n and any permutation π on $[1 \dots n]$, there is a representation of π using at most $\lceil \lg(n!) \rceil + O(n)$ bits to support the operator $\pi^k()$ in time within $O(\log n / \log \log n)$.*

This representation uses space within an additive term within $O(n)$ of the optimal, both on average and in the worst case over all permutations over $[1 \dots n]$.

Compressed Data Structures

Any comparison-based sorting algorithm yields an encoding for permutations, and any adaptive sorting algorithm in the comparison model yields a compression scheme for permutations. Supporting operators on such compressed permutation in less time than required to decompress the whole of it requires some more work:

Runs

Barbay and Navarro [2] described how to segment a partition into $\texttt{nRuns}$ *runs* composed of consecutive positions forming already sorted blocks and how to merge those via a wavelet tree. This yields a data structure compressing a permutation within space optimal over all permutations with $\texttt{nRuns}$ runs of sizes given by the vector $\texttt{vRuns}$. This data structure supports the operators $\pi()$ and $\pi^{-1}()$ in sublinear time within $O(1 + \log \texttt{nRuns})$, with the average supporting time within $O(1 + \mathcal{H}(\texttt{vRuns}))$, which decreases with the entropy of the partition of the permutation into runs. Here, the *entropy* of a sequence of positive integers $X = \langle n_1, n_2, \dots, n_r \rangle$ adding up to n is $\mathcal{H}(X) = \sum_{i=1}^{r} \frac{n_i}{n} \lg \frac{n}{n_i}$.

Theorem 4 *For any strictly positive integer n and any permutation π on $[1 \dots n]$ which can be decomposed into $\texttt{nRuns}$ runs of sizes $\texttt{vRuns} = (r_1, \dots, r_{\texttt{nRuns}})$, there is a representation of π using at most $n\mathcal{H}(\texttt{vRuns}) + O(\texttt{nRuns} \log n) + o(n)$ bits to support the computation of $\pi(i)$ and $\pi^{-1}(i)$ in time within $O(1 + \log \texttt{nRuns})$ in the worst case over $i \in [1 \dots n]$ and in*

time within $O(1 + \mathcal{H}(\texttt{vRuns}))$ *on average when* $i \in [1 \dots n]$ *is uniformly distributed. This compressed data structure can be computed in time within* $O(n(1 + \mathcal{H}(\texttt{vRuns})))$*, which is worst-case optimal in the comparison model over all such permutations decomposed into* nRuns *runs of sizes given by the vector* vRuns.

The partitioning takes only $n - 1$ comparisons, and the construction of the compressed data structure itself is an adaptive sorting algorithm improving over previous results [6, 7].

Heads of Strict Runs

A two-level partition of the permutation yields further compression [2]. The first level partitions the permutation into *strict ascending runs* (maximal ranges of positions satisfying $\pi(i + k) = \pi(i) + k$). The second level partitions the *heads* (first position) of those strict runs into conventional ascending runs. This is analogous to the notion of blocks described by Moffat and Petersson [7] for multisets.

Theorem 5 *For any strictly positive integer* n *and any permutation* π *on* $[1 \dots n]$ *which can be decomposed into* nBlock *strict runs and into* $\texttt{nRuns} \leq \texttt{nBlock}$ *monotone runs, let* vHRuns *be the vector formed by the* nRuns *monotone run lengths in the permutation of strict run heads. Then, there is a representation of* π *using at most* $\texttt{nBlock}\mathcal{H}(\texttt{vHRuns})) + O(\texttt{nBlock} \log \frac{n}{\texttt{nBlock}}) + o(n)$ *bits to support the operator* $\pi()$ *and* $\pi^{-1}()$ *in time within* $O(1 + \log \texttt{nBlock})$*. This compressed data structure can be computed in time within* $O(n(1 + \log \texttt{nBlock}))$.

Shuffled Subsequences

The preorder measures seen so far have considered runs which group contiguous positions in π: this does not need to be always the case. A permutation π over $[1 \dots n]$ can be decomposed in n comparisons into a minimal number nSUS of *Shuffled Up Sequences*, defined as a set of, not necessarily consecutive, subsequences of increasing numbers that have to be removed from π in order to reduce it to the empty sequence [8]. Then those subsequences can be merged using the same techniques as above, which yields a new adaptive sorting algorithm and a new compressed data structure [2]. An optimal partition of a permutation π over $[1 \dots n]$ into a minimal number nSMS of *Shuffled Monotone Sequences*, sequences of not necessarily consecutive subsequences of increasing or decreasing numbers, is NP-hard to compute [9], but if such a permutation is given, the same technique applies [10].

LRM Subsequences

LRM trees partition a sequence of values into consecutive sorted blocks and express the relative position of the first element of each block within a previous block. Such a tree can be computed in $2(n - 1)$ comparisons within the array and overall linear time, through an algorithm similar to that of Cartesian Trees [11]. The interest of LRM trees in the context of adaptive sorting and permutation compression is that the values are increasing in each root-to-leaf branch: they form a partition of the array into subsequences of increasing values. Barbay et al. [3] described how to compute the partition of the LRM tree of minimal size-vector entropy, which yields a compressed data structure asymptotically smaller than $\mathcal{H}(\texttt{vRuns})$-adaptive sorting, smaller in practice than $\mathcal{H}(\texttt{vSUS})$-adaptive sorting, as well as a faster adaptive sorting algorithm.

Number of Inversions

The preorder measure nInv counts the number of pairs (i, j) of positions $1 \leq i < j \leq n$ in a permutation π over $[1 \dots n]$ such that $\pi(i) > \pi(j)$. Its value is exactly the number of comparisons performed by the algorithm `Insertion Sort`, between n and n^2 for a permutation over $[1 \dots n]$. A variant of `Insertion Sort`, named `Local Insertion Sort`, sorts π in $n(1 + \lceil \lg(\texttt{nInv}/n) \rceil)$ comparisons [6, 7].

Simply encoding the n values $(\pi(i) - i)_{i \in [1 \dots n]}$ using the γ' code from Elias [12], and indexing the positions of the beginning of each code by a compressed bit vector, yields a compressed data structure supporting the operator $\pi()$ in constant time. The resulting data structure uses space within $n(1 + 2 \lg \frac{\texttt{nInv}}{n}) + o(n)$ bits. Support for

the operator $\pi^{-1}()$ can be added in two distinct ways, either encoding both π and π^{-1} using this technique within $2n(1 + 2\lg \frac{\texttt{nInv}}{n}) + o(n)$ bits, which supports both operators $\pi()$ and $\pi^{-1}()$ in constant time, or adding support for the operator $\pi^{-1}()$ using Munro et al.'s shortcut succinct index for permutations [1] described previously.

Removing Elements

The preorder measure `nRem` counts the minimum number of elements that must be removed from a permutation so that what remains is already sorted. Its exact value is n minus the length of the *Longest Increasing Subsequence*, which can be computed in time within $O(n \log n)$. Alternatively, the value of `nRem` can be approximated within a constant factor of 2 in $2(n - 1)$ comparisons. Partitioning π into the removed elements and the remaining ones through a bit vector of n bits, representing the order of the $2\texttt{nRem}$ elements in a wavelet tree (using any of the data structures described above), and representing the merging of both into n bits yield a compressed data structure using space within $2n + 2\texttt{nRem}\lg(n/\texttt{nRem}) + o(n)$ bits and supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time, within $O(1 + \log(\texttt{nRem} + 1))$.

Applications

Integer Functions

Munro et al. [1] extended the results on permutations to arbitrary functions from $[1 \ldots n]$ to $[1 \ldots n]$. Again $f^k(i)$ indicates the function iterated k times starting at i: if k is nonnegative, this is straightforward. The case in which k is negative is more complicated as the image is a (possibly empty) multiset over $[1 \ldots n]$.

Whereas π is a set of cycles, f can be viewed as a set of cycles in which each node is the root of a tree. Starting at any node (element of $[1 \ldots n]$), the evaluation moves one step along a branch of the tree, or one step along a cycle. Moving k steps in a positive direction is straightforward, and one moves up a tree and perhaps around a cycle. When k is negative, one must determine all nodes at distance k from the starting location, i, in the direction toward the leaves of the trees. The key technical issue is to run across succinct tree representations picking off all nodes at the appropriate levels. Using a raw encoding of the permutation mapping integers to the nodes, and Munro et al.'s shortcut succinct index [1] to support the operations on it, yields the following result:

Theorem 6 *For any fixed ε, $n > 0$ and $f : [1 \ldots n] \to [1 \ldots n]$, there is a representation of f using $(1 + \varepsilon)n \lg n + O(1)$ bits of space to compute $f^k(i)$ in time within $O(1 + |f^k(i)|)$, for any integer k and for any integer $i \in [1 \ldots n]$.*

Open Problems

Other Measures of Disorder

Moffat and Petersson [7] list many measures of preorder and adaptive sorting techniques. Each measure explored above yields a compressed data structure for permutations supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time. Each adaptive sorting algorithm in the comparison model yields a compression scheme for permutations, but the encoding thus defined does not necessarily support the simple application of the permutation to a single element without decompressing the whole permutation nor the application of the inverse permutation. More work is required in order to decide whether there are compressed data structures for permutations, supporting the operators $\pi()$ and $\pi^{-1}()$ in sublinear time and using space proportional to the other preorder measures [6, 7] (e.g., `Reg`, `Exc`, `Block`, and `Enc`).

Sorting and Encoding Multisets

Munro and Spira [13] showed how to sort multisets through `MergeSort`, `Insertion Sort`, and `Heap Sort`, adapting them with counters to sort in time within $O(n(1 + \mathcal{H}(\langle m_1, \ldots, m_r \rangle)))$ where m_i is the number of occurrences of i in the multiset (note that this is orthogonal to the results described in this chapter that depend on the distribution of the lengths of monotone runs).

It seems easy to combine both approaches (e.g., on `MergeSort` in a single algorithm using both runs and counters), yet quite hard to *analyze* the complexity of the resulting algorithm and compressed data structure. The difficulty measure must depend not only on both the entropy of the partition into runs and the entropy of the partition of the values of the elements but also on the interaction of those partitions.

Compressed Data Structures Supporting $\pi^k()$

In Munro et al.'s "cycle" data structure [1] for supporting the operator $\pi^k()$ (Theorem 2), the raw encoding of the permutation σ representing the cycles of π can be replaced by any compressed data structure such as those described here, with the warning that the compressibility of σ depends not only on π but also on the order in which its cycles are placed in σ. The question if there is a compressed data structure supporting the operator $\pi^k()$ which takes advantage of this order is open.

Recommended Reading

1. Ian Munro J, Raman R, Raman V, Srinivasa Rao S (2012) Succinct representations of permutations and functions. Theorical Computer Science (TCS) 438:74–88
2. Barbay J, Navarro G (2013) On compressing permutations and adaptive sorting. Theorical Computer Science (TCS) 513:109–123
3. Barbay J, Fischer J, Navarro G (2012) LRM-trees: compressed indices, adaptive sorting, and compressed permutations. Theorical Computer Science (TCS) 459:26–41
4. Barbay J (2013) From time to space: fast algorithms that yield small and fast data structures. In: Brodnik A, López-Ortiz A, Raman V, Viola A (eds) Space-efficient data structures, streams, and algorithms (Ian-Fest). Volume 8066 of Lecture Notes in Computer Science. Springer, Heidelberg, pp 97–111
5. Ian Munro J, Raman V (1997) Succinct representation of balanced parentheses, static trees and planar graphs. In: IEEE symposium on Foundations Of Computer Science, Miami Beach, pp 118–126
6. Estivill-Castro V, Wood D (1992) A survey of adaptive sorting algorithms. ACM Computing Survey 24(4):441–476
7. Moffat A, Petersson O (1992) An overview of adaptive sorting. Aust Comput J 24(2):70–77
8. Levcopoulos C, Petersson O (1990) Sorting shuffled monotone sequences. In: Proceedings of the Scandinavian Workshop on Algorithm Theory (SWAT), Bergen. Springer, London, pp 181–191
9. Levcopoulos C, Petersson O (1994) Sorting shuffled monotone sequences. Information Computing 112(1):37–50
10. Barbay J, Claude F, Gagie T, Navarro G, Nekrich Y (2014) Efficient fully-compressed sequence representations. Algorithmica 69(1):232–268
11. Gabow HN, Bentley JL, Tarjan RE (1984) Scaling and related techniques for geometry problems. In: Proceedings of the Symposium on Theorical Computer (STOC), Washington, DC. ACM, pp 135–143
12. Elias P (1975) Universal codeword sets and representations of the integers. IEEE Transaction on Information Theory 21(2):194–203
13. Ian Munro J, Spira PM (1976) Sorting and searching in multisets. SIAM Journal of Computing 5(1):1–8

Succinct Data Structures for Parentheses Matching

Meng He
School of Computer Science, University of Waterloo, Waterloo, ON, Canada

Keywords

Succinct balanced parentheses

Years and Authors of Summarized Original Work

2001; Munro, Raman

Problem Definition

This problem is to design succinct representation of balanced parentheses in a manner in which a number of "natural" queries can be supported quickly, and use it to represent trees and graphs succinctly. The problem of succinctly representing balanced parentheses was initially proposed by Jacobson [6] in 1989, when he proposed *succinct data structures*, i.e., data structures that occupy space close to the information-theoretic

lower bound to represent them, while supporting efficient navigational operations. Succinct data structures provide solutions to manipulate large data in modern applications. The work of Munro and Raman [8] provides an optimal solution to the problem of balanced parentheses representation under the word RAM model, based on which they design succinct trees and graphs.

Balanced Parentheses

Given a balanced parenthesis sequence of length $2n$, where there are n opening parentheses and n closing parentheses, consider the following operations:

- `findclose`(i) (`findopen`(i)), the matching closing (opening) parenthesis for the opening (closing) parenthesis at position i;
- `excess`(i), the number of opening parentheses minus the number of closing parentheses in the sequence up to (and including) position i;
- `enclose`(i), the closest enclosing (matching parenthesis) pair of a given matching parenthesis pair whose opening parenthesis is at position i.

Trees

There are essentially two forms of trees. An *ordinal tree* is a rooted tree in which the children of a node are ordered and specified by their ranks, while in a *cardinal tree* of degree k, each child of a node is identified by a unique number from the set $\{1, 2, \cdots, k\}$. An *binary tree* is a cardinal tree of degree 2. The information-theoretic lower bound of representing an ordinal tree or binary tree of n nodes is $2n - o(n)$ bits, as there are $\binom{2n}{n}/(n+1)$ different ordinal trees or binary trees.

Consider the following operations on ordinal trees (a node is referred to by its preorder number):

- `child`(x, i), the ith child of node x for $i \geq 1$;
- `child_rank`(x), the number of left siblings of node x;
- `depth`(x), the depth of x, i.e., the number of edges in the rooted path to node x;
- `parent`(x), the parent of node x;
- `nbdesc`(x), the number of descendants of node x;
- `height`(x), the height of the subtree rooted at node x;
- `LCA`(x, y), the lowest common ancestor of node x and node y.

On binary trees, the operations `parent`, `nbdesc` and the following operations are considered:

- `leftchild`(x) (`rightchild`(x)), the left (right) child of node x.

Graphs

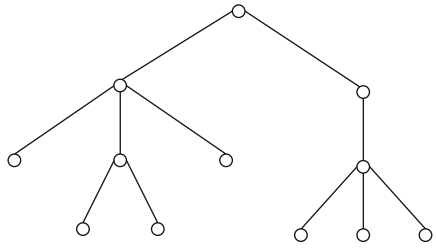
Consider an undirected graph G of n vertices and m edges. Bernhart and Kainen [1] introduced the concept of *page book embedding*. A *k-book embedding* of a graph is a topological embedding of it in a book of k pages that specifies the ordering of the vertices along the spine, and carries each edge into the interior of one page, such that the edges on a given page do not intersect. Thus, a graph with one page is an *outerplanar graph*. The *pagenumber* or *book thickness* [1] of a graph is the minimum number of pages that the graph can be embedded in. A very common type of graphs are planar graphs, and any planar graph can be embedded in at most four pages [15]. Consider the following operations on graphs:

- `adjacency`(x, y), whether vertices x and y are adjacent;
- `degree`(x), the degree of vertex x;
- `neighbors`(x), the neighbors of vertex x.

Key Results

All the results cited are under the word RAM model with word size $\Theta(\lg n)$ bits ($\lg n$ denotes $\lceil \log_2 n \rceil$), where n is the size of the problem considered.

Theorem 1 ([8]) *A sequence of balanced parentheses of length 2n can be represented using*



Balanced parentheses: ((( ) (( ) ( )) ( )) ((( ) ( ) ( ))))

Succinct Data Structures for Parentheses Matching, Fig. 1 An example of the balanced parenthesis sequence of a given ordinal tree

$2n + o(n)$ *bits to support the operations findclose, findopen, excess and enclose in constant time.*

There is a polymorphism between a balanced parenthesis sequence and an ordinal tree: when performing a depth-first traversal of the tree, output an opening parenthesis each time a node is visited, and a closing parenthesis immediately after all the descendants of a node are visited (see Fig. 1 for an example). The work of Munro and Raman proposes a succinct representation of ordinal trees using $2n + o(n)$ bits to support `depth`, `parent` and `nbdesc` in constant time, and `child`(x, i) in $O(i)$ time. Lu and Yeh have further extended this representation to support `child`, `child_rank`, `height` and `LCA` in constant time.

Theorem 2 ([8, 7]) *An ordinal tree of n nodes can be represented using* $2n + o(n)$ *bits to support the operations child, child_rank, parent, depth, nbdesc, height and LCA in constant time.*

A similar approach can be used to represent binary trees:

Theorem 3 ([8]) *A binary tree of n nodes can be represented using* $2n + o(n)$ *bits to support the operations leftchild, rightchild, parent and nbdesc in constant time.*

Finally, balanced parentheses can be used to represent graphs. To represent a one-page graph, the work of Munro and Raman proposes to list the vertices from left to right along the spine, and each node is represented by a pair of parentheses, followed by zero or more closing parentheses and then zero or more opening parentheses, where the number of closing (or opening) parentheses is equal to the number of adjacent vertices to its left (or right) along the spine (see Fig. 2 for an example). This representation can be applied to each page to represent a graph with pagenumber k.

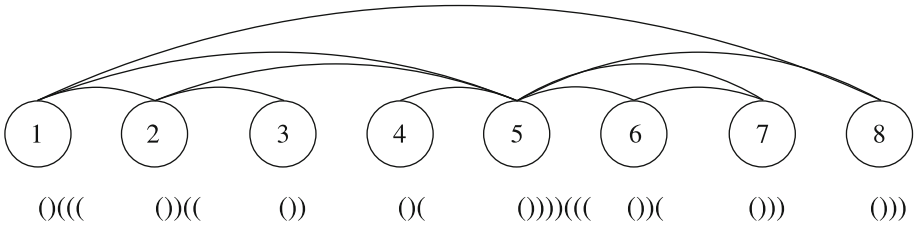
Theorem 4 ([8]) *An outerplanar graph of n vertices and m edges can be represented using* $2n + 2m + o(n + m)$ *bits to support operations adjacency and degree in constant time, and neighbors(x) in time proportional to the degree of x.*

Theorem 5 ([8]) *A graph of n vertices and m edges with pagenumber k can be represented using* $2kn + 2m + o(nk + m)$ *bits to support operations adjacency and degree in* $O(k)$ *time, and neighbors(x) in* $O(d(x) + k)$ *time where d(x) is the degree of x. In particular, a planar graph of n vertices and m nodes can be represented using* $8n + 2m + o(n)$ *bits to support operations adjacency and degree in constant time, and neighbors(x) in* $O(d(x))$ *time where d(x) is the degree of x.*

Applications

Succinct Representation of Suffix Trees

As a result of the growth of the textual data in databases and on the World Wide Web, and also applications in bioinformatics, various indexing techniques have been developed to facilitate pattern searching. Suffix trees [14] are a popular type of text indexes. A suffix tree is constructed over the suffixes of the text as a tree-based data structure, so that queries can be performed by searching the suffixes of the text. It takes $O(m)$ time to use a suffix tree to check whether an arbitrary pattern P of length m is a substring of a given text T of length n, and to count the number of the occurrences, occ, of P in T. $O(occ)$ additional time is required to list all the occurrences



Succinct Data Structures for Parentheses Matching, Fig. 2 An example of the balanced parenthesis sequence of a graph with one page

of P in T. However, a standard representation of a suffix tree requires somewhere between $4n \lg n$ and $6n \lg n$ bits, which is impractical for many applications.

By reducing the space cost of representing the tree structure of a suffix tree (using the work of Munro and Raman), Munro, Raman and Rao [9] have designed space-efficient suffix trees. Given a string of n characters over a fixed alphabet, they can represent a suffix tree using $n \lg n + O(n)$ bits to support the search of a pattern in $O(m + occ)$ time. To achieve this result, they have also extended the work of Munro and Raman to support various operations to retrieve the leaves of a given subtree in an ordinal tree. Based on similar ideas and by applying compressed suffix arrays [5], Sadakane [13] has proposed a different trade-off; his compressed suffix tree occupies $O(n \lg \sigma)$ bits, where σ is the size of the alphabet, and can support any algorithm on a suffix tree with a slight slowdown of a factor of polylog(n).

Succinct Representation of Functions

Munro and Rao [11] have considered the problem of succinctly representing a given function, $f : [n] \to [n]$, to support the computation of $f^k(i)$ for an arbitrary integer k. The straightforward representation of a function is to store the sequence $f(i)$, for $i = 0, 1, \ldots, n-1$. This takes $n \lg n$ bits, which is optimal. However, the computation of $f^k(i)$ takes $\Theta(k)$ time even in the easier case when k is positive. To address this problem, Munro and Rao [11] first extends the representation of balanced parenthesis to support the `next_excess`(i, k) operator, which returns the minimum j such that $j > i$ and `excess`$(j) = k$. They further use this operator to support the `level_anc`(x, i) operator on succinct ordinal trees, which returns the ith ancestor of node x for $i \geq 0$ (given a node x at depth d, its ith ancestor is the ancestor of x at depth $d - i$). Then, using succinct ordinal trees with the support for `level_anc`, they propose a succinct representation of functions using $(1 + \epsilon)n \lg n + O(1)$ bits for any fixed positive constant ϵ, to support $f^k(i)$ in constant time when $k > 0$, and $f^k(i)$ in $O(1 + |f^k(i)|)$ time when $k < 0$.

Multiple Parentheses and Graphs

Chuang et al. [3] have proposed to succinctly represent *multiple parentheses*, which is a string of $O(1)$ types of parentheses that may be unbalanced. They have extended the operations on balanced parentheses to multiple parentheses and designed a succinct representation. Based on the properties of canonical orderings for planar graphs, they have used multiple parentheses and the succinct ordinal trees to represent planar graphs. One of their main results is a succinct representation of planar graphs of n vertices and m edges in $2m + (5 + \epsilon)n + o(m + n)$ bits, for any constant $\epsilon > 0$, to support the operations supported on planar graphs in Theorem 5 in asymptotically the same amount of time. Chiang et al. [2] have further reduced the space cost to $2m + 3n + o(m + n)$ bits. In their paper, they have also shown how to support the operation `wrapped`(i), which returns the number of matching parenthesis pairs whose closest enclosing (matching parenthesis) pair is the pair whose opening parenthesis is at position i, in constant time on balanced parentheses. They

have used it to show how to support the operation `degree`(x), which returns the degree of node x (i.e., the number of its children), in constant time on succinct ordinal trees.

Open Problems

One open research area is to support more operations on succinct trees. For example, it is not known how to support the operation to convert a given node's rank in a preorder traversal into its rank in a level-order traversal.

Another open research area is to further reduce the space cost of succinct planar graphs. It is not known whether it is possible to further improve the encoding of Chiang et al. [2].

A third direction for future work is to design succinct representations of dynamic trees and graphs. There have been some preliminary results by Munro et al. [10] on succinctly representing dynamic binary trees, which have been further improved by Raman and Rao [12]. It may be possible to further improve these results, and there are other related dynamic data structures that do not have succinct representations.

Experimental Results

Geary et al. [4] have engineered the implementation of succinct ordinal trees based on balanced parentheses. They have performed experiments on large XML trees. Their implementation uses orders of magnitude less space than the standard pointed-based representation, while supporting tree traversal operations with only a slight slowdown.

Cross-References

- ▸ Compressed Suffix Array
- ▸ Compressed Text Indexing
- ▸ Rank and Select Operations on Bit Strings
- ▸ Text Indexing

Recommended Reading

1. Bernhart F, Kainen PC (1979) The book thickness of a graph. J Comb Theory B 27(3):320–331
2. Chiang Y-T, Lin C-C, Lu H-I (2005) Orderly spanning trees with applications. SIAM J Comput 34(4):924–945
3. Chuang RC-N, Garg A, He X, Kao M-Y, Lu H-I (2001) Compact encodings of planar graphs via canonical orderings and multiple parentheses. Comput Res Repos. cs.DS/0102005
4. Geary RF, Rahman N, Raman R, Raman V (2006) A simple optimal representation for balanced parentheses. Theor Comput Sci 368(3):231–246
5. Grossi R, Gupta A, Vitter JS (2003) High-order entropy-compressed text indexes. In: Farach-Colton M (ed) Proceedings of the 14th annual ACM-SIAM symposium on discrete algorithms. SIAM, Philadelphia, pp 841–850
6. Jacobson G (1989) Space-efficient static trees and graphs. In: Proceedings of the 30th annual IEEE symposium on foundations of computer science. IEEE, New York, pp 549–554
7. Lu H-I, Yeh C-C (2007) Balanced parentheses strike back. Accepted to ACM Trans Algorithms
8. Munro JI, Raman V (2001) Succinct representation of balanced parentheses and static trees. SIAM J Comput 31(3):762–776
9. Munro JI, Raman V, Rao SS (2001) Space efficient suffix trees. J Algorithms 39(2):205–222
10. Munro JI, Raman V, Storm AJ (2001) Representing dynamic binary trees succinctly. In: Rao Kosaraju S (ed) Proceedings of the 12th annual ACM-SIAM symposium on discrete algorithms. SIAM, Philadelphia, pp 529–536
11. Munro JI, Rao SS (2004) Succinct representations of functions. In: Díaz J, Karhumäki J, Lepistö A, Sannella D (eds) Proceedings of the 31st international colloquium on automata, languages and programming. Springer, Heidelberg, pp 1006–1015
12. Raman R, Rao SS (2003) Succinct dynamic dictionaries and trees. In: Baeten JCM, Lenstra JK, Parrow J, Woeginger GJ (eds) Proceedings of the 30th international colloquium on automata, languages and programming. Springer, Heidelberg, pp 357–368
13. Sadakane K (2007) Compressed suffix trees with full functionality. Theory Comput Syst. Online first. http://dx.doi.org/10.1007/s00224-006-1198-x
14. Weiner P (1973) Linear pattern matching algorithms. In: Proceedings of the 14th annual IEEE symposium on switching and automata theory. IEEE, New York, pp 1–11
15. Yannakakis M (1986) Four pages are necessary and sufficient for planar graphs. In: Hartmanis J (ed) Proceedings of the 18th annual ACM-SIAM symposium on theory of computing. ACM, New York, pp 104–108

Suffix Array Construction

Juha Kärkkäinen
Department of Computer Science, University of Helsinki, Helsinki, Finland

Keywords

Longest common prefix array; Suffix array; Suffix sorting; Text indexing

Years and Authors of Summarized Original Work

2006; Kärkkäinen, Sanders, Burkhardt

Problem Definition

The *suffix array* [4, 15] is the lexicographically sorted array of all the suffixes of a string. It is a popular text index structure with many applications. The subject of this entry is algorithms that construct the suffix array.

More precisely, the input to a suffix array construction algorithm is a *text string* $T = T[0 \ldots n) = t_0 t_1 \ldots t_{n-1}$, i.e., a sequence of n *characters* from an *alphabet* Σ. For $i \in [0 \ldots n]$, let S_i denote the *suffix* $T[i \ldots n) = t_i t_{i+1} \ldots t_{n-1}$. The output is the *suffix array* $SA[0 \ldots n]$ of T, a permutation of $[0 \ldots n]$ satisfying $S_{SA[0]} < S_{SA[1]} < \cdots < S_{SA[n]}$, where $<$ denotes the *lexicographical order* of strings.

Two specific models for the alphabet Σ are considered. An *ordered alphabet* is an arbitrary ordered set with constant time character comparisons. An *integer alphabet* is the integer range $[1 \ldots \sigma]$ for $\sigma = n^{\mathcal{O}(1)}$.

Many applications require that the suffix array is augmented with additional information, most commonly with the *longest common prefix* array $LCP[1 \ldots n]$. An entry $LCP[i]$ of the LCP array is the length of the longest common prefix of the suffixes $S_{SA[i]}$ and $S_{SA[i-1]}$. The *enhanced suffix array* [1] adds two more arrays to obtain a full range of text index functionalities.

There are other important text indexes, most notably suffix trees and compressed text indexes, covered in separate entries. Each of these indexes has their own construction algorithms, but they can also be constructed efficiently from each other. However, in this entry, the focus is on direct suffix array construction algorithms that do not rely on other text indexes.

Key Results

The naive approach to suffix array construction is to use a general sorting algorithm or an algorithm for sorting strings. However, any such algorithm has a worst-case time complexity $\Omega(n^2)$ because the total length of the suffixes is $\Omega(n^2)$.

The first efficient algorithms were based on the *doubling technique* of Karp, Miller, and Rosenberg [10]. The idea is to assign a *rank* to all substrings whose length is a power of two. The rank tells the lexicographic order of the substring among substrings of the same length. Given the ranks for substrings of length h, the ranks for substrings of length $2h$ can be computed using a radix sort step in linear time (doubling). The technique was first applied to suffix array construction by Manber and Myers [15]. The best practical algorithm based on the technique is by Larsson and Sadakane [14].

Theorem 1 (Manber and Myers [15]; Larsson and Sadakane [14]) *The suffix array can be constructed in $\mathcal{O}(n \log n)$ time, which is optimal for the ordered alphabet.*

Faster algorithms for the integer alphabet are based on a different technique, recursion. The basic procedure is as follows.

1. Sort a subset of the suffixes. This is done by constructing a shorter string, whose suffix array gives the order of the desired subset. The suffix array of the shorter string is constructed by recursion.
2. Extend the subset order to full order.

The technique first appeared in suffix tree construction [3], but 2003 saw the independent and simultaneous publication of three linear time suffix array construction algorithms based on the approach but not using suffix trees. Each of the three algorithms uses a different subset of suffixes requiring a different implementation of the second step.

Theorem 2 (Kärkkäinen, Sanders, and Burkhardt [8]; Kim et al. [12]; Ko and Aluru [13]) *The suffix array can be constructed in the optimal linear time for the integer alphabet.*

We will describe the algorithm of Kärkkäinen, Sanders, and Burkhardt [8] called DC3 in more detail. For $k \in \{0, 1, 2\}$, let $\mathcal{R}_k$ be the set of suffixes S_i such that $i \bmod 3 = k$. Let $\mathcal{R}_{12} = \mathcal{R}_1 \cup \mathcal{R}_2$ and define $\mathcal{R}_{01}$ and $\mathcal{R}_{02}$ symmetrically. For example, $\mathcal{R}_{12} = \{S_1, S_2, S_4, S_5, S_7, S_8, \dots\}$. The set $\mathcal{R}_{12}$ is the subset of suffixes sorted first. For $S_i \in \mathcal{R}_{12}$, let $\bar{S}_i$ be the lexicographical rank of S_i in $\mathcal{R}_{12}$. Given those lexicographical ranks, we can compare any two suffixes S_i and S_j in constant time using one of the following ways:

1. If $S_i, S_j \in \mathcal{R}_{12}$, compare the ranks $\bar{S}_i$ and $\bar{S}_j$.
2. If $S_i, S_j \in \mathcal{R}_{01}$, compare the pairs $\langle t_i, \bar{S}_{i+1}\rangle$ and $\langle t_j, \bar{S}_{j+1}\rangle$.
3. If $S_i, S_j \in \mathcal{R}_{02}$, compare the triples $\langle t_i, t_{i+1}, \bar{S}_{i+2}\rangle$ and $\langle t_j, t_{j+1}, \bar{S}_{j+2}\rangle$.

Furthermore, we can radix sort $\mathcal{R}_0$ in linear time by using $\langle t_i, \bar{S}_{i+1}\rangle$ to represent the suffix $S_i \in \mathcal{R}_0$. After this, we can merge $\mathcal{R}_0$ and $\mathcal{R}_{12}$, which takes linear time since we can compare suffixes in constant time.

We still need to describe how to sort $\mathcal{R}_{12}$. Let $\overline{t_i t_{i+1} t_{i+2}}$ be the lexicographical rank of the substring $t_i t_{i+1} t_{i+2}$ among all substrings of length three. Let

$$T_{12} = \overline{t_1 t_2 t_3}\ \overline{t_4 t_5 t_6}\ \overline{t_7 t_8 t_9} \dots \overline{t_2 t_3 t_4}\ \overline{t_5 t_6 t_7}\ \overline{t_8 t_9 t_{10}} \dots .$$

For example if $T =$ `yabbadabbado`, we have

$$T_{12} = \overline{\texttt{abb}}\ \overline{\texttt{ada}}\ \overline{\texttt{bba}}\ \overline{\texttt{do\$}}\ \overline{\texttt{bba}}\ \overline{\texttt{dab}}\ \overline{\texttt{bad}}\ \overline{\texttt{o\$\$}} = 12575648 ,$$

where \$ is a special padding symbol that does not appear in the text and is considered smaller than any normal character. Clearly, sorting the suffixes of T_{12} is equivalent to sorting the set $\mathcal{R}_{12}$. The suffixes of T_{12} are sorted by a recursive call to the algorithm itself. Since the recursive call is for a text of length at most $\lceil 2n/3 \rceil$ and everything outside the recursive call can be done in linear time, the total time complexity of DC3 is $\mathcal{O}(n)$.

The above algorithms and many other suffix array construction algorithms are surveyed in [18]. Worth mentioning among the more recent results are the linear time algorithms of Nong, Zhang, and Chan [17].

The $\Omega(n \log n)$ lower bound for the ordered alphabet mentioned in Theorem 1 comes from the sorting complexity of characters, since the initial characters of the sorted suffixes are the text characters in sorted order. Theorem 2 allows a generalization of this result. For any alphabet, one can first sort the characters of T, remove duplicates, assign a rank to each character, and construct a new string T' over the alphabet $[1 \dots n]$ by replacing the characters of T with their ranks. The suffix array of T' is exactly the same as the suffix array of T. Optimal algorithms for the integer alphabet then give the following result.

Theorem 3 *For any alphabet, the complexity of suffix array construction is the same as the complexity of sorting the characters of the string.*

The result extends to the related arrays.

Theorem 4 (Kasai et al. [11]; Abouelhoda, Kurtz, and Ohlebusch [1]) *The LCP array and the enhanced suffix array can be computed in linear time given the suffix array.*

One of the main advantages of suffix arrays over suffix trees is their smaller space requirement (by a constant factor), and a significant effort has been spent making construction algorithms space efficient, too. The best algorithms need very little extra space.

Theorem 5 (Kärkkäinen, Sanders, and Burkhardt [8]; Nong [16]) *For any* $v = \mathcal{O}(n^{2/3})$*, the suffix array can be constructed in* $\mathcal{O}(n(v + \log n))$ *time and* $\mathcal{O}(n/\sqrt{v})$ *extra space for the ordered alphabet and in* $\mathcal{O}(nv)$ *time and* $\mathcal{O}(n/\sqrt{v})$ *extra space or* $\mathcal{O}(n)$ *time and* $\mathcal{O}(\sigma)$ *extra space for the integer alphabet, where the extra space is the space needed in addition to the input (the string* T*) and the output (the suffix array).*

In the algorithm DC3 described above, all steps can be performed by sorting, prefix sums (assigning lexicographical ranks) and localized computation. This makes it straightforward to adapt to several parallel and hierarchical memory models of computation [8] including the following result for the standard external memory model.

Theorem 6 (Kärkkäinen, Sanders, and Burkhardt [8]) *The suffix array can be constructed in the optimal* $\mathcal{O}(\text{sort}(n))$ *I/Os in the standard external memory model, where* $\text{sort}(n)$ *is the I/O complexity of sorting n elements.*

The above algorithm can be modified to compute the LCP array too in the same I/O complexity [2, 7].

Applications

The suffix array is a simple and powerful text index structure with numerous applications; see [1] and Cross-References. The practical construction of many other text indexes usually starts with the suffix array construction. In particular, the Burrows–Wheeler transform, which is an important technique for text compression and the basis of many compressed text indexes, is easily computed from the suffix array.

Open Problems

Theoretically, the suffix array construction problem is essentially solved. The development of ever more efficient practical algorithms is still going on particularly for external memory and parallel computation. There is currently no external memory algorithm for computing the LCP array from the suffix array in $\mathcal{O}(\text{sort}(n))$ I/Os other than as a side effect of suffix array construction [6].

Experimental Results

Many papers on suffix array construction contain experimental results, but they are usually either out of date (e.g., [18]) or limited in scope (e.g., [16]). The most comprehensive comparison of algorithms is at https://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks. The best practical algorithms for large data are `divsufsort`, which is an $\mathcal{O}(n \log n)$ time algorithm combining several techniques, and `SAIS`, which is an implementation of the linear time algorithm by Gong, Zhang, and Chan [17] (see below for URLs to code). The comparison and the fastest implementation are by the same person, Yuta Mori, but the implementations are widely used and there are no substantial claims for other, faster algorithms.

There are also experiments for suffix array construction in external memory [2, 5] and for LCP array construction [2, 6, 9].

URLs to Code and Data Sets

The input to a suffix array construction algorithm is simply a text, so an abundance of data exists. Links to many text collections are provided at https://code.google.com/p/libdivsufsort/wiki/SACA_Benchmarks. Worth mentioning is also the Pizza&Chili site with its standard text corpus http://pizzachili.dcc.uchile.cl/texts.html and the repetitive text corpus http://pizzachili.dcc.uchile.cl/repcorpus.html.

Notable implementations of suffix array construction algorithms are available at https://code.google.com/p/libdivsufsort/, at https://sites.google.com/site/yuta256/sais, at http://panthema.net/2012/1119-eSAIS-Inducing-Suffix-and-LCP-Arrays-in-External-Memory/ [2], and at https://

www.cs.helsinki.fi/group/pads/SAscan.html [5]. The latter two work in external memory and provide (links to) LCP array construction too.

Cross-References

- Burrows-Wheeler Transform
- Compressed Suffix Array
- Suffix Trees and Arrays
- Suffix Tree Construction

Recommended Reading

1. Abouelhoda MI, Kurtz S, Ohlebusch E (2004) Replacing suffix trees with enhanced suffix arrays. J Discret Algorithms 2(1):53–86
2. Bingmann T, Fischer J, Osipov V (2013) Inducing suffix and LCP arrays in external memory. In: Sanders P, Zeh N (eds) Proceedings of the 15th meeting on algorithm engineering and experiments (ALENEX), New Orleans. SIAM, pp 88–102
3. Farach-Colton M, Ferragina P, Muthukrishnan S (2000) On the sorting-complexity of suffix tree construction. J ACM 47(6):987–1011
4. Gonnet G, Baeza-Yates R, Snider T (1992) New indices for text: PAT trees and PAT arrays. In: Frakes WB, Baeza-Yates R (eds) Information retrieval: data structures & algorithms. Prentice-Hall, Englewood Cliffs
5. Kärkkäinen J, Kempa D (2014) Engineering a lightweight external memory suffix array construction algorithm. In: Iliopoulos CS, Langiu A (eds) Proceedings of the 2nd international conference on algorithms for big data (ICABD), Palermo, pp 53–60
6. Kärkkäinen J, Kempa D (2014) LCP array construction in external memory. In: Gudmundsson J, Katajainen J (eds) Proceedings of the 13th symposium on experimental algorithms (SEA), Copenhagen. Lecture notes in computer science, vol 8504. Springer, pp 412–423
7. Kärkkäinen J, Sanders P (2003) Simple linear work suffix array construction. In: Baeten JCM, Lenstra JK, Parrow J, Woeginger GJ (eds) Proceedings of the 30th international conference on automata, languages and programming (ICALP), Eindhoven. Lecture notes in computer science, vol 2719. Springer, pp 943–955
8. Kärkkäinen J, Sanders P, Burkhardt S (2006) Linear work suffix array construction. J ACM 53(6):918–936
9. Kärkkäinen J, Manzini G, Puglisi SJ (2009) Permuted longest-common-prefix array. In: Kucherov G, Ukkonen E (eds) Proceedings of the 20th annual symposium on combinatorial pattern matching (CPM), Lille. Lecture notes in computer science, vol 5577. Springer, pp 181–192
10. Karp RM, Miller RE, Rosenberg AL (1972) Rapid identification of repeated patterns in strings, trees and arrays. In: Proceedings of the 4th annual ACM symposium on theory of computing (STOC), Denver. ACM, pp 125–136
11. Kasai T, Lee G, Arimura H, Arikawa S, Park K (2001) Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Proceedings of the 12th annual symposium on combinatorial pattern matching (CPM), Jerusalem. Lecture notes in computer science, vol 2089. Springer, pp 181–192
12. Kim DK, Sim JS, Park H, Park K (2005) Constructing suffix arrays in linear time. J Discret Algorithms 3(2–4):126–142
13. Ko P, Aluru S (2005) Space efficient linear time construction of suffix arrays. J Discret Algorithms 3(2–4):143–156
14. Larsson NJ, Sadakane K (2007) Faster suffix sorting. Theor Comput Sci 387(3):258–272
15. Manber U, Myers G (1993) Suffix arrays: a new method for on-line string searches. SIAM J Comput 22(5):935–948
16. Nong G (2013) Practical linear-time O(1)-workspace suffix sorting for constant alphabets. ACM Trans Inf Syst 31(3):Article 15, 15 pages
17. Nong G, Zhang S, Chan WH (2011) Two efficient algorithms for linear time suffix array construction. IEEE Trans Comput 60(10):1471–1484
18. Puglisi SJ, Smyth WF, Turpin A (2007) A taxonomy of suffix array construction algorithms. ACM Comput Surv 39(2):Article 4, 31 pages

Suffix Tree Construction

Jens Stoye
Faculty of Technology, Genome Informatics, Bielefeld University, Bielefeld, Germany

Keywords

Full-text index construction

Years and Authors of Summarized Original Work

1973; Weiner
1976; McCreight

1995; Ukkonen
2000; Farach-Colton, Ferragina, Muthukrishnan

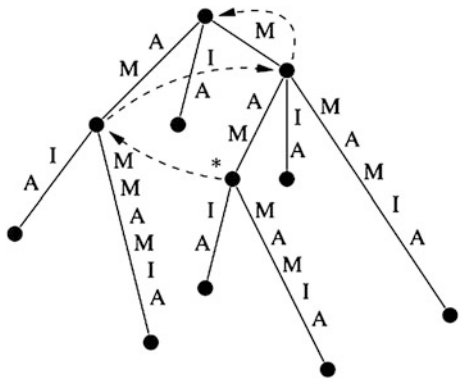
Problem Definition

The suffix tree is perhaps the best-known and most-studied data structure for string indexing with applications in many fields of sequence analysis. After its invention in the early 1970s, several approaches for the efficient construction of the suffix tree of a string have been developed for various models of computation. The most prominent of those that construct the suffix tree in main memory are summarized in this entry.

Notations

Given an alphabet Σ, a *trie* over Σ is a rooted tree whose edges are labeled with strings over Σ such that no two labels of edges leaving the same vertex start with the same symbol. A trie is *compacted* if all its internal vertices, except possibly the root, are branching. Given a finite string $S \in \Sigma^n$, the *suffix tree* of S, $T(S)$, is the compacted trie over Σ such that the concatenations of the edge labels along the paths from the root to the leaves are the suffixes of S. An example is given in Fig. 1.

The concatenation of the edge labels from the root to a vertex v of $T(S)$ is called the *path-label* of v, $P(v)$. For example, the path label of the vertex indicated by the asterisk in Fig. 1 is $P(*) = \text{MAM}$.



Suffix Tree Construction, Fig. 1 The suffix tree for the string $S = \text{MAMMAMIA}$. *Dashed arrows* denote suffix links that are employed by all efficient suffix tree construction algorithms

Constraints

The time complexity of constructing the suffix tree of a string S of length n depends on the size of the underlying alphabet Σ. It may be constant, it may be the alphabet of integers $\Sigma = \{1, 2, \ldots, n\}$, or it may be an arbitrary finite set whose elements can be compared in constant time. Note that the latter case reduces to the previous one if one maps the symbols of the alphabet to the set $\{1, \ldots, n\}$, though at the additional cost of sorting Σ.

Problem 1 (suffix tree construction)

INPUT: A finite string S of length n over an alphabet Σ.
OUTPUT: The suffix tree $T(S)$.

If one assumes that the outgoing edges at each vertex are lexicographically sorted, which is usually the case, the suffix tree allows retrieving the sorted order of S's characters in linear time. Therefore, suffix tree construction inherits the lower bounds from the problem complexity of sorting: $\Omega(n \log n)$ in the general alphabet case and $\Omega(n)$ for integer alphabets.

Key Results

Theorem 1 *The suffix tree of a string of length n can be represented in $O(n \log n)$ bits of space.*

This is easy to see since the number of leaves of $T(S)$ is at most n, and so is the number of internal vertices that, by definition, are all branching, as well as the number of edges. In order to see that each edge label can be stored in $O(\log n)$ bits of space, note that an edge label is always a substring of S. Hence it can be represented by a pair (l, r) consisting of *left pointer* l and *right pointer* r, if the label is $S[l, r]$.

Note that this space bound is not optimal since there are $|\Sigma|^n$ different strings and hence suffix trees, while $n \log n$ bits would allow to represent $n!$ different entities.

Theorem 2 *Suffix trees can be constructed in optimal time, in particular:*

1. For constant-size alphabet, the suffix tree $T(S)$ of a string S of length n can be constructed in $O(n)$ time [11–13]. For general alphabet, these algorithms require $O(n \log n)$ time.
2. For integer alphabet, the suffix tree of S can be constructed in $O(n)$ time [4,9].

Generally, there is a natural strategy to construct a suffix tree: Iteratively all suffixes are inserted into an initially empty structure. Such a strategy will immediately lead to a linear-time construction algorithm if each suffix can be inserted in constant time. Finding the correct position where to insert a suffix, however, is the main difficulty of suffix tree construction.

The first solution for this problem was given by Weiner in his seminal 1973 paper [13]. His algorithm inserts the suffixes from shortest to longest, and the insertion point is found in amortized constant time for constant-size alphabet, using rather a complicated amount of additional data structures. A simplified version of the algorithm was presented by Chen and Seiferas [3]. They give a cleaner presentation of the three types of links that are required in order to find the insertion points of suffixes efficiently, and their complexity proof is easier to follow. Since the suffix tree is constructed while reading the text from right to left, these two algorithms are sometimes called *anti-online* constructions.

A different algorithm was given in 1976 by McCreight [11]. In this algorithm the suffixes are inserted into the growing tree from longest to shortest. This simplifies the update procedure, and the additional data structure is limited to just one type of link: an internal vertex v with path label $P(v) = aw$ for some symbol $a \in \Sigma$ and string $w \in \Sigma^*$ has a *suffix link* to the vertex u with path label $P(u) = w$. In Fig. 1, suffix links are shown as dashed arrows. They often connect vertices above the insertion points of consecutively inserted suffixes, like the vertex with path-label "M" and the root, when inserting suffixes "MAMIA" and "AMIA" in the example of Fig. 1. This property allows reaching the next insertion point without having to search for it from the root of the tree, thus ensuring amortized constant time per suffix insertion. Note that since McCreight's algorithm treats the suffixes from longest to shortest and the intermediate structures are not suffix trees, the algorithm is not an online algorithm.

Another linear-time algorithm for constant-size alphabet is the online construction by Ukkonen [12]. It reads the text from left to right and updates the suffix tree in amortized constant time per added symbol. Again, the algorithm uses suffix links in order to quickly find the insertion points for the suffixes to be inserted. Moreover, since during a single update the edge labels of all leaf edges need to be extended by the new symbol, it requires a trick to extend all these labels in constant time: all the right pointers of the leaf edges refer to the same *end of string* value, which is just incremented.

An even stronger concept than online construction is *real-time* construction, where the worst-case (instead of amortized) time per symbol is considered. Amir et al. [1] present for general alphabet a suffix tree construction algorithm that requires $O(\log n)$ worst-case update time per every single input symbol when the text is read from right to left, and thus requires overall $O(n \log n)$ time, like the other algorithms for general alphabet mentioned so far. They achieve this goal using a binary search tree on the suffixes of the text, enhanced by additional pointers representing the lexicographic and the textual order of the suffixes, called *Balanced Indexing Structure*. This tree can be constructed in $O(\log\ n)$ worst-case time per added symbol and allows maintaining the suffix tree in the same time bound.

The first linear-time suffix tree construction algorithm for integer alphabets was given by Farach-Colton [4]. It uses the so-called *odd-even technique* that proceeds in three steps:

1. Recursively compute the compacted trie of all suffixes of S beginning at odd positions, called the *odd tree* T_o.
2. From T_o compute the *even tree* T_e, the compacted trie of the suffixes beginning at even positions in S.

3. Merge T_o and T_e into the whole suffix tree $T(S)$.

The basic idea of the first step is to encode pairs of characters as single characters. Since at most $n/2$ different such characters can occur, these can be radix-sorted and range-reduced to an alphabet of size $n/2$. Thus, the string S of length n over the integer alphabet $\Sigma = \{1, \ldots, n\}$ is translated in $O(n)$ time into a string S' of length $n/2$ over the integer alphabet $\Sigma' = \{1, \ldots, n/2\}$. Applying the algorithm recursively to this string yields the suffix tree of S'. After translating the edge labels from substrings of S' back to substrings of S, some vertices may exist with outgoing edges whose labels start with the same symbol, because two distinct symbols from Σ' may be pairs with the same first symbol from Σ. In such cases, by local modifications of edge labels or adding additional vertices, the trie property can be regained and the desired tree T_o is obtained.

In the second step, the odd tree T_o from the first step is used to generate the lexicographically sorted list (*lex-ordering* for short) of the suffixes starting at odd positions. Radix-sorting these with the characters at the preceding even positions as keys yields a lex-ordering of the even suffixes in linear time. Together with the longest common prefixes (lcps) of consecutive positions that can be computed in linear time from T_o using constant-time lowest common ancestor queries and the identity

$$\mathrm{lcp}(l_{2i}, l_{2j}) = \begin{cases} \mathrm{lcp}(l_{2i+1}, l_{2j+1}) + 1 & \text{if } S[2i] = S[2j] \\ 0 & \text{otherwise} \end{cases}$$

this ordering allows reconstructing the even tree T_e in linear time.

In the third step, the two tries T_o and T_e are merged into the suffix tree $T(S)$. Conceptually, this is a straightforward procedure: the two tries are traversed in parallel, and every part that is present in one or both of the two trees is inserted in the common structure. However, this procedure is simple only if edges are traversed character by character such that common and differing parts can be observed directly. Such a traversal would, however, require $O(n^2)$ time in the worst case, impeding the desired overall linear running time. Therefore, Farach-Colton suggests to use an oracle that tells for an edge of T_o and an edge of T_e the length of their common prefix.

However, the suggested oracle may overestimate this length, and that is why sometimes the tree generated must be corrected, called *unmerging*. The full details of the oracle and the unmerging procedure can be found in [4].

Overall, if $T(n)$ is the time it takes to build the suffix tree of a string $S \in \{1, \ldots, \mathrm{n}\}^n$, the first step takes $T(n/2) + O(n)$ time and the second and third steps take $O(n)$ time; thus the whole procedure takes $O(n)$ overall time on the RAM model.

Another linear-time construction of suffix trees for integer alphabets can be achieved via linear-time construction of suffix arrays together with longest common prefix tabulation, as described by Kärkkäinen and Sanders in [9].

All previously mentioned algorithms construct the suffix tree in main memory. However, since the data structure may become very large in practice, also methods for building the suffix tree in secondary memory have been studied. Possibly the simplest way is to first construct the suffix array A and the LCP array on disk, as described in the entry ▸ Suffix Array Construction. When this is done, it is only a small final step to construct the suffix tree [4]. The idea is to construct the tree in n phases from left to right, such that after phase i the suffix tree of the strings $A[1], \ldots, A[i]$ has been constructed. Simultaneously, an external-memory stack containing the nodes on the path leading from the root to $A[i]$ is maintained. In phase $i+1$, first, the leaf representing string $A[i+1]$ is created, and then all nodes are popped from the stack whose string length is strictly

greater than LCP[i]. Next, a new node with string depth LCP[i] is created (unless it already exists) whose parent is the top element of the stack and whose children are the last popped element and the new leaf. This new node and the new leaf are finally pushed on the stack. Keeping the two top pages of the stack in internal memory, the algorithm executes a total of $O(n)$ pop and push operations and therefore uses a total of $O(n/B)$ time, where B is the external memory block size.

Other more direct ways to construct the suffix tree on disk have also been developed, e.g., [14, 15].

In some applications the so-called *generalized* suffix tree of several strings is used, a dictionary obtained by constructing the suffix tree of the concatenation of the contained strings. An important question that arises in this context is that of dynamically updating the tree upon insertion and deletion of strings from the dictionary. More specifically, since edge labels are stored as pairs of pointers into the original string, when deleting a string from the dictionary, the corresponding pointers may become invalid and need to be updated. An algorithm to solve this problem in amortized linear time was given by Fiala and Greene [6], and a linear worst-case (and hence real-time) algorithm was given by Ferragina et al. [5].

Applications

The suffix tree supports many applications, most of them in optimal time and space, including exact string matching, set matching, longest common substring of two or more sequences, all-pairs suffix-prefix matching, repeat finding, and text compression. These and several other applications, many of them from bioinformatics, are given in [2] and [8].

Open Problems

Some theoretical questions regarding the expected size and branching structure of suffix trees under more complicated than i. i. d. sequence models are still open. Currently most of the research has moved toward more space-efficient data structures like suffix arrays and compressed string indices or the Burrows-Wheeler Transform.

Experimental Results

Suffix trees are infamous for their high memory requirements. The practical space consumption is between 9 and 11 times the size of the string to be indexed, even in the most space-efficient implementations known [7, 10]. Moreover, [7] also shows that suboptimal algorithms like the very simple quadratic-time *write-only top-down* (WOTD) algorithm can outperform optimal algorithms on many real-world instances in practice, if carefully engineered.

URLs to Code and Data Sets

Several sequence analysis libraries contain code for suffix tree construction. For example, Strmat (http://www.cs.ucdavis.edu/~gusfield/strmat.html) by Gusfield et al. contains implementations of Weiner's and Ukkonen's algorithm. An implementation of the WOTD algorithm by Kurtz can be found at (http://bibiserv.techfak.uni-bielefeld.de/wotd).

Cross-References

- Burrows-Wheeler Transform
- Compressed Suffix Trees
- Suffix Array Construction
- Suffix Trees and Arrays

Recommended Reading

1. Amir A, Kopelowitz T, Lewenstein M, Lewenstein N (2005) Towards real-time suffix tree construction. In: Proceedings of the 12th international symposium on string processing and information retrieval (SPIRE 2005). LNCS, vol 3772. Springer, Berlin, pp 67–78
2. Apostolico A (1985) The myriad virtues of subword trees. In: Apostolico A, Galil Z (eds) Combinatorial algorithms on words. NATO ASI Series, vol F12. Springer, Berlin, pp 85–96
3. Chen MT, Seiferas J (1985) Efficient and elegant subword tree construction. In: Apostolico A, Galil Z (eds) Combinatorial algorithms on words. Springer, New York
4. Farach-Colton M, Ferragina P, Muthukrishnan S (2000) On the sorting-complexity of suffix tree construction. J ACM 47(6):987–1011
5. Ferragina P, Grossi R, Montangero M (1998) A note on updating suffix tree labels. Theor Comput Sci 201:249–262
6. Fiala ER, Greene DH (1989) Data compression with finite windows. Commun ACM 32:490–505

7. Giegerich R, Kurtz S, Stoye J (2003) Efficient implementation of lazy suffix trees. Softw Pract Exp 33:1035–1049
8. Gusfield D (1997) Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York
9. Kärkkäinen J, Sanders P (2003) Simple linear work suffix array construction. In: Proceedings of the 30th international colloquium on automata, languages, and programming (ICALP 2003). LNCS, vol 2719. Springer, Berlin, pp 943–955
10. Kurtz S (1999) Reducing the space requirements of suffix trees. Softw Pract Exp 29:1149–1171
11. McCreight EM (1976) A space-economical suffix tree construction algorithm. J ACM 23:262–272
12. Ukkonen E (1995) On-line construction of suffix trees. Algorithmica 14:249–260
13. Weiner P (1973) Linear pattern matching algorithms. In: Proceedings of the 14th annual IEEE symposium on switching and automata theory. IEEE Press, New York, pp 1–11
14. Cheung C-F, Yu JX, Lu H (2005) Constructing suffix tree for gigabyte sequences with megabyte memory. IEEE Trans Knowl. Data Eng. 17:90–105
15. Tian Y, Tata S, Hankins RA, Patel JM (2005) Practical methods for constructing suffix trees. VLDB J 14:281–299

Suffix Tree Construction in Hierarchical Memory

Paolo Ferragina
Department of Computer Science, University of Pisa, Pisa, Italy

Keywords

Full-text index construction; String B-tree construction; Suffix array construction; Suffix tree construction

Years and Authors of Summarized Original Work

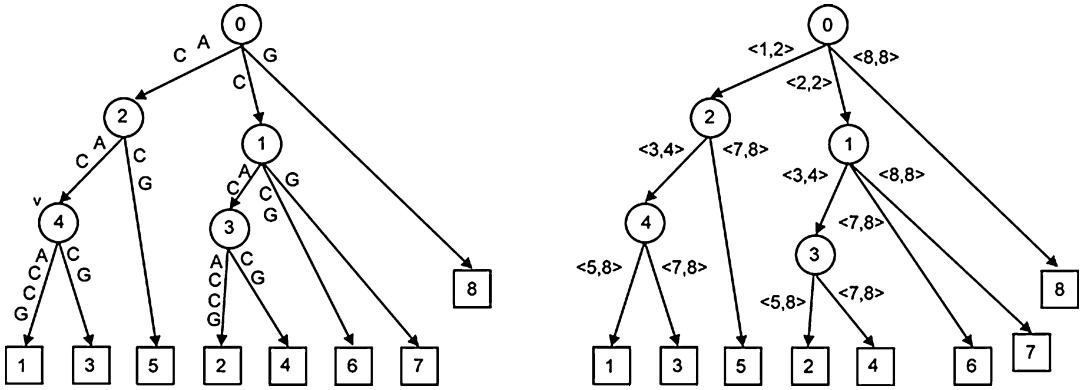
2000; Farach-Colton, Ferragina, Muthukrishnan

Problem Definition

The suffix tree is the ubiquitous data structure of combinatorial pattern matching myriad of situations – just to cite a few, searching, data compression and mining, and bioinformatics [7]. In these applications, the large data sets now available involve the use of numerous memory levels which constitute the storage medium of modern PCs: L1 and L2 caches, internal memory, multiple disks, and remote hosts over a network. The power of this memory organization is that it may be able to offer the expected access time of the fastest level (i.e., cache) while keeping the average cost per memory cell near the one of the cheapest level (i.e., disk), provided that data are properly *cached* and *delivered* to the requiring algorithms. Neglecting questions pertaining to the cost of memory references may even prevent the use of algorithms on large sets of input data. Engineering research is presently trying to improve the input/output subsystem to reduce the impact of these issues, but it is very well known [20] that the improvements achievable by means of a *proper arrangement of data* and a properly *structured algorithmic computation* abundantly surpass the best-expected technology advancements.

The Model of Computation

In order to reason about algorithms and data structures operating on hierarchical memories, it is necessary to introduce a *model of computation* that grasps the essence of real situations so that algorithms that are good in the model are also good in practice. The model considered here is the *external-memory model* [20], which received much attention because of its simplicity and reasonable accuracy. A computer is abstracted to consist of *two memory levels*: the internal memory of size M and the (unbounded) disk memory which operates by reading/writing data in blocks of size B (called *disk pages*). The performance of algorithms is then evaluated by counting (a) the number of disk accesses (I/Os), (b) the internal running time (CPU time), and (c) the number of disk pages occupied by the data structure or used by the algorithm as its working space. This simple model suggests, correctly, that a good external-memory algorithm should exploit both *spatial locality* and *temporal locality*. Of course, "I/O" and "two-level view" refer to any two levels



Suffix Tree Construction in Hierarchical Memory, Fig. 1 The suffix tree of $S = \texttt{ACACACCG}$ on the *left*, and its compact edge-encoding on the *right*. The endmarker # is not shown. Node v spells out the string *ACAC*. Each internal node stores the length of its associated string, and each leaf stores the starting position of its corresponding suffix

of the memory hierarchy with their parameters M and B properly set.

Notation

Let $S[1, n]$ be a string drawn from alphabet Σ, and consider the notation: S_i for the ith *suffix* of string S, $\texttt{lcp}(\alpha, \beta)$ for the *longest common prefix* between the two strings α and β, and $\texttt{lca}(u, v)$ for the *lowest common ancestor* between two nodes u and v in a tree.

The suffix tree of $S[1, n]$, denoted hereafter by $\mathcal{T}_S$, is a tree that stores all suffixes of $S\#$ in a compact form, where $\# \notin \Sigma$ is a special character (see Fig. 1). $\mathcal{T}_S$ consists of n leaves, numbered from 1 to n, and any root-to-leaf path spells out a suffix of $S\#$. The endmarker # guarantees that no suffix is the prefix of another suffix in $S\#$. Each internal node has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can begin with the same character, and sibling edges are ordered lexicographically according to that character. Edge labels are encoded with pairs of integers – say $S[x, y]$ is represented by the pair $\langle x, y \rangle$. As a result, all $\Theta(n^2)$ substrings of S can be represented in $O(n)$ optimal space by $\mathcal{T}_S$'s structure and edge encoding. Furthermore, the rightward scan of the suffix-tree leaves gives the ordered set of S's suffixes, also known as the *suffix array* of S [13]. Notice that the case of a large string collection $\Delta = \{S^1, S^2, \ldots, S^k\}$ *reduces to* the case of one long string $S = S^1 \#_1 S^2 \#_2 \cdots S^k \#_k$, where $\#_i \notin \Sigma$ are special symbols.

Numerous algorithms are known that build the suffix tree optimally in the RAM model (see [3] and references therein). However, most of them exhibit a marked absence of *locality of references* and thus elicit many I/Os when the size of the indexed string is too large to be fit into the internal memory of the computer. This is a serious problem because the slow performance of these algorithms can prevent the suffix tree being used even in medium-scale applications. This encyclopedia's entry surveys algorithmic solutions that deal efficiently with the *construction of suffix trees over large string collections* by executing an optimal number of I/Os. Since it is assumed that the edges leaving a node in $\mathcal{T}_S$ are lexicographically sorted, sorting is an obvious *lower bound* for building suffix trees (consider the suffix tree of a permutation!). The presented algorithms have sorting as their bottleneck, thus establishing that *the complexity of sorting and suffix tree construction match*.

Key Results

Designing a disk-efficient approach to suffix-tree construction has found efficient solutions only in the last few years [4]. The present section surveys

DIVIDE-AND-CONQUER ALGORITHM

(1) Construct the string $S'[j]$ = rank of $\langle S[2j], S[2j+1]\rangle$, and recursively compute $\mathcal{T}_{S'}$.
(2) Derive from $\mathcal{T}_{S'}$ the compacted trie $\mathcal{T}_o$ of all suffixes of S beginning at odd positions.
(3) Derive from $\mathcal{T}_o$ the compacted trie $\mathcal{T}_e$ of all suffixes of S beginning at even positions.
(4) Merge $\mathcal{T}_o$ and $\mathcal{T}_e$ into the whole suffix tree $\mathcal{T}_S$, as follows:
  (4.1) Overmerge $\mathcal{T}_o$ and $\mathcal{T}_e$ into the tree $\mathcal{T}_M$.
  (4.2) Partially unmerge $\mathcal{T}_M$ to get $\mathcal{T}_S$.

Suffix Tree Construction in Hierarchical Memory, Fig. 2 The algorithm that builds the suffix tree directly

SUFFIXARRAY-BASED ALGORITHM

(1) Construct the suffix array $\mathcal{A}_S$ and the array lcp_S of the string S.
(2) Initially set $\mathcal{T}_S$ as a single edge connecting the root to a leaf pointing to suffix $\mathcal{A}_S[1]$.
(2) For $i = 2,\ldots, n$:
  (2.1) Create a new leaf ℓ_i that points to the suffix $\mathcal{A}_S[i]$.
  (2.2) Walk up from ℓ_{i-1} until a node u_i is met whose string-length x_i is $\leq \text{lcp}_S[i]$.
  (2.3) If $x_i = \text{lcp}_S[i]$, leaf ℓ_i is attached to u_i.
  (2.4) If $x_i < \text{lcp}_S[i]$, create node u'_i with string-length x_i, attach it to u_i and leaf ℓ_i to u'_i;

Suffix Tree Construction in Hierarchical Memory, Fig. 3 The algorithm that builds the suffix tree passing through the suffix array

two theoretical approaches which achieve the best (optimal!) I/O-bounds in the worst case; the next section will discuss some practical solutions.

The first algorithm is based on a *Divide-and-Conquer* approach that allows us to reduce the construction process to external-memory sorting and few low-I/O primitives. It builds the suffix tree $\mathcal{T}_S$ by executing four (macro)steps, detailed in Fig. 2. It is not difficult to implement the first three steps in $\text{Sort}(n) = O(\frac{n}{B}\log_{M/B}\frac{n}{B})$ I/Os [20]. The last (merging) step is the most difficult one and its I/O-complexity bounds the cost of the overall approach. Farach-Colton et al. [3] propose an elegant merge for $\mathcal{T}_o$ and $\mathcal{T}_e$: substep (4.1) temporarily relaxes the requirement of getting $\mathcal{T}_S$ in one shot, and thus it blindly (over)merges the paths of $\mathcal{T}_o$ and $\mathcal{T}_e$ by comparing edges only via their first characters; then substep (4.2) refixes $\mathcal{T}_M$ by detecting and undoing in an I/O-efficient manner the (over)merged paths. Note that the time and I/O-complexity of this algorithm follow a nice recursive relation: $T(n) = T(n/2) + O(\text{Sort}(n))$.

Theorem 1 (Farach-Colton et al. [5]) *Given an arbitrary string $S[1,n]$, its suffix tree can be constructed in $O(\text{Sort}(n))$ I/Os, $O(n\log n)$ time and using $O(n/B)$ disk pages.*

The second algorithm [10] is deceptively simple, elegant, and I/O optimal and applies successfully to the construction of other indexing data structures, like the string Btree [5]. The key idea is to derive $\mathcal{T}_S$ from the suffix array $\mathcal{A}_S$ and from the *lcp* array, which stores the longest-common-prefix length of adjacent suffixes in $\mathcal{A}_S$. Its pseudocode is given in Fig. 3. Note that step (1) may deploy any external-memory algorithm for suffix array construction: used here is the elegant and optimal *Skew* algorithm of [9] which takes $O(\text{Sort}(n))$ I/Os. Step (2) takes a total of $O(n/B)$ I/Os by using a stack that stores the nodes on the current rightmost path of $\mathcal{T}_S$ in reversed order, i.e., leaf ℓ_i is on top. Walking upward, splitting edges or attaching nodes in $\mathcal{T}_S$ boils down to popping/pushing nodes from this stack. As a result, the time and I/O-complexity of this algorithm follow the recursive relation: $T(n) = T(2n/3) + O(\text{Sort}(n))$.

Theorem 2 (Kärkkäinen and Sanders 2003, see [10]) *Given an arbitrary string $S[1,n]$, its suffix tree can be constructed in $O(Sort(n))$ I/Os, $O(n \log n)$ time and using $O(n/B)$ disk pages.*

It is not evident which one of these two algorithms is better in practice [10]. The first one exploits a recursion with parameter 1/2 but incurs a large space overhead because of the management of the tree topology; the second one is more space efficient and easier to implement, but exploits a recursion with parameter 2/3.

Applications

The reader is referred to [4] and [7] for a long list of applications of large suffix trees and to [6, 18] for practical implementations.

Open Problems

The recent theoretical and practical achievements mean the idea that "suffix trees are not practical except when the text size to handle is so small that the suffix tree fits in internal memory" is no longer the case [15]. Given a suffix tree, it is known now (see, e.g., [4, 11]) how to map it onto a disk-memory system in order to allow I/O-efficient traversals for subsequent pattern searches. A fortiori, suffix-tree storage, and construction are challenging problems that need further investigation.

Space optimization is closely related to time optimization in a disk-memory system, so the design of *succinct* suffix-tree implementations is a key issue in order to scale to gigabytes of data in reasonable time. This topic is an active area of theoretical research with many fascinating solutions (see, e.g., [16] and the many papers that followed it), which need further exploration in the practical setting.

It is theoretically challenging to design a suffix-tree construction algorithm that takes optimal I/Os and space proportional to the *entropy* of the indexed string. The more compressible is the string, the lighter should be the space requirement of this algorithm. Some results are known [8, 11, 12], but both issues of compression and I/Os have been tackled jointly only recently [6], but more results are foreseen.

Experimental Results

The interest in building large suffix trees arose in the last few years because of the recent advances in sequencing technology, which have allowed the rapid accumulation of DNA and protein data. Some recent papers [1, 2, 9, 17, 18] proposed new practical algorithms that allow us to scale to Gbps/hours. Surprisingly enough, these algorithms are based on *disk-inefficient* schemes, but they properly select the insertion order of the suffixes and exploit carefully the internal memory as a buffer, so that their performance does not suffers significantly from the theoretical I/O-bottleneck.

In [9] the authors propose an *incremental* algorithm, called *PrePar*, which performs multiple passes over the string S and constructs the suffix tree for a *subrange* of suffixes at each pass. For a user-defined parameter q, a suffix subrange is defined as the set of suffixes prefixed by the same q-long string. Suffix subranges induce subtrees of $\mathcal{T}_S$ which can thus be built independently and evicted from internal memory as they are completed. The experiments reported in [9] successfully index 286 Mbps using 2 Gb internal memory.

In [2] the authors propose an improved version of *PrePar*, called *DynaCluster*, that deploys a *dynamic* technique to identify suffix subranges. Unlike *Prepar*, *DynaCluster* does not scan over and over the string S, but it starts from the q-based subranges and then splits them recursively in a DFS-manner if their size is larger than a fixed threshold τ. Splitting is implemented by looking at the next q characters of the suffixes in the subrange. This clustering and lazy-DFS visit of $\mathcal{T}_S$ significantly reduce the number of I/Os incurred by the frequent edge-splitting operations that occur during the suffix-tree construction process and allow it to cope efficiently with skew data.

As a result, *DynaCluster* constructs suffix trees for 200 Mbps with only 16 Mb internal memory.

In [17] authors improved the space requirement and the buffering efficiency, thus being able to construct a suffix tree of 3 Gbps in 30 h, whereas [1] improved the I/O behavior of RAM-algorithms for online suffix-tree construction, by devising a novel low-overhead buffering policy. More recently [14] introduced a new technique, called Elastic Range (ERA), which partitions the tree construction process horizontally and vertically and minimizes I/Os by dynamically adjusting the horizontal partitions independently for each vertical partition, based on the evolving shape of the tree and the available internal memory. This technique is specialized to work also for shared-memory and shared-disk multi-core systems and for parallel shared-nothing architectures. ERA indexes the entire human genome in 19 min on a commodity desktop PC. For comparison, the fastest existing method needs 15 min using 1024 CPUs on an IBM BluGene supercomputer.

Finally [19] observed that increasing memory sizes of current commodity PCs and servers enhance the impact of in-memory tasks on performance. So it is imperative nowadays to reassess the performance of in-memory algorithms and to propose new algorithms that incorporate the characteristics of modern hardware architectures, such as multilevel memory hierarchy and chip multiprocessors (CMPs). Starting from these premises the authors proposed cache-conscious suffix-tree construction algorithms that are tailored to CMP architectures, using novel sample-based cache-partitioning techniques that improved cache performance and exploited on-chip parallelism of CMPs thus achieving satisfactory speedups with increasing number of cores.

Cross-References

- ▸ Cache-Oblivious Sorting
- ▸ Suffix Array Construction
- ▸ Suffix Tree Construction
- ▸ Text Indexing

Recommended Reading

1. Bedathur SJ, Haritsa JR (2004) Engineering a fast online persistent suffix tree construction. In: Proceedings of the 20th international conference on data engineering, Boston, pp 720–731
2. Cheung C, Yu J, Lu H (2005) Constructing suffix tree for gigabyte sequences with megabyte memory. IEEE Trans Knowl Data Eng 17:90–105
3. Farach-Colton M, Ferragina P, Muthukrishnan S (2000) On the sorting-complexity of suffix tree construction. J ACM 47:987–1011
4. Ferragina P (2005) Handbook of computational molecular biology. In: Computer and information science series, ch. 35 on "String search in external memory: algorithms and data structures". Chapman & Hall/CRC, Florida
5. Ferragina P, Grossi R (1999) The string Btree: a new data structure for string search in external memory and its applications. J ACM 46:236–280
6. Ferragina P, Gagie T, Manzini G (2012) Lightweight data indexing and compression in external memory. Algorithmica 63(3):707–730
7. Gusfield D (1997) Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press, Cambridge
8. Hon W, Sadakane K, Sung W (2009) Breaking a time-and-space barrier in constructing full-text indices. SIAM J Comput 38(6):2162–2178
9. Hunt E, Atkinson M, Irving R (2002) Database indexing for large DNA and protein sequence collections. Int J Very Large Data Bases 11:256–271
10. Kärkkäinen J, Sanders P, Burkhardt S (2006) Linear work suffix array construction. J ACM 53:918–936
11. Ko P, Aluru S (2007) Optimal self-adjusting trees for dynamic string data in secondary storage. In: Symposium on string processing and information retrieval (SPIRE), Santiago. LNCS, vol 4726, pp 184–194. Springer, Berlin
12. Mäkinen V, Navarro G (2008) Dynamic entropy-compressed sequences and full-text indexes. ACM Trans Algorithm 4(3)
13. Manber U, Myers G (1993) Suffix arrays: a new method for on-line string searches. SIAM J Comput 22:935–948
14. Mansour E, Allam A, Skiadopoulos S, Kalnis P (2011) ERA: efficient serial and parallel suffix tree construction for very long strings. PVLDB 5(1):49–60
15. Navarro G, Baeza-Yates R (2000) A hybrid indexing method for approximate string matching. J Discr Algorithms 1:21–49
16. Navarro G, Mäkinen V (2007) Compressed full text indexes. ACM Comput Surv 39(1): Article no 2
17. Tian Y, Tata S, Hankins RA, Patel JM (2005) Practical methods for constructing suffix trees. VLDB J 14(3):281–299

18. Thomo A, Barsky M, Stege U (2010) A survey of practical algorithms for suffix tree construction in external memory. Softw Pract Experience 40(11):965–988
19. Tsirogiannis D, Koudas N (2010) Suffix tree construction on modern hardware. In: Proceedings of the 13th international conference on extending database technology (EDBT), Lausanne, pp 263–274
20. Vitter J (2002) External memory algorithms and data structures: dealing with MASSIVE DATA. ACM Comput Surv 33:209–271

Suffix Trees and Arrays

Alberto Apostolico[1] and Fabio Cunial[2]
[1]College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
[2]Department of Computer Science, Helsinki Institute for Information Technology (HIIT), University of Helsinki, Helsinki, Finland

Keywords

Full-text indexing; Pattern matching; String searching; Suffix array; Suffix tree

Years and Authors of Summarized Original Work

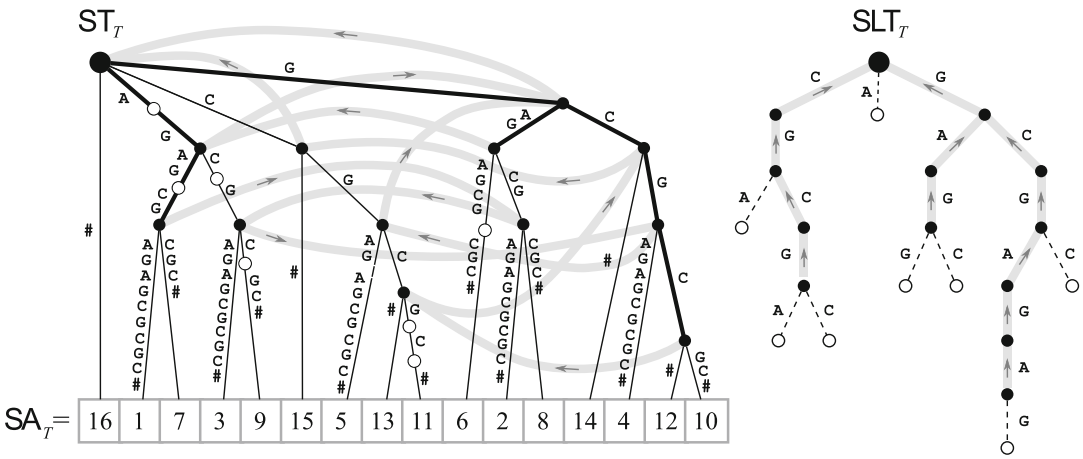
1973; McCreight
1973; Weiner
1993; Manber, Myers
1995; Ukkonen

The suffix tree is one of the oldest full-text inverted indexes and one of the most persistent subjects of study in the theory of algorithms. With extensions and refinements, including succinct and compressed variants that provide some of its expressive power in smaller space, it constitutes a fundamental conceptual tool in the design of string algorithms. The companion structure represented by the suffix array is as powerful as the suffix tree in many applications, but it requires significantly less space. The uses of these data structures are so numerous that it is difficult to account for all of them, while even more are being discovered. Salient applications include searching for a pattern in a text in time proportional to the size of the pattern, various computations on regularities such as repeats and palindromes within a text, statistical tables of substring occurrences, data compression by textual substitution, as well as ancillary yet fundamental tasks in string searching with errors, and more.

Problem Definition

It is well known that searching among n keys in an unsorted table takes optimal linear time. When multiple searches are expected, however, it becomes worth to sort the table once and for all, whereby each subsequent search will require only logarithmic time. It is similarly possible to build an inverted index on a long text so that the search for any query string will take time proportional to the length of the query rather than that of the text. It turns out that the data structures built for this purpose support many more applications, which are the topic of this entry.

Formally, let T be a string of length n on alphabet $\Sigma = [1 \dots \sigma]$, let $\underline{T}$ be its reverse, and let $\# \notin \Sigma$ be a shorthand for zero. To simplify the exposition, we assume throughout that σ is a constant. The *suffix tree* $\mathsf{ST}_T = (\perp, V, E)$ of T is a tree rooted at node $\perp \in V$ with set of nodes V and set of labeled edges E (Fig. 1, left). Edge labels are pointers to substrings of $T\#$: we denote by $\ell(e)$, and equivalently by $\ell(u, v)$, the label of edge $e = (u, v) \in E$, and we denote by $\ell(v)$ the string $\ell(\perp, v_1) \cdot \ell(v_1, v_2) \cdot \dots \cdot \ell(v_{k-1}, v)$, where $\perp, v_1, v_2, \dots, v_{k-1}, v$ is a path in ST_T. We say that node v has *string depth* $|\ell(v)|$. Let $v \in V$ be an internal node, and let $w_1, w_2, \dots, w_k$ be its children: then, $2 \leq k \leq \sigma + 1$, and labels $\ell(v, w_1), \ell(v, w_2), \dots, \ell(v, w_k)$ start with distinct characters. The children of v are ordered lexicographically according to the labels of edges $(v, w_1), (v, w_2), \dots, (v, w_k)$. There is a bijection between the leaves of ST_T and the suffixes of $T\#$, so every leaf is annotated with the starting position of its corresponding suffix. Moreover, if leaf $v \in V$ is associated with the suffix that



Suffix Trees and Arrays, Fig. 1 Relationship between the suffix tree, the suffix array (*left*), and the suffix-link tree (*right*) of string $T = \texttt{AGAGCGAGAGCGCGC\#}$. *Thin black lines*, edges of ST_T; *thick gray lines*, suffix links; *thin dashed lines*, implicit Weiner links; *thick black lines*, the subtree of ST_T induced by maximal repeats. *Black dots*, nodes of ST_T; *large black dot*, $\perp$; *white dots*, destinations of implicit Weiner links. *Squares*, leaves of ST_T and cells of SA_T; numbers, starting position of each suffix in T. For clarity, implicit Weiner links are not overlaid to ST_T, and suffix links from the leaves of ST_T are not drawn

starts at position i, then $\ell(v) = T[i \ldots n]\#$. Since ST_T has exactly $n + 1$ leaves and every internal node has at least two children, there are at most n internal nodes; thus, ST_T takes $O(n)$ space. We drop the subscript from ST whenever the underlying string is clear from the context.

A substring W of $T\#$ is called *right maximal* if both Wa and Wb occur in T, with $\{a, b\} \subseteq \Sigma \cup \{\#\}$ and $a \neq b$. Clearly a substring W is right maximal iff $W = \ell(v)$ for some $v \in V$. Moreover, assume that $\ell(v) = aW$ for some $v \in V$, $a \in \Sigma$, and $W \in \Sigma^*$. Since aW is right maximal, string W is right maximal as well; therefore, there is a node $w \in V$ with $\ell(w) = W$. Thus, the set of labels $\{\ell(v) : v \in V\}$ enjoys the *suffix closure* property, in the sense that if a string W belongs to the set so does every one of its suffixes. We say that there is a *suffix link from v to w labeled by a*, and we write $\texttt{suffixLink}(v) = w$. Clearly, if v is a leaf, then $\texttt{suffixLink}(v)$ is either a leaf or $\perp$. The graph induced by V and by suffix links is a trie rooted at $\perp$: such trie is called the *suffix-link tree* SLT_T of string T (Fig. 1, right). Inverting the direction of all suffix links yields the so-called *explicit Weiner links*. Given a node v and a symbol $a \in \Sigma$, it might happen that string $a\ell(v)$ does occur in T but that it is not the label of any node in V: all such left extensions of nodes in V that end in the middle of an edge of ST are called *implicit Weiner links*. A node in V can have more than one outgoing Weiner link, and all such Weiner links have different labels. The number of suffix links (or, equivalently, of explicit Weiner links) is upper-bounded by $2n-2$, and the same bound holds for the number of implicit Weiner links: in some applications, we thus assume that ST is augmented with unary nodes that correspond to all the destinations of implicit Weiner links. A substring W of $T\#$ is called *left maximal* if both aW and bW occur in $T\#$, with $\{a, b\} \subseteq \Sigma \cup \{\#\}$ and $a \neq b$, where $T\#$ is interpreted as a circular string. A string that is both left and right maximal is called *maximal repeat*. The set of all left-maximal strings enjoys the *prefix closure* property; therefore, there is a bijection between the maximal repeats and the nodes that lie in some paths of ST that start from the root (Fig. 1, left).

The *suffix array* $\mathsf{SA}_T[1 \ldots n + 1]$ of string T is the permutation of $[1 \ldots n + 1]$ such that $\mathsf{SA}_T[k] = i$ iff suffix $T[i \ldots n]\#$ has position k in the list of all suffixes of $T\#$ taken in lexicographic order. In this case, we say that suffix $T[i \ldots n]\#$ has *lexicographic rank* k. Clearly

$\mathsf{SA}_T[1] = n + 1$. The *inverse suffix array* of string T is an array $R_T[1 \ldots n + 1]$ such that $R_T[\mathsf{SA}[i]] = i$ for all $i \in [1 \ldots n + 1]$. A substring W of $T\#$ corresponds to a unique, contiguous *interval* (i_W, j_W) of SA_T, which contains all the suffixes of $T\#$ that are prefixed by W. An additional structure that complements the suffix array in many applications is the *longest common prefix array* $\mathsf{LCP}_T[2 \ldots n + 1]$, which stores at position i the length of the longest prefix shared by suffix $T[\mathsf{SA}_T[i] \ldots n]\#$ and by suffix $T[\mathsf{SA}_T[i - 1] \ldots n]\#$. Clearly $\mathsf{LCP}_T[k] \geq |W|$ for all $k \in [i_W + 1 \ldots j_W]$. Again, we drop the subscript from SA, R, and LCP whenever the underlying string is clear from the context.

Suffix tree, suffix array, and LCP array are strongly intertwined, and they have connections to other substring recognizers, like the *directed acyclic word graph* (DAWG) and its compact variant (CDAWG). SA can be thought of as the ordered set of leaves of ST, and ST can be thought of as a search tree built on top of SA (Fig. 1, left). The full ST, including suffix links, can be built from SA and LCP with a $O(n)$-time scan [1], and SA can be built from ST with a $O(n)$ traversal. LCP itself can be built from SA in $O(n)$ time [18]. A number of ingenious algorithms have been proposed to build ST and SA in linear time directly from the string itself, even in the case of polynomial alphabets: see [10, 17, 19, 20, 25, 32, 33] for a sampler of such algorithms, and see [28] for a detailed taxonomy. Some applications require to maintain the suffix tree after edits to the underlying string: see [12, 13, 22] for a sampler of such algorithms. Finally, see [21] for a comparative study of space-efficient allocations of suffix trees.

Key Results

Suffix trees are extremely versatile indexes that allow one to solve a variety of string matching and analysis problems [2, 9, 14]. We review few such problems, classifying the corresponding algorithmic solutions based on the way they walk on the suffix tree and on the information they store in each node. This classification exposes recurrent design patterns, it highlights which parts of the suffix tree are needed by each application, and it helps decide which algorithms can be implemented on top of more succinct but less powerful representations of the suffix tree. The emphasis of this section is on the power of different traversals of the suffix tree, not necessarily on the most efficient solution of each string analysis problem.

Top-Down

Exact searching inside a string S of length n is the most natural example of top-down traversal of ST_S. Given a query string W, we can just match its characters from the root of ST in $O(|W|)$ time to determine whether W occurs in S or not. Since edges are labeled by substrings of S, the search for W can end in the middle of an edge (u, v): we say that v is the *locus* of W in ST, and we denote it by $\texttt{locus}(W)$. This approach generalizes to a set of patterns $W_1, W_2, \ldots, W_k$ of total length m, by building the suffix tree of the concatenation $W = W_1\#_1 W_2\#_2 \cdots \#_{k-1} W_k$ and by traversing ST_S and ST_W synchronously, where $i \neq j$ implies $\#_i \neq \#_j$ and $\#_i \neq \#$.

The total number of (possibly overlapping) occurrences of the label $\ell(v)$ of a node v of ST equals the number of leaves in the subtree rooted at v, which can be computed by a bottom-up traversal of the tree. All strings that end in the middle of edge (u, v) start exactly at the same positions as $\ell(v)$ in S; therefore, ST with frequency annotation allows one to return the frequency in S of any string W in $O(|W|)$ time. An important consequence of this is the fact that the number of *distinct frequencies* assumed by nonempty substrings of S is at most $|S|$. It is also possible to annotate every node of ST with the smallest and largest leaf in its subtree, supporting $O(|W|)$-time queries on the first and last occurrence in S of any string W. More generally, traversing the tree rooted at $\texttt{locus}(W)$ in $O(|W| + k)$ time allows one to print all the k starting positions of W in S.

Finding all the occurrences of W in S can also be done in $O(|W| \log n)$ time, by binary searching SA_S for strings $W\#$ and $W\$$, where $\$ = \sigma + 1$: the result of these searches are, respectively, the starting and ending position of the interval of all suffixes prefixed by W. Knowing this interval allows one to derive the number of occurrences of W in S in constant time and to output the starting positions of such occurrences in time linear in the size of the output. Using simple properties of LCP_S, it is possible to reduce the time of binary search to $O(|W| + \log n)$, by reusing information during the search [24].

The top-down navigation of a suitably annotated suffix tree of S allows one also to compute the Lempel-Ziv factorization of S [23]. Recall that this factorization scans the string from left to right, and it determines at every position i the longest prefix of $S[i \ldots n]$ that equals a prefix of $S[j \ldots n]$, where $j < i$. Let W be such longest prefix: the factorization outputs the tuple $(j, |W|, S[i + |W|])$. Clearly we can find all this information by annotating every node v of ST with the index j of the smallest leaf in the subtree rooted at v. Then, we can just match suffix $S[i \ldots n]$ from the root of ST until a mismatch occurs or until we find a node with index greater than i. More advanced solutions embed the factorization in an online, one-pass construction of ST [29].

Bottom-Up

A *square* is a string WW where $W \in \Sigma^+$ is not in the form Z^k with $k > 1$ for any $Z \in \Sigma^+$. Clearly, if a square WW occurs at position i in S, then there is a node v in ST_S such that $|\ell(v)| \geq |W|$ and such that leaves i and $i + |W|$ belong to the subtree rooted at v. The converse is also true [4]. Thus, we can output all the repeats of S by using the following bottom-up traversal of ST. Assume without loss of generality that all nodes in ST have exactly two children. Every node u of ST builds its list of occurrences, sorted by position in S, using the lists of its children. Then, it scans its list once to find all pairs of positions at distance at most $|\ell(v)|$ in S that are consecutive in the list: every such pair is a square, and positions at distance at most $|\ell(v)|$ that are not consecutive induce squares that are implied by the consecutive positions.

Let v and w be the two children of v, and assume without loss of generality that the list of occurrences of v is smaller than the list occurrences of w. Then, the list of node u can be built by extracting all elements from the list of node v and by inserting them into the list of node w. As a consequence of such insertions, the occurrences in the list of v move to a list that is at least twice the size of the original list: it follows that an occurrence can be pushed into at most $O(\log n)$ lists; therefore, the total number of extractions and insertions is bounded by $O(n \log n)$. If the lists of occurrences are implemented with balanced trees, the total time to extract all squares from S is $O(n \log^2 n)$. More advanced approaches manage to shave a logarithm, reaching optimal $O(n \log n)$ time [4], and to reduce the complexity to $O(n + \tau)$, where τ is the size of the output [15, 31].

The algorithm for detecting squares can be adapted to compute all the *maximal palindromes* of S, by applying it to string $T = S\#\underline{S}\$$. Note that a variant of the same algorithm can be implemented using the suffix array. First, it is easy to see that a bottom-up, in-order traversal of the internal nodes of ST_S can be simulated by a linear scan of SA_S and of LCP_S, maintaining a stack [1]. It follows that, for every interval (i_v, j_v) in SA of a node v in ST, we can just check whether $\mathsf{SA}[k] + |\ell(v)| \in [i_v \ldots j_v]$ and $S[\mathsf{SA}[k] + |\ell(v)|] \neq S[\mathsf{SA}[k] + 2|\ell(v)|]$, for every $k \in [i_v \ldots j_v]$: in this case, the occurrence of square $\ell(v)$ at position $\mathsf{SA}[k]$ is called *branching*. It is easy to see that all squares can be derived from squares with branching occurrences [31]. Moreover, if the occurrence at position $\mathsf{SA}[k]$ is branching, then suffixes $\mathsf{SA}[k] + |\ell(v)|$ and $\mathsf{SA}[k] + 2|\ell(v)|$ belong to distinct children of node v in ST: we can thus discard the child w of v with the largest number of leaves and check for every $k \in [i \ldots j]$ that does not belong to the interval of w whether $\mathsf{SA}[k] - |\ell(v)| \in [i_v \ldots j_v]$ and $S[\mathsf{SA}[k]] \neq S[\mathsf{SA}[k] + |\ell(v)|]$. The child of v with largest interval can be determined in constant time during the simulated bottom-up traversal of ST, and since the largest

interval is always excluded, the algorithm runs in $O(n \log n)$ time.

Given a collection of k strings of total length n, let S be the concatenation of all such strings, each terminated by a distinct symbol that does not belong to Σ. A bottom-up navigation of ST_S (called also the *generalized suffix tree* of the collection) allows one to compute the length of a longest string that occurs in $x \leq k$ strings. To solve this problem, we can annotate each leaf v of ST with a bitvector `which` of length k, such that $\texttt{which}[i] = 1$ iff the suffix associated with v starts inside string i. Then, every node of ST can be annotated with the same bitvector via a bottom-up, $O(nk)$ traversal, in which we compute the bitvector of a node by taking the logical `or` of the bitvectors of its children. More advanced algorithms solve this problem in $O(n)$ time [8]. As a byproduct, this annotation allows one to answer queries on the number of strings in the collection that contain a given substring, a problem known as *document counting*. A germane problem is that of *document listing*, in which we are given a pattern and we are asked to return the set of all documents that contain one or more copies of the pattern [26].

Top-Down and Suffix Links

Given two strings S and T, of length n and m, respectively, the *matching statistics array* $\mathsf{MS}_{S,T}[1 \ldots n]$ is such that $\mathsf{MS}_{S,T}[i]$ stores the length of the longest string that starts at position i in S and that occurs in T [33]. We can compute $\mathsf{MS}_{S,T}$ by scanning S from left to right, while simultaneously issuing child and suffix-link queries on ST_T. This results in a peculiar walk on ST_T that consists of alternating sequences of suffix-tree edges and of suffix links (we can also compute $\mathsf{MS}_{S,T}$ symmetrically, by scanning S from right to left and by simultaneously issuing parent and Weiner-link queries on ST_T [27]).

Specifically, assume that we are at position i in S, and let $W = S[i \ldots i + \mathsf{MS}_{S,T}[i] - 1]$. Note that W can end in the middle of an edge (u, v) of ST_T: let $W = aXY$ where $a \in \Sigma$, $X \in \Sigma^*$, $aX = \ell(u)$, and $Y \in \Sigma^*$. Moreover, let $u' = \texttt{suffixLink}(u)$ and $v' = \texttt{suffixLink}(v)$. Note that suffix links can project edge (u, v) onto a *path* $u', v_1, v_2, \ldots, v_k, v'$, where $v_j \in V$ for $j \in [1 \ldots k]$. Since $\mathsf{MS}_{S,T}[i+1] \geq \mathsf{MS}_{S,T}[i]-1$, the first step to compute $\mathsf{MS}_{S,T}[i + 1]$ is to find the position of XY in ST_T: we call this phase of the algorithm the *repositioning phase*. To implement the repositioning phase, it suffices to take the suffix link from u, to follow the outgoing edge from u' whose label starts by the first character of Y, and then to iteratively jump to the next internal node of ST_T and to choose the next outgoing edge according to the corresponding character of Y. After repositioning, we start matching the new characters of S on ST_T, i.e., we read characters $S[i + \mathsf{MS}_{S,T}[i]], S[i + \mathsf{MS}_{S,T}[i] + 1], \ldots$ until such an extension becomes impossible in ST_T. We call this phase of the algorithm the *matching phase*. Note that no character of S that has been read during the repositioning phase of $\mathsf{MS}_{S,T}[i + 1]$ will be read again during the repositioning phase of $\mathsf{MS}_{S,T}[i + k]$ with $k > 1$: it follows that every position j of S is consumed at most twice, once in the matching phase of some $\mathsf{MS}_{S,T}[i]$ with $i \leq j$ and once in the repositioning phase of some $\mathsf{MS}_{S,T}[k]$ with $i < k < j$. Since every mismatch can be charged to the position of which it concludes the matching statistics, the total number of mismatches encountered by the algorithm is bounded by the length of S.

These algorithms can be adapted to compute the *shortest unique substring array* $\mathsf{SUS}_S[1 \ldots n]$, which stores at index i the length of the shortest substring of S that occurs only at position i [33]. The average of the matching statistics vector can be used to estimate the cross-entropy of the probability distributions of two stationary, ergodic, stochastic processes with finite memory that generated S and T [11]. Moreover, a number of compositional similarity measures between two strings S and T can be computed by scanning S and by simultaneously navigating ST_T as in matching statistics: this has the advantage of building and annotating the suffix tree of just the shortest string [30]. Matching statistics on a suitably annotated suffix tree of T allows one also to approximate the probability that S was generated by the same variable-length Markov process that produced

T, another measure of similarity not based on sequence alignment [3].

Top-Down in the Suffix-Link Tree

A number of statistical applications require to annotate the nodes of ST_S with *empirical probabilities* rather than with raw frequencies. The empirical probability $p_S(W)$ of a string W is essentially the number of its occurrences $f_S(W)$ divided by the maximum number of occurrences that W can have in a string of length $|S| = n$. This number cannot exceed $n-|W|+1$, but it also depends on the number of overlaps that W has with itself, i.e., on the number of proper *borders* of W: thus, we set $p_S(W) = f_S(W)/b(W)$, where $b(W)$ is the length of the shortest period of W. Note that p_S can change inside an edge of ST. However, if we are interested only in the empirical probability of nodes of ST, we can compute all such values in overall linear time, by mapping the longest-border computation in the KMP algorithm onto a depth-first navigation *of the suffix-link tree* [5].

The exact computation of the *variance* of the frequency of a string W in S can be itself mapped onto the computation of the longest proper border of W. Under suitable statistical assumptions, computing the expectation and variance of the frequency of all right-maximal substrings of S suffices to detect all substrings of S with anomalous frequency: it is thus possible to discover all statistically frequent and rare substrings of S in overall linear time [5].

Any Order

A single pass over all nodes of ST *in any order*, coupled with a number of checks on the children and on the Weiner links of each node, suffices to solve a number of string analysis problems in linear time.

A string W is a *maximal unique match* (MUM) between two strings S and T if it occurs exactly once in S and exactly once in T and if neither aW nor Wb occur in both S and T for any $\{a, b\} \subseteq \Sigma$ (for simplicity, we disregard cases in which W occurs at the beginning or at the end of a string) [14]. Clearly W must be a right-maximal substring of $U = S\#T\$$, where # and \$ are separators not belonging to Σ. Therefore, we just need to iterate over every node v of ST_U in any order, checking the following conditions: (1) v has exactly two leaves as children; (2) the suffixes that correspond to such leaves start before and after position $|S| + 1$ in U, respectively; and (3) v has two Weiner links. A similar approach extends to MUMs of more than two strings, as well as to *maximal* (not necessarily unique) *exact matches* between two strings and to the *maximal repeats* [7] and the *minimal absent words* of a single string [16].

Symmetrically, it is easy to detect the MUMs of two strings S and T by a linear scan of the suffix array of $U = S\#T\$$ and of the corresponding LCP array. Indeed, a MUM corresponds to an interval $(i, i + 1)$ of size two in SA_U such that $\mathsf{LCP}_U[i] < \mathsf{LCP}_U[i + 1]$, $\mathsf{LCP}_U[i + 2] < \mathsf{LCP}_U[i + 1]$, $U[\mathsf{SA}_U[i] - 1] \neq U[\mathsf{SA}_U[i + 1] - 1]$, and $\mathsf{SA}_U[i] < |S| + 1 < \mathsf{SA}_U[i + 1]$. Similar criteria allow one to detect maximal repeats, *supermaximal repeats* [14], and maximal exact matches [1].

String Depth Annotation

Assume that every node v of ST_S is annotated with $|\ell(v)|$. Recall that the *shortest unique substring array* $\mathsf{SUS}_S[1 \ldots n]$ is such that $\mathsf{SUS}_S[i]$ is the length of the shortest substring of S that occurs only at position i. Since $S[i \ldots i + \mathsf{SUS}[i] - 1] = Wa$ where $a \in \Sigma$, since $\texttt{locus}(Wa)$ is a leaf v, and since $\texttt{locus}(W) = \texttt{parent}(v)$, traversing the nodes of ST in any order suffices to compute $\mathsf{SUS}[i]$ for every i. String depth annotations, coupled with a traversal of the nodes of ST in any order, suffice also to compute measures of compositional complexity of S, like the total number of distinct substrings, possibly of a fixed length k.

Frequency Annotation

Recall that $f_S(W)$ is the number of occurrences of string W in S. Assume that we want to compute $p(a|W) = f_S(Wa)/f_S(W)$ for all substrings W of S and for all characters $a \in \Sigma$ such that Wa is a substring of S. Such values are called *conditional probabilities*. Clearly $p(a|W) = 1$ if W ends in the middle of an edge of ST_S: it is thus sufficient to compute conditional probabilities

for the nodes of ST, and this can be done by traversing the nodes of ST in any order and by accessing their children.

String Depth and Frequency Annotation
Assume that every node v of ST_S is also annotated with the number of leaves in the subtree rooted at v. Then, traversing the nodes of ST in any order allows one to compute the longest substring of S that repeats at least τ times, or the most frequent string of length at least τ, for any user-specified threshold τ. String depth and frequency annotations, coupled with a traversal of the nodes of ST in any order, allow one also to compute the number of distinct substrings that occur τ times in S, for every frequency τ in a user-specified range.

Given a substring W of S, let $\texttt{right}(W)$ be the set of characters that occur in S after W. More formally, $\texttt{right}(W) = \{a \in \Sigma : f_S(Wa) > 0\}$. The *kth order empirical entropy* of S is defined as follows:

$$\mathsf{H}(S,k) = \frac{1}{|S|} \sum_{W \in \Sigma^k} \sum_{a \in \texttt{right}(W)} f_S(Wa) \log \left(\frac{f_S(W)}{f_S(Wa)} \right)$$

To compute $\mathsf{H}(S,k)$, it suffices again to traverse the nodes of ST_S in any order, to check whether $|\ell(v)| = k$, and to cumulate the contribution of v to $\mathsf{H}(S,k)$ by reading the frequency of its children. Strings of length k that end in the middle of an edge of ST do not contribute to $\mathsf{H}(S,k)$.

In a similar fashion, given a string S on alphabet Σ, let $\mathbf{S}$ be a vector indexed by all strings in Σ^k for a fixed $k > 0$, such that $\mathbf{S}[W]$ contains the frequency of string W in S. We call $\mathbf{S}$ the k-mer *composition vector* of string S. Given two strings S and T, assume that we want to compute a function $\kappa(S,T)$ that depends only on $N = \sum_{W \in \Sigma^k} f(\mathbf{S}[W], \mathbf{T}[W])$, $D_S = \sum_{W \in \Sigma^k} g(\mathbf{S}[W])$, and $D_T = \sum_{W \in \Sigma^k} h(\mathbf{T}[W])$, where f, g, and h are user-specified functions. $\kappa(S,T)$ if often called *k-mer kernel* in text classification. It is possible to compute $\kappa(S,T)$ in overall linear time by traversing the nodes of the generalized suffix tree of S and T in any order. A similar traversal of ST allows one to compute $\kappa(S,T)$ on composition vectors that are indexed by all possible substrings, of any length. In practice the frequencies used in composition vectors are normalized by their expected values under IID or Markov probability distributions: a number of kernels based on such normalized counts can still be computed in overall linear time by traversing the nodes of ST in any order [6].

Positional Annotations
Given two strings S and T, the longest string W that occurs in both S and T is clearly a right-maximal substring of the concatenation $U = S\#T\$$, where # and \$ are separators not belonging to Σ. Consider thus ST_U, and assume that every node v is annotated with $|\ell(v)|$ and with a bit $\texttt{flag}(v)$ set to one iff the subtree rooted at v contains at least one leaf that starts before position $|S|+1$ in U and at least one leaf starting after position $|S|+1$ in U. Such annotation can be carried out in a bottom-up traversal of ST. We can compute W by iterating over the nodes $v \in \mathsf{ST}$ with $\texttt{flag}(v) = 1$ and by cumulating the maximum of the lengths of the encountered labels. The set of *all common substrings* between S and T is the set of all prefixes of the labels of nodes $v \in \mathsf{ST}$ such that $\texttt{flag}(v) = 1$ and $\texttt{flag}(w) = 0$ for every child w of v. This approach generalizes immediately to more than two strings, and it allows one to compute the length of the longest substring common to *at least* τ strings in a collection of k strings in $O(k|U|)$ time and space. More advanced approaches solve this problem in $O(|U|)$ time [8].

Applications

The primitives discussed above find application in a wide set of domains. A list of the most salient ones includes exact and approximate string

searching, string compression, statistical pattern discovery, alignment-free string comparison, string kernels in learning theory, sequence analysis, and assembly in bioinformatics.

Cross-References

- Approximate Tandem Repeats
- Burrows-Wheeler Transform
- Compressed Suffix Array
- Compressed Suffix Trees
- Document Retrieval on String Collections
- Indexed Approximate String Matching
- Indexed Two-Dimensional String Matching
- Lempel-Ziv Compression
- Lowest Common Ancestors in Trees
- Pattern Matching on Compressed Text
- String Matching
- Suffix Array Construction
- Suffix Tree Construction

Recommended Reading

1. Abouelhoda MI, Kurtz S, Ohlebusch E (2004) Replacing suffix trees with enhanced suffix arrays. J Discret Algorithms 2(1):53–86
2. Apostolico A (1985) The myriad virtues of subword trees. In: Apostolico A, Galil Z (eds) Combinatorial algorithms on words. Springer, Berlin/New York, pp 85–96
3. Apostolico A, Bejerano G (2000) Optimal amnesic probabilistic automata or how to learn and classify proteins in linear time and space. J Comput Biol 7(3–4):381–393
4. Apostolico A, Preparata FP (1983) Optimal off-line detection of repetitions in a string. Theor Comput Sci 22(3):297–315
5. Apostolico A, Bock ME, Lonardi S, Xu X (2000) Efficient detection of unusual words. J Comput Biol 7(1–2):71–94
6. Apostolico A, Denas O et al (2008) Fast algorithms for computing sequence distances by exhaustive substring composition. Algorithms Mol Biol 3(13)
7. Beller T, Berger K, Ohlebusch E (2012) Space-efficient computation of maximal and supermaximal repeats in genome sequences. In: 19th international symposium on string processing and information retrieval (SPIRE 2012), Cartagena de Indias. Lecture notes in computer science, vol 7608. Springer, pp 99–110
8. Chi L, Hui K (1992) Color set size problem with applications to string matching. In: Combinatorial pattern matching, Tucson. Springer, pp 230–243
9. Crochemore M, Hancart C, Lecroq T (2007) Algorithms on strings. Cambridge University Press, New York
10. Farach M (1997) Optimal suffix tree construction with large alphabets. In: Proceedings of the 38th annual symposium on foundations of computer science, 1997, Miami Beach. IEEE, pp 137–143
11. Farach M, Noordewier M, Savari S, Shepp L, Wyner A, Ziv J (1995) On the entropy of DNA: algorithms and measurements based on memory and rapid convergence. In: Proceedings of the sixth annual ACM-SIAM symposium on discrete algorithms (SODA '95), San Francisco. Society for Industrial and Applied Mathematics, pp 48–57
12. Ferragina P (1997) Dynamic text indexing under string updates. J Algorithms 22(2):296–328
13. Fiala ER, Greene DH (1989) Data compression with finite windows. Commun ACM 32(4):490–505. doi:10.1145/63334.63341, http://doi.acm.org/10.1145/63334.63341
14. Gusfield D (1997) Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, Cambridge/New York
15. Gusfield D, Stoye J (2004) Linear time algorithms for finding and representing all the tandem repeats in a string. J Comput Syst Sci 69(4):525–546. doi:10.1016/j.jcss.2004.03.004, http://dx.doi.org/10.1016/j.jcss.2004.03.004
16. Herold J, Kurtz S, Giegerich R (2008) Efficient computation of absent words in genomic sequences. BMC Bioinform 9(1):167
17. Kärkkäinen J, Sanders P, Burkhardt S (2006) Linear work suffix array construction. J ACM 53(6):918–936
18. Kasai T, Lee G, Arimura H, Arikawa S, Park K (2001) Linear-time longest-common-prefix computation in suffix arrays and its applications. In: Combinatorial pattern matching, Jerusalem. Springer, pp 181–192
19. Kim DK, Sim JS, Park H, Park K (2005) Constructing suffix arrays in linear time. J Discret Algorithms 3(2):126–142
20. Ko P, Aluru S (2003) Space efficient linear time construction of suffix arrays. In: Combinatorial pattern matching, Morelia. Springer, pp 200–210
21. Kurtz S (1999) Reducing the space requirement of suffix trees. Softw Pract Exp 29:1149–1171
22. Larsson NJ (1996) Extended application of suffix trees to data compression. In: Data compression conference, Snowbird, pp 190–199
23. Lempel A, Ziv J (1976) On the complexity of finite sequences. IEEE Trans Inf Theory 22:75–81
24. Manber U, Myers G (1993) Suffix arrays: a new method for on-line string searches. SIAM J Comput 22(5):935–948
25. McCreight EM (1976) A space-economical suffix tree construction algorithm. J ACM 23(2):262–272

26. Muthukrishnan S (2002) Efficient algorithms for document retrieval problems. In: Proceedings of the thirteenth annual ACM-SIAM symposium on discrete algorithms (SODA '02), San Francisco. Society for Industrial and Applied Mathematics, Philadelphia, pp 657–666. http://dl.acm.org/citation.cfm?id=545381.545469
27. Ohlebusch E, Gog S, Kügel A (2010) Computing matching statistics and maximal exact matches on compressed full-text indexes. In: XXth international symposium on string processing and information retrieval (SPIRE 2010), Los Cabos, pp 347–358
28. Puglisi SJ, Smyth WF, Turpin AH (2007) A taxonomy of suffix array construction algorithms. ACM Comput Surv 39(2):4
29. Rodeh M, Pratt VR, Even S (1981) Linear algorithm for data compression via string matching. J ACM 28(1):16–24
30. Smola AJ, Vishwanathan S (2003) Fast kernels for string and tree matching. In: Becker S, Thrun S, Obermayer K (eds) Advances in neural information processing systems (NIPS '03) 15, Vancouver. MIT, pp 585–592
31. Stoye J, Gusfield D (2002) Simple and flexible detection of contiguous repeats using a suffix tree. Theor Comput Sci 270(1):843–856
32. Ukkonen E (1995) On-line construction of suffix trees. Algorithmica 14(3):249–260
33. Weiner P (1973) Linear pattern matching algorithms. In: IEEE conference record of 14th annual symposium on switching and automata theory (SWAT '08), Iowa City, 1973. IEEE, pp 1–11

Sugiyama Algorithm

Nikola S. Nikolov
Department of Computer Science and Information Systems, University of Limerick, Limerick, Republic of Ireland

Keywords

Barycentric method; Crossing minimization; Hierarchical graph drawing; Layered graph drawing; Sugiyama algorithm; Sugiyama framework

Years and Authors of Summarized Original Work

1981; Sugiyama, Tagawa, Toda

Problem Definition

Given a directed graph (digraph) $G(V, E)$ with a set of vertices V and a set of edges E, the Sugiyama algorithm solves the problem of finding a 2D hierarchical drawing of G subject to the following readability requirements:

(a) Vertices are drawn on horizontal lines without overlapping; each line represents a level in the hierarchy; all edges point downwards.
(b) Short-span edges (i.e., edges between adjacent levels) are drawn with straight lines.
(c) Long-span edges (i.e., edges between nonadjacent levels) are drawn as close to straight lines as possible.
(d) The number of edge crossings is the minimum.
(e) Vertices connected to each other are placed as close to each other as possible.
(f) The layout of edges coming into (or going out of) a vertex is balanced, i.e., edges are evenly spaced around a common target (or source) vertex.

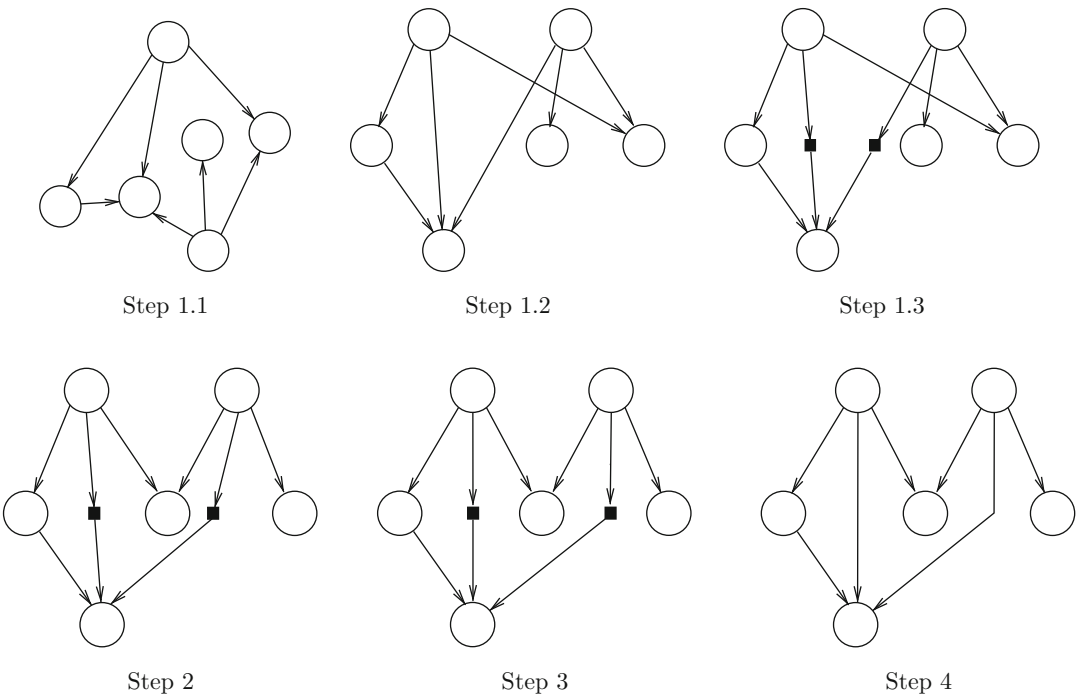
Requirements (a) and (b) are easy to meet and they are imposed as mandatory basic drawing rules. Requirements (c)–(f) are much harder to satisfy and typically they are met approximately [1, 4, 11].

Key Results

Sugiyama et al. propose a four-step procedure for finding a hierarchical drawing of a digraph subject to the readability requirements listed above. It is known as the Sugiyama algorithm, the Sugiyama method, or the Sugiyama framework [19]. The steps of the Sugiyama framework are illustrated in Fig. 1.

The Sugiyama Framework

Step 1: Preparatory step for transforming the input digraph G into a proper hierarchy.



Sugiyama Algorithm, Fig. 1 Illustration of the steps of the Sugiyama framework

Step 1.1: Transform the input digraph G into a directed acyclic graph (dag) by reversing the direction of some edges.

Step 1.2: Transform the dag into a multilevel digraph, called a *hierarchy*, by partitioning V into l levels (or layers) $V_1, V_2, \ldots, V_l$ such that for each edge $e = (v, w) \in E$ if $v \in V_i$ then $w \in V_{i+1}$. Levels are drawn on horizontal lines which determine the $y-$coordinates of the vertices.

Step 1.3 Transform the hierarchy into a *proper hierarchy* by introducing *dummy vertices* along long-span edges; one dummy vertex at each crossing of a long-span edge with a level.

Step 2: For each level V_i, specify a linear order σ_i of the vertices in V_i with the goal of minimizing the total number of edge crossing.

Step 3: Determine the $x-$coordinates of the vertices subject to requirements (c), (e), and (f) while preserving the linear order in the levels.

Step 4: Draw G in a 2D drawing area where dummy vertices are removed and the long-span edges are restored.

Steps 1.3 and 4 are trivial as computational problems. Steps 1.1 and 1.2 can be solved easily if the only readability requirements are those listed above. However, some sensible additional requirements can turn Steps 1.1 and 1.2 into difficult combinatorial optimization problems. For example, if we want to minimize the number of reversed edges at Step 1.1, then we need to solve the MINIMUM FEEDBACK ARC SET problem which is NP-hard [12]. Similarly, if we impose upper bounds on both the number of levels and the number of vertices per level, then the problem in Step 1.2, known as the *layering* problem, becomes NP-complete [4].

Following the work of Sugiyama et al., two types of solutions to the layering problem have been proposed in the research literature. The first type of layer assignment algorithm is list-scheduling algorithms (adapted from the area of static precedence-constrained multiprocessor

scheduling) which produce layer assignments with either the minimum number of levels or a specified maximum number of vertices per level [4]. These include the longest-path algorithm [13] and the Coffman-Graham algorithm [3] as well as the proposed by Nikolov et al. [15] MinWidth and StretchWidth heuristics which take into account the dummy vertices. The second type of algorithm employs network simplex and branch-and-cut techniques, respectively, for minimizing the number of dummy vertices with or without constraints on the number of levels and the number of vertices per level [9, 10].

Steps 2 and 3 are already hard to solve with the readability requirements listed above. It has also been suggested to precede Step 2 by an edge concentration or edge bundling step for achieving a more readable drawing [14, 16]. The other key results in the work of Sugiyama et al., besides defining the four-step framework, are efficient heuristics for Steps 2 and 3, respectively.

Reduction of the Number of Edge Crossings

Consider a proper hierarchy $G(V, E, \mathcal{L})$ with a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$, a set of edges $E = \{e_1, e_2, \ldots, e_m\}$, and a partitioning $\mathcal{L} = \{V_1, V_2, \ldots, V_l\}$ of the vertex set V into l levels (the result of Step 1.3). Let $\sigma_i : V_i \to \{1, 2, \ldots |V_i|\}$ be a linear order of the vertices in level V_i and let S_i be the set of all possible orders σ_i. The problem at Step 2 of the Sugiyama algorithm is to find a set of linear orders $\sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_l\} \in S_1 \times S_2 \times \ldots \times S_l$ such that the total number of edge crossings is the minimum. Let $K(G, \sigma)$ be the total number of edge crossings for a hierarchy G and a set of linear orders σ, and let $K(V_i, V_{i+1}, \sigma_i, \sigma_{i+1})$ be the number of edge crossings between layers V_i and V_{i+1} with linear orders σ_i and σ_{i+1}, respectively.

The algorithm, proposed by Sugiyama et al. for Step 2, is a heuristic which consists in initially choosing a random order σ_1 for the vertices in level V_1 and then repeatedly executing the following five-step procedure, called Down-Up, until either σ does not change or an initially given maximum number of iterations is reached.

The Down-Up Procedure

Step A: $i \leftarrow 1$.

Step B: With a fixed linear order σ_i, find a linear order σ_{i+1} which minimizes $K(V_i, V_{i+1}, \sigma_i, \sigma_{i+1})$.

Step C: If $i < n - 1$, then $i \leftarrow i + 1$ and go to Step B. Otherwise, go to Step D.

Step D: With a fixed linear order σ_{i+1}, find a linear order σ_i which minimizes $K(V_i, V_{i+1}, \sigma_i, \sigma_{i+1})$.

Step E: If $i > 1$, then $i \leftarrow i - 1$ and go to Step D. Otherwise, stop.

Both Step B and Step D involve minimizing the number of edge crossings between two adjacent layers with the linear order in one of them being fixed. This problem is known as the ONE-SIDED CROSSING MINIMIZATION (OSCM) problem, which has been shown to be NP-hard [5]. Based on previous work by Warfield [20], Sugiyama et al. show how OSCM can be reduced to the MINIMUM FEEDBACK SET problem and propose a heuristic method, called the *barycentric method*, for solving it. Let $A = (a_{ij})$ be the adjacency matrix of G. In essence, with a fixed linear order σ_i, the barycentric method orders the vertices in level V_{i+1} in the increasing order of their barycenters B_j, defined with Eq. (1).

$$B_j = \sum_{k=1}^{|V_i|} a_{kj}\sigma_i(v_k) \Big/ \sum_{k=1}^{|V_i|} a_{kj}, \qquad j \in \{1, 2, \ldots, |V_{i+1}|\} \tag{1}$$

Sugiyama et al. evaluate the Down-Up procedure experimentally with 800 randomly generated hierarchies as well as with five hierarchies from practical applications. Their conclusion is

that the proposed heuristic is effective. It was observed that in most cases the Down-Up procedure requires a single iteration. Reportedly, the heuristic was successfully extended for the case when vertices in each level are partitioned into subsets where the vertices in each subset must be arranged adjacently.

Step 2 is probably the best studied part of the Sugiyama framework. Numerous improvements to the original technique as well as alternative algorithms for crossing minimization have been proposed since the introduction of the Sugiyama framework [1, 4, 5, 7, 9, 11]. Notable among them is the 3-approximation *median method* proposed by Eades and Wormald [5] for solving the OSCM problem. Having the order of the vertices in level V_i fixed, the median method consists of placing each vertex in level V_{i+1} at a position which corresponds to the median of the positions of its neighbors in level V_i. Since the median method is an approximation algorithm, it guarantees to find a solution without edge crossings if such exists.

Determination of x-Coordinates of Vertices

For Step 3 of their framework, Sugiyama et al. propose a version of the Down-Up procedure with the barycenter of a vertex based on the x-coordinates of the connected to it vertices in an adjacent level. Consider the *down* part of the Down-Up procedure (the *up* part is symmetrical). If the x-coordinates of the vertices in level V_i are known, the barycenters B_j^* of the vertices in level V_{i+1} are defined with Eq. (2).

$$B_j^* = \sum_{k=1}^{|V_i|} a_{kj} x(v_k) / \sum_{k=1}^{|V_i|} a_{kj}, \quad j \in \{1, 2, \ldots, |V_{i+1}|\} \qquad (2)$$

The x-coordinates of the vertices in level V_{i+1} are determined according to their priority. The highest priority has the dummy vertices (introduced in Step 1.3), and the priority of each other vertex in level V_{i+1} is the number of vertices in level V_i connected to it. The x-coordinate of each vertex $v_j \in V_{i+1}$ is the integer number which is the closest to B_j^* available horizontal position (without changing the linear order from Step 2 and without displacing already placed vertices with higher priority). In finding this position, it is allowed to displace vertices with a priority lower than the priority of v_j, where this displacement should be as little as possible.

Sugiyama et al. evaluate the effectiveness of this method for improving the readability requirements (c), (e), and (f) experimentally. Reportedly, they have extended their heuristic for the case when the dimensions of the vertices are not insignificant. Both the Step 2 and the Step 3 heuristics were successfully applied to a hierarchy with more than 500 vertices.

Alternative algorithms for Step 3 have been proposed by Gansner et al. [9], Eades et al. [6], and Sander [17]. Probably, the best solution for Step 3 to date is the $O(|V|)$ algorithm of Brandes and Köpf [2]. It assigns x-coordinates to vertices by computing four *extreme* vertex alignments which are then combined into a final layout with at most two bends per edge.

Applications

Hierarchical graph drawings are useful for providing insight into hierarchical structures in complex systems. In recent years, the Sugiyama algorithm has found an important application for visual analysis of large social and biological networks [8, 18].

Cross-References

▸ List Scheduling

Recommended Reading

1. Bastert O, Matuszewski C (2000) Layered drawings of digraphs. In: Kaufman M, Wagner D (eds) Drawing graphs: method and models. Lecture notes in computer science, vol 2025. Springer, Berlin/Heidelberg, pp 87–120

2. Brandes U, Köpf B (2002) Fast and simple horizontal coordinate assignment. In: Mutzel P, Jünger M, Leipert S (eds) Graph drawing. Lecture notes in computer science, vol 2265. Springer, Berlin/Heidelberg, pp 31–44
3. Coffman EG Jr, Graham R (1972) Optimal scheduling for two-processor systems. Acta Inform 1(3):200–213
4. Eades P, Sugiyama K (1990) How to draw a directed graph. J Inf Process 13(4):424–437
5. Eades P, Wormald NC (1994) Edge crossings in drawings of bipartite graphs. Algorithmica 11(4):379–403
6. Eades P, Lin X, Tamassia R (1996) An algorithm for drawing a hierarchical graph. Int J Comput Geom Appl 6(2):145–156
7. Eppstein D, Goodrich MT, Meng JY (2007) Confluent layered drawings. Algorithmica 47(4):439–452
8. Fu X, Hong SH, Nikolov N, Shen X, Wu Y, Xu K (2007) Visualization and analysis of email networks. In: Asia-Pacific symposium on visualization, Sydney, pp 1–8
9. Gansner ER, Koutsofios E, North SC, Vo KP (1993) A technique for drawing directed graphs. IEEE Trans Softw Eng 19(3):214–230
10. Healy P, Nikolov NS (2002) How to layer a directed acyclic graph. In: Mutzel P, Jünger M, Leipert S (eds) Graph drawing. Lecture notes in computer science, vol 2265. Springer, Berlin/Heidelberg, pp 16–30
11. Healy P, Nikolov NS (2013) Hierarchical drawing algorithms. In: Tamassia R (ed) Handbook of graph drawing and visualization. Discrete mathematics and its applications, chap 13. Chapman and Hall/CRC, Boca Raton/London/New York, pp 409–454
12. Lempel A, Cederbaum I (1966) Minimum feedback arc and vertex sets of a directed graph. IEEE Trans Circuit Theory 13(4):399–403
13. Mehlhorn K (1984) Data structures and algorithms, Volume 2: graph algorithms and NP-completeness. Springer, Heidelberg
14. Newbery FJ (1989) Edge concentration: a method for clustering directed graphs. In: Proceedings of the 2nd international workshop on software configuration management (SCM '89), Princeton. ACM, pp 76–85
15. Nikolov NS, Tarassov A, Branke J (2005) In search for efficient heuristics for minimum-width graph layering with consideration of dummy nodes. J Exp Algorithmics 10:1–27
16. Pupyrev S, Nachmanson L, Kaufmann M (2011) Improving layered graph layouts with edge bundling. In: Brandes U, Cornelsen S (eds) Graph drawing. Lecture notes in computer science, vol 6502. Springer, Berlin/Heidelberg, pp 329–340
17. Sander G (1996) A fast heuristic for hierarchical Manhattan layout. In: Brandenburg FJ (ed) Graph drawing. Lecture notes in computer science, vol 1027. Springer, Berlin/Heidelberg, pp 447–458
18. Schwikowski B, Uetz P, Fields S (2000) A network of protein-protein interactions in yeast. Nat Biotechnol 18(12):1257–1261
19. Sugiyama K, Tagawa S, Toda M (1981) Methods for visual understanding of hierarchical system structures. IEEE Trans Syst Man Cybern 11(2):109–125
20. Warfield JN (1977) Crossing theory and hierarchical mapping. IEEE Trans Syst Man Cybern 7(7):502–523

Superiority and Complexity of the Spaced Seeds

Louxin Zhang
Department of Mathematics, National University of Singapore, Singapore, Singapore

Keywords

Homology search; NP-hardness; Sensitivity and hit probability; Spaced seeds

Years and Authors of Summarized Original Work

2006; Ma, Li, Zhang

Problem Definition

In the 1970s, sequence alignment was introduced to demonstrate the similarity of the sequences of genes and proteins [12]. A DNA sequence is a finite sequence over four nucleotides – adenine, guanine, cytosine, and thymine, whereas a protein sequence is over 20 amino acids. Homologous proteins have similar biological functions. Since they evolve from a common ancestral sequence, the sequences of homologous proteins and their encoding genes are often highly similar. Therefore, the DNA or amino acid sequence of a protein is often aligned with the sequences of well-studied proteins to infer the biological functions of the protein.

Formally, an alignment of two sequences, S and T, on an alphabet $\mathcal{B}$ is a two-row matrix with the following properties:

1. The letters in S are listed in order, interspersed with space symbols "–," in a row, where "–" represents the fact that a letter is missing at a position.
2. The letters in T are listed in the other row in the same manner.
3. Each column does not contain two "–."

An alignment of S and T poses a model of the evolution from their least common ancestral sequence to themselves. An alignment is scored using a scoring matrix that has a score for every pair of letters in $\mathcal{B} \cup \{-\}$. The score of an alignment is defined to be the sum of the scores of the pairs of letters appearing in the columns of the alignment.

Proteins often have multiple functions. Two proteins having a common function often have one or several highly similar regions in their DNA and amino acid sequences. Such "conserved" regions are found by solving the local alignment problem:

Input: Two sequences $S = s_1 s_2 \ldots s_m$ and $T = t_1 t_2 \cdots t_n$ on an alphabet.
Find: Two subsequences $S' = s_i s_{i+1} \cdots s_j$ $(i \leq j)$ and $T' = t_k t_{k+1} \cdots t_l$ $(k \leq l)$ such that the alignment score of S' and T' is as large as possible.

The alignments between their subsequences are called *local alignments* of S and T.

A dynamic programming approach takes quadratic time to solve the local alignment problem [13]. Unfortunately, it is not fast enough for homology search against a database with millions of DNA or protein sequences. Therefore, a filtration technique was adopted to design fast algorithms for homology search in the 1990s [1], by which good local alignments between two sequences are found by first identifying short consecutive matches of a specified length between the sequences, called *seed hits*, and then extending them to obtain good local alignments.

The filtration technique has a dilemma over sensitivity and speed. Employing a long seed will miss some good local alignments between two sequences, decreasing sensitivity; on the other hand, using a short seed will waste time on extending many seed hits into local alignments that are not biologically meaningful, resulting in low speed.

In PatternHunter [10], Ma, Tromp, and Li introduced the idea of optimized spaced seeds to achieve good balance between the sensitivity and speed of the filtration approach. PatternHunter by default looks for nucleotide match in 11 positions in every region of 18 bases long, specified by the string $111 * 1 * *1 * 1 * *11 * 111$, to trigger the process of local alignment. Such hit patterns, called *spaced seeds*, led to surprisingly higher sensitivity as well as speed than the consecutive seed 11111111111 that has the same number of match positions [10]. Moreover, sensitivity can further be improved by employing multiple spaced seeds that are longer than 18 bases [8,14]. This motivates the study of how to find the optimal spaced seeds of given length and weight [2–5,7].

Key Results

A spaced seed Q can be represented by a string of 1's and $*$'s, where 1's give the match positions in a seed hit. The number of 1's in Q is called its *weight*, denoted by w_Q; the length of the corresponding string is called its *length*, denoted by L_Q. The relative positions in Q are denoted by $\mathcal{RP}(Q)$. For example, for $Q = 111 * 1 * *1 * 1 * *11 * 111$, $\mathcal{RP}(Q) = \{0, 1, 2, 4, 7, 9, 12, 13, 15, 16, 17\}$.

An alignment containing no –'s is called a *ungapped alignment*. A local ungapped alignment can be modeled as a 0-1 sequence by translating match columns (containing two identical letters) into 1's and mismatch columns into 0's. Hence, a hit of Q identifies an alignment if the relative positions of Q match 1's in a region in the corresponding 0-1 string of the alignment.

Assume match occurs independently with probability p at a position in a local ungapped alignment. The *sensitivity* of Q in detecting a local alignment of n columns of two sequences with identity p is then defined to be the probability that Q hits a Bernoulli random sequence, called a *uniform region*, in which 1

and 0 appear with probability p and $(1 - p)$, respectively. A spaced seed is *optimal* for aligning sequences with identity p of length n if it has the largest hit probability over a uniform region of length n in which 1 appears with probability p at a position.

A straightforward method for identifying optimal spaced seeds is to exhaustively examine all the spaced seeds of given length and weight by keeping the largest sensitivity (or hit probability) over a uniform region. Unfortunately, the sensitivity of a spaced seed is unlikely computable in polynomial time.

Theorem 1 *Computing the sensitivity of a spaced seed over a uniform region is NP-hard.*

The hit probability of a spaced seed over a uniform region can be computed using a dynamic programming approach [7] or using recurrence relations [4,5]. Not surprisingly, these approaches become impractical for identifying long spaced seeds, because their complexities are an exponential function in the difference of the length and weight of spaced seeds under consideration. Here a simple polynomial-time approximation scheme is presented.

WISESAMPLE ALGORITHM

Input: A spaced seed Q, a positive integer n, $0 < p < 1$, and $\epsilon > 0$.
Find: An estimate of hit probability Q in a uniform region of length n in which bit 1 appears at a position with probability p.

Initialize an array A: $A[i] \leftarrow 0$ for $j = 1, 2, \ldots, n - L_Q$;
$N \leftarrow \lceil 6\epsilon^{-2} n^2 \log n \rceil$;
Repeats N times
  $R[i] \leftarrow 1$ for $i \in \mathcal{RP}(Q)$;
  $R[i] \leftarrow 1$ with probability p for $i \in \{1, 2, \ldots, n\} - \mathcal{RP}(Q)$;
  For $i = 1, 2, \ldots, L - L_Q$
    If Q does not hit the subregion $R[1, i + L_Q - 1]$
      $A[i] \leftarrow A[i] + 1$;
Output $p^{w_Q} \left(1 + N^{-1} \sum_{j=1}^{n-L_Q} n_j\right)$.

Theorem 2 *Let Q be a spaced seed and its hit probability be x on a uniform region with identity p of length n.* WISESAMPLE *outputs an estimate y of x on input Q, n, p, and $\epsilon >$ such that $|y - x| \leq \epsilon x$ with high probability.*

Let Q be a spaced seed and R a uniform region with identity p of length n. Following convention in renewal theory, Q hits R at position k if and only if $R[k - L_Q + i_j + 1] = 1$ for all $1 \leq j \leq w_Q$. Let A_k be the event that Q hits R at position k and $\bar{A}_k$ be the complement event of A_k. Then the probability f_k that Q **first** hits R at the k-th position is:

$$f_k = \Pr[\bar{A}_0 \bar{A}_1 \cdots \bar{A}_{k-2} A_{k-1}].$$

The hit probability $Q_n(p)$ of Q on R is equal to:

$$Q_n(p) = \Pr[A_0 \cup A_1 \cup \cdots \cup A_{n-1}].$$

When seed hits are extended into local alignments, two seed hits will give one local alignment if they overlap. Therefore, the sensitivity of a spaced seed is closely related to the number of its nonoverlapping hits in a uniform region. A nonoverlapping hit of a spaced seed is a recurrent event with the following convention: If a hit at position k is selected as a nonoverlapping hit, then the next nonoverlapping hit is the first hit at or after position $k + L_Q$.

The average distance, μ_Q, between two successive nonoverlapping hits of Q is defined to be

$$\mu_Q = \sum_{j \geq L_Q} j f_j.$$

A spaced seed is *nonuniform* if $g.c.d.(\mathcal{RP}(Q)) = 1$.

Theorem 3 *For any nonuniform spaced seed* Q,

$$\mu_Q \leq \sum_{j=1}^{w_Q} p^{-j} + (L_Q - w_q) - (1-p)\left(p^{2-w_Q} - 1\right)/p.$$

Buhler et al. [3] proved that for any spaced seed Q, there are two constants α_Q and λ_Q that are independent of n such that $\lim_{n\to\infty}(1 - Q_n(p))/\left(\alpha_Q \lambda_Q\right) = 1$, where λ_Q is the largest eigenvalue of the transition matrix of a Markov chain model constructed from Q.

Theorem 4 *For the consecutive seed* B *of weight* w,

$$\frac{1}{\sum_{j=1}^{w} p^{-j} - w + 1} \leq \lambda_B \leq 1 - \frac{1}{\sum_{j=1}^{w}(p^{-j} + p^{j-1}) - w}.$$

For a spaced seed Q,

$$1 - \frac{1}{\mu_Q - L_Q + 1} \leq \lambda_Q \leq 1 - \frac{1}{\mu_Q}.$$

If $L_Q < (1-p)\left[p^{2-w_Q} - 1\right]/p + 1$, by Theorems 3 and 4, $\lambda_Q \leq \lambda_B$. This implies that Q has a larger hit probability than the consecutive seed of the same weight in a long uniform region with identity p.

The detailed proofs of these results can be found in [11, 15].

Applications

Spaced seed approach finds applications in homology search and comparison of genome sequences. PatternHunter was used to compare the mouse and human genomes in the mouse genome project [6]. MegaBLAST and BLASTZ have adopted spaced seeds for homology search. Recently, the approach has also been used in mapping short reads into reference genome sequences.

Interestingly, spaced seed design is found to be closely related to optimal Golomb ruler design [9].

Open Problems

It is proved to be NP-hard to identify the optimal spaced seeds over a nonuniform region [8].

Open problem 1 Is it NP-hard to find the optimal spaced seed of a given length and weight over a uniform region?

It has been shown that a uniform spaced seed has a lower hit probability than the consecutive seed of the same weight over any uniform region [4, 7]. But the following problem is open:

Open problem 2 For any nonuniform spaced seed Q and $0 < p < 1$, is there $n(p, Q)$ such that Q has a larger hit probability than the consecutive seed of the same weight over a uniform region with identity p of length $n \geq n(p, Q)$?

Cross-References

- Local Alignment (with Affine Gap Weights)
- Local Alignment (with Concave Gap Weights)

Recommended Reading

1. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. J Mol Biol 215(3):403–410
2. Brejovà B, Brown D, Vinař T (2004) Optimal spaced seeds for homologous coding regions. J Bioinformatics Comput Biol 1:595–610
3. Buhler J, Keich U, Sun Y (2004) Designing seeds for similarity search in genomic DNA. J Comput Syst Sci 70:342–363
4. Choi KP, Zhang LX (2004) Sensitivity analysis and efficient method for identifying optimal spaced seeds. J Comput Syst Sci 68:22–40

5. Choi KP, Zeng F, Zhang LX (2004) Good spaced seeds for homology search. Bioinformatics 20:1053–1059
6. Intl Mouse Genome Sequencing Consortium (2002) Initial sequencing and comparative analysis of the mouse genome. Nature 409:520–562
7. Keich U, Li M, Ma B, Tromp J (2004) On spaced seeds for similarity search. Discret Appl Math 3:253–263
8. Li M, Ma B, Kisman D, Tromp J (2004) PatternHunter II: highly sensitive and fast homology search. J Bioinformatics Comput Biol 2:417–440
9. Ma B, Yao H (2009) Seed optimization for iid similarities is no easier than optimal Golomb ruler design. Inf Process Lett 109(19):1120–1124
10. Ma B, Tromp J, Li M (2002) PatternHunter: faster and more sensitive homology search. Bioinformatics 18:440–445
11. Ma B, Li M (2007) On the complexity of the spaced seeds. J Comput Syst Sci 73:1024–1034
12. Needleman SB, Wunsch CD (1970) A general method applicable to the search for similarities in the amino acid sequence of two proteins. J Mol Biol 48:443–453
13. Smith TF, Waterman MS (1980) Identification of common molecular subsequences. J Mol Biol 147:195–197
14. Sun Y, Buhler J (2004) Designing multiple simultaneous seeds for DNA similarity search. In: Proceedings RECOMB'04, 2004, San Diego, pp 76–85
15. Zhang LX (2007) Superiority of spaced seeds for homology search. IEEE/ACM Trans Comput Biol Bioinformatics (TCBB) 4:496–505

Support Vector Machines

Nello Cristianini[1] and Elisa Ricci[2]
[1]Department of Engineering Mathematics, and Computer Science, University of Bristol, Bristol, UK
[2]Department of Electronic and Information Engineering, University of Perugia, Perugia, Italy

Keywords

Kernel Methods; Large Margin Methods; Support Vector Machines

Years and Authors of Summarized Original Work

1992; Boser, Guyon, Vapnik

Problem Definition

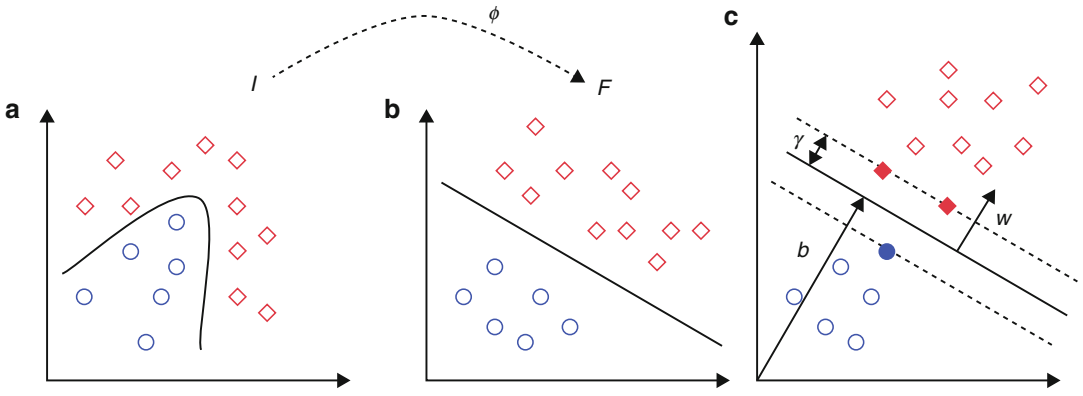
In 1992 Vapnik and coworkers [1] proposed a supervised algorithm for classification that has since evolved into what are now known as support vector machines (SVMs) [2]: a class of algorithms for classification, regression, and other applications that represent the current state of the art in the field. Among the key innovations of this method were the explicit use of convex optimization, statistical learning theory, and kernel functions.

Classification

Given a *training set* $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_\ell, y_\ell)\}$ of data points $\mathbf{x}_i$ from $X \subseteq \mathbb{R}^n$ with corresponding labels y_i from $Y = \{-1, +1\}$, generated from an unknown distribution, the task of classification is to learn a function $g: X \to Y$ that correctly classifies new examples $(\mathbf{x}, y)$ (i.e., such that $g(\mathbf{x}) = y$) generated from the same underlying distribution as the training data.

A good classifier should guarantee the best possible generalization performance (e.g., the smallest error on unseen examples). Statistical learning theory [3], from which SVMs originated, provides a link between the expected generalization error for a given training set and a property of the classifier known as its capacity. The SV algorithm effectively regulates the capacity by considering the function corresponding to the hyperplane that separates, according to the labels, the given training data and it is maximally distant from them (*maximal margin hyperplane*). When no linear separation is possible, a nonlinear mapping into a higher dimensional *feature space* is realized. The hyperplane found in the feature space corresponds to a nonlinear decision boundary in the input space.

Let $\phi : I \subseteq \mathbb{R}^n \to F \subseteq \mathbb{R}^N$ be a mapping from the input space I to the feature space F (Fig. 1a). In the learning phase, the algorithm finds a hyperplane defined by the equation $\langle \mathbf{w}, \phi(\mathbf{x}_i) \rangle = b$ such that the *margin*



Support Vector Machines, Fig. 1 (**a**) The feature map simplifies the classification task. (**b**) A maximal margin hyperplane with its support vectors highlighted

$$\gamma = \min_{1 \le i \le \ell} \; y_i(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle - b)$$
$$= \min_{1 \le i \le \ell} \; y_i g(\mathbf{x}_i) \qquad (1)$$

is maximized, where $\langle , \rangle$ denotes the inner product, $\mathbf{w}$ is a ℓ-dimensional vector of weights, and b is a threshold.

The quantity $(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle - b)/||\mathbf{w}||$ is the signed distance of the sample $\mathbf{x}_i$ from the hyperplane. When multiplied by the label y_i, it gives a positive value for correct classification and a negative value for an uncorrect one. Given a new data point $\mathbf{x}$, a label is assigned evaluating the decision function:

$$g(\mathbf{x}) = \text{sign}(\langle \mathbf{w}, \phi(\mathbf{x})\rangle - b) \qquad (2)$$

Maximizing the Margin

For linearly separable classes, there exists a hyperplane $(\mathbf{w}, b)$ such that

$$y_i(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle - b) \ge \gamma, \quad i = 1, \ldots, \ell. \qquad (3)$$

Imposing $||\mathbf{w}||^2 = 1$, the choice of the hyperplane such that the margin is maximized is equivalent to the following optimization problem:

$$\max_{\mathbf{w},b,\gamma} \gamma$$
$$\text{subject to } y_i(\langle \mathbf{w}, \phi(\mathbf{x}_i)\rangle - b) \ge \gamma, \; i = 1, \ldots, \ell, \qquad (4)$$
$$\text{and } ||\mathbf{w}||^2 = 1.$$

An efficient solution can be found in the dual space by introducing the Lagrange multipliers α_i, $i = 1, \ldots, \ell$. The problem (4) can be recast in the following dual form:

$$\max_{\alpha} \sum_{i=1}^{\ell} \alpha_i - \sum_{i=1}^{\ell} \sum_{j=1}^{\ell} \alpha_i \alpha_j y_i y_j \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j)\rangle \qquad (5)$$
$$\text{subject to } \sum_{i=1}^{\ell} \alpha_i y_i = 0, \quad \alpha_i \ge 0.$$

This formulation shows how the problem reduces to a *convex* (quadratic) optimization task. A key property of solutions α^* of this kind of problems is that they must satisfy the Karush-Kuhn-Tucker (KKT) conditions that ensure that only a subset of training examples needs to be associated to a nonzero α_i. This property is called *sparseness* of the SVM solution and is crucial in practical applications.

In the solution α^*, often only a subset of training examples is associated to nonzero α_i. These are called *support vectors* and correspond to the points that lie closest to the separating hyperplane (Fig. 1b). For the maximal margin hyperplane, the weight vector $\mathbf{w}^*$ is given by a linear function of the training points:

$$\mathbf{w}^* \sum_{i=1}^{\ell} \alpha_i^* y_i \phi(\mathbf{x}_i). \qquad (6)$$

S

Then the decision function (2) can equivalently be expressed as

$$g(\mathbf{x}) = \text{sign}(\sum_{i=1}^{\ell} \alpha_i^* y_i \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}) \rangle - b). \quad (7)$$

For a support vector $\mathbf{x}_i$, it is $\langle \mathbf{w}^*, \phi(\mathbf{x}_i) \rangle - b = y_i$ from which the optimum bias b^* can be computed. However, it is better to average the values obtained by considering all the support vectors [2]. Both the quadratic programming (QP) problem (5) and the decision function (7) depend only on the dot product between data points. The matrix of dot products with elements $K_{ij} = K(\mathbf{x}_i, \mathbf{x}_j) = \langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle$ is called the *kernel matrix*. In the case of linear separation, we simply have $K(\mathbf{x}_i, \mathbf{x}_j) = \langle \mathbf{x}_i, \mathbf{x}_j \rangle$, but in general, one can use functions that provide nonlinear decision boundaries. Widely used kernels are the polynomial $K(\mathbf{x}_i, \mathbf{x}_j) = (\langle \mathbf{x}_i, \mathbf{x}_j \rangle + 1)^d$ or the Gaussian $K(\mathbf{x}_i, \mathbf{x}_j) = e^{-\frac{||\mathbf{x}_i - \mathbf{x}_j||^2}{\sigma^2}}$ where d and σ are user-defined parameters.

Key Results

In the framework of learning from examples, SVMs have shown several advantages compared to traditional neural network models (which represented the state of the art in many classification tasks up to 1992). The statistical motivation for seeking the maximal margin solution is to minimize an upper bound on the test error that is independent of the number of dimensions and inversely proportional to the separation margin (and the sample size). This directly suggests embedding of the data in a high-dimensional space where a large separation margin can be achieved; this can be done efficiently with kernels using techniques from convex optimization. The sparseness of the solution, implied by the KKT conditions, adds to the efficiency of the result.

The initial formulation of SVMs by Vapnik and coworkers [1] has been extended by many other researchers. Here we summarize some key contributions.

Soft Margin

In the presence of noise the SV algorithm can be subject to overfitting. In this case one needs to tolerate some training errors in order to obtain a better generalization power. This has led to the development of the *soft margin* classifiers [4]. Introducing the slack variables $\xi_i \geq 0$, optimal class separation can be obtained by

$$\begin{aligned} &\min_{\mathbf{w},b\gamma,\xi} -\gamma + C \sum_{i=1}^{\ell} \xi_i \\ &\text{subject to } y_i(\langle \mathbf{w}, \phi(\mathbf{x}_i) \rangle - b) \geq \gamma - \xi_i, \xi_i \geq 0 \\ &i = 1, \ldots, \ell \text{ and } ||\mathbf{w}||^2 = 1. \end{aligned} \quad (8)$$

The constant C is user defined and controls the trade-off between the maximization of the margin and the number of classification errors. The dual formulation is the same as (5) with the only difference in the bound constraints ($0 \leq \alpha_i \leq C, \quad i = 1, \ldots, \ell$). The choice of soft margin parameter is one of the two main design choices (together with the kernel function) in applications. It is an elegant result [5] that the entire set of solutions for all possible values of C can be found with essentially the same computational cost as finding a single solution: this set is often called the *regularization path*.

Regression

A SV algorithm for regression, called support vector regression (SVR), was proposed in 1996 [6]. A linear algorithm is used in the kernel-induced feature space to construct a function such that the training points are inside a tube of given radius ε. As for classification the regression function only depends on a subset of the training data.

Speeding Up the Quadratic Program

Since the emergence of SVMs, many researchers have developed techniques to effectively solve the problem (5): a quite time-consuming task,

especially for large training sets. Most methods decompose large-scale problems into a series of smaller ones. The most widely used method is that of Platt [7] and it is known as sequential minimal optimization.

Kernel Methods

In SVMs, both the learning problem and the decision function can be formulated only in terms of dot products between data points. Other popular methods (i.e., principal component analysis, canonical correlation analysis, fisher discriminant) have the same property. This fact has led to a huge number of algorithms that effectively use kernels to deal with nonlinear functions keeping the same complexity as the linear case. They are referred to as *kernel methods* [8, 9].

Choosing the Kernel

The main design choice when using SVMs is the selection of an appropriate kernel function, a problem of model selection that roughly relates to the choice of a topology for a neural network. It is a nontrivial result [10] that also this key task can be translated into a convex optimization problem (a semi-definite program) under general conditions. A kernel can be optimally selected from a kernel space resulting from all linear combinations of a basic set of kernels.

Kernels for General Data

Kernels are not just useful tools to allow us to deploy methods of linear statistics in a nonlinear setting. They also allow us to apply them to nonvectorial data: kernels have been designed to operate on sequences, graphs, text, images, and many other kinds of data [8].

Applications

Since their emergence, SVMs have been widely used in a huge variety of applications. To give some examples, good results have been obtained in text categorization, handwritten character recognition, and biosequence analysis.

Text Categorization

In automatic text categorization, text documents are classified into a fixed number of predefined categories based on their content. In the works performed by Joachims [11] and Dumais et al. [12], documents are represented by vectors with the so-called bag-of-words approach used in the information retrieval field. The distance between two documents is given by the inner product between the corresponding vectors. Experiments on the collection of Reuters news stories showed good results for SVMs compared to other classification methods.

Handwritten Character Recognition

This is the first real-world task on which SVMs were tested. In particular two publicly available data sets (USPS and NIST) have been considered since they are usually used for benchmarking classifiers. A lot of experiments, mainly summarized in [13], were performed which showed that SVMs can perform as well as other complex systems without incorporating any detailed prior knowledge about the task.

Bioinformatics

SVMs have been widely used also in bioinformatics. For example, Jaakkola and Haussler [14] applied SVMs to the problem of protein homology detection, i.e., the task of relating new protein sequences to proteins whose properties are already known. Brown et al. [15] describe a successful use of SVMs for the automatic categorization of gene expression data from DNA microarrays.

URL to Code

Many free software implementations of SVMs are available at the website

- www.support-vector.net/software.html

Two in particular deserve a special mention for their efficiency:

- *SVMlight*: Joachims T. Making large-scale SVM learning practical. In: Schölkopf B, Burges CJC, and Smola AJ (eds) Advances in Kernel Methods Support Vector Learning, MIT Press, 1999. Software available at http://svmlight.joachims.org
- *LIBSVM*: Chang CC, and Lin CJ, LIBSVM: a library for support vector machines, 2001. Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm

Cross-References

▸ PAC Learning
▸ Perceptron Algorithm

Recommended Reading

1. Boser B, Guyon I, Vapnik V (1992) A training algorithm for optimal margin classifiers. In: Proceedings of the fifth annual workshop on computational learning theory, Pittsburgh
2. Cristianini N, Shawe-Taylor J (2000) An introduction to support vector machines and other kernel-based learning methods. Cambridge University Press, Cambridge. Book website: www.support-vector.net
3. Vapnik V (1995) The nature of statistical learning theory. Springer, New York
4. Cortes C, Vapnik V (1995) Support-vector network. Mach Learn 20:273–297
5. Hastie T, Rosset S, Tibshirani R, Zhu J (2004) The entire regularization path for the support vector machine. J Mach Learn Res 5:1391–1415
6. Drucker H, Burges CJC, Kaufman L, Smola A, Vapnik V (1997) Support vector regression machines. Adv Neural Inf Process Syst (NIPS) 9:155–161. MIT
7. Platt J (1999) Fast training of support vector machines using sequential minimal optimization. In: Schölkopf B, Burges CJC, Smola AJ (eds) Advances in kernel methods support vector learning. MIT, Cambridge, pp 185–208
8. Shawe-Taylor J, Cristianini N (2004) Kernel methods for pattern analysis. Cambridge University Press, Cambridge. Book website: www.kernel-methods.net
9. Scholkopf B, Smola AJ (2002) Learning with kernels. MIT, Cambridge
10. Lanckriet GRG, Cristianini N, Bartlett P, El Ghaoui L, Jordan MI (2004) Learning the kernel matrix with semidefinite programming. J Mach Learn Res 5:27–72
11. Joachims T (1998) Text categorization with support vector machines. In: Proceedings of European conference on machine learning (ECML), Chemnitz
12. Dumais S, Platt J, Heckerman D, Sahami M (1998) Inductive learning algorithms and representations for text categorization. In: 7th international conference on information and knowledge management, Bethesda
13. LeCun Y, Jackel LD, Bottou L, Brunot A, Cortes C, Denker JS, Drucker H, Guyon I, Muller UA, Sackinger E, Simard P, Vapnik V (1995) Comparison of learning algorithms for handwritten digit recognition. In: Fogelman-Soulie F, Gallinari P (eds) Proceedings international conference on artificial neural networks (ICANN), Paris, vol 2. EC2, pp 5360
14. Jaakkola TS, Haussler D (1999) Probabilistic kernel regression models. In: Proceedings of the 1999 Conference on AI and Statistics, Fort Lauderdale
15. Brown M, Grundy W, Lin D, Cristianini N, Sugnet C, Furey T, Ares M Jr, Haussler D (2000) Knowledge-based analysis of mircoarray gene expression data using support vector machines. Proc Natl Acad Sci 97(1):262–267

Surface Reconstruction

Nina Amenta
Department of Computer Science, University of California, Davis, CA, USA

Keywords

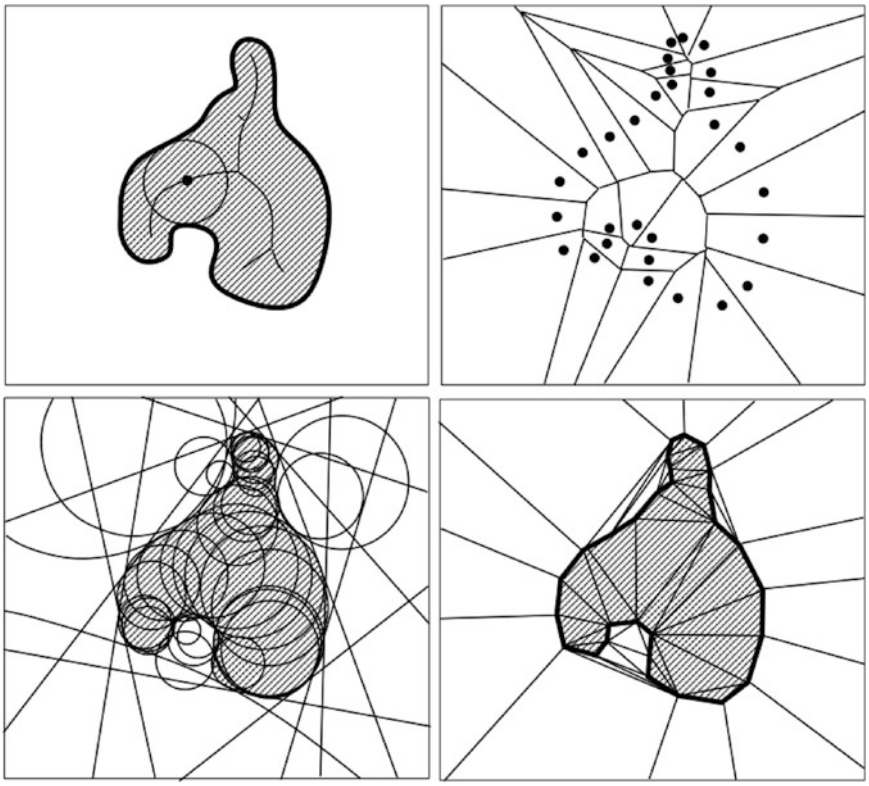
Delaunay triangulation; Local feature size; Medial axis; Surface reconstruction; Voronoi diagram

Years and Authors of Summarized Original Work

1999; Amenta, Bern
2000; Amenta, Choi, Dey, Leekha
2001; Amenta, Choi, Kolluri
2004; Dey, Goswami

Problem Definition

Surface reconstruction, here, is the problem of producing a piecewise-linear representation of a two-dimensional surface S in $\mathbb{R}^3$, given as input a set P of point samples from the surface. Very



Surface Reconstruction, Fig. 1 The medial axis of an object; the Voronoi diagram of a set of samples from the object boundary; the set of polar balls, with those inside the object shaded; the corresponding cells of the weighted Voronoi diagram, again with those inside the object shaded

sparse sets of point samples clearly do not convey much about S, so in order to prove correctness, we need to assume that the sample P is somehow sufficiently dense. The minimum required density could vary across the surface, with more detailed areas requiring denser sampling. This idea is captured in the following definition [2]. Let S be a two-dimensional surface in $\mathbb{R}^3$. The *medial axis* of S is the closure of the set of points that have more than one nearest point on S; a two-dimensional example is shown in Fig. 1, top left.

Definition 1 The *local feature size* $f(x)$ at a point x is the minimum distance from x to the medial axis of S.

The distance from the medial axis to the surface is zero at a sharp feature such as a corner or a crease, so we usually assume that S is smooth. The algorithms described here make the following ϵ-*sampling assumption*: the minimum distance, at any surface point x, to the nearest sample point is at most $\epsilon f(x)$, for some small constant ϵ. This leads to algorithms that are provably correct in the following sense.

INPUT*: A point set P that is an ϵ-sample from a smooth surface S without boundary.*

OUTPUT*: A piecewise-linear manifold without boundary, homeomorphic to S, that everywhere lies within distance $O(\epsilon f(x))$ of S.* The monograph [7] is an excellent reference for this line of research.

Key Results

One key idea is that in the neighborhood of any point $p \in P$ sampled from S, the surface is well approximated by a plane. Specifically, for any surface point x closer to p than to any other

sample, the distance of x from the tangent plane at p is $O(\epsilon f(x))$, as is the difference between the surface normal at x and the surface normal at p [2] (with the corrected proof [3]). Another key idea is that some subset of the Voronoi vertices of P approximates the medial axis of S, as in Fig. 1, top right.

Crust Algorithm

The crust algorithm [2] approximates the medial axis with a subset of the three-dimensional Voronoi vertices, called the poles. Each sample point in $p \in P$ selects the vertex of its Voronoi cell farthest from p as its first pole and the vertex farthest in the opposite direction as its second. We then eliminate any Delaunay triangle all of whose circumspheres contain a pole; this is easy to implement by computing the Delaunay triangulation of the set P augmented with the set of poles and eliminating any output triangle adjacent to a pole. A subset of the remaining surface triangles can then be selected as the piecewise-linear output surface.

Cocone Algorithm

The cocone algorithm [4] provides a simpler way of selecting a set of surface Delaunay triangles, requiring only one Voronoi diagram computation. It relies on the fact that the direction vector from a sample $p \in P$ to its first pole is within $O(\epsilon)$ of the surface normal at p, under the ϵ-sampling assumption. We define the cocone at p as the complement of a double cone, such that the angle between the cone surface and this approximate normal vector is at least $\pi - \pi/8$. We consider the intersection of the cocone at p with the Voronoi cell of p; the Delaunay triangles dual to any edge in this intersection are marked as potential surface triangles. Triangles marked by all three of their vertices are included in the set of surface triangles.

Powercrust Algorithm

While it is easy in theory to select a subset of the surface triangles to form a piecewise-linear output surface, it can be difficult in practice when the sampling density fails to meet the assumption, as is inevitable at sharp features. The power crust algorithm [5] eliminates this issue by producing a piecewise-linear output surface. The Voronoi ball centered at a pole is the ball with its nearest input samples on the boundary; see Fig. 1, lower left. We begin by labeling the Voronoi balls of all of the poles either as inside or outside the object bounded by S, using an iterative algorithm. We then compute the weighted Voronoi diagram, also known as the power diagram, of these polar Voronoi balls. Any Voronoi face separating the cell of an inner pole from the cell of an outer pole is output as part of the surface (Fig. 1, lower right). The faces of the piecewise-linear output surface are convex polygons but not in general triangles.

Noisy Samples

When the input sample points have noise, not every pole will be near the medial axis. Nonetheless, if the level of noise is everywhere small relative to the local feature size f, some subset of Voronoi vertices will still approximate the medial axis. In [8], this idea is developed into a provably correct algorithm. In addition to the ϵ-sampling assumption, we need to assume that the noise level is $O(\epsilon^2 f(x))$ and that the distance from any sample p to the kth nearest sample p' is $O(\epsilon f(x))$. This allows us to recognize a Voronoi vertex of p as a pole only when it is significantly farther from p than the k-nearest neighbors of p. These poles are then labeled as either inner or outer. This algorithm produces a triangulation of the boundary of the union of the inner polar balls as the output surface.

Complexity

The complexity of all of these algorithms depends on the complexity of the Voronoi diagram. While in general the Voronoi diagram of n points in $\mathbb{R}^3$ might have complexity $O(n^2)$, Attali, Boissonnat, and Lietier [6] proved that the complexity of the Voronoi diagram for points distributed uniformly on a nondegenerate smooth surface in $\mathbb{R}^3$ is $O(n \lg n)$. Another idea, employed by Funke and Ramos [9] and advanced by Cheng et al. [12], is to replace the Voronoi diagram with a less computationally expensive structure to get an $O(n \lg n)$ algorithm.

Applications

Interest in this problem was motivated by the advent of laser-range and LiDAR scanners [10], which produce depth maps sampled by point clouds. It is often reasonable to assume noise-free surface samples, since there are preprocessing methods, such as moving least squares (MLS) [1], that attract noisy point clouds onto nearby surfaces; there has also been theoretical work on MLS. MLS, or simply local plane-fitting, can be used to produce a normal vector at each sample point. Another common assumption is that the normal vectors can be consistently oriented. Poisson surface reconstruction [11] is an optimization technique that constructs manifold surfaces from possibly noisy points with normals. Because of its very efficient implementations, it is currently the most popular method in practice.

Open Problems

Subsequent work in surface reconstruction, both in computer graphics and in computational geometry, has focused on the identification and reconstruction of sharp features and then using them to construct surfaces that are non-manifold. Proving that the complexity of the Voronoi diagram of points distributed on a generic smooth surface with noise or with boundary is $o(n^2)$ remains open.

URLs to Code and Data Sets

There is code available for the cocone algorithm (http://web.cse.ohio-state.edu/~tamaldey/cocone.html), with several subsequent variants. There is also code for the power crust algorithm (http://www.cs.ucdavis.edu/~amenta/powercrust.html). There is a set of benchmark data sets for surface reconstruction (http://www.cs.utah.edu/~bergerm/recon_bench).

Cross-References

▸ Curve Reconstruction
▸ Manifold Reconstruction

Recommended Reading

1. Alexa M, Behr J, Cohen-Or D, Fleishman S, Levin D, Silva CT (2003) Computing and rendering point set surfaces. IEEE Trans Vis Comput Graph 9(1):3–15
2. Amenta N, Bern M (1999) Surface reconstruction by Voronoi filtering. Discret Comput Geom 22(4):481–504
3. Amenta N, Dey TK (2007) Normal variation for adaptive feature size, arXiv
4. Amenta N, Choi S, Dey TK, Leekha N (2000) A simple algorithm for homeomorphic surface reconstruction. In: Proceedings of the sixteenth annual symposium on computational geometry, Hong Kong. ACM, pp 213–222
5. Amenta N, Choi S, Kolluri RK (2001) The power crust, unions of balls, and the medial axis transform. Comput Geom Theory Appl 19(2):127–153
6. Attali D, Boissonnat JD, Lieutier A (2003) Complexity of the Delaunay triangulation of points on surfaces: the smooth case. In: Proceedings of the nineteenth annual symposium on computational geometry, San Diego. ACM, pp 201–210
7. Dey TK (2006) Curve and surface reconstruction: algorithms with mathematical analysis. Cambridge monographs on applied and computational mathematics. Cambridge University Press, Leiden
8. Dey TK, Goswami S (2004) Provable surface reconstruction from noisy samples. In: Proceedings of the twentieth annual symposium on computational geometry, Brooklyn. ACM, pp 330–339
9. Funke S, Ramos EA (2002) Smooth-surface reconstruction in near-linear time. In: Proceedings of the thirteenth annual ACM-SIAM symposium on discrete algorithms, San Francisco. Society for Industrial and Applied Mathematics, pp 781–790
10. Hoppe H, DeRose T, Duchamp T, McDonald J, Stuetzle W (1992) Surface reconstruction from unorganized points. ACM Trans Graph (TOG) 26(2):71–78
11. Kazhdan M, Bolitho M, Hoppe H (2006) Poisson surface reconstruction. In: Proceedings of the fourth eurographics symposium on geometry processing, Cagliari, pp 61–70
12. Cheng S-W, Jin J, Lau M-K (2012) A fast and simple surface reconstruction algorithm. In: Proceedings of the 28th annual symposium on computational geometry, Chapel Hill, pp 69–78

Symbolic Model Checking

Adnan Aziz[1] and Amit Prakash[2]
[1]Department of Electrical and Computer Engineering, University of Texas, Austin, TX, USA
[2]Microsoft, MSN, Redmond, WA, USA

Keywords

Formal hardware verification

Years and Authors of Summarized Original Work

1990; Burch, Clarke, McMillan, Dill

Problem Definition

Design verification is the process of taking a design and checking that it works correctly. More specifically, every design verification paradigm has three components [6]: (1) a language for specifying the design in an unambiguous way, (2) a language for specifying properties that are to be checked of the design, and (3) a checking procedure, which determines whether the properties hold off the design.

The verification problem is very general: it arises in low-level designs, e.g., checking that a combinational circuit correctly implements arithmetic, as well as high-level designs, e.g., checking that a library written in high-level language correctly implements an abstract data type.

Hardware Verification

The verification of hardware designs is particularly challenging. Verification is difficult in part because the large number of concurrent operations make it very difficult to conceive of and construct all possible corner cases, e.g., one unit initiating a transaction at the same cycle as another receiving an exception. In addition, software models used for simulation run orders of several magnitude slower than the final chip operates at. Faulty hardware is usually impossible to correct after fabrication, which means that the cost of a defect is very high, since it takes several months to go through the process of designing and fabricating new hardware. Wile et al. [15] provide a comprehensive account of hardware verification.

State Explosion

Since the number of state-holding elements in digital hardware is bounded, the number of possible states that the design can be in is infinite, so complete automated verification is, in principle, possible. However, the number of states that a hardware design can reach from the initial state can be exponential in the size of the design; this phenomenon is referred to as "state explosion." In particular, algorithms for verifying hardware that explicitly record visited states, e.g., in a hash table, have very high time complexity, making them infeasible for all but the smallest designs. The problem of complete hardware verification is known to be PSPACE-hard, which means that any approach must be based on heuristics.

Hardware Model

A hardware design is formally described using *circuits* [4, 8]. A *combinational circuit* consists of *Boolean combinational elements* connected by *wires*. The Boolean combinational elements are *gates* and *primary inputs*. Gates come in three types: *NOT, AND*, and *OR*. The NOT gate functions as follows: it takes a single Boolean-valued *input* and produces a single Boolean-valued *output* which takes value 0 if the input is 1 and 1 if the input is 0. The AND gate takes two Boolean-valued inputs and produce a single output; the output is 1 if both inputs are 1 and 0 otherwise. The OR gate is similar to AND, except that its output is 1 if one or both inputs are 1. A circuit can be represented as a directed graph where the nodes represent the gates and

wires represent edges in the direction of signal flow.

A circuit can be represented by a directed graph where the nodes represent the gates and primary inputs, and edges represent wires in the direction of signal flow. Circuits are required to be acyclic, that is, there is no cycle of gates. The absence of cycles implies that a Boolean assignment to the primary inputs can be propagated through the gates in topological order.

A *sequential circuit* extends the notion of circuit described above by adding *stateful elements*. Specifically, a sequential circuit includes *registers*. Each register has a single input, which is referred to as its *next-state input*.

A *valuation* on a set V is a function whose domain is V. A *state* in a sequential circuit is a Boolean-valued valuation on the set of registers. An *input* to a sequential circuit is a Boolean-valued valuation on the set of primary inputs. Given a state s and an input i, the logic gates in the circuit uniquely define a Boolean-valued valuation t to the set of register inputs – this is referred to as the next state of the circuit at state s under input i and say s *transitions* to t on input i. It is convenient to denote such a transition by $s \xrightarrow{i} t$.

A sequential circuit can naturally be identified with a *finite state machine* (FSM), which is a graph defined over the set of all states; an edge (s, t) exists in the FSM graph if there exists an input i, state s transitions to t on input i.

Invariant Checking

An *invariant* is a set of states; informally, the term is used to refer to a set of states that are "good" in some sense. One common way to specify an invariant is to write a Boolean formula on the register variables – the states which satisfy the formula are precisely the states in the invariant.

Given states r and s, define r to be *reachable* from s if there is a sequence of inputs $i_0, i_1, \ldots, i_{n-1}$ such that $s = s_0 \xrightarrow{i_0} s_1 \xrightarrow{i_1} \cdots s_n = t$. A fundamental problem in hardware verification is the following: given an invariant A, and a state s, does there exists a state r reachable from s which is not in A?

Key Results

Symbolic model checking (SMC) is a heuristic approach to hardware verification. It is based on the idea that rather than representing and manipulating states one at a time, it is more efficient to use symbolic expressions to represent and manipulate sets of states.

A key idea in SMC is that given a set $A \subset \{0,1\}^n$, a Boolean function A can be constructed such that f_A: $\{0,1\}^n \to \{0,1\}$ given by $f(\alpha_1, \ldots, \alpha_n) = 1$ iff $(\alpha_1, \ldots, \alpha_n) \in A$. Note that given a characteristic function f_A, A can be obtained and vice versa.

There are many ways in which a Boolean function can be represented: formulas in DNF, general Boolean formulas, combinational circuits, etc. In addition to an efficient representation for state sets, the ability to perform fast computations with sets of states is also important, for example, in order to determine if an invariant holds, it is required to compute the set of states reachable from a given state. BDDs [2] are particularly well suited to representing Boolean functions, as they combine succinct representation with efficient manipulation; they are the data structure underlying SMC.

Image Computation

A key computation that arises in verification is determining the *image* of a set of states A in a design D – the image of A is the set of all states t for which there exists a state in A and an input i such that state s transitions to t under input i. The image of A is denoted by $\mathrm{Img}(A)$.

The *transition relation* of a design is the set of (s, i, t) triples such that s transitions to t under input i. Let the design have n registers and m primary inputs; then the transition relation is subset of $\{0,1\}^n \times \{0,1\}^m \times \{0,1\}^n$.

Conceptually, the transition relation completely captures the dynamics of the design – given an initial state, and input sequence, the evolution of the design is completely determined by the transition relation.

Since the transition relation is a subset of $\{0,1\}^{n+m+n}$, it has a characteristic function $f_T : \{0,1\}^{n+m+n} \rightarrow \{0,1\}$. View f_T as being defined over the variables $x_0, \ldots, x_{n-1}$, $i_0, \ldots, i_{m-1}$, $y_0, \ldots, y_{n-1}$. Let the set of states A be represented by the function f_A defined over variables $x_0, \ldots, x_{n-1}$. Then the following identity holds

$$\mathrm{Img}(A) = (\exists x_0 \cdot \exists x_{n-1} \exists i_0 \cdots \exists i_{m-1})(f_A \cdot f_T).$$

The identity holds because $(\beta_0, \ldots, \beta_{n-1})$ satisfies the right-hand side expression exactly when there are values $\alpha_0, \ldots, \alpha_{n-1}$, and $\iota_0, \ldots, \iota_{m-1}$ such that $(\alpha_0, \ldots, \alpha_{n-1}) \in A$ and the state $(\alpha_0, \ldots, \alpha_{n-1})$ transitions to $(\beta_0, \ldots, \beta_{n-1})$ on input $(\iota_0, \ldots, \iota_{m-1})$.

Invariant Checking

The set of all states reachable from a given set A is the limit as n tends to infinity of the sequence of states $R_0, R_1, \ldots$ defined below:

$$R_0 = A$$
$$R_{i+1} = R_i \cup \mathrm{Img}(R_i).$$

Since for all i, $R_i \subseteq R_{i+1}$ and the number of distinct state sets is finite, the limit is reached in some finite number of steps, i.e., for some n, it must be that $R_{n+1} = R_n$. It is straightforward to show that the limit is exactly equal to the set of states reachable from A – the basic idea is to inductively construct input sequences that lead from states in A to R_i and to show that state t is reachable from a state in A under an input sequence of length l, then t must be in R_l.

Given BDDs F and G representing functions f and g, respectively, there is an algorithm based on dynamic programming for performing conjunction, i.e., for computing the BDD for $f \cdot g$. The algorithm has polynomial complexity, specifically $O(|F| \cdot |G|)$, where $|B|$ denotes the number of nodes in the BDD B. There are similar algorithms for performing disjunction ($f + g$) and computing cofactors (f_x and $f_{x'}$). Together these yield an algorithm for the operation of existential quantification, since $(\exists x) f = f_x + f'_x$.

It is straightforward to build BDDs for f_A and $f_T : A$ is typically given using a propositional formula, and the BDD for f_A can be built up using functions for conjunction, disjunction, and negation. The BDD for f_T is built using from the BDDs for the next-state nodes, over the register and primary input variables. Since the only gate types are AND, OR, and NOT, the BDD can be built using the standard BDD operators for conjunction, disjunction, and negation. Let the next-state functions be $f_0, \ldots, f_{n-1}$; then f_T is $(y_0 = f_0) \cdot (y_1 = f_1) \cdot \cdots \cdot (y_{n-1} = f_{n-1})$, and so the BDD for f_T can be constructed using the usual BDD operators.

Since the image computation operation can be expressed in terms of f_A and F_T, and conjunction and existential quantification operations, it can be performed using BDDs. The computation of R_i involves an image operation, and a disjunction, and since BDDs are canonical, the test for fixed point is trivial.

Applications

The primary application of the technique described above is for checking properties of hardware designs. These properties can be invariants described using propositional formulae over the register variables, in which case the approach above is directly applicable. More generally, properties can be expressed in a *temporal logic* [5], specifically through formulae which express acceptable sequences of outputs and transitions.

CTL is one common temporal logic. A CTL formula is given by the following grammar: if x is a variable corresponding to a register, then $\mathbf{x}$ is a CTL formula; otherwise, if φ and ψ are CTL

formulas, then so as $(\neg\phi)$, $(\phi \vee \psi)$, $(\phi \wedge \psi)$, $(\phi \rightarrow \psi)$, and $EX\phi$, $E\phi U\psi$, and $EG\phi$.

A CTL formula is interpreted as being true at a state; a formula **x** is true at a state if that register is 1 in that state. Propositional connectives are handled in the standard way, e.g., a state satisfies a formula $(\phi \wedge \psi)$ if it satisfies both φ and ψ. A state s satisfies $EG\phi$ if there exists a state t such that s transitions to, and t satisfies φ. A state s satisfies $E\phi U\psi$ if there exists a sequence of inputs $i_0, \ldots, i_n$ leading through state $s_0 = s$, $s_1, s_2, \ldots, s_{n+1}$ such that s_{n+1} satisfies ψ, and all states $s_i, i \leq n+1$ satisfy φ. A state s satisfies $EG\phi$ if there exists an infinite sequence of inputs $i_0, i_1, \ldots$ leading through state $s_0 = s, s_1, s_2, \ldots$ such that all states s_i satisfy φ.

CTL formulas can be checked by a straightforward extension of the technique described above for invariant checking. One approach is to compute the set of states in the design satisfying subformulas of φ, starting from the subformulas at the bottom of the parse tree for φ. A minor difference between invariant checking and this approach is that the latter relies on *pre-image* computation; the pre-image of A is the set of all states t for which there exists an input i such that t transitions under i to a state in A.

Symbolic analysis can also be used to check the equivalence of two designs by forming a new design which operates the two initial designs in parallel and has a single output that is set to 1 if the two initial designs differ [14]. In practice this approach is too inefficient to be useful, and techniques which rely more on identifying common substructures across designs are more successful.

The complement of the set of reachable states can be used to identify parts of the design which are redundant and to propagate don't care conditions from the input of the design to internal nodes [12].

Many of the ideas in SMC can be applied to software verification – the basic idea is to "finitize" the problem, e.g., by considering integers to lie in a restricted range or setting an a priori bound on the size of arrays [7].

Experimental Results

Many enhancements have been made to the basic approach described above. For example, the BDD for the entire transition relation can grow large, so *partitioned transition relations* [11] are used instead; these are based on the observation that $\exists x.(f \cdot g) = f \cdot \exists x.g$, in the special case that f is independent of x. Another optimization is the use of *don't cares*; for example, when computing the image of A, the BDD for f_T can be simplified with respect to transitions originating at A' [13]. Techniques based on SAT have enjoyed great success recently. These approaches case the verification problem in terms of satisfiability of a CNF formula. They tend to be used for bounded checks, i.e., determining that a given invariant holds on all input sequences of length k [1]. Approaches based on *transformation-based verification* complement symbolic model checking by simplifying the design prior to verification. These simplifications typically remove complexity that was added for performance rather than functionality, e.g., pipeline registers.

The original paper by Clarke et al. [3] reported results on a toy example, which could be described in a few dozen lines of a high-level language. Currently, the most sophisticated model checking tool for which published results are ready is SixthSense, developed at IBM [10].

A large number of papers have been published on applying SMC to academic and industrial designs. Many report success on designs with an astronomical number of states – these results become less impressive when taking into consideration the fact that a design with n registers has 2^n states.

It is very difficult to define the complexity of a design. One measure is the number of registers in the design. Realistically, a hundred registers is at the limit of design complexity that can be handled using symbolic model checking. There are cases of designs with many more registers that have been successfully verified with symbolic model checking, but these registers are invariably part of a very regular structure, such as a memory array.

Data Sets

The SMV system described in [9] has been updated, and its latest incarnation nuSMV (http://nusmv.irst.itc.it/) includes a number of examples.

The VIS (http://embedded.eecs.berkeley.edu/pubs/downloads/vis) system from UC Berkeley and UC Boulder also includes a large collection of verification problems, ranging from simple hardware circuits to complex multiprocessor cache systems.

The SIS (http://embedded.eecs.berkeley.edu/pubs/downloads/sis/) system from UC Berkeley is used for logic synthesis. It comes with a number of sequential circuits that have been used for benchmarking symbolic reachability analysis.

Cross-References

▸ Binary Decision Graph

Recommended Reading

1. Biere A, Cimatti A, Clarke E, Fujita M, Zhu Y (1999) Symbolic model checking using sat procedures instead of BDDs. In: ACM design automation conference, New Orleans
2. Bryant R (1986) Graph-based algorithms for Boolean function manipulation. IEEE Trans Comput C-35:677–691
3. Burch JR, Clarke EM, McMillan KL, Dill DL (1992) Symbolic model checking: 10^{20} states and beyond. Inf Comput 98(2):142–170
4. Cormen TH, Leiserson CE, Rivest RH, Stein C (2001) Introduction to algorithms. MIT, Cambridge
5. Emerson EA (1990) Temporal and modal logic. In: van Leeuwen J (ed) Formal models and semantics. Volume B of handbook of theoretical computer science. Elsevier Science, Amsterdam, pp 996–1072
6. Gupta A (1993) Formal hardware verification methods: a survey. Form Method Syst Des 1:151–238
7. Jackson D (2006) Software abstractions: logic, language, and analysis. MIT, Cambridge
8. Katz R (1993) Contemporary logic design. Benjamin/Cummings Publishing Company, Redwood City
9. McMillan KL (1993) Symbolic model checking. Kluwer Academic, Boston
10. Mony H, Baumgartner J, Paruthi V, Kanzelman R, Kuehlmann A (2004) Scalable automated verification via expert-system guided transformations. In: Formal methods in CAD, Austin
11. Ranjan R, Aziz A, Brayton R, Plessier B, Pixley C (1995) Efficient BDD algorithms for FSM synthesis and verification. In: Proceedings of the international workshop on logic synthesis, Tahoe City, May 1995
12. Savoj H (1992) Don't cares in multi-level network optimization. Ph.D. thesis, Electronics Research Laboratory, College of Engineering, University of California, Berkeley
13. Shiple TR, Hojati R, Sangiovanni-Vincentelli AL, Brayton RK (1994) Heuristic minimization of BDDs using don't cares. In: ACM design automation conference, San Diego, June 1994
14. Touati H, Savoj H, Lin B, Brayton RK, Sangiovanni-Vincentelli AL (1990) Implicit state enumeration of finite state machines using BDDs. In: IEEE international conference on computer-aided design, Santa Clara, pp 130–133, Nov 1990
15. Wile B, Goss J, Roesner W (2005) Comprehensive functional verification. Morgan-Kaufmann

Symmetric Graph Drawing

Seokhee Hong
School of Information Technologies, University of Sydney, Sydney, NSW, Australia

Keywords

Graph automorphism; Graph drawing; Planar graph; Symmetry

Years and Authors of Summarized Original Work

2006; Hong, McKay and Eades

Problem Definition

Symmetry is one of the most important aesthetic criteria in graph drawing that clearly reveals the structure and properties of a graph. Many graphs in Graph Theory textbooks are often symmetric.

A symmetry of a drawing D of a graph G induces an *automorphism* ϕ of the graph G, a permutation of the vertex set that preserves

adjacency. If an automorphism ϕ can be displayed as a symmetry in a drawing of the graph G, then it is called a *geometric automorphism* [6]. A geometric automorphism ϕ of a *planar* graph G is a *planar automorphism*, if there is a *planar* drawing of G which displays ϕ. Note that not every automorphism is geometric, and not every geometric automorphism is planar.

In general, algorithms for constructing symmetric drawings of graphs have two steps:

1. *Symmetry finding step*: Find the geometric automorphisms of a graph
2. *Symmetry drawing step*: Draw the graph displaying these automorphisms as symmetries.

Note that the first step is more difficult than the second step. For example, finding automorphism of a graph is isomorphism-hard; however finding geometric automorphism of a graph is NP-hard in general [18]. For planar graphs, computing isomorphism (therefore, automorphism) of a graph can be solved in linear time [7, 17]. However, finding the *best plane embedding* of planar graphs that displays the maximum number of symmetries in a drawing of a planar graph is challenging, because a planar graph can have exponential number of possible plane embeddings.

Furthermore, the product of two geometric automorphisms is not necessarily geometric, because they may be displayed by different drawings. A subgroup A of the automorphism group of a graph is a *geometric automorphism group*, if there is a single drawing of the graph that displays every element of A. Therefore, to construct a maximally symmetric drawing of a graph, one needs to compute a *maximum size* geometric automorphism group for the graph. Therefore, the main research problem for Symmetric Graph Drawing can be defined as below.

Symmetric Graph Drawing Problem
Input: A graph G.
Output: A maximum size geometric automorphism group A of G, A symmetric drawing D of G that displays all elements of A.

Key Results

There are two types of symmetry in two-dimensional drawings: *rotational symmetry* (i.e., a rotation about a point) and *axial* (or *reflectional*) *symmetry* (i.e., a reflection about an axis). The *order* of an automorphism α is the smallest positive integer k such that α^k equals the identity I. A group-theoretic characterization of geometric automorphism group was given by Eades and Lin [6] as follows:

- A group of order 2 generated by an *axial automorphism*;
- A *cyclic group* of order k generated by a *rotational automorphism*;
- A *dihedral group* of order $2k$ generated by a rotational automorphism of order k and an axial automorphism. In this case there are k axial symmetries.

In two dimensions, the problem of determining whether a given graph can be drawn symmetrically is NP-complete in general [18]. Exact algorithms are devised based on Branch and Cut approach by Buchheim and Junger [3] and a group-theoretic approach by Abelson et al. [1]. Linear-time algorithms are available for trees and outerplanar graphs by Manning and Atallah [19, 20] and for series-parallel digraphs by Hong et al. [14]. Linear-time algorithms are presented for maximally symmetric drawings of triconnected planar graphs by Hong et al. [15] and for biconnected, oneconnected, and disconnected planar graphs by Hong and Eades [10, 12, 13]. Hong and Nagamochi presented a linear-time algorithm for constructing a *symmetric convex* drawings of internally trconnected planar graphs [16]. For a survey on symmetric drawings of graphs in two dimensions, see [5].

In three dimensions, the problem of determining whether a graph can be drawn symmetrically in three dimensions is *NP-hard* in general [8]. A group-theoretic characterization of symmetric drawing in n-dimensions and exact algorithms based on a group-theoretic approach are given

by Abelson et al. [1]. Linear-time algorithms are available for trees by Hong and Eades [9], series-parallel digraphs by Hong et al. [11], and biconnected and oneconnected planar graphs [8].

In this article, we review a linear-time algorithm for constructing maximally symmetric straight-line drawings of *triconnected planar graphs* by Hong, McKay, and Eades [15]. The following theorem summarizes their main results.

Theorem 1 *There is a linear-time algorithm that constructs straight-line drawings of maximally symmetric planar drawings of triconnected planar graphs.*

Computing a Planar Automorphism Group of Maximum Size

We first review the first step of the algorithm, i.e., symmetry finding step for triconnected planar graphs [15]. A geometric automorphism group A of a graph G is a *planar automorphism group*, if there is a *planar* drawing of the graph that displays every element of A.

Suppose that A is a group acting on a set X. The *stabilizer* of $x \in X$, denoted by $stab_A(x)$, is $\{g \in A \mid g(x) = x\}$, and the *orbit* of x, denoted by $orbit_A(x)$, is $\{g(x) \mid g \in A\}$. We say that $g \in A$ *fixes* $x \in X$ if $g(x) = x$; if g fixes x for every $g \in A$, then A fixes x. If $X' \subseteq X$ and $\phi(x') \in X'$ for all $x' \in X'$, then g *fixes* X'. Automorphisms $g_1, g_2, \ldots, g_k$ are called *generators* of $\langle g_1, g_2, \ldots, g_k \rangle$; the group consists of all permutations formed from products of elements of $\{g_1, g_2, \ldots, g_k\}$.

Hong et al. [15] characterize planar automorphisms as below.

Lemma 1 *Let G be a triconnected planar graph. An automorphism of G is a planar automorphism if and only if it fixes a face of G.*

To find the best plane embedding to compute a planar automorphism group with a maximum size, the algorithm uses the Stabilizer-Orbit theorem in group theory [2].

Theorem 2 (Stabilizer-Orbit theorem) *Suppose that A is a group acting on a set X and let $x \in X$. Then $|A| = |orbit_A(x)| \times |stab_A(x)|$.*

The overall algorithm computing a maximum size planar automorphism group of a triconnected planar graph can be described as follows;

Algorithm `Compute_Max_PAG`

1. Find a plane embedding which has a maximum size planar automorphism group.
2. Perform "star triangulation" for the given embedding.
3. Compute the *generators* of the planar automorphism group of the new embedding.

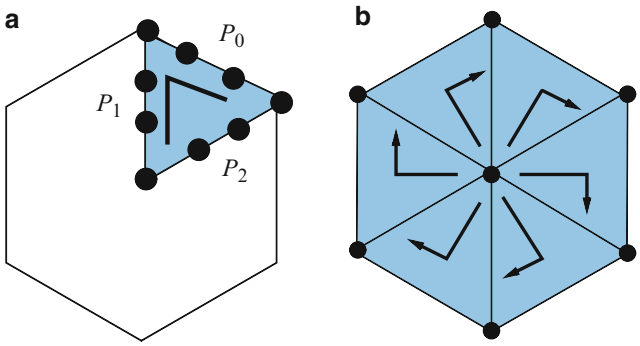
The first step of `Compute_Max_PAG` uses two applications of an algorithm of Fontet [7], which computes the orbits on vertices of the (full) automorphism group of a triconnected planar graph in linear time.

Theorem 3 *Fontet's algorithm [7] can be used to find a plane embedding of a triconnected graph G such that the corresponding planar automorphism group is maximized in linear time.*

Proof Based on Lemma 1, we take a dual graph of G^* of G and compute the orbits of G^* using Fontet's algorithm [7]. Choose an orbit O of minimum size; the stabilizer O has the maximum size, by Theorem 2. Taking a face $f \in O$ as the outer face of the plane embedding of G, we have an embedding that displays the maximum number of symmetries.

Once the outer face and thus the plane embedding is chosen, the second step of `Compute_Max_PAG` performs *star triangulation*, i.e., triangulate each internal face f by inserting a new vertex v in the face and joining v to each vertex of f. Clearly, this step takes linear time and simplifies the drawing algorithm.

The final step of `Compute_Max_PAG` is to compute the planar automorphism group for star-triangulated plane graph. Since an explicit representation of the planar automorphism group may take more than linear space, for a more compact representation, an algorithm for computing *minimal generators* was devised. For details on a linear-time algorithm for computing generators of a planar automorphism group, see [15].



Symmetric Graph Drawing, Fig. 1 Example of (**a**) a wedge and (**b**) merging step

Overview of the Drawing Algorithm

We now review a linear-time drawing algorithm for constructing a symmetric drawing of a triconnected planar graph that achieves that maximum with straight-line edges. The main characteristic of symmetric drawings is the repetition of congruent drawings of isomorphic subgraphs. To exploit this property, the drawing algorithm uses a divide and conquer approach: (i) divide the graph into isomorphic subgraphs; (ii) compute a drawing for a subgraph; and (iii) merge multiple copies of drawings of subgraphs to construct a symmetric drawing of the whole graph. Overall, each step of the drawing algorithm runs in linear time.

The input of the drawing algorithm is a triconnected planar graph with fixed plane embedding and a specified outer face, which maximize the number of symmetries. The symmetric drawing algorithm takes a different approach for each type of planar automorphism group: i.e., *cyclic* case, *one axial* case, and *dihedral* case.

The Cyclic Case

Here we describe how to display k rotational symmetries. Note that after star triangulation, there is a *central* vertex c, which is fixed by the planar automorphism group for $k \geq 3$. If $k = 2$, there exits either a central vertex or a central edge. If there is a central edge, then we preprocess the graph by inserting a dummy central vertex c into the central edge with two dummy edges.

The rotational symmetric drawing algorithm consists of three steps:

Algorithm **Cyclic**

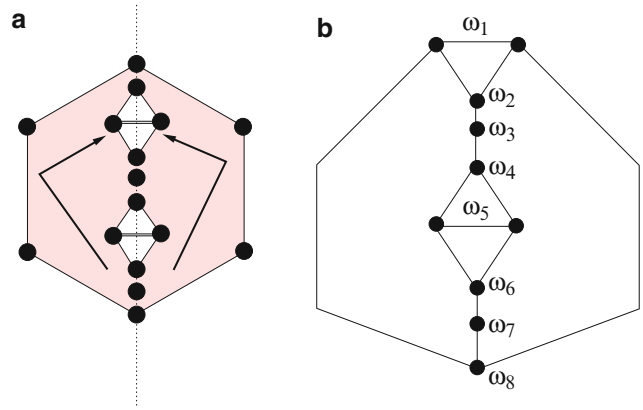
1. `Find_Wedge_Cyclic`.
2. `Draw_Wedge_Cyclic`.
3. `Merge_Wedges_Cyclic`.

The first step is to find a subgraph *wedge* W, which takes linear time:

Algorithm `Find_Wedge_Cyclic`

1. Find the *central* vertex c.
2. Find a shortest path P_1, from c to a vertex v_1 on the outer face, using breadth first search.
3. Find the path P_2 which is a mapping of P_1 under a minimal generator of the rotation.
4. Find the wedge W (see Fig. 1a), an induced subgraph of G enclosed by the cycle formed from P_1, P_2 and a path P_0 along the outer face from v_1 to v_2.

The second step, `Draw_Wedge_Cyclic`, constructs a drawing D of the wedge W using Algorithm `CYN`, the linear-time convex drawing algorithm by Chiba et al. [4], such that P_1, P_2, and P_0 are drawn as straight lines. The input to Algorithm `CYN` is an internally triconnected plane graph G with given outer face S and a straight-line drawing S^* of S as a *weakly convex polygon*, i.e., not every vertex of the outer face needs to be at an apex (i.e., the interior angle is less than π) of the polygon. Algorithm `CYN` chooses a vertex v and deletes it from G together with incident edges and divides the resulting graph $G' = G - v$ into the biconnected components $B_1, B_2, \ldots, B_p$, $p \geq 1$. It defines a convex polygon S_i^* of the outer facial cycle S_i of



Symmetric Graph Drawing, Fig. 2 Example of (**a**) a fixed string of diamonds and (**b**) ω_ℓ

each B_i and recursively applies the algorithm to draw B_i with S_i^* as outer boundary. For details, see [4].

The last step, `Merge_Wedges_Cyclic`, constructs a drawing of the whole graph G by replicating the drawing D of W, k times. Note that this merge step relies on the fact that P_1 and P_2 are drawn as straight lines. See Fig. 1b.

It is clear that Algorithm `Cyclic` constructs a straight-line drawing of a triconnected plane graph which shows k rotational symmetry in linear time.

One Axial Symmetry

Consider a drawing of a star-triangulated plane graph with one axial symmetry. There are fixed vertices, edges, and/or fixed faces on the axis; we need to characterize the subgraph formed by these.

A *diamond* is either a triangle or the 4-vertex graph. A *string of diamonds* is a graph formed from a path $P = (v_1, v_2, \ldots, v_k)$, $k \geq 2$, by a number (zero or greater) of "splitting" operations, as follows. If $1 \leq i \leq k-1$, then the edge (v_i, v_{i+1}) may be replaced by a diamond. Alternatively, each of the end edges (v_1, v_2) and (v_{k-1}, v_k) may be replaced by a triangle. Note that a string of diamonds is basically a path consisting of edges and diamonds; each end of the path may be a triangle; see Fig. 2a.

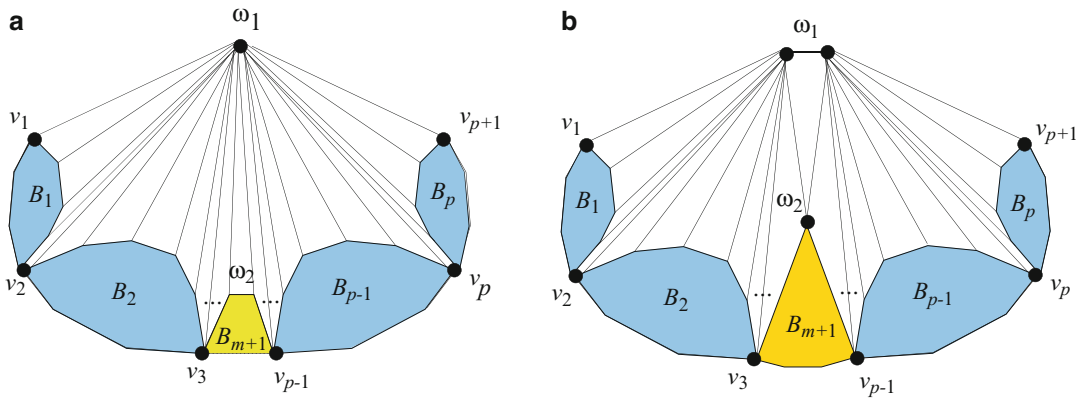
To display a single axial symmetry, we need two steps. First we identify the *fixed string of diamonds*; then use Algorithm `Symmetric_CYN`, a modified version of Algorithm `CYN`. More formally, the algorithm `One_Axial` is described below.

Algorithm `One_Axial`

1. Find a fixed string of diamonds. Suppose that $\omega_1, \omega_2, \ldots, \omega_k$ are the fixed edges and vertices in the fixed string of diamonds, in order from the outer face (ω_1 is on the outer face). For each ℓ, ω_ℓ may be a vertex or an edge (see Fig. 2b).
2. Choose a symmetric convex polygon S^* for the outer face S of G.
3. `Symmetric_CYN`$(1, S^*, G, y_1)$.

The main ingredient in Algorithm `One_Axial` is Algorithm `Symmetric_CYN`. To modify Algorithm `CYN` to display a single axial symmetry, the following three conditions should be satisfied:

- Choose the first vertex or edge on the fixed string of diamonds ω_1 (see Fig. 3).
- Let $D(B_i)$ be the drawing of B_i and α be the axial symmetry. Then, $D(B_i)$ should be a reflection of $D(B_j)$, where $B_j = \alpha(B_i)$, $i = 1, 2, \ldots, m$ and $m = \lfloor p/2 \rfloor$: To satisfy this condition, define S_j^* to be the reflection of S_i^*, $i = 1, 2, \ldots, m$. Then we apply Algorithm `CYN` for $B_i, i = 1, 2, \ldots, m$ and construct $D(B_j)$ using a reflection of $D(B_i)$.
- If p is odd, then $D(B_{m+1})$ should display axial symmetry: To satisfy this condition, we recursively apply Algorithm `Symmetric_CYN` to B_{m+1}.



Symmetric Graph Drawing, Fig. 3 Example of a symmetric version of CYN

Note that the position of ω_2 in Fig. 3 can be chosen arbitrarily along the axis of symmetry of S^* within S^*. This means that we can specify the positions of the fixed vertices and middle edges along the axis of symmetry a priori, that is, as input to the algorithm. The Algorithm `Symmetric_CYN` can be described as below:

Algorithm `Symmetric_CYN`

input: ℓ: index of vertex or middle edge on the fixed string of diamonds.
input: S^*: a weakly convex polygon of the outer face of S of G.
input: G: a triangulated planar graph.
input: y_ℓ: a position on the axis of symmetry for the fixed vertex or the fixed edge ω_ℓ.

1. Delete ω_ℓ from G together with edges incident to ω_ℓ. Divide the resulting graph $G' = G - \omega_\ell$ into the blocks $B_1, B_2, \ldots, B_p$, $p \geq 1$, ordered anticlockwise around the outer face. Let $m = \lfloor p/2 \rfloor$.
2. Determine a convex polygon S_i^* of the outer facial cycle S_i of each B_i such that B_i with S_i^* satisfy the conditions for convex drawing algorithm `CYN` and S_{p-i+1}^* is a reflection of S_i^*.
3. For each $i = 1$ to m,
   (a) Construct a drawing $D(B_i)$ of B_i using Algorithm `CYN`.
   (b) Construct $D(B_{p-i+1})$ as a reflection of $D(B_i)$.
4. If p is odd, then construct a drawing $D(B_{m+1})$ using `Symmetric_CYN`$(\ell + 1, S_{m+1}^*, B_{m+1}, y_{\ell+1})$.
5. Merge the $D(B_i)$ to form a drawing of G, placing ω_ℓ at y_ℓ.

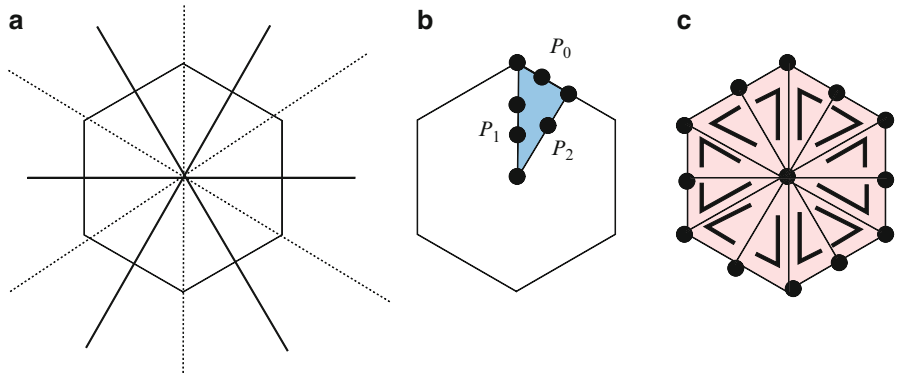
Since Algorithm `CYN` [4] runs in linear time, clearly Algorithm `Symmetric_CYN` and Algorithm `One_Axial` takes linear time.

The Dihedral Case

We now review an algorithm for displaying a dihedral group $< \rho, \alpha >$, where ρ is a rotation of order k and α is an axial automorphism. As with the cyclic case, we assume that there is a central vertex.

The drawing algorithm adopts the same strategy as for the cyclic case: (i) divide the graph into "wedges"; (ii) draw each wedge; and (iii) merge the drawings of wedges to construct a symmetric drawing of the whole graph. However, the dihedral case is more difficult than the cyclic case, because an axial symmetry in the dihedral group can have fixed faces as well as fixed edges; i.e., the boundary of a wedge may be a fixed string of diamonds as in the one axial case. To achieve dihedral symmetry, the axis of symmetry must be the perpendicular bisector of the middle edge of each diamond. This makes the merging operation more difficult.

Consider a drawing of a triconnected planar graph with a dihedral symmetry group of size $2k$.



Symmetric Graph Drawing, Fig. 4 Wedge for the dihedral case

There are k axial symmetries, with axes at angles of $\pi i/k$, $0 \leq i \leq k-1$, to the x axis, as in Fig. 4a. Roughly speaking, a wedge is the area between two adjacent axes, as in Fig. 4b. Note that in these wedges, the boundaries P_1 and P_2 may be strings of diamonds. These may terminate in a triangle.

As with the cyclic case, Algorithm `Dihedral` has three steps: (i) `Find_Wedge_Dihedral`, (ii) `Draw_Wedge_Dihedral`, and (iii) `Merge_Wedges_Dihedral`.

The first step is to define the "wedge" subgraph by finding two fixed strings of diamonds. Note that one can find the central vertex c and the two fixed strings of diamonds P_1 and P_2 in linear time using the generators of the group.

Algorithm `Find_Wedge_Dihedral`

1. Find the central vertex c.
2. Find a string of diamonds P_1 that is fixed by α, from c to a vertex v or an edge e on the outer face.
3. Traverse the outer face, clockwise from v (or e) to the vertex v' or edge e' that is fixed by $\rho^{-1}\alpha\rho$. Let P_0 denote the path so traversed.
4. Find the string of diamonds P_2 for $\rho^{-1}\alpha\rho$, from c to v' (or e').
5. Define the wedge W to be the subgraph enclosed by P_0, P_1, and P_2, including the vertices and edges of P_0, P_1, and P_2.

The second step, `Draw_Wedge_Dihedral`, constructs a drawing of the wedge, which is the most complicated step of the drawing algorithm. This step must ensure that the middle edge of each diamond on the boundary is orthogonal to the axis of reflection.

Roughly speaking, the algorithm `Draw_Wedge_Dihedral` runs as follows: (i) Find all *special diamonds* of P_1 and P_2 that share fixed vertices or fixed edges, and draw them first using algorithm `Draw_Special_Diamonds`; (ii) choose the positions of all the fixed vertices of P_1 and P_2 that have not been drawn so far; (iii) subdivide the wedge in various ways to form "subwedges"; (iv) draw each of these subwedges using Algorithms `CYN` and `Symmetric_CYN` accordingly. For details, see [15].

The final step, Algorithm `Merge_Wedges_Dihedral` simply constructs a drawing for the whole graph by merging the drawing D of the wedge W. Clearly each step of Algorithm `Dihedral` takes linear time.

Cross-References

▸ Convex Graph Drawing

Recommended Reading

1. Abelson D, Hong S, Taylor DE (2007) Geometric automorphism groups of graphs. Discret Appl Math 155(17):2211–2226. Elsevier
2. Armstrong MA (1988) Groups and symmetry. Springer, New York

3. Buchheim C, Junger M (2003) Detecting symmetries by branch and cut. Math Progr (Ser B) 98:369–384
4. Chiba N, Yamanouchi T, Nishizeki T (1984) Linear algorithms for convex drawings of planar graphs. In: Adrian Bondy J, Murty USR (eds) Progress in graph theory. Academic, Toronto/Orlando, pp 153–173
5. Eades P, Hong S (2013) Detection and display of symmetries. In: Tamassia R (ed) Handbook of graph drawing and visualisation. Chapman and Hall/CRC
6. Eades P, Lin X (2000) Spring algorithms and symmetry. Theor Comput Sci 240(2):379–405
7. Fontet M (1976) Linear algorithms for testing isomorphism of planar graphs. In: Proceedings of third colloquium on automata, languages and programming, Edinburgh, pp 411–423
8. Hong S (2002) Drawing graphs symmetrically in three dimensions. In: Proceeding of graph drawing 2001, Vienna. Lecture notes in computer science, vol 2265. Springer, pp 189–204
9. Hong S, Eades P (2003) Drawing trees symmetrically in three dimensions. Algorithmica 36(2):153–178. Springer
10. Hong S, Eades P (2003) Symmetric layout of disconnected graphs. In: Algorithms and computation (Proceedings of ISAAC 2003, Kyoto). Lecture notes in computer science, vol 2906. Springer, Berlin/New York, pp 405–414
11. Hong S, Eades P (2004) Linkless symmetric drawings of series parallel digraphs. Comput Geom Theory Appl 29(3):191–222. Elsevier
12. Hong S, Eades P (2005) Drawing planar graphs symmetrically II: biconnected graphs. Algorithmica 42(2):159–197. Springer
13. Hong S, Eades P (2006) Drawing planar graphs symmetrically III: oneconnected graphs. Algorithmica 44(1):67–100. Springer
14. Hong S, Eades P, Lee S (2000) Drawing series parallel digraphs symmetrically. Comput Geom Theory Appl 17(3–4):165–188
15. Hong S, McKay B, Eades P (2006) A linear time algorithm for constructing maximally symmetric straight-line drawings of triconnected planar graphs. Discret Comput Geom 36(2):283–311. Springer
16. Hong S, Nagamochi H (2010) Linear time algorithm for symmetric convex drawings of planar graphs. Algorithmica 58(2):433–460. Springer
17. Hopcroft JE, Wong JK (1974) Linear time algorithm for isomorphism of planar graphs. In: Proceedings of ACM symposium on theory of computing, Seattle, pp 172–184
18. Lubiw A (1981) Some NP-complete problems similar to graph isomorphism. SIAM J Comput 10(1):11–21
19. Manning J, Atallah MJ (1988) Fast detection and display of symmetry in trees. Congr Numer 64:159–169
20. Manning J, Atallah MJ (1992) Fast detection and display of symmetry in outerplanar graphs. Discret Appl Math 39:13–35

Synchronizers, Spanners

Michael Elkin
Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel

Keywords

Low-stretch spanning subgraphs; Network synchronization

Years and Authors of Summarized Original Work

1985; Awerbuch

Problem Definition

Consider a communication network, modeled by an n-vertex undirected unweighted graph $G = (V, E)$, for some positive integer n. Each vertex of G hosts a processor of unlimited computational power; the vertices have unique identity numbers, and they communicate via the edges of G by sending messages of size $O(\log n)$ each.

In the *synchronous* setting, the communication occurs in discrete *rounds*, and a message sent in the beginning of a round R arrives at its destination before the round R ends. In the *asynchronous* setting, each vertex maintains its own clock, and clocks of distinct vertices may disagree. It is assumed that each message sent (in the asynchronous setting) arrives at its destination within a certain time τ after it was sent, but the value of τ is not known to the processors.

It is generally much easier to devise algorithms that apply to the synchronous setting (henceforth, synchronous algorithms) rather than to the asynchronous one (henceforth, asynchronous algorithms). In [1] Awerbuch initiated the study of simulation techniques that translate synchronous algorithms to asynchronous ones. These simulation techniques are called *synchronizers*.

To devise the first synchronizers, Awerbuch [1] constructed a certain graph partition which is of its own interest. In particular, Peleg and Schäffer noticed [8] that this graph partition induces a subgraph with certain interesting properties. They called this subgraph a *graph spanner*. Formally, for a positive integer parameter k, a k-spanner of a graph $G = (V, E)$ is a subgraph $G' = (V, H)$, $H \subseteq E$, such that for every edge $e = (v, u) \in E$, the distance between the vertices v and u in H, $\text{dist}'_G(v, u)$, is at most k.

Key Results

Awerbuch devised three basic synchronizers, called α, β, and γ. The synchronizer α is the simplest one; using it results in only a constant overhead in time, but in a very significant overhead in communication. Specifically, the latter overhead is linear in the number of edges of the underlying network. Unlike the synchronizer α, the synchronizer β requires a somewhat costly initialization stage. In addition, using it results in a significant time overhead (linear in the number of vertices n), but it is more communication efficient than α. Specifically, its communication overhead is linear in n.

Finally, the synchronizer γ represents a trade-off between the synchronizers α and β. Specifically, this synchronizer is parametrized by a positive integer parameter k. When k is small, then the synchronizer behaves similarly to the synchronizer α, and when k is large, it behaves similarly to the synchronizer β. A particularly important choice of k is $k = \log n$. At this point on the trade-off curve, the synchronizer γ has a logarithmic in n time overhead and a linear in n communication overhead. The synchronizer γ has, however, a quite costly initialization stage.

The main result of [1] concerning spanners is that for every $k = 1, 2, \ldots$, and every n-vertex unweighted undirected graph $G = (V, E)$, there exists an $O(k)$-spanner with $O(n^{1+1/k})$ edges. (This result was explicated by Peleg and Schäffer [8].)

Applications

Synchronizers are extensively used for constructing asynchronous algorithms. The first applications of synchronizers are constructing the *breadth-first-search tree* and computing the *maximum flow*. These applications were presented and analyzed by Awerbuch in [1]. Later synchronizers were used for maximum matching [10], for computing shortest paths [7], and for other problems.

Graph spanners were found useful for a variety of applications in distributed computing. In particular, some constructions of synchronizers employ graph spanners [1, 9]. In addition, spanners were used for routing [4] and for computing almost shortest paths in graphs [5].

Open Problems

Synchronizers with improved properties were devised by Awerbuch and Peleg [3] and Awerbuch et al. [2]. Both these synchronizers have polylogarithmic time and communication overheads. However, the synchronizers of Awerbuch and Peleg [3] require a large initialization time. (The latter is at least linear in n.) On the other hand, the synchronizers of [2] are randomized. A major open problem is to obtain *deterministic* synchronizers with polylogarithmic time and communication overheads and sublinear in n initialization time. In addition, the degrees of the logarithm in the polylogarithmic time and communication overheads in synchronizers of [2, 3] are quite large. Another important open problem is to construct synchronizers with improved parameters.

In the area of spanners, spanners that distort large distances to a significantly smaller extent than they distort small distances were constructed by Elkin and Peleg in [6]. These spanners fall short from achieving a *purely additive distortion*. Constructing spanners with a purely additive distortion is a major open problem.

Cross-References

▸ Sparse Graph Spanners

Recommended Reading

1. Awerbuch B (1985) Complexity of network synchronization. J ACM 4:804–823
2. Awerbuch B, Patt-Shamir B, Peleg D, Saks ME (1992) Adapting to asynchronous dynamic networks. In: Proceedings of the 24th annual ACM symposium on theory of computing, Victoria, 4–6 May 1992, pp 557–570
3. Awerbuch B, Peleg D (1990) Network synchronization with polylogarithmic overhead. In: Proceedings of the 31st IEEE symposium on foundations of computer science, Sankt Louis, 22–24 Oct 1990, pp 514–522
4. Awerbuch B, Peleg D (1992) Routing with polynomial communication-space tradeoff. SIAM J Discret Math 5:151–162
5. Elkin M (2001) Computing almost shortest paths. In: Proceedings of the 20th ACM symposium on principles of distributed computing, Newport, 26–29 Aug 2001, pp 53–62
6. Elkin M, Peleg D (2001) Spanner constructions for general graphs. In: Proceedings of the 33th ACM symposium on theory of computing, Heraklion, 6–8 July 2001, pp 173–182
7. Lakshmanan KB, Thulasiraman K, Comeau MA (1989) An efficient distributed protocol for finding shortest paths in networks with negative cycles. IEEE Trans Softw Eng 15:639–644
8. Peleg D, Schäffer A (1989) Graph spanners. J Graph Theory 13:99–116
9. Peleg D, Ullman JD (1989) An optimal synchronizer for the hypercube. SIAM J Comput 18:740–747
10. Schieber B, Moran S (1986) Slowing sequential algorithms for obtaining fast distributed and parallel algorithms: maximum matchings. In: Proceedings of 5th ACM symposium on principles of distributed computing, Calgary, 11–13 Aug 1986, pp 282–292