E

Edit Distance Under Block Operations

S. Cenk Sahinalp
Laboratory for Computational Biology, Simon Fraser University, Burnaby, BC, USA

Keywords

Block edit distance

Years and Authors of Summarized Original Work

2000; Cormode, Paterson, Sahinalp, Vishkin
2000; Muthukrishnan, Sahinalp

Problem Definition

Given two strings $S = s_1 s_2 \ldots s_n$ and $R = r_1 r_2 \ldots r_m$ (wlog let $n \geq m$) over an alphabet $\sigma = \{\sigma_1, \sigma_2, \ldots \sigma_\ell\}$, the *standard edit distance* between S and R, denoted $ED(S, R)$ is the minimum number of *single character edits*, specifically *insertions*, *deletions* and *replacements*, to transform S into R (equivalently R into S).

If the input strings S and R are permutations of the alphabet σ (so that $|S| = |R| = |\sigma|$) then an analogous *permutation edit distance* between S and R, denoted $PED(S, R)$ can be defined as the minimum number of single character *moves*, to transform S into R (or vice versa).

A generalization of the standard edit distance is *edit distance with moves*, which, for input strings S and R is denoted $EDM(S, R)$, and is defined as the minimum number of character edits and *substring (block) moves* to transform one of the strings into the other. A move of block $s[j, k]$ to position h transforms $S = s_1 s_2 \ldots s_n$ into $S' = s_1 \ldots s_{j-1} s_{k+1} s_{k+2} \ldots s_{h-1} s_j \ldots s_k s_h \ldots s_n$ [4].

If the input strings S and R are permutations of the alphabet σ (so that $|S| = |R| = |\sigma|$) then $EDM(S, R)$ is also called as the transposition distance and is denoted $TED(S, R)$ [1].

Perhaps the most general form of the standard edit distance that involves edit operations on blocks/substrings is the *block edit distance*, denoted $BED(S, R)$. It is defined as the minimum number of single character edits, block moves, as well as *block copies and block uncopies* to transform one of the strings into the other. Copying of a block $s[j, k]$ to position h transforms $S = s_1 s_2 \ldots s_n$ into $S' = s_1 \ldots s_j s_{j+1} \ldots s_k \ldots s_{h-1} s_j \ldots s_k s_h \ldots s_n$. A block uncopy is the inverse of a block copy: it deletes a block $s[j, k]$ provided there exists $s[j', k'] = s[j, k]$ which does not overlap with $s[j, k]$ and transforms S into $S' = s_1 \ldots s_{j-1} s_{k+1} \ldots s_n$.

Throughout this discussion all edit operations have unit cost and they may overlap; i.e., a character can be edited on multiple times.

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

Key Results

There are exact and approximate solutions to computing the edit distances described above with varying performance guarantees. As can be expected, the best available running times as well as the approximation factors for computing these edit distances vary considerably with the edit operations allowed.

Exact Computation of the Standard and Permutation Edit Distance

The fastest algorithms for exactly computing the standard edit distance have been available for more than 25 years.

Theorem 1 (Levenshtein [9]) *The standard edit distance ED(S, R) can be computed exactly in time $O(n \cdot m)$ via dynamic programming.*

Theorem 2 (Masek-Paterson [11]) *The standard edit distance ED(S, R) can be computed exactly in time $O(n + n \cdot m/\log^2_{|\sigma|} n)$ via the "four-Russians trick".*

Theorem 3 (Landau-Vishkin [8]) *It is possible to compute ED(S, R) in time $O(n \cdot ED(S, R))$.*

Finally, note that if S and R are permutations of the alphabet σ, $PED(S, R)$ can be computed much faster than the standard edit distance for general strings: Observe that $PED(S, R) = n - LCS(S, R)$ where $LCS(S, R)$ represents the longest common subsequence of S and R. For permutations S, R, $LCS(S, R)$ can be computed in time $O(n \cdot \log\log n)$ [3].

Approximate Computation of the Standard Edit Distance

If some approximation can be tolerated, it is possible to considerably improve the $\tilde{O}(n \cdot m)$ time ($\tilde{O}$ notation hides polylogarithmic factors) available by the techniques above. The fastest algorithm that *approximately* computes the standard edit distance works by *embedding* strings S and R from alphabet σ into shorter strings S' and R' from a larger alphabet σ' [2]. The embedding is achieved by applying a general version of the *Locally Consistent Parsing* [13, 14] to partition the strings R and S into *consistent blocks* of size c to $2c - 1$; the partitioning is consistent in the sense that identical (long) substrings are partitioned identically. Each block is then replaced with a label such that identical blocks are identically labeled. The resulting strings S' and R' preserve the edit distance between S and R approximately as stated below.

Theorem 4 (Batu-Ergun-Sahinalp [2]) *ED(S, R) can be computed in time $\tilde{O}(n^{1+\epsilon})$ within an approximation factor of* $\min\{n^{\frac{1-\epsilon}{3}+o(1)}, (ED(S, R)/n^{\epsilon})^{\frac{1}{2}+o(1)}\}$.

For the case of $\epsilon = 0$, the above result provides an $\tilde{O}(n)$ time algorithm for approximating $ED(S, R)$ within a factor of $\min\{n^{\frac{1}{3}+o(1)}, ED(S, R)^{\frac{1}{2}+o(1)}\}$.

Approximate Computation of Edit Distances Involving Block Edits

For all edit distance variants described above which involve blocks, there are no known polynomial time algorithms; in fact it is NP-hard to compute $TED(S, R)$ [1], $EDM(S, R)$ and $BED(S, R)$ [10]. However, in case S and R are permutations of σ, there are polynomial time algorithms that approximate transposition distance within a constant factor:

Theorem 5 (Bafna-Pevzner [1]) *TED(S, R) can be approximated within a factor of 1.5 in $O(n^2)$ time.*

Furthermore, even if S and R are arbitrary strings from σ, it is possible to approximately compute both $BED(S, R)$ and $EDM(S, R)$ in near linear time. More specifically obtain an embedding of S and R to binary vectors $f(S)$ and $f(R)$ such that:

Theorem 6 (Muthukrishnan-Sahinalp [12]) $\frac{\|f(S)-f(R)\|_1}{\log^* n} \leq BED(S, R) \leq \|f(S) - f(R)\|_1 \cdot \log n$.

In other words, the Hamming distance between $f(S)$ and $f(R)$ approximates $BED(S, R)$ within a factor of $\log n \cdot \log^* n$. Similarly for $EDM(S, R)$, it is possible to embed S and R to integer valued vectors $F(S)$ and $F(R)$ such that:

Theorem 7 (Cormode-Muthukrishnan [4]) $\frac{||F(S)-F(R)||_1}{\log^* n} \leq EDM(S,R) \leq ||F(S) - F(R)||_1 \cdot \log n$.

In other words, the L_1 distance between $F(S)$ and $F(R)$ approximates $EDM(S, R)$ within a factor of $\log n \cdot \log^* n$.

The embedding of strings S and R into binary vectors $f(S)$ and $f(R)$ is introduced in [5] and is based on the Locally Consistent Parsing described above. To obtain the embedding, one needs to hierarchically partition S and R into growing size *core* blocks. Given an alphabet σ, Locally Consistent Parsing can identify only a limited number of substrings as core blocks. Consider the lexicographic ordering of these core blocks. Each dimension i of the embedding $f(S)$ simply indicates (by setting $f(S)[i] = 1$) whether S includes the ith core block corresponding to the alphabet σ as a substring. Note that if a core block exists in S as a substring, Locally Consistent Parsing will identify it.

Although the embedding above is exponential in size, the resulting binary vector $f(S)$ is very sparse. A simple representation of $f(S)$ and $f(R)$, exploiting their sparseness can be computed in time $O(n \log^* n)$ and the Hamming distance between $f(S)$ and $f(R)$ can be computed in linear time by the use of this representation [12].

The embedding of S and R into integer valued vectors $F(S)$ and $F(R)$ are based on similar techniques. Again, the total time needed to approximate $EDM(S, R)$ within a factor of $\log n \cdot \log^* n$ is $O(n \log^* n)$.

Applications

Edit distances have important uses in computational evolutionary biology, in estimating the evolutionary distance between pairs of genome sequences under various edit operations. There are also several applications to the *document exchange problem* or *document reconciliation problem* where two copies of a text string S have been subject to edit operations (both single character and block edits) by two parties resulting in two versions S_1 and S_2, and the parties communicate to reconcile the differences between the two versions. An information theoretic lower bound on the number of bits to communicate between the two parties is then $\Omega(BED(S,R)) \cdot \log n$. The embedding of S and R to binary strings $f(S)$ and $f(R)$ provides a simple protocol [5] which gives a near-optimal tradeoff between the number of rounds of communication and the total number of bits exchanged and works with high probability.

Another important application is to the Sequence Nearest Neighbors (SNN) problem, which asks to preprocess a set of strings $S_1, \ldots, S_k$ so that given an on-line query string R, the string S_i which has the lowest distance of choice to R can be computed in time polynomial with $|R|$ and polylogarithmic with $\sum_{j=1}^{k} |S_j|$. There are no known exact solutions for the SNN problem under any edit distance considered here. However, in [12], the embedding of strings S_i into binary vectors $f(S_i)$, combined with the Approximate Nearest Neighbors results given in [6] for Hamming Distance, provides an approximate solution to the SNN problem under block edit distance as follows.

Theorem 8 (Muthukrishnan-Sahinalp [12]) *It is possible to preprocess a set of strings $S_1, \ldots, S_k$ from a given alphabet σ in $O(poly(\sum_{j=1}^{k} |S_j|))$ time such that for any on-line query string R from σ one can compute a string S_i in time $O(polylog(\sum_{j=1}^{k} |S_j|) \cdot poly(|R|))$ which guarantees that for all $h \in [1,k]$, $BED(S_i, R) \leq BED(S_h, R) \cdot \log(\max_j |S_j|) \cdot \log^*(\max_j |S_j|)$.*

Open Problems

It is interesting to note that when dealing with permutations of the alphabet σ the problem of computing both character edit distances and block edit distances become much easier; one can compute $PED(S, R)$ exactly and $TED(S, R)$ within an approximation factor of 1.5 in $\tilde{O}(n)$ time. For arbitrary strings, it is an open question whether one can approximate $TED(S, R)$ or $BED(S, R)$ within a factor of $o(\log n)$ in polynomial time.

One recent result in this direction shows that it is not possible to obtain a polylogarithmic approximation to $TED(S, R)$ via a greedy strategy [7]. Furthermore, although there is a lower bound of $\Omega(n^{\frac{1}{3}})$ on the approximation factor that can be achieved for computing the standard edit distance in $\tilde{O}(n)$ time by the use of string embeddings, there is no general lower bound on how closely one can approximate $ED(S, R)$ in near linear time.

Cross-References

▶ Approximate String Matching

Recommended Reading

1. Bafna V, Pevzner PA (1998) Sorting by transpositions. SIAM J Discret Math 11(2):224–240
2. Batu T, Ergün F, Sahinalp SC (2006) Oblivious string embeddings and edit distance approximations. In: Proceedings of the ACM-SIAM SODA, pp 792–801
3. Besmaphyatnikh S, Segal M (2000) Enumerating longest increasing subsequences and patience sorting. Inf Process Lett 76(1–2):7–11
4. Cormode G, Muthukrishnan S (2002) The string edit distance matching problem with moves. In: Proceedings of the ACM-SIAM SODA, pp 667–676
5. Cormode G, Paterson M, Sahinalp SC, Vishkin U (2000) Communication complexity of document exchange. In: Proceedings of the ACM-SIAM SODA, pp 197–206
6. Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of fimensionality. In: Proceedings of the ACM STOC, pp 604–613
7. Kaplan H, Shafrir N (2005) The greedy algorithm for shortest superstrings. Info Process Lett 93(1):13–17
8. Landau G, Vishkin U (1989) Fast parallel and serial approximate string matching. J Algorithms 10:157–169
9. Levenshtein VI (1965) Binary codes capable of correcting deletions, insertions, and reversals. Dokl Akad Nauk SSSR 163(4):845–848 (Russian). (1966) Sov Phys Dokl 10(8):707–710 (English translation)
10. Lopresti DP, Tomkins A (1997) Block edit models for approximate string matching. Theor Comput Sci 181(1):159–179
11. Masek W, Paterson M (1980) A faster algorithm for computing string edit distances. J Comput Syst Sci 20:18–31
12. Muthukrishnan S, Sahinalp SC (2000) Approximate nearest neighbors and sequence comparison with block operations. In: Proceedings of the ACM STOC, pp 416–424
13. Sahinalp SC, Vishkin U (1994) Symmetry breaking for suffix tree construction. In: ACM STOC, pp 300–309
14. Sahinalp SC, Vishkin U (1996) Efficient approximate and dynamic matching of patterns using a labeling paradigm. In: Proceeding of the IEEE FOCS, pp 320–328

Efficient Decodable Group Testing

Hung Q. Ngo[1] and Atri Rudra[2]
[1]Computer Science and Engineering, The State University of New York, Buffalo, NY, USA
[2]Department of Computer Science and Engineering, State University of New York, Buffalo, NY, USA

Keywords

Coding theory; Nonadaptive group testing; Sublinear-time decoding

Years and Authors of Summarized Original Work

2011; Ngo, Porat, Rudra

Problem Definition

The basic group testing problem is to identify the unknown set of *positive items* from a large population of *items* using as few *tests* as possible. A test is a subset of items. A test returns positive if there is a positive item in the subset. The semantics of "positives," "items," and "tests" depend on the application.

In the original context [3], group testing was invented to solve the problem of identifying syphilis-infected blood samples from a large collection of WWII draftees' blood samples. In this case, items are blood samples, which are positive if they are infected. A test is a *pool* (group) of blood samples. Testing a group of samples at a time will save resources if the test outcome is negative. On the other hand, if the test outcome is positive, then all we know is that at least one sample in the pool is positive, but we do not know which one(s).

In *nonadaptive combinatorial group testing* (NACGT), we assume that the number of positives is at most d for some fixed integer d and that all tests have to be specified in advance before any test outcome is known. The NACGT paradigm has found numerous applications in many areas of mathematics, computer science, and computational biology [4, 9, 10].

A NACGT strategy with t tests on a universe of N items is represented by a $t \times N$ binary matrix $\mathbf{M} = (m_{ij})$, where $m_{ij} = 1$ iff item j belongs to test i. Let $\mathbf{M}_i$ and $\mathbf{M}^j$ denote row i and column j of $\mathbf{M}$, respectively. Abusing notation, we will also use $\mathbf{M}_i$ (respectively, $\mathbf{M}^j$) to denote the set of rows (respectively, columns) corresponding to the 1-entries of row i (respectively, column j). In other words, $\mathbf{M}_i$ is the ith pool, and $\mathbf{M}^j$ is the set of pools that item j belongs to.

Let $D \subset [N]$ be the unknown subset of positive items, where $|D| \le d$. Let $\boldsymbol{y} = (y_i)_{i=1}^t \in \{0,1\}^t$ denote the test outcome vector, i.e., $y_i = 1$ iff the ith test is positive. Then, the test outcome vector is precisely the (Boolean) union of the positive columns: $\boldsymbol{y} = \bigcup_{j \in D} \mathbf{M}^j$. The task of identifying the unknown subset D from the test outcome vector $\boldsymbol{y}$ is called *decoding*.

The main problem In many modern applications of NACGT, there are two key requirements for an NACGT scheme:

1. *Small number of tests*. "Tests" are computationally expensive in many applications.
2. *Efficient decoding*. As the item universe size N can be extremely large, it would be ideal for the decoding algorithm to run in time sublinear in N and more precisely in poly$(d, \log N)$ time.

Key Results

To be able to uniquely identify an arbitrary subset D of at most d positives, it is necessary and sufficient for the test outcome vectors $\boldsymbol{y}$ to be different for distinct subsets D of at most d positives. An NACGT matrix with the above property is called *d-separable*. However, in general such matrices only admit the brute force $\Omega(N^d)$-time decoding algorithm. A very natural decoding algorithm called the *naïve decoding algorithm* runs much faster, in time $O(tN)$.

Definition 1 (Naïve decoding algorithm) Eliminate all items that participate in negative tests; return the remaining items.

This algorithm does not work for arbitrary d-separable matrices. However, if the test matrix $\mathbf{M}$ satisfies a slightly stronger property called *d-disjunct*, then the naïve decoding algorithm is guaranteed to work correctly.

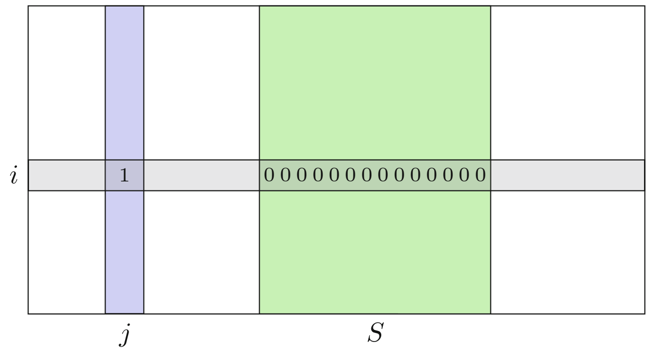
Definition 2 (Disjunct matrix) A $t \times N$ binary matrix $\mathbf{M}$ is said to be d-disjunct iff $\mathbf{M}^j \setminus \bigcup_{k \in S} \mathbf{M}^k \neq \emptyset$ for any set S of d columns and any $j \notin S$. (See Fig. 1.)

Minimize Number of Tests

It is remarkable that d-disjunct matrices not only allow for linear time decoding, which is a vast improvement over the brute-force algorithm for separable matrices, but also have asymptotically the same number of tests as d-separable matrices [4]. Let $t(d, N)$ denote the minimum number of rows of an N-column d-disjunct matrix. It has been known for about 40 years [5] that $t(\Omega(\sqrt{N}), N) = \Theta(N)$, and for $d = O(\sqrt{N})$ we have

$$\Omega\left(\frac{d^2}{\log d}\log N\right) \le t(d, N) \le O(d^2 \log N). \tag{1}$$

A $t \times N$ d-disjunct matrix with $t = O(d^2 \log N)$, rows can be constructed randomly or even deterministically (see [11]). However, the decoding time $O(tN)$ of the naïve decoding algorithm is still too slow for modern applications, where in most cases $d \ll N$ and thus $t \ll N$.



Efficient Decodable Group Testing, Fig. 1 A d-disjunct matrix has the following property: for any subset S of d (not necessarily contiguous) columns, and any column j that is not present in S, there exists a row i that has a 1 in column j and all zeros in S

Efficient Decoding

An ideal decoding time would be in the order of poly$(d, \log N)$, which is sublinear in N for practical ranges of d. Ngo, Porat, and Rudra [10] showed how to achieve this goal using a couple of ideas: (a) two-layer test matrix construction and (b) code concatenation using a list recoverable code.

(a) Two-layer test matrix construction The idea is to construct **M** by stacking on top of one another two matrices: a "filtering" matrix **F** and an "identification" matrix **D**. (See Fig. 2.) The filtering matrix is used to quickly identify a "small" set of L candidate items including *all* the positives. Then, the identification matrix is used to pinpoint precisely the positives. For example, let **D** be any d-disjunct matrix, and that from the tests corresponding to the rows of **F**, we can produce a set S of L = poly$(d, \log N)$ candidate items in time poly$(d, \log N)$. Then, by running the naïve decoding algorithm on S using test results corresponding to the rows of **D**, we can identify all the positives in time poly$(d, \log N)$. To formalize the notion of "filtering matrix," we borrow a concept from coding theory, where producing a small list of candidate codewords is the *list decoding problem* [6].

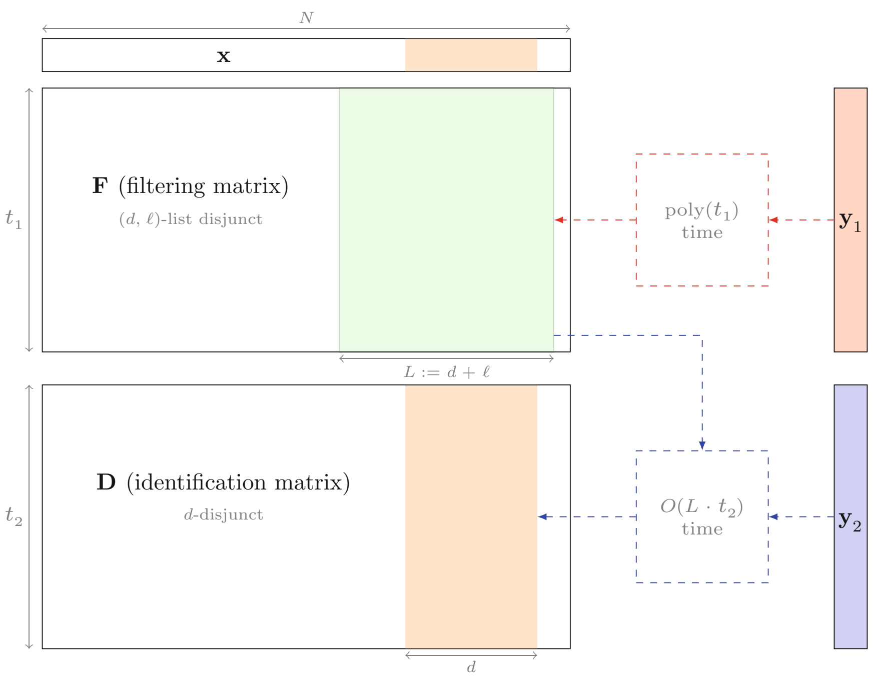
Definition 3 (List-disjunct matrix) Let $d + \ell \leq N$ be positive integers. A matrix **F** is (d, ℓ)-list disjunct if and only if $\bigcup_{j \in T} \mathbf{M}^j \setminus \bigcup_{k \in S} \mathbf{M}^k \neq \emptyset$ for any two disjoint sets S and T of columns of **F** with $|S| = d$ and $|T| = \ell$. (See Fig. 3.)

Note that a matrix is d-disjunct matrix iff it is $(d, 1)$-list disjunct. However, the relaxation to $\ell = \Theta(d)$ allows the existence (and construction) of $(d, O(d))$-list-disjunct matrices with $\Theta(d \log(N/d))$ rows. The existence of such small list-disjunct matrices is crucially used in the second idea below.

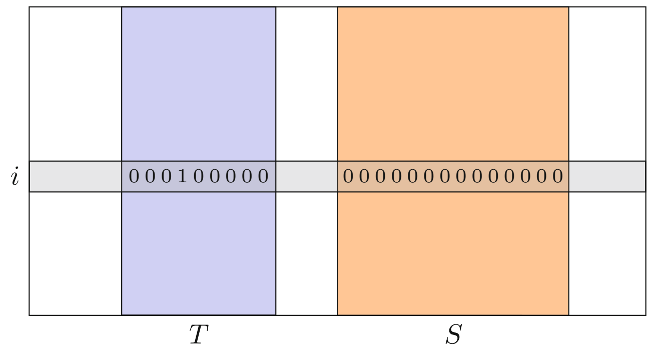
(b) Code Concatenation with list recoverable codes A $t \times N$ (d, ℓ)-list-disjunct matrix admits $O(tN)$-decoding time using the naïve decoding algorithm. However, to achieve poly$(d, \log N)$ decoding time overall, we will need to construct list-disjunct matrices that allow for a poly$(d, \log N)$ decoding time. In particular, to use such a matrix as a filtering matrix, it is necessary that ℓ = poly(d). To construct efficiently decodable list-disjunct matrices, we need other ideas. Ngo, Porat, and Rudra [10] used a connection to list recoverable codes [6] to construct such matrices. This connection was used to construct $(d, O(d^{3/2}))$-list-disjunct matrices with $t = o(d^2 \log_d N)$ rows that can be decoded in poly(t) time. This along with the construction in Fig. 2 implies the following result:

Theorem 1 ([10]) *Given any d-disjunct matrix, it can be converted into another matrix with $1 + o(1)$ times as many rows that is also efficiently decodable (even if the original matrix was not).*

Other constructions of list-disjunct matrices with worse parameters were obtained earlier by Indyk, Ngo and Rudra [7], and Cheraghchi [1] using connections to expanders and randomness extractors.



Efficient Decodable Group Testing, Fig. 2 The vector $\boldsymbol{x}$ denotes the characteristic vector of the d positives (illustrated by the *orange box*). The final matrix is the stacking of $\mathbf{F}$, which is a (d, ℓ)-list-disjunct matrix, and $\mathbf{D}$, which is a d-disjunct matrix. The result vector is naturally divided into $\boldsymbol{y}_1$ (the part corresponding to $\mathbf{F}$ and denoted by the *red vector*) and $\boldsymbol{y}_2$ (the part corresponding to $\mathbf{D}$ and denoted by the *blue vector*). The decoder first uses $\boldsymbol{y}_1$ to compute a superset of the set of positives (denoted by *green box*), which is then used with $\boldsymbol{y}_2$ to compute the final set of positives. The first step of the decoding is represented by the *red-dotted box*, while the second step (naïve decoder) is denoted by the *blue-dotted box*



Efficient Decodable Group Testing, Fig. 3 A (d, ℓ)-list-disjunct matrix satisfies the following property: for any subset S of size d and any disjoint subset T of size ℓ, there exists a row i that has a 1 in at least one column in T and all zeros in S

Applications

Heavy hitter is one of the most fundamental problems in data streaming [8]. Cormode and Muthukrishnan [2] showed that an NACGT scheme that is efficiently decodable and is also *explicit* solves a natural version of the heavy hitter problem. An explicit construction means one needs an algorithm that outputs a column or a specific entry of $\mathbf{M}$ instead of storing the entire matrix $\mathbf{M}$ which can be extremely space consuming. This is possible with Theorem 1 by picking the filtering and decoding matrices to be explicit.

Another important generalization of NACGT matrices are those that can handle errors in the test outcomes. Again this is possible with the construction of Fig. 2 if the filtering and decoding matrices are also error tolerant. The list-disjunct matrices constructed by Cheraghchi are also error tolerant [1].

Open Problems

The outstanding open problem in group testing theory is to close the gap (1). An explicit construction of (d, d)-list-disjunct matrices is not known; solving this problem will lead to a scheme that is (near-)optimal in all desired objectives.

Recommended Reading

1. Cheraghchi M (2013) Noise-resilient group testing: Limitations and constructions. Discret Appl Math 161(1–2):81–95
2. Cormode G, Muthukrishnan S (2005) What's hot and what's not: tracking most frequent items dynamically. ACM Trans Database Syst 30(1):249–278
3. Dorfman R (1943) The detection of defective members of large populations. Ann Math Stat 14(4):436–440
4. Du DZ, Hwang FK (2000) Combinatorial group testing and its applications. Series on applied mathematics, vol 12, 2nd edn. World Scientific, River Edge
5. D'yachkov AG, Rykov VV (1982) Bounds on the length of disjunctive codes. Problemy Peredachi Informatsii 18(3):7–13
6. Guruswami V (2004) List decoding of error-correcting codes (Winning thesis of the 2002 ACM doctoral dissertation competition). Lecture notes in computer science, vol 3282. Springer
7. Indyk P, Ngo HQ, Rudra A (2010) Efficiently decodable non-adaptive group testing. In: Proceedings of the twenty first annual ACM-SIAM symposium on discrete algorithms (SODA'2010). ACM, New York, pp 1126–1142
8. Muthukrishnan S (2005) Data streams: algorithms and applications. Found Trends Theor Comput Sci 1(2)
9. Ngo HQ, Du DZ (2000) A survey on combinatorial group testing algorithms with applications to DNA library screening. In: Discrete mathematical problems with medical applications (New Brunswick, 1999). DIMACS series in discrete mathematics and theoretical computer science, vol 55. American Mathematical Society, Providence, pp 171–182
10. Ngo HQ, Porat E, Rudra A (2011) Efficiently decodable error-correcting list disjunct matrices and applications – (extended abstract). In: ICALP (1), pp 557–568
11. Porat E, Rothschild A (2011) Explicit nonadaptive combinatorial group testing schemes. IEEE Trans Inf Theory 57(12):7982–7989

Efficient Dominating and Edge Dominating Sets for Graphs and Hypergraphs

Andreas Brandstädt[1,2] and Ragnar Nevries[1]
[1]Computer Science Department, University of Rostock, Rostock, Germany
[2]Department of Informatics, University of Rostock, Rostock, Germany

Keywords

Efficient domination; Efficient edge domination; Exact cover

Years and Authors of Summarized Original Work

2010; Brandstädt, Hundt, Nevries
2011; Brandstädt, Mosca
2012; Brandstädt, Leitert, Rautenbach
2013; Brandstädt, Milanič, Nevries
2014; Brandstädt, Giakoumakis
2014; Nevries

Problem Definition

For a hypergraph $H = (V, \mathcal{E})$, a subset of edges $\mathcal{E}' \subseteq \mathcal{E}$ is an *exact cover* of H, if every vertex of V is contained in exactly one hyperedge of $\mathcal{E}'$, that is, for all $e, f \in \mathcal{E}'$ with $e \neq f$, $e \cap f = \emptyset$ and $\bigcup \mathcal{E}' = V$. The Exact Cover (XC) problem asks for the existence of an exact cover in a given hypergraph H. Exact Cover is in Karp's famous list of 21 NP-complete problems; it is NP-complete even for 3-element hyperedges (problem X3C [SP2] in [14]).

Let G be a finite simple undirected graph with vertex set V and edge set E. A vertex *dominates* itself and all its neighbors, i.e., every vertex $v \in V$ dominates its closed neighborhood $N[v] = \{u \mid u = v$ or $uv \in E\}$. A vertex subset D of G is an *efficient dominating* (*e.d.*) set, if, for every vertex $v \in V$, there is exactly one $d \in D$ dominating v [1, 2]. An edge subset M of G is an *efficient edge dominating* (*e.e.d.*) set, if it is an efficient dominating set in the line graph $L(G)$ of G [15]. Efficient dominating sets are sometimes also called *independent perfect dominating sets*, and efficient edge dominating sets are also known as *dominating induced matchings*.

The Efficient Domination (ED) problem for a graph G asks for the existence of an e.d. set in G. The Efficient Edge Domination (EED) problem asks for the existence of an e.d. set in the line graph $L(G)$.

For a graph G, let $\mathcal{N}(G)$ denote its closed neighborhood hypergraph, that is, for every vertex $v \in V$, the closed neighborhood $N[v]$ is a hyperedge in $\mathcal{N}(G)$; note that this is a multiset since distinct vertices may have the same closed neighborhood. For a graph G, the *square* G^2 has the same vertex set as G and two vertices, x and y, are adjacent in G^2, if and only if their distance in G is at most 2. Note that G^2 is isomorphic to $L(\mathcal{N}(G))$.

By definition, the ED problem on a graph G is the same as the Exact Cover problem on its closed neighborhood hypergraph $\mathcal{N}(G)$, and the EED problem is the same as the Exact Cover problem on $L(\mathcal{N}(G))$.

Key Results

ED and EED are NP-complete; their complexity on special graph classes was studied in various papers – see, e.g., [2, 3, 12, 16–18, 20, 22, 24, 25] for ED and [5, 7, 11, 15, 19, 21] for EED. In particular, ED remains NP-complete for chordal graphs as well as for (very restricted) bipartite graphs such as chordal bipartite graphs, and EED is NP-complete for bipartite graphs but solvable in linear time for chordal graphs.

ED for Graphs

A key tool in [8] is a reduction of ED for G to the maximum-weight independent set problem for G^2, which is based on the following observation:

For a hypergraph $H = (V, \mathcal{E})$ and $e \in \mathcal{E}$, let $\omega(e) := |e|$ be an edge weight function. For the line graph $L(H)$, let $\alpha_\omega(L(H))$ denote the maximum weight of an independent vertex set in $L(H)$. The weight of any independent vertex set in $L(H)$ is at most $|V|$, and H has an exact cover, if and only if $\alpha_\omega(L(H)) = |V|$. Using the fact that G^2 is isomorphic to $L(\mathcal{N}(G))$ and ED on G corresponds to Exact Cover on $\mathcal{N}(G)$, this means that ED on G can be reduced to the maximum weight of an independent vertex set in G^2, similarly for EED. This unified approach helps to answer some open questions on ED and EED for graph classes; one example is ED for strongly chordal graphs: Since for a dually chordal graph G, its square G^2 is chordal, ED is solvable in polynomial time for dually chordal graphs and thus for strongly chordal graphs [8] (recall that ED is NP-complete for chordal graphs). Similar properties of powers lead to polynomial time for ED on AT-free graphs using known results [8]. For P_5-free graphs having an e.d., G^2 is P_4-free [9].

ED is NP-complete for planar bipartite graphs of maximum degree 3 [9]. In [23], this is sharpened by adding a girth condition: ED is NP-complete for planar bipartite graphs of maximum degree 3 and girth at least g, for every fixed g.

From the known results, it follows that ED is NP-complete for F-free graphs whenever F contains a cycle or a claw. Thus, F can be assumed to be cycle- and claw-free (see, e.g., [9]); such graphs F are called *linear forests*. For $(P_3 + P_3)$-free graphs and thus for P_7-free graphs, ED is NP-complete. ED is robustly solvable in time $O(nm)$ for P_5-free graphs and for $(P_4 + P_2)$-free graphs [9, 23]. For every fixed $k \geq 1$, ED is solvable in polynomial time for $(P_5 + kP_2)$-free graphs [4]. For P_6-free graphs, the complexity of ED is an open problem, and correspondingly for $(P_6 + kP_2)$-free graphs; these are the only open cases for F-free graphs.

EED for Graphs

The fact that graphs having an e.e.d. are K_4-free leads to a simple linear time algorithm for EED on chordal graphs. More generally, EED is solvable in polynomial time for hole-free graphs and thus for weakly chordal graphs and for chordal bipartite graphs [7]. This also follows from the fact that, for a weakly chordal graph G, $L(G)^2$ is weakly chordal [10] and from the reduction of EED for G to the maximum-weight independent set problem for $L(G)^2$. In [23], this is improved to a robust $O(nm)$ time algorithm for EED on hole-free graphs. In [8], we show that EED is solvable in linear time for dually chordal graphs.

One of the open problems for EED was its complexity on P_k-free graphs. In [5], we show that EED is solvable in linear time for P_7-free graphs. The complexity of EED remains open for P_k-free graphs, $k \geq 8$. In [11], EED is solved in polynomial time on claw-free graphs. EED is NP-complete for planar bipartite graphs of maximum degree 3 [7]. In [23], it is shown that EED is NP-complete for planar bipartite graphs of maximum degree 3 and girth at least g, for every fixed g.

XC, ED, and EED for Hypergraphs

The notion of α-*acyclicity* [13] is one of the most important and most frequently studied hypergraph notions. Among the many equivalent conditions describing α-acyclic hypergraphs, we take the following: For a hypergraph $H = (V, \mathcal{E})$, a tree T with node set $\mathcal{E}$ and edge set E_T is a *join tree* of H, if, for all vertices $v \in V$, the set of hyperedges $\mathcal{E}_v := \{e \in \mathcal{E} \mid v \in e\}$ containing v induces a subtree of T. H is α-*acyclic*, if it has a join tree. Let $H^* := (\mathcal{E}, \{\mathcal{E}_v \mid v \in V\})$ be the *dual hypergraph* of H. The hypergraph $H = (V, \mathcal{E})$ is a *hypertree*, if there is a tree T with vertex set V such that, for all $e \in \mathcal{E}$, $T[e]$ is connected. Obviously, H is α-acyclic, if and only if its dual H^* is a hypertree.

By a result of Duchet, Flament, and Slater (see, e.g., [6]), it is known that H is a hypertree, if and only if H has the Helly property and its line graph $L(H)$ is chordal. In its dual version, it says that H is α-acyclic, if and only if H is conformal and its 2-section graph is chordal. In [8], we show:

(i) ED and XC are NP-complete for α-acyclic hypergraphs but solvable in polynomial time for hypertrees.
(ii) EED is NP-complete for hypertrees but solvable in polynomial time for α-acyclic hypergraphs.

Recommended Reading

1. Bange DW, Barkauskas AE, Slater PJ (1988) Efficient dominating sets in graphs. In: Ringeisen RD, Roberts FS (eds) Applications of discrete mathematics. SIAM, Philadelphia, pp 189–199
2. Bange DW, Barkauskas AE, Host LH, Slater PJ (1996) Generalized domination and efficient domination in graphs. Discret Math 159:1–11
3. Biggs N (1973) Perfect codes in graphs. J Comb Theory (B) 15:289–296
4. Brandstädt A, Giakoumakis V (2014) Efficient domination for $(P_5 + kP_2)$-free graphs. Manuscript, arXiv:1407.4593v1
5. Brandstädt A, Mosca R (2011) Dominating induced matchings for P_7-free graphs in linear time. Technical report CoRR, arXiv:1106.2772v1; Extended abstract in: Asano T, Nakano S-I, Okamoto Y, Watanabe O (eds) Algorithms and computation. LNCS, vol 7074. Springer, pp 100–109
6. Brandstädt A, Le VB, Spinrad JP (1999) Graph classes: a survey. SIAM monographs on discrete mathematics and applications, vol 3. SIAM, Philadelphia
7. Brandstädt A, Hundt C, Nevries R (2010) Efficient edge domination on hole-free graphs in polynomial time. In: Lópes-Ortiz A (ed) LATIN 2010: theoretical informatics, Oaxaca. LNCS, vol 6034. Springer, pp 650–661
8. Brandstädt A, Leitert A, Rautenbach D (2012) Efficient dominating and edge dominating sets for graphs and hypergraphs. Technical report CoRR, arXiv:1207.0953v2 and in: Choa K-M, Hsu T-S, Lee D-T (eds) Algorithms and computation. LNCS, vol 7676. Springer, pp 267–277
9. Brandstädt A, Milanič M, Nevries R (2013) New polynomial cases of the weighted efficient domination problem. Technical report CoRR, arXiv:1304.6255 and in: Chatterjee K, Sgall J (eds) Mathematical foundations of computer science 2013. LNCS, vol 8087. Springer, pp 195–206

10. Cameron K, Sritharan R, Tang Y (2003) Finding a maximum induced matching in weakly chordal graphs. Discret Math 266:133–142
11. Cardoso DM, Korpelainen N, Lozin VV (2011) On the complexity of the dominating induced matching problem in hereditary classes of graphs. Discret Appl Math 159:521–531
12. Chang GJ, Pandu Rangan C, Coorg SR (1995) Weighted independent perfect domination on co-comparability graphs. Discret Appl Math 63:215–222
13. Fagin R (1983) Degrees of acyclicity for hypergraphs and relational database schemes. J ACM 30:514–550
14. Garey MR, Johnson DS (1979) Computers and intractability – a guide to the theory of NP-completeness. Freeman, San Francisco
15. Grinstead DL, Slater PL, Sherwani NA, Holmes ND (1993) Efficient edge domination problems in graphs. Inf Process Lett 48:221–228
16. Kratochvíl J (1991) Perfect codes in general graphs, Rozpravy Československé Akad. Věd Řada Mat. Přírod Vd' 7. Akademia, Praha
17. Liang YD, Lu CL, Tang CY (1997) Efficient domination on permutation graphs and trapezoid graphs. In: Jiang T, Lee DT (eds) Computing and combinatorics, Shanghai. LNCS, vol 1276. Springer, pp 232–241
18. Lin Y-L (1998) Fast algorithms for independent domination and efficient domination in trapezoid graphs. In: Chwa K-Y, Ibarra OH (eds) Algorithms and computation, Taejon. LNCS, vol 1533. Springer, pp 267–275
19. Lu CL, Tang CY (1998) Solving the weighted efficient edge domination problem on bipartite permutation graphs. Discret Appl Math 87:203–211
20. Lu CL, Tang CY (2002) Weighted efficient domination problem on some perfect graphs. Discret Appl Math 117:163–182
21. Lu CL, Ko M-T, Tang CY (2002) Perfect edge domination and efficient edge domination in graphs. Discret Appl Math 119:227–250
22. Milanič M (2011) A hereditary view on efficient domination. Extended abstract In: Adacher L, Flamini M, Leo G, Nicosia G, Pacifici A, Piccialli V (eds) Proceedings of the 10th cologne-twente workshop on graphs and combinatorial optimization, Frascati, pp 203–206. Full version: Hereditary efficiently dominatable graphs. J Graph Theory 73:400–424
23. Nevries R (2014) Efficient domination and polarity. Ph.D. thesis, University of Rostock, Germany
24. Yen C-C (1992) Algorithmic aspects of perfect domination. Ph.D. thesis, National Tsing Hua University, Taiwan
25. Yen C-C, Lee RCT (1996) The weighted perfect domination problem and its variants. Discret Appl Math 66:147–160

Efficient Methods for Multiple Sequence Alignment with Guaranteed Error Bounds

S.M. Yiu and Francis Y.L. Chin
Department of Computer Science, University of Hong Kong, Hong Kong, China

Keywords

Multiple global alignment; Multiple string alignment

Years and Authors of Summarized Original Work

1993; Gusfield

Problem Definition

Multiple sequence alignment is an important problem in computational biology. Applications include finding highly conserved subregions in a given set of biological sequences and inferring the evolutionary history of a set of taxa from their associated biological sequences (e.g., see [9]). There are a number of measures proposed for evaluating the goodness of a multiple alignment, but prior to this work, no efficient methods are known for computing the optimal alignment for any of these measures. The work of Gusfield [7] gives two computationally efficient multiple alignment approximation algorithms for two of the measures with approximation ratio of less than 2. For one of the measures, they also derived a randomized algorithm, which is much faster and with high probability and reports a multiple alignment with small error bounds. To the best knowledge of the entry authors, this work is the first to provide approximation algorithms (with guarantee error bounds) for this problem.

Notations and Definitions

Let X and Y be two strings of alphabet Σ. The pairwise alignment of X and Y maps X and Y into strings X' and Y' that may contain spaces, denoted by '_', where (1) $|X'| = |Y'| = \ell$ and (2) removing spaces from X' and Y' returns X and Y, respectively. The score of the alignment is defined as $d(X', Y') = \sum_{i=1}^{\ell} s(X'(i), Y'(i))$ where $X'(i)$ (and $Y'(i)$) denotes the ith character in X' (and Y') and $s(a, b)$ with $a, b \in \Sigma \cup$ '_' is the distance-based scoring scheme that satisfies the following assumptions:

1. $s('\_', '\_') = 0$;
2. Triangular inequality: for any three characters, $x, y, z, s(x, z) \le s(x, y) + s(y, z))$.

Let $\chi = X_1, X_2, \ldots, X_k$ be a set of $k > 2$ strings of alphabet Σ. A multiple alignment A of these k strings maps $X_1, X_2, \ldots, X_k$ to $X'_1, X'_2, \ldots, X'_k$ that may contain spaces such that (1) $|X'_1| = |X'_2| = \cdots = |X'_k| = \ell$ and (2) removing spaces from X'_i returns X_i for all $1 \le i \le k$. The multiple alignment A can be represented as a $k \times \ell$ matrix.

The Sum of Pairs (SP) Measure

The score of a multiple alignment A, denoted by SP(A), is defined as the sum of the scores of pairwise alignments induced by A, that is, $\sum_{i<j} d(X'_i, X'_j) = \sum_{i<j} \sum_{p=1}^{\ell} s(X'_i[p], X'_j[p])$ where $1 \le i < j \le k$.

Problem 1 (Multiple Sequence Alignment with Minimum SP Score)

INPUT: A set of k strings, a scoring scheme s.
OUTPUT: A multiple alignment A of these k strings with minimum SP(A).

The Tree Alignment (TA) Measure

In this measure, the multiple alignment is derived from an evolutionary tree. For a given set χ of k strings, let $\chi' \supseteq \chi$. An evolutionary tree T'_χ for χ is a tree with at least k nodes, where there is a one-to-one correspondence between the nodes and the strings in χ'. Let $X'_u \in \chi'$ be the string for node u. The score of T'_χ, denoted by TA (T'_χ), is defined as $\sum_{e=(u,v)} D(X'_u, X'_v)$ where e is an edge in T'_χ and $D(X'_u, X'_v)$ denotes the score of the optimal pairwise alignment for X'_u and X'_v. Analogously, the multiple alignment of χ under the TA measure can also be represented by a $|\chi'| \times \ell$ matrix, where $|\chi'| \ge k$, with a score defined as $\sum_{e=(u,v)} d(X'_u, X'_v)$ (e is an edge in T'_χ), similar to the multiple alignment under the SP measure in which the score is the summation of the alignment scores of all pairs of strings. Under the TA measure, since it is always possible to construct the $|\chi'| \times \ell$ matrix such that $d(X'_u, X'_v) = D(X'_u, X'_v)$ for all $e = (u, v)$ in T'_χ and we are usually interested in finding the multiple alignment with the minimum TA value, so $D(X'_u, X'_v)$ is used instead of $d(X'_u, X'_v)$ in the definition of TA (T'_χ).

Problem 2 (Multiple Sequence Alignment with Minimum TA Score)

INPUT: A set of k strings, a scoring scheme s.
OUTPUT: An evolutionary tree T for these k strings with minimum TA(T).

Key Results

Theorem 1 *Let A^* be the optimal multiple alignment of the given k strings with minimum SP score. They provide an approximation algorithm (the center star method) that gives a multiple alignment A such that* $\frac{SP(A)}{SP(A^*)} \le \frac{2(k-1)}{k} = 2 - \frac{2}{k}$.

The center star method is to derive a multiple alignment which is consistent with the optimal pairwise alignments of a center string with all the other strings. The bound is derived based on the triangular inequality of the score function. The time complexity of this method is $O(k^2\ell^2)$, where ℓ^2 is the time to solve the pairwise alignment by dynamic programming and k^2 is needed to find the center string, X_c, which gives the minimum value of $\sum_{i \ne c} D(X_c, X_i)$.

Theorem 2 *Let A^* be the optimal multiple alignment of the given k strings with minimum SP score. They provide a randomized algorithm that gives a multiple alignment A such that $\frac{SP(A)}{SP(A^*)} \leq 2 + \frac{1}{r-1}$ with probability at least $1 - (\frac{r-1}{r})^p$ for any $r > 1$ and $p \geq 1$. Instead of computing $\binom{k}{2}$ optimal pairwise alignments to find the best center string, the randomized algorithm only considers p randomly selected strings to be candidates for the best center string; thus, this method needs to x compute only $(k-1)p$ optimal pairwise alignments in $O(kp\ell^2)$ time where $1 \leq p \leq k$.*

Theorem 3 *Let T^* be the optimal evolutionary tree of the given k strings with minimum TA score. They provide an approximation algorithm that gives an evolutionary tree T such that $\frac{TA(T)}{TA(T^*)} \leq \frac{2(k-1)}{k} = 2 - \frac{2}{k}$.*

In the algorithm, they first compute all the $\binom{k}{2}$ optimal pairwise alignments to construct a graph with every node representing a distinct string X_i and the weight of each edge (X_i, X_j) as $D(X_i X_j)$. This step determines the overall time complexity $O(k^2\ell^2)$. Then, they find a minimum spanning tree from the graph. The multiple alignment has to be consistent with the optimal pairwise alignments represented by the edges of this minimum spanning tree.

Applications

Multiple sequence alignment is a fundamental problem in computational biology. In particular, multiple sequence alignment is useful in identifying those common structures, which may only be weakly reflected in the sequence and not easily revealed by pairwise alignment. These common structures may carry important information for their evolutionary history, critical conserved motifs, and common 3D molecular structure, as well as biological functions.

More recently, multiple sequence alignment is also used in revealing noncoding RNAs (ncRNAs) [2]. In this type of multiple alignment, we are not only align the underlying sequences but also the secondary structures of the RNAs. Researchers believe that ncRNAs that belong to the same family should have common components giving a similar secondary structure. The multiple alignment can help to locate and identify these common components.

Open Problems

A number of open problems related to the work of Gusfield remain open. For the SP measure, the center star method can be extended to the q-star method ($q > 2$) with approximation ratio of $2 - q/k$ [1, 10], sect. 7.5 of [11]). Whether there exists an approximation algorithm with better approximation ratio or with better time complexity is still unknown. For the TA measure, to be the best knowledge of the entry authors, the approximation ratio in Theorem 3 is currently the best result.

Another interesting direction related to this problem is the constrained multiple sequence alignment problem [12] which requires the multiple alignment to contain certain aligned characters with respect to a given constrained sequence. The best known result [6] is an approximation algorithm (also follows the idea of center star method) which gives an alignment with approximation ratio of $2 - 2/k$ for k strings.

For the complexity of the problem, Wang and Jiang [13] were the first to prove the NP-hardness of the problem with SP score under a *nonmetric* distance measure over a 4-symbol alphabet. More recently, in [5], the multiple alignment problem with SP score, star alignment, and TA score have been proved to be NP-hard for all binary or larger alphabets under *any metric*. Developing efficient approximation algorithms with good bounds for any of these measures is desirable.

Experimental Results

Two experiments have been reported in the paper showing that the worst-case error bounds in

Theorems 1 and 2 (for the SP measure) are pessimistic compared to the typical situation arising in practice.

The scoring scheme used in the experiments is $s(a,b) = 0$ if $a = b$; $s(a,b) = 1$ it either a or b is a space; otherwise $s(a,b) = 2$. Since computing the optimal multiple alignment with minimum SP score has been shown to be NP-hard, they evaluate the performance of their algorithms using the lower bound of $\sum_{i<j} D(X_i, X_j)$ (recall that $D(X_i, X_j)$ is the score of the optimal pairwise alignment of X_i and X_j). They have aligned 19 similar amino acid sequences with average length of 60 of homeoboxs from different species. The ratio of the scores of reported alignment by the center star method to the lower bound is only 1.018 which is far from the worst-case error bound given in Theorem 1. They also aligned 10 not-so-similar sequences near the homeoboxes, and the ratio of the reported alignment to the lower bound is 1.162. Results also show that the alignment obtained by the randomized algorithm is usually not far away from the lower bound.

Data Sets

The exact sequences used in the experiments are not provided.

Cross-References

▶ Statistical Multiple Alignment

Recommended Reading

1. Bafna V, Lawler EL, Pevzner PA (1997) Approximation algorithms for multiple sequence alignment. Theor Comput Sci 182:233–244
2. Dalli D, Wilm A, Mainz I, Stegar G (2006) STRAL: progressive alignment of non-coding RNA using base pairing probability vectors in quadratic time. Bioinformatics 22(13):1593–1599
3. Do C, Brudno M, Batzoglou S (2004) ProbCons: probabilistic consistency-based multiple alignment of amino acid sequences. In: Proceedings of the thirteenth national conference on artificial intelligence, pp 703–708, San Jose. AAAI Press
4. Edgar R (2004) MUSCLE: multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Res 32:1792–1797
5. Elias I (2003) Setting the intractability of multiple alignment. In: Proceedings of the 14th annual international symposium on algorithms and computation (ISAAC 2003), Kyoto, pp 352–363
6. Francis YL, Chin NLH, Lam TW, Prudence WHW (2005) Efficient constrained multiple sequence alignment with performance guarantee. J Bioinform Comput Biol 3(1):1–18
7. Gusfield D (1993) Efficient methods for multiple sequence alignment with guaranteed error bounds. Bull Math Biol 55(1):141–154
8. Notredame C, Higgins D, Heringa J (2000) T-coffee: a novel method for multiple sequence alignments. J Mol Biol 302:205–217
9. Pevsner J (2003) Bioinformatics and functional genomics. Wiley, New York
10. Pevzner PA (1992) Multiple alignment, communication cost, and graph matching. SIAM J Appl Math 52:1763–1779
11. Pevzner PA (2000) Computational molecular biology: an algorithmic approach. MIT, Cambridge
12. Tang CY, Lu CL, Chang MDT, Tsai YT, Sun YJ, Chao KM, Chang JM, Chiou YH, Wu CM, Chang HT, Chou WI (2002) Constrained multiple sequence alignment tool development and its application to RNase family alignment. In: Proceedings of the first IEEE computer society bioinformatics conference (CSB 2002), Stanford, pp 127–137
13. Wang L, Jiang T (1994) On the complexity of multiple sequence alignment. J Comput Biol 1:337–48
14. Ye Y, Cheung D, Wang Y, Yiu SM, Qing Z, Lam TW, Ting HF GLProbs: aligning multiple sequences adaptively. IEEE/ACM Trans Comput Biol Bioinformatics. doi:http://doi.ieeecomputersociety.org/10.1109/TCBB.2014.2316820

Efficient Polynomial Time Approximation Scheme for Scheduling Jobs on Uniform Processors

Klaus Jansen
Department of Computer Science, University of Kiel, Kiel, Germany

Keywords

Approximation schemes; Integer linear programming; Scheduling; Sensitivity analysis

Years and Authors of Summarized Original Work

2009; Jansen
2011; Jansen, Robenek
2014; Chen, Jansen, Zhang

Problem Definition

We consider the following fundamental problem in scheduling theory. Suppose that there is a set $\mathcal{J}$ of n independent jobs J_j with processing time p_j and a set $\mathcal{P}$ of m nonidentical processors P_i that run at different speeds s_i. If job J_j is executed on processor P_i, then processor P_i needs p_j/s_i time units to complete the job. The goal is to find an assignment $a : \mathcal{J} \rightarrow \mathcal{P}$ for the jobs to the processors that minimizes the total length of the schedule $\max_{i=1,\ldots,m} \sum_{J_j : a(J_j)=P_i} p_j/s_i$. This is the minimum time needed to complete all jobs on the processors. The problem is denoted $Q||C_{\max}$ and it is also called the minimum makespan problem on uniform parallel processors. By simplicity we may assume that the number m of processors is bounded by the number of jobs; otherwise select only the fastest n machines in $O(m)$ time.

Key Results

The scheduling problem on uniform and also identical processors is NP-hard [7] and the existence of a polynomial time algorithm for it would imply $P = NP$. Hochbaum and Shmoys [9, 10] presented a family of polynomial time approximation algorithms $\{A_\epsilon | \epsilon > 0\}$ for both scheduling problems, where each algorithm A_ϵ generates a schedule of length $(1 + \epsilon)\,OPT(I)$ for each instance I and has running time polynomial in the input size $|I|$. Such a family of algorithms is called a polynomial time approximation scheme (PTAS). It is allowed that the running time of each algorithm A_ϵ is exponential in $1/\epsilon$. The running time of the PTAS for uniform processors by Hochbaum and Shmoys [10] is $(n/\epsilon)^{O(1/\epsilon^2)}$.

Two restricted classes of approximation schemes were defined to classify different faster approximation scheme. An efficient polynomial time approximation scheme (EPTAS) is a PTAS with running time $f(1/\epsilon)\,poly(|I|)$ for some function f, while a fully polynomial time approximation scheme (FPTAS) runs in time $poly(1/\epsilon, |I|)$; polynomial in $1/\epsilon$ and the size $|I|$ of the instance. Since the scheduling problem on identical and also uniform processors is NP-hard in the strong sense (it contains bin packing as special case), we cannot hope for an FPTAS. For identical processors, Hochbaum and Shmoys (see [8]) and Alon et al. [1] gave an EPTAS with running time $f(1/\epsilon) + O(n)$, where f is doubly exponential in $1/\epsilon$.

Known Techniques

Hochbaum and Shmoys [9] introduced the dual approximation approach for identical and uniform processors and used the relationship between these scheduling problems and the bin packing problem. This relationship between scheduling on identical processors and bin packing problem had been exploited already by Coffman et al. [3]. Using the dual approximation approach, Hochbaum and Shmoys [9] proposed a PTAS for scheduling on identical processors with running time $(n/\epsilon)^{O(1/\epsilon^2)}$.

The main idea in the approach is to guess the length of the schedule by using binary search and to consider the corresponding bin packing instance with scaled identical bin size equal to 1. Then they distinguish between large items with size $> \epsilon$ and small items with size $\leq \epsilon$. For the large items they use a dynamic programming approach to calculate the minimum number of bins needed to pack them all. Afterward, they pack the remaining small items in a greedy way in enlarged bins of size $1 + \epsilon$ (i.e., they pack into any bin that currently contains items of total size at most 1; and if no such bin exists, then they open a new bin).

Furthermore, Hochbaum and Shmoys (see [8]) and Alon et al. [1] achieved an improvement to linear time by using an integer linear program for the cutting stock formulation of bin packing for the large items and a result on integer linear programming with a fixed number of variables by Lenstra [15]. This gives an EPTAS for identical processors with running time $f(1/\epsilon) + O(n)$ where f is doubly exponential in $1/\epsilon$.

For uniform processors, the decision problem for the scheduling problem with makespan at most T can be viewed as a bin packing problem with different bin sizes. Using an ϵ-relaxed version of this bin packing problem, Hochbaum and Shmoys [10] were also able to obtain a PTAS for scheduling on uniform processors with running time $(n/\epsilon)^{O(1/\epsilon^2)}$. The main underlying idea in their algorithm is a clever rounding technique and a nontrivial dynamic programming approach over the different bins ordered by their sizes.

New Results

Recently, Jansen [11] proposed an EPTAS for scheduling jobs on uniform machines:

Theorem 1 ([11]) *There is an EPTAS (a family of algorithms $\{A_\epsilon | \epsilon > 0\}$) which, given an instance I of $Q||C_{\max}$ with n jobs and m processors with different speeds and a positive number $\epsilon > 0$, produces a schedule for the jobs of length $A_\epsilon(I) \leq (1+\epsilon)OPT(I)$. The running time of A_ϵ is*

$$2^{O(1/\epsilon^2 \log^3(1/\epsilon))} + poly(n).$$

Interestingly, the running time of the EPTAS is only single exponential in $1/\epsilon$.

Integer Linear Programming and Grouping Techniques

The new algorithm uses the dual approximation method by Hochbaum and Shmoys [10] to transform the scheduling problem into a bin packing problem with different bin sizes. Next, the input is structured by rounding bin sizes and processing times to values of the form $(1+\delta)^i$ and $\delta(1+\delta)^i$ with $i \in \mathbb{Z}$ where δ depends on ϵ. After sorting the bins according to their sizes, $c_1 \geq \ldots \geq c_m$, three groups of bins are built: $\mathcal{B}_1$ with the largest K bins (where K is constant). Let G be the smallest index such that capacity $c_{K+G+1} \leq \gamma c_K$ where $\gamma < 1$ depends on ϵ; such an index G exists for $c_m \leq \gamma c_K$. In this case $\mathcal{B}_2$ is the set of the next G largest bins where the maximum size $c_{\max}(\mathcal{B}_2) = c_{K+1}$ divided by the minimum size $c_{\min}(\mathcal{B}_2) = c_{K+G}$ is bounded by a constant $1/\gamma$ and $\mathcal{B}_3$ is the set with the remaining smaller bins of size smaller than γc_K. This generates a gap of constant size between the capacities of bins in $\mathcal{B}_1$ and $\mathcal{B}_3$. If the rate c_m/c_K, where c_m is the smallest bin size, is larger the constant γ, then a simpler instance is obtained with only two groups $\mathcal{B}_1$ and $\mathcal{B}_2$ of bins.

For $\mathcal{B}_1$ all packings for the very large items are computed (those which only fit there). If there is a feasible packing, then a mixed integer linear program (MILP) or an integer linear program (ILP) in the simpler case is used to place the other items into the bins. The placement of the large items into the second group $\mathcal{B}_2$ is done via integral configuration variables; similar to the ILP formulation for bin packing by Fernandez de la Vega and Lueker [6]. Fractional configuration variables are used for the placement of large items into $\mathcal{B}_3$. Furthermore, additional fractional variables are taken to place small items into $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$. The MILP has only a constant number of integral variables and, therefore, can be solved via the algorithm by Lenstra or Kannan [14, 15].

In order to avoid that the running time is doubly exponential in $1/\epsilon$, a recent result by Eisenbrand and Shmonin [5] about integer cones is used. To apply their result a system of equalities for the integral configuration variables is considered and the corresponding coefficients are rounded. Then each feasible solution of the modified MILP contains at most $O(1/\delta \log^2(1/\delta))$ integral variables with values larger than zero. By choosing the strictly positive integral variables in the MILP, the number of integral configuration variables is reduced from $2^{O(1/\delta \log(1/\delta))}$ to $O(1/\delta \log^2(1/\delta))$. The number of choices is bounded by $2^{O(1/\delta^2 \log^3(1/\delta))}$.

Afterward, the fractional variables in the MILP solution are rounded to integral values using ideas from scheduling job shops [13] and scheduling on unrelated machines [16]. The effect of the rounding is that most of the items can be placed directly into the bins. Only a few of them cannot be placed this way, and here is where the K largest bins and the gap between $\mathcal{B}_1$ and $\mathcal{B}_3$ come into play. It can be proved that these items can be moved to the K largest bins by increasing their sizes only slightly.

Algorithm Avoiding the MILP

Recently an EPTAS for scheduling on uniform machines is presented by Jansen and Robenek [12] that avoids the use of an MILP or ILP solver. In the new approach instead of solving (M)ILPs, an LP-relaxation and structural information about the "closest" ILP solution is used.

In the following the main techniques are described for identical processors. For a given LP-solution x, the distance to the closest ILP solution y in the infinity norm is studied, i.e., $\|x - y\|_\infty$. For the constraint matrix A_δ of the considered LP, this distance is defined by

$$\max\text{-gap}(A_\delta) := \max\{\min\{\|y^\star - x^\star\|_\infty : y^\star \text{ solution of ILP}\} : x^\star \text{solution of LP}\}.$$

Let $C(A_\delta)$ denote an upper bound for max-gap (A_δ). The running time of the algorithm is $2^{O(1/\epsilon \log(1/\epsilon)\log(C(A_\delta)))} + poly(n)$. The algorithm for uniform processors is more complex, but we obtain a similar running time $2^{O(1/\epsilon \log(1/\epsilon)\log(C(\tilde{A}_\delta)))} + poly(n)$, where the constraint matrix $\tilde{A}_\delta$ is slightly different. For the details we refer to [12].

It can be proved using a result by Cook et al. [4] that $C(A_\delta), C(\tilde{A}_\delta) \leq 2^{O(1/\epsilon \log^2(1/\epsilon))}$. Consequently, the algorithm has a running time at most $2^{O(1/\epsilon^2 \log^3(1/\epsilon))} + poly(n)$, the same as in [11]. But, to our best knowledge, no instance is known to take on the value $2^{O(1/\epsilon \log^2(1/\epsilon))}$ for max-gap(A_δ). We conjecture $C(A_\delta) \leq poly(1/\epsilon)$. If that holds, the running time of the algorithm would be $2^{O(1/\epsilon \log^2(1/\epsilon))} + poly(n)$ and thus improve the result in [11].

Lower Bounds

Recently, Chen, Jansen, and Zhang [2] proved the following lower bound on the running time: For scheduling on an arbitrary number of identical machines, denoted by $P||C_{\max}$, a polynomial time approximation scheme (PTAS) of running time $2^{O((1/\epsilon)^{1-\delta})} * poly(n)$ for any $\delta > 0$ would imply that the exponential time hypothesis (ETH) for 3-SAT fails.

Open Problems

The main open question is whether there is an EPTAS for scheduling jobs on identical and uniform machines with a running time $2^{O(1/\epsilon \log^c(1/\epsilon))} * poly(n)$.

Experimental Results

None is reported.

Recommended Reading

1. Alon N, Azar Y, Woeginger GJ, Yadid T (1998) Approximation schemes for scheduling on parallel machines. J Sched 1:55–66
2. Chen L, Jansen K, Zhang G (2014) On the optimality of approximation schemes for the classical scheduling problem. In: Symposium on discrete algorithms (SODA 2014), Portland
3. Coffman EG, Garey MR, Johnson DS (1978) An application of bin packing to multiprocessor scheduling. SIAM J Comput 7:1–17
4. Cook W, Gerards AMH, Schrijver A, Tardos E (1986) Sensitivity theorems in integer linear programming. Math Program 34:251–264
5. Eisenbrand F, Shmonin G (2006) Caratheodory bounds for integer cones. Oper Res Lett 34:564–568
6. Fernandez de la Vega W, Lueker GS (1981) Bin packing can be solved within $1 + \epsilon$ in linear time. Combinatorica 1:349–355
7. Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. W.H. Freeman, San Francisco
8. Hochbaum DS (1997) Various notions of approximations: good, better, best, and more. In: Hochbaum DS (ed) Approximation algorithms for NP-hard problems, chap 9. Prentice Hall, Boston, pp 346–398
9. Hochbaum DS, Shmoys DB (1987) Using dual approximation algorithms for scheduling problems: practical and theoretical results. J ACM 34:144–162
10. Hochbaum DS, Shmoys DB (1988) A polynomial approximation scheme for scheduling on uniform processors: using the dual approximation approach. SIAM J Comput 17:539–551
11. Jansen K (2010) An EPTAS for scheduling jobs on uniform processors: using an MILP relaxation with a constant number of ntegral vasriables. SIAM J Discret Math 24(2):457–485
12. Jansen K, Robenek C (2011) Scheduling jobs on identical and uniform processors revisited. In: Workshop on approximation and online algorithms, WAOA 2011. LNCS, vol 7164, pp 109–122 and Technical report, University of Kiel, Saarbrücken, TR-1109

13. Jansen K, Solis-Oba R, Sviridenko M (2003) Makespan minimization in job shops: a linear time approximation acheme. SIAM J Discret Math 16:288–300
14. Kannan R (1987) Minkowski's convex body theorem and integer programming. Math Oper Res 12:415–440
15. Lenstra HW (1983) Integer programming with a fixed number of variables. Math Oper Res 8:538–548
16. Lenstra JK, Shmoys DB, Tardos E (1990) Approximation algorithms for scheduling unrelated parallel machines. Math Program 24:259–272

Engineering Algorithms for Computational Biology

David A. Bader
College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

Keywords

High-performance computational biology

Years and Authors of Summarized Original Work

2002; Bader, Moret, Warnow

Problem Definition

In the 50 years since the discovery of the structure of DNA, and with new techniques for sequencing the entire genome of organisms, biology is rapidly moving towards a data-intensive, computational science. Many of the newly faced challenges require high-performance computing, either due to the massive-parallelism required by the problem, or the difficult optimization problems that are often combinatoric and NP-hard. Unlike the traditional uses of supercomputers for regular, numerical computing, many problems in biology are irregular in structure, significantly more challenging to parallelize, and integer-based using abstract data structures.

Biologists are in search of biomolecular sequence data, for its comparison with other genomes, and because its structure determines function and leads to the understanding of biochemical pathways, disease prevention and cure, and the mechanisms of life itself. Computational biology has been aided by recent advances in both technology and algorithms; for instance, the ability to sequence short contiguous strings of DNA and from these reconstruct the whole genome and the proliferation of high-speed microarray, gene, and protein chips for the study of gene expression and function determination. These high-throughput techniques have led to an exponential growth of available genomic data.

Algorithms for solving problems from computational biology often require parallel processing techniques due to the data- and compute-intensive nature of the computations. Many problems use polynomial time algorithms (e.g., all-to-all comparisons) but have long running times due to the large number of items in the input; for example, the assembly of an entire genome or the all-to-all comparison of gene sequence data. Other problems are compute-intensive due to their inherent algorithmic complexity, such as protein folding and reconstructing evolutionary histories from molecular data, that are known to be NP-hard (or harder) and often require approximations that are also complex.

Key Results

None

Applications

Phylogeny Reconstruction

A phylogeny is a representation of the evolutionary history of a collection of organisms or genes (known as taxa). The basic assumption of process necessary to phylogenetic reconstruction is repeated divergence within species or genes. A phylogenetic reconstruction is usually depicted as a tree, in which modern taxa are depicted at the leaves and ancestral taxa occupy internal nodes, with the edges of the tree denoting evolutionary relationships among the taxa. Reconstructing phylogenies is a major component of modern research programs in biology and medicine (as well as linguistics). Naturally, scientists are interested in phylogenies for the sake of knowledge, but such analyses also have many uses in applied research and in the commercial arena. Existing phylogenetic reconstruction techniques suffer from serious problems of running time (or, when fast, of accuracy). The problem is particularly serious for large data sets: even though data sets comprised of sequence from a single gene continue to pose challenges (e.g., some analyses are still running after 2 years of computation on medium-sized clusters), using whole-genome data (such as gene content and gene order) gives rise to even more formidable computational problems, particularly in data sets with large numbers of genes and highly-rearranged genomes.

To date, almost every model of speciation and genomic evolution used in phylogenetic reconstruction has given rise to NP-hard optimization problems. Three major classes of methods are in common use. Heuristics (a natural consequence of the NP-hardness of the problems) run quickly, but may offer no quality guarantees and may not even have a well-defined optimization criterion, such as the popular *neighbor-joining* heuristic [9]. Optimization based on the criterion of *maximum parsimony* (MP) [4] seeks the phylogeny with the least total amount of change needed to explain modern data. Finally, optimization based on the criterion of *maximum likelihood* (ML) [5] seeks the phylogeny that is the most likely to have given rise to the modern data.

Heuristics are fast and often rival the optimization methods in terms of accuracy, at least on datasets of moderate size. Parsimony-based methods may take exponential time, but, at least for DNA and amino acid data, can often be run to completion on datasets of moderate size. Methods based on maximum likelihood are very slow (the point estimation problem alone appears intractable) and thus restricted to very small instances, and also require many more assumptions than parsimony-based methods, but appear capable of outperforming the others in terms of the quality of solutions when these assumptions are met. Both MP- and ML-based analyses are often run with various heuristics to ensure timely termination of the computation, with mostly unquantified effects on the quality of the answers returned.

Thus there is ample scope for the application of high-performance algorithm engineering in the area. As in all scientific computing areas, biologists want to study a particular dataset and are willing to spend months and even years in the process: accurate branch prediction is the main goal. However, since all exact algorithms scale exponentially (or worse, in the case of ML approaches) with the number of taxa, speed remains a crucial parameter – otherwise few datasets of more than a few dozen taxa could ever be analyzed.

Experimental Results

As an illustration, this entry briefly describes a high-performance software suite, GRAPPA (Genome Rearrangement Analysis through Parsimony and other Phylogenetic Algorithms) developed by Bader et al. *GRAPPA* extends Sankoff and Blanchette's breakpoint phylogeny algorithm [10] into the more biologically-meaningful inversion phylogeny and provides a highly-optimized code that can make use of distributed- and shared-memory parallel systems (see [1, 2, 6, 7, 8, 11] for details). In [3], Bader et al. gives the first linear-time algorithm and fast implementation for computing inversion distance between two signed permutations. *GRAPPA* was run on a 512-processor IBM Linux cluster with Myrinet and obtained a 512-fold speedup (linear speedup with respect to the number of processors): a complete breakpoint analysis (with the more demanding inversion distance used in lieu of breakpoint distance) for the 13 genomes in the Campanulaceae data set ran in less than 1.5 h in an October 2000 run, for a *million-fold* speedup over the original implementation. The latest version features significantly improved bounds and new distance correction methods and, on the same dataset, exhibits a speedup factor of *over one billion*. GRAPPA achieves this speedup through a combination of parallelism and high-performance algorithm engineering.

Although such spectacular speedups will not always be realized, many algorithmic approaches now in use in the biological, pharmaceutical, and medical communities may benefit tremendously from such an application of high-performance techniques and platforms.

This example indicates the potential of applying high-performance algorithm engineering techniques to applications in computational biology, especially in areas that involve complex optimizations: Bader's reimplementation did not require new algorithms or entirely new techniques, yet achieved gains that turned an impractical approach into a usable one.

Cross-References

- ▶ Distance-Based Phylogeny Reconstruction (Fast-Converging)
- ▶ Distance-Based Phylogeny Reconstruction: Safety and Edge Radius
- ▶ Efficient Methods for Multiple Sequence Alignment with Guaranteed Error Bounds
- ▶ High Performance Algorithm Engineering for Large-Scale Problems
- ▶ Local Alignment (with Affine Gap Weights)
- ▶ Local Alignment (with Concave Gap Weights)
- ▶ Multiplex PCR for Gap Closing (Whole-Genome Assembly)
- ▶ Peptide De Novo Sequencing with MS/MS
- ▶ Perfect Phylogeny Haplotyping
- ▶ Phylogenetic Tree Construction from a Distance Matrix
- ▶ Sorting Signed Permutations by Reversal (Reversal Distance)
- ▶ Sorting Signed Permutations by Reversal (Reversal Sequence)
- ▶ Sorting by Transpositions and Reversals (Approximate Ratio 1.5)
- ▶ Substring Parsimony

Recommended Reading

1. Bader DA, Moret BME, Warnow T, Wyman SK, Yan M (2001) High-performance algorithm engineering for gene-order phylogenies. In: DIMACS workshop on whole genome comparison. Rutgers University, Piscataway
2. Bader DA, Moret BME, Vawter L (2001) Industrial applications of high-performance computing for phylogeny reconstruction. In: Siegel HJ (ed) Proceedings of the SPIE commercial applications for high-performance computing, vol 4528. Denver, pp 159–168
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. Farris JS (1983) The logical basis of phylogenetic analysis. In: Platnick NI, Funk VA (eds) Advances in cladistics. Columbia University Press, New York, pp 1–36
5. Felsenstein J (1981) Evolutionary trees from DNA sequences: a maximum likelihood approach. J Mol Evol 17:368–376
6. Moret BME, Bader DA, Warnow T, Wyman SK, Yan M (2001) GRAPPA: a high performance computational tool for phylogeny reconstruction from gene-order data. In: Proceedings of the botany, Albuquerque
7. Moret BME, Bader DA, Warnow T (2002) High-performance algorithm engineering for computational phylogenetics. J Supercomput 22:99–111, Special issue on the best papers from ICCS'01
8. Moret BME, Wyman S, Bader DA, Warnow T, Yan M (2001) A new implementation and detailed study of breakpoint analysis. In: Proceedings of the 6th Pacific symposium biocomputing (PSB 2001), Hawaii, Jan 2001, pp 583–594
9. Saitou N, Nei M (1987) The neighbor-joining method: a new method for reconstruction of phylogenetic trees. Mol Biol Evol 4:406–425
10. Sankoff D, Blanchette M (1998) Multiple genome rearrangement and breakpoint phylogeny. J Comput Biol 5:555–570
11. Yan M (2004) High performance algorithms for phylogeny reconstruction with maximum parsimony. PhD thesis, Electrical and Computer Engineering Department, University of New Mexico, Albuquerque, Jan 2004

Engineering Algorithms for Large Network Applications

Christos Zaroliagis
Department of Computer Engineering and Informatics, University of Patras, Patras, Greece

Years and Authors of Summarized Original Work

2002; Schulz, Wagner, Zaroliagis

Problem Definition

Dealing effectively with applications in large networks, it typically requires the efficient solution of one ore more underlying algorithmic problems. Due to the size of the network, a considerable effort is inevitable in order to achieve the desired efficiency in the algorithm.

One of the primary tasks in large network applications is to answer queries for finding best routes or paths as efficiently as possible. Quite often, the challenge is to process a vast number of such queries on-line: a typical situation encountered in several real-time applications (e.g., traffic information systems, public transportation systems) concerns a query-intensive scenario, where a central server has to answer a huge number of on-line customer queries asking for their best routes (or optimal itineraries). The main goal in such an application is to reduce the (average) response time for a query.

Answering a best route (or optimal itinerary) query translates in computing a minimum cost (shortest) path on a suitably defined directed graph (digraph) with nonnegative edge costs. This in turn implies that the core algorithmic problem underlying the efficient answering of queries is the single-source single-target shortest path problem.

Although the straightforward approach of precomputing and storing shortest paths for all pairs of vertices would enabling the optimal answering of shortest path queries, the quadratic space requirements for digraphs with more than 10^5 vertices makes such an approach prohibitive for large and very large networks. For this reason, the main goal of almost all known approaches is to keep the space requirements as small as possible. This in turn implies that one can afford a heavy (in time) preprocessing, which does not blow up space, in order to speed-up the query time.

The most commonly used approach for answering shortest path queries employs Dijkstra's algorithm and/or variants of it. Consequently, the main challenge is how to reduce the algorithm's *search-space* (number of vertices visited), as this would immediately yield a better query time.

Key Results

All results discussed concern answering of *optimal* (or *exact* or *distance-preserving*) shortest paths under the aforementioned query-intensive scenario, and are all based on the following generic approach. A preprocessing of the input network $G = (V, E)$ takes place that results in a data structure of size $O(|V| + |E|)$ (i.e., linear to the size of G). The data structure contains additional information regarding certain shortest paths that can be used later during querying.

Depending on the pre-computed additional information as well as on the way a shortest path query is answered, two approaches can be distinguished. In the first approach, *graph annotation*, the additional information is attached to vertices or edges of the graph. Then, speed-up techniques to Dijkstra's algorithm are employed that, based on this information, decide quickly which part of the graph does not need to be searched. In the second approach, an *auxiliary graph* G' is constructed hierarchically. A shortest path query is then answered by searching only a small part of G', using Dijkstra's algorithm enhanced with heuristics to further speed-up the query time.

In the following, the key results of the first [3, 4, 9, 11] and the second approach [1, 2, 5, 7, 8, 10] are discussed, as well as results concerning modeling issues.

First Approach: Graph Annotation

The first work under this approach concerns the study in [9] on large railway networks. In that paper, two new heuristics are introduced: the *angle-restriction* (that tries to reduce the search space by taking advantage of the geometric layout of the vertices) and the *selection of stations* (a subset of vertices is selected among which all pairs shortest paths are pre-computed). These two heuristics along with a combination of the classical *goal-directed* or A^* *search* turned out to be rather efficient. Moreover, they motivated two important generalizations [10, 11] that gave further improvements to shortest path query times.

The full exploitation of geometry-based heuristics was investigated in [11], where both street and railway networks are considered. In that paper, it is shown that the search space of Dijkstra's algorithm can be significantly reduced (to 5–10 % of the initial graph size) by extracting geometric information from a given layout of the graph and by encapsulating pre-computed shortest path information in resulted geometric objects, called *containers*. Moreover, the dynamic case of the problem was investigated, where edge costs are subject to change and the geometric containers have to be updated.

A powerful modification to the classical Dijkstra's algorithm, called *reach-based routing*, was presented in [4]. Every vertex is assigned a so-called *reach value* that determines whether a particular vertex will be considered during Dijkstra's algorithm. A vertex is excluded from consideration if its reach value is small; that is, if it does not contribute to any path long enough to be of use for the current query.

A considerable enhancement of the classical A^* *search* algorithm using landmarks (selected vertices like in [9, 10]) and the triangle inequality with respect to the shortest path distances was shown in [3]. Landmarks and triangle inequality help to provide better lower bounds and hence boost A^* search.

Second Approach: Auxiliary Graph

The first work under this approach concerns the study in [10], where a new hierarchical decomposition technique is introduced called *multi-level graph*. A multi-level graph $\mathcal{M}$ is a digraph which is determined by a sequence of subsets of V and which extends E by adding multiple levels of edges. This allows to efficiently construct, during querying, a subgraph of $\mathcal{M}$ which is substantially smaller than G and in which the shortest path distance between any of its vertices is equal to the shortest path distance between the same vertices in G. Further improvements of this approach have been presented recently in [1]. A refinement of the above idea was introduced in [5], where the multi-level overlay graphs are introduced. In such a graph, the decomposition hierarchy is not determined by application-specific information as it happens in [9, 10].

An alternative hierarchical decomposition technique, called *highway hierarchies*, was presented in [7]. The approach takes advantage of the inherent hierarchy possessed by real-world road networks and computes a hierarchy of coarser views of the input graph. Then, the shortest path query algorithm considers mainly the (much smaller in size) coarser views, thus achieving dramatic speed-ups in query time. A revision and improvement of this method was given in [8]. A powerful combination of the highway hierarchies with the ideas in [3] was reported in [2].

Modeling Issues

The modeling of the original best route (or optimal itinerary) problem on a large network to a shortest path problem in a suitably defined directed graph with appropriate edge costs also plays a significant role in reducing the query time. Modeling issues are thoroughly investigated in [6]. In that paper, the first experimental comparison of two important approaches (time-expanded versus time-dependent) is carried out, along with new extensions of them towards realistic modeling. In addition, several new heuristics are introduced to speed-up query time.

Applications

Answering shortest path queries in large graphs has a multitude of applications, especially in traffic information systems under the aforementioned scenario; that is, a central server has to answer, as fast as possible, a huge number of on-line customer queries asking for their best routes or itineraries. Other applications of the above scenario involve route planning systems for cars, bikes and hikers, public transport systems for itinerary information of scheduled vehicles (like trains or buses), answering queries in spatial databases, and web searching. All the above applications concern real-time systems in which users continuously enter their requests for finding their best connections or routes. Hence, the main goal is to reduce the (average) response time for answering a query.

Open Problems

Real-world networks increase constantly in size either as a result of accumulation of more and more information on them, or as a result of the digital convergence of media services, communication networks, and devices. This scaling-up of networks makes the scalability of the underlying algorithms questionable. As the networks continue to grow, there will be a constant need for designing faster algorithms to support core algorithmic problems.

Experimental Results

All papers discussed in section "Key Results" contain important experimental studies on the various techniques they investigate.

Data Sets

The data sets used in [6, 11] are available from http://lso-compendium.cti.gr/ under problems 26 and 20, respectively.

The data sets used in [1, 2] are available from http://www.dis.uniroma1.it/~challenge9/.

URL to Code

The code used in [9] is available from http://doi.acm.org/10.1145/351827.384254.

The code used in [6, 11] is available from http://lso-compendium.cti.gr/ under problems 26 and 20, respectively.

The code used in [3] is available from http://www.avglab.com/andrew/soft.html.

Cross-References

- ▸ Implementation Challenge for Shortest Paths
- ▸ Shortest Paths Approaches for Timetable Information

Recommended Reading

1. Delling D, Holzer M, Müller K, Schulz F, Wagner D (2006) High-performance multi-level graphs. In: 9th DIMACS challenge on shortest paths, Rutgers University, Nov 2006
2. Delling D, Sanders P, Schultes D, Wagner D (2006) Highway hierarchies star. In: 9th DIMACS challenge on shortest paths, Rutgers University, Nov 2006
3. Goldberg AV, Harrelson C (2005) Computing the shortest path: *A** search meets graph theory. In: Proceedings of the 16th ACM-SIAM symposium on discrete algorithms – SODA. ACM, New York/SIAM, Philadelphia, pp 156–165
4. Gutman R (2004) Reach-based routing: a new approach to shortest path algorithms optimized for road networks. In: Algorithm engineering and experiments – ALENEX (SIAM, 2004). SIAM, Philadelphia, pp 100–111
5. Holzer M, Schulz F, Wagner D (2006) Engineering multi-level overlay graphs for shortest-path queries. In: Algorithm engineering and experiments – ALENEX (SIAM, 2006). SIAM, Philadelphia, pp 156–170
6. Pyrga E, Schulz F, Wagner D, Zaroliagis C (2007) Efficient models for timetable information in public transportation systems. ACM J Exp Algorithmic 12(2.4):1–39
7. Sanders P, Schultes D (2005) Highway hierarchies hasten exact shortest path queries. In: Algorithms – ESA 2005. Lecture notes in computer science, vol 3669. pp 568–579
8. Sanders P, Schultes D (2006) Engineering highway hierarchies. In: Algorithms – ESA 2006. Lecture notes in computer science, vol 4168. pp 804–816
9. Schulz F, Wagner D, Weihe K (2000) Dijkstra's algorithm on-line: an empirical case study from public railroad transport. ACM J Exp Algorithmics 5(12): 1–23
10. Schulz F, Wagner D, Zaroliagis C (2002) Using multi-level graphs for timetable information in railway systems. In: Algorithm engineering and experiments – ALENEX 2002. Lecture notes in computer science, vol 2409. pp 43–59
11. Wagner D, Willhalm T, Zaroliagis C (2005) Geometric containers for efficient shortest path computation. ACM J Exp Algorithm 10(1.3):1–30

Engineering Geometric Algorithms

Dan Halperin
School of Computer Science, Tel-Aviv University, Tel Aviv, Israel

Keywords

Certified and efficient implementation of geometric algorithms; Geometric computing with certified numerics and topology

Years and Authors of Summarized Original Work

2004; Halperin

Problem Definition

Transforming a theoretical geometric algorithm into an effective computer program abounds with hurdles. Overcoming these difficulties is the concern of *engineering geometric algorithms*, which deals, more generally, with the design and implementation of certified and efficient solutions to algorithmic problems of geometric nature. Typical problems in this family include the construction of Voronoi diagrams, triangulations, arrangements of curves and surfaces (namely, space subdivisions), two- or higher-dimensional search structures, convex hulls and more.

Geometric algorithms strongly couple topological/combinatorial structures (e.g., a graph describing the triangulation of a set of points) on the one hand, with numerical information (e.g., the coordinates of the vertices of the triangulation) on the other. Slight errors in the numerical calculations, which in many areas of science and engineering can be tolerated, may lead to detrimental mistakes in the topological structure, causing the computer program to crash, to loop infinitely, or plainly to give wrong results.

Straightforward implementation of geometric algorithms as they appear in a textbook, using standard machine arithmetic, is most likely to fail. This entry is concerned only with *certified* solutions, namely, solutions that are guaranteed to construct the exact desired structure or a good approximation of it; such solutions are often referred to as *robust*.

The goal of engineering geometric algorithms can be restated as follows: *Design and implement geometric algorithms that are at once robust and efficient in practice*.

Much of the difficulty in adapting in practice the existing vast algorithmic literature in computational geometry comes from the assumptions that are typically made in the theoretical study of geometric algorithms that (1) the input is in general position, namely, degenerate input is precluded, (2) computation is performed on an ideal computer that can carry out real arithmetic to infinite precision (so-called real RAM), and (3) the cost of operating on a small number of simple geometric objects is "unit" time (e.g., equal cost is assigned to intersecting three spheres and to comparing two integer numbers).

Now, in real life, geometric input is quite often degenerate, machine precision is limited, and operations on a small number of simple geometric objects within the same algorithm may differ 100-fold and more in the time they take to execute (when aiming for certified results). Just implementing an algorithm carefully may not suffice and often redesign is called for.

Key Results

Tremendous efforts have been invested in the design and implementation of robust computational-geometry software in recent years. Two notable large-scale efforts are the CGAL library [1] and the geometric part of the LEDA library [14]. These are jointly reviewed in the survey by Kettner and Näher [13]. Numerous other relevant projects, which for space constraints are not reviewed here, are surveyed by Joswig [12] with extensive references to papers and Web sites.

A fundamental engineering decision to take when coming to implement a geometric algorithm is what will the underlying arithmetic be, that is, whether to opt for exact computation or use the machine floating-point arithmetic. (Other less commonly used options exist as well.) To date, the CGAL and LEDA libraries are almost exclusively based on exact computation. One of the reasons for this exclusivity is that exact computation emulates the ideal computer (for restricted problems) and makes the adaptation of algorithms from theory to software easier. This is facilitated by major headway in developing tools for efficient computation with rational or algebraic numbers (GMP [3], LEDA [14], CORE [2] and more). On top of these tools, clever techniques for reducing the amount of exact computation were developed, such as floating-point filters and the higher-level geometric filtering.

The alternative is to use the machine floating-point arithmetic, having the advantage of being very fast. However, it is nowhere near the ideal infinite precision arithmetic assumed in the theoretical study of geometric algorithms and algorithms have to be carefully redesigned. See, for example, the discussion about imprecision in the manual of QHULL, the convex hull program by Barber et al. [5]. Over the years a variety of specially tailored floating-point variants of algorithms have been proposed, for example, the carefully crafted VRONI package by Held [11], which computes the Voronoi diagram of points and line segments using standard floating-point arithmetic, based on the topology-oriented approach of Sugihara and Iri. While VRONI works very well in practice, it is not theoretically certified. *Controlled perturbation* [9] emerges as a systematic method to produce certified approximations of complex geometric constructs while using floating-point arithmetic: the input is perturbed such that all predicates are computed accurately even with the limited-precision machine arithmetic, and a method is given to bound the necessary magnitude of perturbation that will guarantee the successful completion of the computation.

Another decision to take is how to represent the output of the algorithm, where the major issue is typically how to represent the coordinates of vertices of the output structure(s). Interestingly, this question is crucial when using exact computation since there the output coordinates can be prohibitively large or simply impossible to finitely enumerate. (One should note though that many geometric algorithms are *selective* only, namely, they do not produce new geometric entities but just select and order subsets of the input coordinates. For example, the output of an algorithm for computing the convex hull of a set of points in the plane is an ordering of a subset of the input points. No new point is computed. The discussion in this paragraph mostly applies to algorithms that output new geometric constructs, such as the intersection point of two lines.) But even when using floating-point arithmetic, one may prefer to have a more compact bit-size representation than, say, machine doubles. In this direction there is an effective, well-studied solution for the case of polygonal objects in the plane, called *snap rounding*, where vertices and intersection points are snapped to grid vertices while retaining certain topological properties of the exact desired structure. Rounding with guarantees is in general a very difficult problem, and already for polyhedral objects in 3-space the current attempts at generalizing snap rounding are very costly (increasing the complexity of the rounded objects to the third, or even higher, power).

Then there are a variety of engineering issues depending on the problem at hand. Following are two examples of engineering studies where the experience in practice is different from what the asymptotic resource measures imply. The examples relate to fundamental steps in many geometric algorithms: decomposition and point location.

Decomposition

A basic step in many geometric algorithms is to decompose a (possibly complex) geometric object into simpler subobjects, where each subobject typically has constant descriptive complexity. A well-known example is the triangulation of a polygon. The choice of decomposition may have a significant effect on the efficiency in practice of various algorithms that rely on decomposition. Such is the case when constructing Minkowski sums of polygons in the plane. The Minkowski sum of two sets A and B in $\mathbb{R}^d$ is the vector sum of the two sets $A \oplus B = \{a + b | a \in A, b \in B\}$. The simplest approach to computing Minkowski sums of two polygons in the plane proceeds in three steps: triangulate each polygon, then compute the sum of each triangle of one polygon with each triangle of the other, and finally take the union of all the subsums. In asymptotic measures, the choice of triangulation (over alternative decompositions) has no effect. In practice though, triangulation is probably the worst choice compared with other convex decompositions, even fairly simple heuristic ones (not necessarily optimal), as shown by experiments on a dozen different decomposition methods [4]. The explanation is that triangulation increases the overall complexity of the subsums and in turn makes the union stage more complex – indeed by a constant factor, but a noticeable factor in practice. Similar phenomena were observed in other situations as well. For example, when using the prevalent vertical decomposition of arrangements – often it is too costly compared with sparser decompositions (i.e., decompositions that add fewer extra features).

Point Location

A recurring problem in geometric computing is to process given planar subdivision (planar map), so as to efficiently answer *point-location* queries: Given a point q in the plane, which face of the map contains q? Over the years a variety of point-location algorithms for planar maps were implemented in CGAL, in particular, a hierarchical search structure that guarantees logarithmic query time after expected $O(n \log n)$ preprocessing time of a map with n edges. This algorithm is referred to in CGAL as the *RIC* point-location algorithm after the preprocessing method which uses randomized incremental construction. Several simpler, easier-to-program algorithms for point location were also implemented. None of the latter beats the RIC algorithm in query time. However, the RIC is by far the slowest of all the implemented algorithms in terms of preprocessing, which in many scenarios renders it less effective. One of the simpler methods devised is a variant of the well-known *jump-and-walk* approach to point location. The algorithm scatters points (so-called *landmarks*) in the map and maintains the landmarks (together with their containing faces) in a nearest-neighbor search structure. Once a query q is issued it finds the nearest landmark ℓ to q, and "walks" in the map from ℓ toward q along the straight line segment connecting them. This landmark approach offers query time that is only slightly more expensive than the RIC method while being very efficient in preprocessing. The full details can be found in [10]. This is yet another consideration when designing (geometric) algorithms: the cost of preprocessing (and storage) versus the cost of a query. Quite often the effective (practical) tradeoff between these costs needs to be deduced experimentally.

Applications

Geometric algorithms are useful in many areas. Triangulations and arrangements are examples of basic constructs that have been intensively studied in computational geometry, carefully implemented and experimented with, as well as used in diverse applications.

Triangulations

Triangulations in two and three dimensions are implemented in CGAL [7]. In fact, CGAL offers many variants of triangulations useful for different applications. Among the applications where CGAL triangulations are employed are meshing, molecular modeling, meteorology, photogrammetry, and geographic information systems (GIS). For other available triangulation packages, see the survey by Joswig [12].

Arrangements

Arrangements of curves in the plane are supported by CGAL [15], as well as envelopes of surfaces in three-dimensional space. Forthcoming is support also for arrangements of curves on surfaces. CGAL arrangements have been used in motion planning algorithms, computer-aided design and manufacturing, GIS, computer graphics, and more (see Chap. 1 in [6]).

Open Problems

In spite of the significant progress in certified implementation of effective geometric algorithms, the existing theoretical algorithmic solutions for many problems still need adaptation or redesign to be useful in practice. One example where progress can have wide repercussions is devising effective decompositions for curved geometric objects (e.g., arrangements) in the plane and for higher-dimensional objects. As mentioned earlier, suitable decompositions can have a significant effect on the performance of geometric algorithms in practice.

Certified fixed-precision geometric computing lags behind the exact computing paradigm in terms of available robust software, and moving forward in this direction is a major challenge. For example, creating a certified floating-point counterpart to CGAL is a desirable (and highly intricate) task.

Another important tool that is largely missing is consistent and efficient rounding of geometric objects. As mentioned earlier, a fairly satisfactory solution exists for polygonal objects in the plane. Good techniques are missing for curved objects in the plane and for higher-dimensional objects (both linear and curved).

URL to Code

http://www.cgal.org

Cross-References

▸ LEDA: a Library of Efficient Algorithms
▸ Robust Geometric Computation

Recommended Reading

Conferences publishing papers on the topic include the ACM Symposium on Computational Geometry (SoCG), the Workshop on Algorithm Engineering and Experiments (ALENEX), the Engineering and Applications Track of the European Symposium on Algorithms (ESA), its predecessor and the Workshop on Experimental Algorithms (WEA). Relevant journals include the *ACM Journal on Experimental Algorithmics, Computational Geometry: Theory and Applications* and the *International Journal of Computational Geometry and Applications*. A wide range of relevant aspects are discussed in the recent book edited by Boissonnat and Teillaud [6], titled *Effective Computational Geometry for Curves and Surfaces*.

1. The CGAL project homepage. http://www.cgal.org/. Accessed 6 Apr 2008
2. The CORE library homepage. http://www.cs.nyu.edu/exact/core/. Accessed 6 Apr 2008
3. The GMP webpage. http://gmplib.org/. Accessed 6 Apr 2008
4. Agarwal PK, Flato E, Halperin D (2002) Polygon decomposition for efficient construction of Minkowski sums. Comput Geom Theory Appl 21(1–2):39–61
5. Barber CB, Dobkin DP, Huhdanpaa HT (2008) Imprecision in QHULL. http://www.qhull.org/html/qh-impre.htm. Accessed 6 Apr 2008
6. Boissonnat J-D, Teillaud M (eds) (2006) Effective computational geometry for curves and surfaces. Springer, Berlin
7. Boissonat J-D, Devillers O, Pion S, Teillaud M, Yvinec M (2002) Triangulations in CGAL. Comput Geom Theory Appl 22(1–3):5–19
8. Fabri A, Giezeman G-J, Kettner L, Schirra S, Schönherr S (2000) On the design of CGAL a computational geometry algorithms library. Softw Pract Exp 30(11):1167–1202
9. Halperin D, Leiserowitz E (2004) Controlled perturbation for arrangements of circles. Int J Comput Geom Appl 14(4–5):277–310
10. Haran I, Halperin D (2006) An experimental study of point location in general planar arrangements. In: Proceedings of 8th workshop on algorithm engineering and experiments, Miami, pp 16–25
11. Held M (2001) VRONI: an engineering approach to the reliable and efficient computation of Voronoi diagrams of points and line segments. Comput Geom Theory Appl 18(2):95–123
12. Joswig M (2004) Software. In: Goodman JE, O’Rourke J (eds) Handbook of discrete and computational geometry, chapter 64, 2nd edn. Chapman & Hall/CRC, Boca Raton, pp 1415–1433

13. Kettner L, Näher S (2004) Two computational geometry libraries: LEDA and CGAL. In: Goodman JE, O'Rourke J (eds) Handbook of discrete and computational geometry, chapter 65, 2nd edn. Chapman & Hall/CRC, Boca Raton, pp 1435–1463
14. Mehlhorn K, Näher S (2000) LEDA: a platform for combinatorial and geometric computing. Cambridge University Press, Cambridge
15. Wein R, Fogel E, Zukerman B, Halperin D (2007) Advanced programming techniques applied to CGAL's arrangement package. Comput Geom Theory Appl 36(1–2):37–63

Enumeration of Non-crossing Geometric Graphs

Shin-ichi Tanigawa
Research Institute for Mathematical Sciences (RIMS), Kyoto University, Kyoto, Japan

Keywords

Enumeration; Non-crossing (crossing-free) geometric graphs; Triangulations

Years and Authors of Summarized Original Work

2009; Katoh, Tanigawa
2014; Wettstein

Problem Definition

Let P be a set of n points in the plane in general position, i.e., no three points are collinear. A *geometric graph on* P is a graph on the vertex set P whose edges are straight-line segments connecting points in P. A geometric graph is called *non-crossing* (or *crossing-free*) if any pair of its edges does not have a point in common except possibly their endpoints. We denote by $\mathcal{P}(P)$ the set of all non-crossing geometric graphs on P (which are also called *plane straight-line graphs* on P). A graph class $\mathcal{C}(P) \subseteq \mathcal{P}(P)$ can be defined by imposing additional properties such as connectivity, degree bound, or cycle-freeness. Examples of $\mathcal{C}(P)$ are the set of *triangulations* (i.e., inclusion-wise maximal graphs in $\mathcal{P}(P)$), the set of *non-crossing perfect matchings*, the set of *non-crossing spanning k-connected graphs*, the set of *non-crossing spanning trees*, and the set of *non-crossing spanning cycles* (i.e., simple polygons). The problem is to enumerate all graphs in $\mathcal{C}(P)$ for a given set P of n points in the plane.

The following notations will be used to denote the cardinality of $\mathcal{C}(P)$: $\mathsf{tri}(P)$ for triangulations, $\mathsf{pg}(P)$ for plane straight-line graphs, $\mathsf{st}(P)$ for non-crossing spanning trees, and $\mathsf{cg}(P)$ for non-crossing spanning connected graphs.

Key Results

Enumeration of Triangulations

The first efficient enumeration algorithm for triangulations was given by Avis and Fukuda [3] as an application of their reverse search technique. The algorithm relies on well-known properties of Delaunay triangulations.

A triangulation T on P is called *Delaunay* if no point in P is contained in the interior of the circumcircle of a triangle in T. If it is assumed for simplicity that no four points in P lie on a circle, then the Delaunay triangulation on P exists and is unique. The Delaunay triangulation has the lexicographically largest angle vector among all triangulations on P, where the angle vector of a triangulation is the list of all the angles sorted in nondecreasing order.

For a triangulation T, a *Lawson edge* is an edge ab which is incident to two triangles, say abc and abd in T, and the circumcircle of abc contains d in its interior. *Flipping* a Lawson edge ab (i.e., replacing ab with another diagonal edge cd) always creates a triangulation having a lexicographically larger angle vector. Moreover a triangulation has a Lawson edge if and only if it is not Delaunay. In other words, any triangulation can be converted to the Delaunay triangulation by flipping Lawson edges.

In the algorithm by Avis and Fukuda, a rooted search tree on the set of triangulations is defined such that the root is the Delaunay triangulation and the parent of a non-Delaunay triangulation T

is a triangulation obtained by flipping the smallest Lawson edge in T (assuming a fixed total ordering on edges). Since the Delaunay triangulation can be computed in $O(n \log n)$ time, all the triangulations can be enumerated by tracing the rooted search tree based on the reverse search technique. A careful implementation achieves $O(n \cdot \mathsf{tr}(P))$ time with $O(n)$ space.

An improved algorithm was given by Bespamyatnikh [5], which runs in $O(\log \log n \cdot \mathsf{tr}(P))$ time with $O(n)$ space. His algorithm is also based on the reverse search technique, but the rooted search tree is defined by using the lexicographical ordering of edge vectors rather than angle vectors. This approach was also applied to the enumeration of pointed pseudo-triangulations [4]. See [6] for another approach.

Enumeration of Non-crossing Geometric Graphs

In [3], Avis and Fukuda also developed an enumeration algorithm for non-crossing spanning trees, whose running time is $O(n^3 \cdot \mathsf{sp}(P))$. This was improved to $O(n \log n \cdot \mathsf{sp}(P))$ by Aichholzer et al. [1]. They also gave enumeration algorithms for plane straight-line graphs and non-crossing spanning connected graphs with running time $O(n \log n \cdot \mathsf{pg}(P))$ and $O(n \log n \cdot \mathsf{sc}(P))$, respectively.

Katoh and Tangiawa [8] proposed a simple enumeration technique for wider classes of non-crossing geometric graphs. The same approach was independently given by Razen and Welzl [9] for counting the number of plane straight-line graphs, and the following description in terms of Delaunay triangulations is from [9].

Since each graph in $\mathcal{C}(P)$ is a subgraph of a triangulation, one can enumerate all graphs in $\mathcal{C}(P)$ by first enumerating all triangulations and then enumerating all graphs in $\mathcal{C}(P)$ in each triangulation. The output may contain duplicates, but one can avoid duplicates by enumerating only graphs in $\{G \in \mathcal{C}(P) \mid L(T) \subseteq E(G) \subseteq E(T)\}$ for each triangulation T, where $L(T)$ denotes the set of the Lawson edges in T. This enumeration framework leads to an algorithm with time complexity $O((t^{\mathrm{pre}} + \log \log n)\mathsf{tri}(P) + t \cdot \mathsf{c}(P))$ and space complexity $O(n + s)$ provided that graphs in $\{G \in \mathcal{C}(G) \mid L(T) \subseteq E(G) \subseteq E(T)\}$ can be enumerated in $O(t)$ time per graph with $O(t^{\mathrm{pre}})$ time preprocessing and $O(s)$ space for each triangulation T. For example, in the case of non-crossing spanning trees, one can use a fast enumeration algorithm for spanning trees in a given undirected graph to solve each subproblem, and the current best implementation gives an enumeration algorithm for non-crossing spanning trees with time complexity $O(n \cdot \mathsf{tri}(P) + \mathsf{st}(P))$.

For plane straight-line graphs and spanning connected graphs, $\mathsf{pg}(P) \geq (\sqrt{8})^n \mathsf{tri}(P)$ [9] and $\mathsf{cs}(P) \geq 1.51^n \mathsf{tri}(P)$ [8] hold for any P in general position. Hence $\mathsf{tri}(P)$ is dominated by $\mathsf{pg}(P)$ and $\mathsf{cs}(P)$, respectively, and plane straight-line graphs or non-crossing spanning connected graphs can be enumerated in constant time on average with $O(n)$ space [8]. The same technique can be applied to the set of non-crossing spanning 2-connected graphs. It is not known whether there is a constant $c > 1$ such that $\mathsf{st}(P) \geq c^n \mathsf{tri}(P)$ for every P in general position.

In [8] an approach that avoids enumerating all triangulations was also discussed. Suppose that a nonempty subset $\mathcal{I}$ of $\mathcal{P}(P)$ satisfies a monotone property, i.e., for every $G, G' \in \mathcal{P}(P)$ with $G \subseteq G'$, $G' \in \mathcal{I}$ implies $G \in \mathcal{I}$, and suppose that $\mathcal{C}(P)$ is the set of all maximal elements in $\mathcal{I}$. Then all graphs in $\mathcal{C}(P)$ can be enumerated just by enumerating all triangulations T on P with $L(T) \in \mathcal{I}$, and this can be done efficiently based on the reverse search technique. This approach leads to an algorithm for enumerating non-crossing minimally rigid graphs in $O(n^2)$ time per output with $O(n)$ space, where a graph $G = (V, E)$ is called *minimally rigid* if $|E| = 2|V| - 3$ and $|E'| \leq 2|V'| - 3$ for any subgraph $G' = (V', E')$ with $|V'| \geq 2$.

Enumeration of Non-crossing Perfect Matchings

Wettstein [10] proposed a new enumeration (and counting) technique for non-crossing geometric graphs. This is motivated from a counting algorithm of triangulations by Alvarez and

Seidel [2] and can be used for enumerating, e.g., non-crossing perfect matchings, plane straight-line graphs, convex subdivisions, and triangulations. The following is a sketch of the algorithm for non-crossing perfect matchings.

A matching can be reduced to an empty graph by removing edges one by one. By fixing a rule for the removing edge in each matching, one can define a rooted search tree $\mathcal{T}$ on the set of non-crossing matchings, and the set of non-crossing matchings can be enumerated by tracing $\mathcal{T}$. To reduce time complexity, the first idea is to trace only a subgraph $\mathcal{T}'$ of $\mathcal{T}$ induced by a subclass of non-crossing matchings by a clever choice of removing edges. Another idea is a compression of the search tree $\mathcal{T}'$ by using an equivalence relation on the subclass of non-crossing matchings. The resulting graph $\mathcal{G}$ is a digraph on the set of equivalence classes, where there is a one-to-one correspondence between non-crossing perfect matchings and directed paths of length $n/2$ from the root. A crucial observation is that $\mathcal{G}$ has at most $2^n n^3$ edges while the number of non-crossing perfect matchings is known to be at least $\text{poly}(n) \cdot 2^n$ for any P in general position [7]. Hence non-crossing perfect matchings can be enumerated in polynomial time on average by first constructing $\mathcal{G}$ and then enumerating all the dipaths of length $n/2$ in $\mathcal{G}$. It was also noted in [10] that the algorithm can be polynomial-time delay, but still the space complexity is exponential in n.

Open Problems

A challenging open problem is to design an efficient enumeration algorithm for the set of non-crossing spanning cycles, the set of highly connected triangulations, or the set of degree-bounded triangulations or non-crossing spanning trees. It is also not known whether triangulations can be enumerated in constant time per output.

Cross-References

▶ Enumeration of Paths, Cycles, and Spanning Trees
▶ Reverse Search; Enumeration Algorithms
▶ Voronoi Diagrams and Delaunay Triangulations

Recommended Reading

1. Aichholzer O, Aurenhammer F, Huemer C, Krasser H (2007) Gray code enumeration of plane straight-line graphs. Graphs Comb 23(5):467–479
2. Alvarez V, Seidel R (2013) A simple aggregative algorithm for counting triangulations of planar point sets and related problems. In: Proceedings of the 29th annual symposium on computational geometry (SoCG2013), Rio de Janeiro. ACM
3. Avis D, Fukuda K (1996) Reverse search for enumeration. Discret Appl Math 65(1–3):21–46
4. Bereg S (2005) Enumerating pseudo-triangulations in the plane. Comput Geom Theory Appl 30(3):207–222
5. Bespamyatnikh S (2002) An efficient algorithm for enumeration of triangulations. Comput Geom Theory Appl 23(3):271–279
6. Brönnimann H, Kettner L, Pocchiola M, Snoeyink J (2006) Counting and enumerating pointed pseudotriangulations with the greedy flip algorithm. SIAM J Comput 36(3):721–739
7. García A, Noy M, Tejel J (2000) Lower bounds on the number of crossing-free subgraphs of K_n. Comput Geom Theory Appl 16(4):211–221
8. Katoh N, Tanigawa S (2009) Fast enumeration algorithms for non-crossing geometric graphs. Discret Comput Geom 42(3):443–468
9. Razen A, Welzl E (2011) Counting plane graphs with exponential speed-up. In: Calude C, Rozenberg G, Salomaa A, Maurer HA (eds) Rainbow of computer science. Springer, Berlin/Heidelberg, pp 36–46
10. Wettstein M (2014) Counting and enumerating crossing-free geometric graphs. In: Proceedings of the 30th annual symposium on computational geometry (SoCG2014), Kyoto. ACM

Enumeration of Paths, Cycles, and Spanning Trees

Roberto Grossi
Dipartimento di Informatica, Università di Pisa, Pisa, Italy

Keywords

Amortized analysis; Arborescences; Cycles; Elementary circuits; Enumeration algorithms; Graphs; Paths; Spanning trees

Years and Authors of Summarized Original Work

1975; Johnson
1975; Read and Tarjan
1994; Shioura, Tamura, Uno
1995; Kapoor, Ramesh
1999; Uno
2013; Birmelé, Ferreira, Grossi, Marino, Pisanti, Rizzi, Sacomoto, Sagot

Problem Definition

Let $G = (V, E)$ be a (directed or undirected) graph with $n = |V|$ vertices and $m = |E|$ edges. A *walk* of length k is a sequence of vertices $v_0, \ldots, v_k \in V$ such that v_i and v_{i+1} are connected by an edge of E, for any $0 \le i < k$. A *path* π of length k is a walk $v_0, \ldots, v_k$ such that any two vertices v_i and v_j are distinct, for $0 \le i < j \le k$: this is also called *st-path* where $s = v_0$ and $t = v_k$. A *cycle* (or, equivalently, *elementary circuit*) C of length $k + 1$ is a path $v_0, \ldots, v_k$ such that v_k and v_0 are connected by an edge of E.

We denote by $\mathcal{P}_{st}(G)$ the set of *st*-paths in G for any two given vertices $s, t \in V$ and by $\mathcal{C}(G)$ the set of cycles in G. Given a graph G, the problem of *st-path enumeration* asks for generating all the paths in $\mathcal{P}_{st}(G)$. The problem of *cycle enumeration* asks for generating all the cycles in $\mathcal{C}(G)$.

We denote by $\mathcal{S}(G)$ the set of spanning trees in a connected graph G, where a spanning tree $T \subseteq E$ is a set of $|T| = n - 1$ edges such that no cycles are contained in T and each vertex in V is incident to at least an edge of T. Given a connected graph G, the problem of *spanning tree enumeration* asks for generating all the spanning trees in $\mathcal{S}(G)$.

Typical costs of enumeration algorithms are proportional to the output size times a polynomial function of the graph size. Sometimes enumeration is meant with the stronger property of *listing*, where each solution is explicitly output. In the latter case, we define an algorithm for a listing problem to be *optimally output sensitive* if its running time is $O(n + m + K)$ where K is the following output cost for the enumeration problem at hand, namely, $\mathcal{P}_{st}(G)$, $\mathcal{C}(G)$, or $\mathcal{S}(G)$.

- $K = \sum_{\pi \in \mathcal{P}_{st}(G)} |\pi|$ where $|\pi|$ is the number of nodes in the *st*-path π.
- $K = \sum_{C \in \mathcal{C}(G)} |C|$ where $|C|$ is the number nodes in the cycle C.
- $K = \sum_{T \in \mathcal{S}(G)} |T| = |\mathcal{S}(G)| \cdot (n - 1)$ for spanning trees.

Although the above is a notion of optimality for listing solutions explicitly, it is possible in some cases that the enumeration algorithm can efficiently encode the differences between consecutive solutions in the sequence produced by the enumeration. This is the case of spanning trees, where a cost of $K = |\mathcal{S}(G)|$ is possible when they are implicitly represented during enumeration. This is called CAT (constant amortized time) enumeration in [28].

Key Results

Some possible approaches to attack the enumeration problems are listed below, where the term "search" is meant as an exploration of the space of solutions.

Backtrack search. A backtracking algorithm finds the solutions for a listing problem by exploring the search space and abandoning a partial solution (thus the name "backtracking") that cannot be completed to a valid one.

Binary partition search. An algorithm divides the search space into two parts. In the case of graphs, this is generally done by taking an edge (or a vertex) and (i) searching for all solutions that include that edge (resp. vertex) and (ii) searching for all solutions that do not include that edge (resp. vertex). Point (i) can sometimes be implemented by contracting the edge, i.e., merging the endpoints of the edge and their adjacency list.

Differential encoding search. The space of solutions is encoded in such a way that consecutive solutions differ by a constant number of modifications. Although not every enumeration problem has properties that allow such encoding, this technique leads to very efficient algorithms.

Reverse search. This is a general technique to explore the space of solutions by reversing a local search algorithm. This approach implicitly generates a tree of the search space that is traversed by the reverse search algorithm. One of the properties of this tree is that it has bounded height, a useful fact for proving the time complexity of the algorithm.

Although there is some literature on techniques for enumeration problems [38, 39, 41], many more techniques and "tricks" have been introduced when attacking particular problems. For a deep understanding of the topic, the reader is recommended to review the work of researchers such as David Avis, Komei Fukuda, Shin-ichi Nakano, and Takeaki Uno.

Path and Cycles

Listing all the cycles in a graph is a classical problem whose efficient solutions date back to the early 1970s. In particular, at the turn of the 1970s, several algorithms for enumerating all cycles of an undirected graph were proposed. There is a vast body of work, and the majority of the algorithms listing all the cycles can be divided into the following three classes (see [1, 23] for excellent surveys).

Search space algorithms. Cycles are looked for in an appropriate search space. In the case of undirected graphs, the *cycle vector space* [6] turned out to be the most promising choice: from a basis for this space, all vectors are computed, and it is tested whether they are a cycle. Since the algorithm introduced in [43], many algorithms have been proposed: however, the complexity of these algorithms turns out to be exponential in the dimension of the vector space and thus in n. For the special case of planar graphs, the paper in [34] describes an algorithm listing all the cycles in $O((|\mathcal{C}(G)| + 1)n)$ time.

Backtrack algorithms. All paths are generated by backtrack, and, for each path, it is tested whether it is a cycle. One of the first algorithms based on this approach is the one proposed in [37], which is however exponential in $|\mathcal{C}(G)|$. By adding a simple pruning strategy, this algorithm has been successively modified in [36]: it lists all the cycles in $O(nm(|\mathcal{C}(G)| + 1))$ time. Further improvements were proposed in [16], [35], and [27], leading to $O((|\mathcal{C}(G)| + 1)(m + n))$ time algorithms that work for both directed and undirected graphs. Apart from the algorithm in [37], all the algorithms based on this approach are *polynomial-time delay*, that is, the time elapsed between the outputting of two cycles is polynomial in the size of the graph (more precisely, $O(nm)$ in the case of the algorithm of [36] and $O(m)$ in the case of the other three algorithms).

Algorithms using the powers of the adjacency matrix. This approach uses the so-called variable adjacency matrix, that is, the formal sum of edges joining two vertices. A nonzero element of the pth power of this matrix is the sum of all walks of length p: hence, to compute all cycles, we compute the nth power of the variable adjacency matrix. This approach is not very efficient because of the non-simple walks. All algorithms based on this approach (e.g., [26] and [45]) basically differ only on the way they avoid to consider walks that are neither paths nor cycles.

For directed graphs, the best known algorithm for listing cycles is Johnson's [16]. It builds upon Tarjan's backtracking search [36], where the search starts from the least vertex of each strongly connected component. After that, a new strongly connected component is discovered, and the search starts again from the least vertex in it. When exploring a strongly connected component with a recursive backtracking procedure, it uses an enhanced marking system to avoid visiting the same cycle multiple times. A vertex is marked each time it enters the backtracking stack. Upon leaving the stack, if a cycle is found, then the vertex is unmarked. Otherwise, it remains marked until another vertex involved in a cycle is popped from the stack, and there exists a path of marked vertices (not in the stack) between these two vertices. This strategy is implemented using a collection of lists B, one list per vertex containing its marked neighbors not in the stack. Unmarking is done by a recursive procedure. The complexity of the algorithm is $O(n + m + |\mathcal{C}(G)|m)$ time and $O(n + m)$ space.

For undirected graphs, Johnson's bound can be improved with an optimal output-sensitive algorithm [2]. First of all, the cycle enumeration problem is reduced to the st-path enumeration by considering any spanning tree of the given graph G and its non-tree edges $b_1, b_2, \ldots, b_r$. Then, for $i = 1, 2, \ldots, r$, the cycles in $\mathcal{C}(G)$ can be listed as st-paths in $G \setminus \{b_1, \ldots, b_i\}$, where s and t are the endpoint of non-tree edge b_i. Hence, the subproblem to be solved with an optimal output-sensitive algorithm is the st-path enumeration problem. Binary partition search is adopted to avoid duplicated output, but the additional ingredient is the notion of *certificate*, which is a suitable data structure that maintains the biconnected components of the residual graph and guarantees that each recursive call thus produces at least one solution. Its amortized analysis is based on a lower bound on the number of st-paths that can be listed in the residual graph, so as to absorb the cost of maintaining the certificate. The final cost is $O(m+n+\sum_{\pi \in \mathcal{P}_{st}(G)} |\pi|)$ time and $O(n+m)$ space, which is optimal for listing.

Spanning Trees

Listing combinatorial structures in graphs has been a long-time problem of interest. In his 1970 book [25], Moon remarks that "many papers have been written giving algorithms, of varying degrees of usefulness, for listing the spanning trees of a graph" (citation taken from [28]). Among others, he cites [7, 9, 10, 13, 42] – some of these early papers date back to the beginning of the twentieth century. More recently, in the 1960s, Minty proposed an algorithm to list all spanning trees [24].

The first algorithmic solutions appeared in the 1960s [24] and the combinatorial papers even much earlier [25]. Other results from Welch, Tiernan, and Tarjan for this and other problems soon followed [36, 37, 43] and used backtracking search. Read and Tarjan presented an algorithm taking $O(m + n + |\mathcal{S}(G)| \cdot m)$ time and $O(m + n)$ space [27]. Gabow and Myers proposed the first algorithm [11] which is optimal when the spanning trees are explicitly listed, taking $O(m + n + |\mathcal{S}(G)| \cdot n)$ time and $O(m + n)$ space.

When the spanning trees are implicitly enumerated, Kapoor and Ramesh [17] showed that an elegant incremental representation is possible by storing just the $O(1)$ information needed to reconstruct a spanning tree from the previously enumerated one, requiring a total of $O(m + n + |\mathcal{S}(G)|)$ time and $O(mn)$ space [17], later reduced to $O(m)$ space by Shioura et al. [32]. These methods use the reverse search where the elements are the spanning trees. The rule for moving along these elements and for their differential encoding is based upon the observation that adding a non-tree edge and removing a tree edge of the cycle thus formed produces another spanning tree from the current one. Some machinery is needed to avoid duplicated spanning trees and to spend $O(1)$ amortized cost per generated spanning tree.

A simplification of the incremental enumeration of spanning trees is based on matroids and presented by Uno [39]. It is a binary partition search giving rise to a binary enumeration tree, where the two children calls generated by the current call correspond to the fact that the current edge is either contracted in $O(n)$ time or deleted in $O(m - n)$ time. There is a trimming and balancing phase in $O(n(m - n))$ time: trimming removes the edges that do not appear in any of the spanning trees that will be generated by the current recursive call and contracts the edges that appear in all of these spanning trees. Balancing splits the recursive calls as in the divide-and-conquer paradigm. A crucial property proved in [39] is that the residual graph will generate at least $\Omega(n(m - n))$ spanning trees, and thus the total cost per call, which is dominated by trimming and balancing, can be amortized as $O(1)$ per spanning tree. The method in [39] works also for directed spanning trees (arborescences) with an amortized $O(\log n)$ time cost per directed spanning tree.

Applications

The classical problem of listing all the cycles of a graph has been extensively studied for its many applications in several fields, ranging from the mechanical analysis of chemical structures [33] to the design and analysis of reliable communication networks and the graph isomorphism problem [43]. Almost 40 years after, the problem of efficiently listing all cycles of a graph is still an active area of research (e.g., [14, 15, 22, 29, 30, 44]). New application areas have emerged in the last decade, such as bioinformatics: for example, two algorithms for this problem have been proposed in [20] and [21] while studying biological interaction graphs, with important network properties derived for feedback loops, signaling paths, and dependency matrix, to name a few.

When considering weighted cycles, the paper in [19] proves that there is no polynomial total time algorithm (unless $P = NP$) to enumerate negative-weight (simple) cycles in directed weighted graphs. Uno [40] and Ferreira et al. [8] considered the enumeration of chordless cycles and paths. A chordless or induced cycle (resp., path) in an undirected graph is a cycle (resp., path) such that the subgraph induced by its vertices contains exactly the edges of the cycle (resp., path). Both chordless cycles and paths are very natural structures in undirected graphs with an important history, appearing in many papers in graph theory related to chordal graphs, perfect graphs, and co-graphs (e.g., [4, 5, 31]), as well as many NP-complete problems involving them (e.g., [3, 12, 18]).

As for spanning trees, we refer to the section "K-best enumeration" of this book.

Recommended Reading

1. Bezem G, Leeuwen Jv (1987) Enumeration in graphs. Technical Report RUU-CS-87-07, Utrecht University
2. Birmelé E, Ferreira R, Grossi R, Marino A, Pisanti N, Rizzi R, Sacomoto G, Sagot MF (2013) Optimal listing of cycles and st-paths in undirected graphs. In: Proceedings of the twenty-fourth annual ACM-SIAM symposium on discrete algorithms, New Orleans. SIAM, pp 1884–1896
3. Chen Y, Flum J (2007) On parameterized path and chordless path problems. In: IEEE conference on computational complexity, San Diego, pp 250–263
4. Chudnovsky M, Robertson N, Seymour P, Thomas R (2006) The strong perfect graph theorem. Ann Math 164:51–229
5. Conforti M, Rao MR (1992) Structural properties and decomposition of linear balanced matrices. Math Program 55:129–168
6. Diestel R (2005) Graph theory. Graduate texts in mathematics. Springer, Berlin/New York
7. Duffin R (1959) An analysis of the wang algebra of networks. Trans Am Math Soc 93:114–131
8. Ferreira RA, Grossi R, Rizzi R, Sacomoto G, Sagot M (2014) Amortized $\tilde{O}(|V|)$-delay algorithm for listing chordless cycles in undirected graphs. In: Proceedings of European symposium on algorithms. LNCS, vol 8737. Springer, Berlin/Heidelberg, pp 418–429
9. Feussner W (1902) Uber stromverzweigung in netzformigen leitern. Ann Physik 9:1304–1329
10. Feussner W (1904) Zur berechnung der stromstarke in netzformigen leitern. Ann Physik 15:385–394
11. Gabow HN, Myers EW (1978) Finding all spanning trees of directed and undirected graphs. SIAM J Comput 7(3):280–287
12. Haas R, Hoffmann M (2006) Chordless paths through three vertices. Theor Comput Sci 351(3):360–371
13. Hakimi S (1961) On trees of a graph and their generation. J Frankl Inst 272(5):347–359
14. Halford TR, Chugg KM (2004) Enumerating and counting cycles in bipartite graphs. In: IEEE Communication Theory Workshop, Cancun
15. Horváth T, Gärtner T, Wrobel S (2004) Cyclic pattern kernels for predictive graph mining. In: Proceedings of 10th ACM SIGKDD, Seattle, pp 158–167
16. Johnson DB (1975) Finding all the elementary circuits of a directed graph. SIAM J Comput 4(1):77–84
17. Kapoor S, Ramesh H (1995) Algorithms for enumerating all spanning trees of undirected and weighted graphs. SIAM J Comput 24:247–265
18. Kawarabayashi K, Kobayashi Y (2008) The induced disjoint paths problem. In: Lodi A, Panconesi A, Rinaldi G (eds) IPCO. Lecture notes in computer science, vol 5035. Springer, Berlin/Heidelberg, pp 47–61
19. Khachiyan L, Boros E, Borys K, Elbassioni K, Gurvich V (2006) Generating all vertices of a polyhedron is hard. In: Proceedings of the seventeenth annual ACM-SIAM symposium on discrete algorithm, society for industrial and applied mathematics, Philadelphia, SODA '06, Miami, pp 758–765
20. Klamt S et al (2006) A methodology for the structural and functional analysis of signaling and regulatory networks. BMC Bioinform 7:56
21. Klamt S, von Kamp A (2009) Computing paths and cycles in biological interaction graphs. BMC Bioinform 10:181
22. Liu H, Wang J (2006) A new way to enumerate cycles in graph. In: AICT and ICIW, Washington, DC, USA pp 57–59
23. Mateti P, Deo N (1976) On algorithms for enumerating all circuits of a graph. SIAM J Comput 5(1):90–99
24. Minty G (1965) A simple algorithm for listing all the trees of a graph. IEEE Trans Circuit Theory 12(1):120–120

25. Moon J (1970) Counting labelled trees. Canadian mathematical monographs, vol 1. Canadian Mathematical Congress, Montreal
26. Ponstein J (1966) Self-avoiding paths and the adjacency matrix of a graph. SIAM J Appl Math 14:600–609
27. Read RC, Tarjan RE (1975) Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. Networks 5(3):237–252
28. Ruskey F (2003) Combinatorial generation. Preliminary working draft University of Victoria, Victoria
29. Sankar K, Sarad A (2007) A time and memory efficient way to enumerate cycles in a graph. In: Intelligent and advanced systems, Kuala Lumpur pp 498–500
30. Schott R, Staples GS (2011) Complexity of counting cycles using Zeons. Comput Math Appl 62:1828–1837
31. Seinsche D (1974) On a property of the class of n-colorable graphs. J Comb Theory, Ser B 16(2):191–193
32. Shioura A, Tamura A, Uno T (1994) An optimal algorithm for scanning all spanning trees of undirected graphs. SIAM J Comput 26:678–692
33. Sussenguth E (1965) A graph-theoretical algorithm for matching chemical structures. J Chem Doc 5:36–43
34. Syslo MM (1981) An efficient cycle vector space algorithm for listing all cycles of a planar graph. SIAM J Comput 10(4):797–808
35. Szwarcfiter JL, Lauer PE (1976) A search strategy for the elementary cycles of a directed graph. BIT Numer Math 16:192–204
36. Tarjan RE (1973) Enumeration of the elementary circuits of a directed graph. SIAM J Comput 2(3):211–216
37. Tiernan JC (1970) An efficient search algorithm to find the elementary circuits of a graph. Commun ACM 13:722–726
38. Uno T (1998) New approach for speeding up enumeration algorithms. Algorithms and computation. Springer, Berlin/Heidelberg, pp 287–296
39. Uno T (1999) A new approach for speeding up enumeration algorithms and its application for matroid bases. In: COCOON, Tokyo, pp 349–359
40. Uno T (2003) An output linear time algorithm for enumerating chordless cycles. In: 92nd SIGAL of information processing society Japan, Tokyo pp 47–53, (in Japanese)
41. Uno T (2003) Two general methods to reduce delay and change of enumeration algorithms. National Institute of Informatics, Technical Report NII-2003-004E, Tokyo, Apr. 2003
42. Wang K (1934) On a new method for the analysis of electrical networks. Nat Res Inst for Eng Academia Sinica Memoir (2):19
43. Welch JT Jr (1966) A mechanical analysis of the cyclic structure of undirected linear graphs. J ACM 13:205–210
44. Wild M (2008) Generating all cycles, chordless cycles, and hamiltonian cycles with the principle of exclusion. J Discret Algorithms 6:93–102
45. Yau S (1967) Generation of all hamiltonian circuits, paths, and centers of a graph, and related problems. IEEE Trans Circuit Theory 14:79–81

Equivalence Between Priority Queues and Sorting

Rezaul A. Chowdhury
Department of Computer Sciences, University of Texas, Austin, TX, USA
Stony Brook University (SUNY), Stony Brook, NY, USA

Keywords

AC^0 operation; Pointer machine; Priority queue; Sorting; Word RAM

Synonyms

Heap

Years and Authors of Summarized Original Work

2007 (2002); Thorup

Problem Definition

A *priority queue* is an abstract data structure that maintains a set Q of elements, each with an associated value called a *key*, under the following set of operations [5, 6]:

insert(Q, x, k): Inserts element x with key k into Q.

find-min(Q): Returns an element of Q with the minimum key but does not change Q.

delete(Q, x, k): Deletes element x with key k from Q.

Additionally, the following operations are often supported:

delete-min(Q): Deletes an element with the minimum key value from Q and returns it.

decrease-key(Q, x, k): Decreases the current key k' of x to k assuming $k < k'$.

meld(Q_1 , Q_2): Given priority queues Q_1 and Q_2, returns the priority queue $Q_1 \cup Q_2$.

Observe that a *delete-min* can be implemented as a *find-min* followed by a *delete*, a *decrease-key* as a *delete* followed by an *insert*, and a *meld* as a series of *find-min*, *delete* and *insert*. However, more efficient implementations of *decrease-key* and *meld* often exist [5, 6].

Priority queues have many practical applications including event-driven simulation, job scheduling on a shared computer, and computation of shortest paths, minimum spanning forests, minimum cost matching, optimum branching, etc. [5, 6].

A priority queue can trivially be used for sorting by first inserting all keys to be sorted into the priority queue and then by repeatedly extracting the current minimum. The major contribution of Mikkel Thorup's 2002 article (Full version published in 2007) titled "Equivalence between Priority Queues and Sorting" [17] is a reduction showing that the converse is also true. Taken together, these two results imply that priority queues are computationally equivalent to sorting, that is, asymptotically, the per key cost of sorting is the update time of a priority queue.

A result similar to those in the current work [17] was presented earlier by the same author [14] which resulted in monotone priority queues (i.e., meaning that the extracted minimums are nondecreasing) with amortized time bounds only. In contrast, the current work [17] constructs general priority queues with worst-case bounds.

In addition to establishing the equivalence between priority queues and sorting, Thorup's reductions [17] are also used to translate several known sorting results into new results on priority queues.

Background

Some relevant background information is summarized below which will be useful in understanding the key results in section "Key Results."

- A standard *word RAM* models what one programs in a standard imperative programming language such as C. In addition to direct and indirect addressing and conditional jumps, there are functions, such as addition and multiplication, operating on a constant number of words. The memory is divided into words, addressed linearly starting from 0. The running time of a program is the number of instructions executed and the space is the maximal address used. The word length is a machine-dependent parameter which is big enough to hold a key and at least logarithmic in the number of input keys so that they can be addressed.
- A pointer machine is like the word RAM except that addresses cannot be manipulated.
- The AC^0 complexity class consists of constant-depth circuits with unlimited fan-in [18]. Standard AC^0 operations refer to the operations available via C but where the functions on words are in AC^0. For example, this includes addition but not multiplication.
- Integer keys will refer to nonnegative integers. However, if the input keys are signed integers, the correct ordering of the keys is obtained by flipping their sign bits and interpreting them as unsigned integers. Similar tricks work for floating point numbers and integer fractions [14].
- The atomic heaps of Fredman and Willard [7] are used in one of Thorup's reductions [17]. These heaps can support updates and searches in sets of $\mathcal{O}\left(\log^2 n\right)$ keys in $\mathcal{O}(1)$ worst-case time [20]. However, atomic heaps use multiplication operations which are not in AC^0.

Key Results

The main results in this paper are two reductions from priority queues to sorting. The stronger of the two, stated in Theorem 1, is for integer priority queues running on a standard word RAM.

Theorem 1 *If for some nondecreasing function S, up to n integer keys can be sorted in $S(n)$ time per key, an integer priority queue can be im-*

plemented supporting find-min *in constant time, and updates, i.e.,* insert *and* delete, *in* $\mathcal{O}(S(n))$ *time. Here* n *is the current number of keys in the queue. The reduction uses linear space. The reduction runs on a standard word RAM assuming that each integer key is contained in a single word.*

The reduction above provides the following new bounds for linear space integer priority queues improving previous bounds given by Han [8] and Thorup [14], respectively:

1. **(Deterministic)** $\mathcal{O}(\log\log n)$ update time using a sorting algorithm by Han [9].
2. **(Randomized)** $\mathcal{O}\left(\sqrt{\log\log n}\right)$ expected update time using a sorting algorithm given by Han and Thorup [10].

The reduction in Theorem 1 employs atomic heaps [7] which, in addition to being very complicated, use AC^0 operations. The following slightly weaker recursive reduction which does not restrict the domain of the keys is completely combinatorial.

Theorem 2 *If for some nondecreasing function* S*, up to* n *keys can be sorted in* $S(n)$ *time per key, a priority queue can be implemented supporting* find-min *in constant time, and updates in* $T(n)$ *time where* n *is the current number of keys in the queue and* $T(n)$ *satisfies the recurrence:*

$$T(n) = \mathcal{O}(S(n)) + T(\mathcal{O}(\log n))$$

The reduction runs on a pointer machine in linear space using only standard AC^0 *operations.*

This reduction implies the following new integer priority queue bounds not implied by Theorem 1, which improve previous bounds given by Thorup in 1998 [13] and 1997 [15], respectively:

1. **(Deterministic in AC^0)** $\mathcal{O}\left((\log\log n)^{1+\epsilon}\right)$ update time for any constant $\epsilon > 0$ using a standard AC^0 sorting algorithm given by Han and Thorup [10].
2. **(Randomized in AC^0)** $\mathcal{O}(\log\log n)$ expected update time using a randomized AC^0 sorting algorithm given by Thorup [15].

The Reduction in Theorem 1

Given a sorting routine that can sort up to n keys in $S(n)$ time per key, the priority queue is constructed as follows. All keys are assumed to be distinct.

The data structure has two major components: a partially sorted list of keys called a *base list* and a set of *level buffers* (also called *update buffers*). Most keys of the priority queue reside in the base list partitioned into logarithmic-sized disjoint sets called *base sets*. While the keys inside any given base set are not required to be sorted, each of those keys must be larger than every key in the base set (if any) appearing before it in the list. Keys inside each base set are stored in a doubly linked list allowing constant time updates. The first base set in the list containing the smallest key among all base sets is also maintained in an atomic heap so that the current minimum can be found in constant time. Each level buffer has a different capacity and accumulates updates (*insert/delete*) with key values in a different range. Smaller level buffers accept updates with smaller keys. An atomic heap is used to determine in constant time which level buffer collects a new update. When a level buffer accumulates enough updates, they first enter a sorting phase and then a merging phase. In the merging phase each update is applied on the proper base set in the key list, and invariants on base set size and ranges of level buffers are fixed. These phases are not executed immediately, instead they are executed in fixed time increments over a period of time. A level buffer continues to accept new updates, while some updates accepted by it earlier are still in the sorting phase, and some even older updates are in the merging phase. Every time it accepts a new update, $\mathcal{O}(S(n))$ time is spent on the sorting phase associated with it and $\mathcal{O}(1)$ time on its merging phase including rebalancing of base sets and scanning. This strategy allows the sorting and merging phases to complete execution by the time the level buffer becomes full again and thus keeping the movement of updates through different phases smooth while maintaining an $\mathcal{O}(S(n))$ worst-case time bound per update. Moreover, the size and capacity constraints ensure that the smallest key in the data structure is available in $\mathcal{O}(1)$ time. More details are given below.

The Base List: The base list consists of base sets $A_1, A_2, \ldots, A_k$, where $\frac{\Phi}{4} \le |A_i| \le \Phi$ for $i < k$, and $|A_k| \le \Phi$ for some $\Phi = \Theta(\log n)$. The exact value of Φ is chosen carefully to make sure that it conforms with the requirements of the delicate worst-case base set rebalancing protocol used by the reduction. The base sets are partitioned by *base splitters* $s_0, s_1, \ldots, s_{k+1}$, where $s_0 = -\infty$, $s_{k+1} = \infty$, and for $i = 1, \ldots, k-1$, $\max A_{i-1} < s_i \le \min A_i$. If a base set becomes too large or too small, it is split or joined with an adjacent set, respectively.

Level Buffers: Among the base splitters $l + 2 = \Theta(\log n)$ are chosen to become *level splitters* $t_0, t_1, \ldots, t_l, t_{l+1}$ with $t_0 = s_0 = -\infty$ and $t_{l+1} = s_{k+1} = \infty$, so that for $j > 0$, the number of keys in the base list below t_j is around $4^{j+1}\Phi$. These splitters are placed in an atomic heap. As the base list changes the level splitters are moved, as needed, in order to maintain their exponential distribution.

Associated with each level splitter t_j, $1 \le j \le l$, is a *level buffer* B_j containing keys in $[t_{j-1}, t_{j+2})$, where $t_{l+2} = \infty$. Buffer B_j consists of an *entrance* buffer, a *sorter*, and a *merger*, each with capacity for 4^j keys. Level j works in a cycle of 4^j steps. The cycle starts with an empty entrance, at most 4^j updates in the sorter, and a sorted list of at most 4^j updates in the merger. In each step one may accept an update for the entrance, spend $S\left(4^j\right) = \mathcal{O}(S(n))$ time in the sorter and $\mathcal{O}(1)$ time in merging the sorted list in the merger with the $\mathcal{O}\left(4^j\right)$ base splitters in $[t_{j-1}, t_{j+2})$ and scanning for a new t_j among them. Therefore, after 4^j such steps, the sorted list is correctly merged with the base list, a new t_j is found, and a new sorted list is produced. The sorter then takes the role of the merger, the entrance becomes the sorter, and the empty merger becomes the new entrance.

Handling Updates: When a new update key k (*insert*/*delete*) is received, the atomic heap of level splitters is used to find in $\mathcal{O}(1)$ time the t_j such that $k \in [t_{j-1}, t_j)$. If $k \in [t_0, t_1)$, its position is identified among the $\mathcal{O}(1)$ base splitters below t_1, and the corresponding base set is updated in $\mathcal{O}(1)$ time using the doubly linked list and the atomic heap (if exists) over the keys of that set. If $k \in [t_{j-1}, t_j)$ for some $j > 1$, the update is placed in the entrance of B_j, performing one step of the cycle of B_j in $\mathcal{O}(S(n))$ time. Additionally, during each update another splitter t_r is chosen in a round-robin fashion, and a step of a cycle of level r is executed in $\mathcal{O}(S(n))$ time. This additional work ensures that after every l updates some progress is made on moving each level splitter.

A *find-min* returns the minimum element of the base list which is available in $\mathcal{O}(1)$ time.

The Reduction in Theorem 2

This reduction follows from the previous reduction by replacing the atomic heap containing the level splitters with a data structure similar to a level buffer and the atomic heap over the keys of the first base set with a recursively defined priority queue satisfying the following recurrence for update time: $T(n) = \mathcal{O}(S(n)) + T(\mathcal{O}(\Phi))$.

Further Improvement

Alstrup et al. [1] presented a general reduction that transforms a priority queue to support *insert* in $\mathcal{O}(1)$ time while keeping the other bounds unchanged. This reduction can be used to reduce the cost of insertion to a constant in Theorems 1 and 2.

Applications

Thorup's equivalence results [17] can be used to translate known sorting results into new results on priority queues for integers and strings in different computational models (see section "Key Results"). These results can also be viewed as a new means of proving lower bounds for sorting via priority queues.

A new RAM priority queue that matches the bounds in Theorem 1 and also supports *decrease-key* in $\mathcal{O}(1)$ time is presented by Thorup [16]. This construction combines Andersson's exponential search trees [2] with the priority queues implied by Theorem 1. The reduction in Theorem 1 is also used by Pagh et al. [12] in order to develop an adaptive integer sorting algorithm for the word RAM and by Arge and Thorup [3] to develop a sorting algorithm that is simultaneously I/O efficient and internal memory efficient in the RAM model of computation. Cohen et al. [4] use a priority queue generated through this reduction to obtain a simple and fast amortized implementation of a reservoir sampling scheme that provides variance optimal unbiased estimation of subset sums. Reductions from meldable priority queues to sorting presented by Mendelson et al. [11] use the reductions from non-meldable priority queues to sorting given in [17].

An external-memory version of Theorem 1 has been proved by Wei and Yi [19].

Open Problems

One major open problem is to find a general reduction (if one exists) that allows us to decrease the value of a key in constant time. Another open question is whether the gap between the bounds implied by Theorems 1 and 2 can be reduced or removed. For example, for a hypothetical linear time-sorting algorithm, Theorem 1 implies a priority queue with an update time of $\mathcal{O}(1)$, while Theorem 2 implies only $\mathcal{O}(\log^* n)$-time updates.

Cross-References

- ▶ Cache-Oblivious Sorting
- ▶ External Sorting and Permuting
- ▶ Minimum Spanning Trees
- ▶ Single-Source Shortest Paths
- ▶ String Sorting
- ▶ Suffix Tree Construction

Recommended Reading

1. Alstrup S, Husfeldt T, Rauhe T, Thorup M (2005) Black box for constant-time insertion in priority queues (note). ACM TALG 1(1):102–106
2. Andersson A (1996) Faster deterministic sorting and searching in linear space. In: Proceedings of the 37th FOCS, Burlington, pp 135–141
3. Arge L, Thorup M (2013) RAM-efficient external memory sorting. In: Proceedings of the 24th ISAAC, Hong Kong, pp 491–501
4. Cohen E, Duffield N, Kaplan H, Lund C, Thorup M (2009) Stream sampling for variance-optimal estimation of subset sums. In: Proceedings of the 20th SODA, New York, pp 1255–1264
5. Cormen T, Leiserson C, Rivest R, Stein C (2009) Introduction to algorithms. MIT, Cambridge
6. Fredman M, Tarjan R (1987) Fibonacci heaps and their uses in improved network optimization algorithms. J ACM 34(3):596–615
7. Fredman M, Willard D (1994) Trans-dichotomous algorithms for minimum spanning trees and shortest paths. J Comput Syst Sci 48:533–551
8. Han Y (2001) Improved fast integer sorting in linear space. Inf Comput 170(8):81–94. Announced at STACS'00 and SODA'01
9. Han Y (2004) Deterministic sorting in $O(n \log\log n)$ time and linear space. J Algorithms 50(1):96–105. Announced at STOC'02
10. Han Y, Thorup M (2002) Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In: Proceedings of the 43rd FOCS, Vancouver, pp 135–144
11. Mendelson R, Tarjan R, Thorup M, Zwick U (2006) Melding priority queues. ACM TALG 2(4):535–556. Announced at SODA'04
12. Pagh A, Pagh R, Thorup M (2004) On adaptive integer sorting. In: Proceedings of the 12th ESA, Bergen, pp 556–579
13. Thorup M (1998) Faster deterministic sorting and priority queues in linear space. In: Proceedings of the 9th SODA, San Francisco, pp 550–555
14. Thorup M (2000) On RAM priority queues. SIAM J Comput 30(1):86–109. Announced at SODA'96
15. Thorup M (2002) Randomized sorting in $O(n \log\log n)$ time and linear space using addition, shift, and bit-wise boolean operations. J Algorithms 42(2):205–230. Announced at SODA'97
16. Thorup M (2004) Integer priority queues with decrease key in constant time and the single source shortest paths problem. J Comput Syst Sci (special issue on STOC'03) 69(3):330–353
17. Thorup M (2007) Equivalence between priority queues and sorting. J ACM 54(6):28. Announced at FOCS'02
18. Vollmer H (1999) Introduction to circuit complexity: a uniform approach. Springer, Berlin/New York

19. Wei Z, Yi K (2014) Equivalence between priority queues and sorting in external memory. In: Proceedings of 22nd ESA, Wroclaw, pp 830–841
20. Willard D (2000) Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. SIAM J Comput 29(3):1030–1049. Announced at SODA'92

Estimating Simple Graph Parameters in Sublinear Time

Oded Goldreich[1] and Dana Ron[2]
[1]Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel
[2]School of Electrical Engineering, Tel-Aviv University, Ramat-Aviv, Israel

Keywords

Graph parameters; Sublinear-time algorithms

Years and Authors of Summarized Original Work

2008; Goldreich, Ron

Problem Definition

A *graph parameter* σ is a real-valued function over graphs that is invariant under graph isomorphism. For example, the average degree of the graph, the average distance between pairs of vertices, and the minimum size of a vertex cover are graph parameters. For a fixed graph parameter σ and a graph $G = (V, E)$, we would like to compute an estimate of $\sigma(G)$. To this end we are given query access to G and would like to perform this task in time that is *sublinear* in the size of the graph and with high success probability. In particular, this means that we do not read the entire graph but rather only access (random) parts of it (via the query mechanism). Our main focus here is on a very basic graph parameter: its average degree, denoted $\overline{d}(G)$.

The estimation algorithm is given an approximation parameter $\epsilon > 0$. It should output a value $\hat{d}$ such that with probability at least $2/3$ (over the random choices of the algorithm) it holds that $\overline{d}(G) \leq \hat{d} \leq (1+\epsilon)\cdot\overline{d}(G)$. (The error probability can be decreased to 2^{-k} by invoking the algorithm $\Theta(k)$ times and outputting the median value.) For any vertex $v \in V = [n]$ of its choice, where $[n] \overset{\text{def}}{=} \{1, \ldots, n\}$, the estimation algorithm may query the degree of v, denoted $d(v)$. We refer to such queries as *degree queries*. In addition, the algorithm may ask for the ith neighbor of v, for any $1 \leq i \leq d(v)$. These queries are referred to as *neighbor queries*. We assume for simplicity that G does not contain any isolated vertices (so that, in particular, $\overline{d}(G) \geq 1$). This assumption can be removed.

Key Results

The problem of estimating the average degree of a graph in sublinear time was first studied by Feige [7]. He considered this problem when the algorithm is allowed only degree queries, so that the problem is a special case of estimating the average value of a function given query access to the function. For a general function $d : [n] \to [n-1]$, obtaining a constant-factor estimate of the average value of the function (with constant success probability) requires $\Omega(n)$ queries to the function. Feige showed that when d is the degree function of a graph, for any $\gamma \in (0, 1]$, it is possible to obtain an estimate of the average degree that is within a factor of $(2+\gamma)$ by performing only $O(\sqrt{n}/\gamma)$ (uniformly selected) queries. He also showed that in order to go below a factor of 2 in the quality of the estimate, $\Omega(n)$ queries are necessary.

However, given that the object in question is a graph, it is natural to allow the algorithm to query the neighborhood of vertices of its choice and not only their degrees; indeed, the aforementioned problem definition follows this natural convention. Goldreich and Ron [10] showed that by giving the algorithm this extra power, it is possible to break the factor-2 barrier. They provide an algorithm that, given $\epsilon > 0$, outputs a $(1+\epsilon)$-factor estimate of the average degree (with probability at least $2/3$) after performing $O(\sqrt{n}\,\cdot$

poly$(\log n, 1/\epsilon)$) degree and neighbor queries. In fact, since a degree query to vertex v can be replaced by $O(\log d(v)) = O(\log n)$ neighbor queries, which implement a binary search, degree queries are not necessary. Furthermore, when the average degree increases, the performance of the algorithm improves, as stated next.

Theorem 1 *There exists an algorithm that makes only neighbor queries to the input graph and satisfies the following condition. On input $G = (V, E)$ and $\epsilon \in (0, 1)$, with probability at least 2/3, the algorithm halts within $O\left(\sqrt{n/\overline{d}(G)} \cdot \text{poly}(\log n, 1/\epsilon)\right)$ steps and outputs a value in $[\overline{d}(G), (1+\epsilon) \cdot \overline{d}(G)]$.*

The running time stated in Theorem 1 is essentially optimal in the sense that (as shown in [10]) a $(1+\epsilon)$-factor estimate requires $\Omega(\sqrt{n/(\epsilon\overline{d}(G))})$ queries, for every value of n, for $\overline{d}(G) \in [2, o(n)]$, and for $\epsilon \in [\omega(n^{-1/4}), o(n/\overline{d}(G))]$.

The following is a high-level description of the algorithm and the ideas behind its analysis. For the sake of simplicity, we only show how to obtain a $(1+\epsilon)$-factor estimate by performing $O\left(\sqrt{n} \cdot \text{poly}(\log n, 1/\epsilon)\right)$ queries (under the assumption that $\overline{d}(G) \geq 1$). For the sake of the presentation, we also allow the algorithm to perform degree queries. We assume that $\epsilon \leq 1/2$, or else we run the algorithm with $\epsilon = 1/2$. We first show how to obtain a $(2+\epsilon)$-approximation by performing only degree queries and then explain how to improve the approximation by using neighbor queries as well.

Consider a partition of the graph vertices into *buckets* $B_1, \ldots, B_r$, where

$$B_i \overset{\text{def}}{=} \{v : (1+\epsilon/8)^{i-1} \leq d(v) < (1+\epsilon/8)^i\} \tag{1}$$

and $r = O(\log n/\epsilon)$. By this definition,

$$\frac{1}{n}\sum_{i=1}^{r} |B_i|\cdot(1+\epsilon/8)^i \in \left[\overline{d}(G), (1+\epsilon/8) \cdot \overline{d}(G)\right]. \tag{2}$$

Suppose we could obtain an estimate $\hat{b}_i$ of the size of each bucket B_i such that $\hat{b}_i \in [(1-\epsilon/8)|B_i|, (1+\epsilon/8)|B_i|]$. Then

$$\frac{1}{n}\sum_{i=1}^{r} \hat{b}_i \cdot (1+\epsilon/8)^i \in \left[(1-\epsilon/8) \cdot \overline{d}(G), (1+3\epsilon/8) \cdot \overline{d}(G)\right]. \tag{3}$$

Now, for each B_i, if we uniformly at random select $\Omega\left(\frac{n}{|B_i|} \cdot \frac{\log r}{\epsilon^2}\right)$ vertices, then, by a multiplicative Chernoff bound, with probability $1 - O(1/r)$, the fraction of sampled vertices that belong to B_i is in the interval $\left[(1-\epsilon/8)\frac{|B_i|}{n}, (1+\epsilon/8)\frac{|B_i|}{n}\right]$. By querying the degree of each sampled vertex, we can determine to which bucket it belongs and obtain an estimate of $|B_i|$. Unfortunately, if B_i is much smaller than $\sqrt{n}$, then the sample size required to estimate $|B_i|$ is much larger than the desired $O\left(\sqrt{n} \cdot \text{poly}(\log n, 1/\epsilon)\right)$. Let $L \overset{\text{def}}{=} \{i : |B_i| \geq \sqrt{\epsilon n/8r}\}$ denote the set of indices of *large* buckets. The basic observation is that if, for each $i \in L$, we have an estimate $\hat{b}_i \in [(1-\epsilon/8)|B_i|, (1+\epsilon/8)|B_i|]$, then

$$\frac{1}{n}\sum_{i \in L} \hat{b}_i \cdot (1+\epsilon/8)^i \in \left[(1/2-\epsilon/4) \cdot \overline{d}(G), (1+3\epsilon/8) \cdot \overline{d}(G)\right]. \tag{4}$$

The reasoning is essentially as follows. Recall that $\sum_v d(v) = 2|E|$. Consider an edge (u, v) where $u \in B_j$ and $v \in B_k$. If $j, k \in L$, then this edge contributes twice to the sum in Eq. (4): once when $i = j$ and once when $i = k$. If $j \in L$ and $k \notin L$ (or vice verse), then this edge contributes only once. Finally, if $j, k \notin L$, then the edge does not contribute at all, but there are at most $\epsilon n/8$ edges of this latter type. Since it is possible to obtain such estimates $\hat{b}_i$ for all $i \in L$ simultaneously, with constant success probability, by sampling $O\left((\sqrt{n} \cdot \text{poly}(\log n, 1/\epsilon)\right)$ vertices, we can get a

$(2+\epsilon)$-factor estimate by performing this number of degree queries. Recall that we cannot obtain an approximation factor below 2 by performing $o(n)$ queries if we use only degree queries.

In order to obtain the desired factor of $(1+\epsilon)$, we estimate the number of edges (u,v) such that $u \in B_j$ and $v \in B_k$ with $j \in L$ and $k \notin L$, which are counted only once in Eq. (4). Here is where neighbor queries come into play. For each $i \in L$ (more precisely, for each i such that $\hat{b}_i$ is sufficiently large), we estimate $e_i \stackrel{\text{def}}{=} |\{(u,v) : u \in B_i,\ v \in B_k \text{ for } k \notin L\}|$. This is done by uniformly sampling *neighbors* of vertices in B_i, querying their degree, and therefore estimating the fraction of edges incident to vertices in B_i whose other endpoint belongs to B_k for $k \notin L$. If we denote the estimate of e_i by $\hat{e}_i$, then we can get that by performing $O\left((\sqrt{n} \cdot \text{poly}(\log n, 1/\epsilon)\right)$ neighbor queries, with high constant probability, the $\hat{e}_i$'s are such that

$$\frac{1}{n}\sum_{i\in L}\left(\hat{b}_i \cdot (1+\epsilon/8)^i + \hat{e}_i\right) \in \left[(1-\epsilon/2)\cdot \overline{d}(G), (1+\epsilon/2)\cdot \overline{d}(G)\right]. \tag{5}$$

By dividing the left-hand side in Eq. (5) by $(1-\epsilon/2)$, we obtain the $(1+\epsilon)$-factor we sought.

Estimating the Average Distance

Another graph parameter considered in [10] is the average distance between vertices. For this parameter, the algorithm is given access to a *distance-query oracle*. Namely, it can query the distance between any two vertices of its choice. As opposed to the average degree parameter where neighbor queries could be used to improve the quality of the estimate (and degree queries were not actually necessary), distance queries are crucial for estimating the average distance, and neighbor queries are not of much use. The main (positive) result concerning the average distance parameter is stated next.

Theorem 2 *There exists an algorithm that makes only distance queries to the input graph and satisfies the following condition. On input $G = (V,E)$ and $\epsilon \in (0,1)$, with probability at least 2/3, the algorithm halts within $O\left(\sqrt{n/\overline{D}(G)} \cdot \text{poly}(1/\epsilon)\right)$ steps and outputs a value in $[\overline{D}(G), (1+\epsilon)\cdot\overline{D}(G)]$, where $\overline{D}(G)$ is the average of the all-pairs distances in G. A corresponding algorithm exists for the average distance to a given vertex $s \in V$.*

Comments for the Recommended Reading

The current entry falls within the scope of sublinear-time algorithms (see, e.g., [4]).

Other graph parameters that have been studied in the context of sublinear-time algorithms include the minimum weight of a spanning tree [2, 3, 5], the number of stars [11] and the number of triangles [6], the minimum size of a vertex cover [13–15, 17], the size of a maximum matching [14, 17], and the distance to having various properties [8, 13, 16]. Related problems over weighted graphs that represent distance metrics were studied in [12] and [1].

Recommended Reading

1. Bădoiu M, Czumaj A, Indyk P, Sohler C (2005) Facility location in sublinear time. In: Automata, languages and programming: thirty-second international colloquium (ICALP), Lisbon, pp 866–877
2. Chazelle B, Rubinfeld R, Trevisan L (2005) Approximating the minimum spanning tree weight in sublinear time. SIAM J Comput 34(6):1370–1379
3. Czumaj A, Sohler C (2009) Estimating the weight of metric minimum spanning trees in sublinear time. SIAM J Comput 39(3):904–922
4. Czumaj A, Sohler C (2010) Sublinear-time algorithms, in [9]
5. Czumaj A, Ergun F, Fortnow L, Magen A, Newman I, Rubinfeld R, Sohler C (2005) Approximating the weight of the euclidean minimum spanning tree in sublinear time. SIAM J Comput 35(1):91–109
6. Eden T, Levi A, Ron D, Seshadhri C (2015) Approximately counting triangles in sublinear time. In: Proceedings of FOCS 2015, Berkeley. (To appear) (see also arXiv:1504.00954 and arXiv:1505.01927)
7. Feige U (2006) On sums of independent random variables with unbounded variance, and estimating

the average degree in a graph. SIAM J Comput 35(4):964–984
8. Fischer E, Newman I (2007) Testing versus estimation of graph properties. SIAM J Comput 37(2):482–501
9. Goldreich O (ed) (2010) Property testing: current research and surveys. LNCS, vol 6390. Springer, Heidelberg
10. Goldreich O, Ron D (2008) Approximating average parameters of graphs. Random Struct Algorithms 32(4):473–493
11. Gonen M, Ron D, Shavitt Y (2011) Counting stars and other small subgraphs in sublinear time. SIAM J Discret Math 25(3):1365–1411
12. Indyk P (1999) Sublinear-time algorithms for metric space problems. In: Proceedings of the thirty-first annual ACM symposium on the theory of computing (STOC), Atlanta, pp 428–434
13. Marko S, Ron D (2009) Distance approximation in bounded-degree and general sparse graphs. Trans Algorithms 5(2):article number 22
14. Nguyen HN, Onak K (2008) Constant-time approximation algorithms via local improvements. In: Proceedings of the forty-ninth annual symposium on foundations of computer science (FOCS), Philadelphia, pp 327–336
15. Parnas M, Ron D (2007) Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. Theor Comput Sci 381(1–3):183–196
16. Parnas M, Ron D, Rubinfeld R (2006) Tolerant property testing and distance approximation. J Comput Syst Sci 72(6):1012–1042
17. Yoshida Y, Yamamoto M, Ito H (2009) An improved constant-time approximation algorithm for maximum matchings. In: Proceedings of the fourty-first annual ACM symposium on the theory of computing (STOC), Bethesda, pp 225–234

Euclidean Traveling Salesman Problem

Artur Czumaj
Department of Computer Science, Centre for Discrete Mathematics and Its Applications, University of Warwick, Coventry, UK

Keywords

Approximation algorithms; Euclidean graphs; PTAS; TSP

Years and Authors of Summarized Original Work

1998; Arora
1999; Mitchell

Problem Definition

This entry considers geometric optimization $\mathcal{NP}$-hard problems like the Euclidean traveling salesman problem and the Euclidean Steiner tree problem. These problems are geometric variants of standard graph optimization problems, and the restriction of the input instances to geometric or Euclidean case arises in numerous applications (see [1, 2]). The main focus of this entry is on the Euclidean traveling salesman problem.

The Euclidean Traveling Salesman Problem (TSP)

For a given set S of n points in the Euclidean space $\mathbb{R}^d$, find the minimum length path that visits each point exactly once. The cost $\delta(x, y)$ of an edge connecting a pair of points $x, y \in \mathbb{R}^d$ is equal to the Euclidean distance between points x and y, that is, $\delta(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}$, where $x = (x_1, \ldots, x_d)$ and $y = (y_1, \ldots, y_d)$. More generally, the distance could be defined using other norms, such as ℓ_p norms for any $p > 1$, $\delta(x, y) = \left(\sum_{i=1}^{p} (x_i - y_i)^p\right)^{1/p}$.

For a given set S of points in Euclidean space $\mathbb{R}^d$, for a certain integer d, $d \geq 2$, a *Euclidean graph* (network) is a graph $G = (S, E)$, where E is a set of straight-line segments connecting pairs of points in S. If all pairs of points in S are connected by edges in E, then G is called a *complete Euclidean graph on S*. The cost of the graph is equal to the sum of the costs of the edges of the graph, $\text{cost}(G) = \sum_{(x,y)\in E} \delta(x, y)$.

A *polynomial-time approximation scheme (PTAS)* is a family of algorithms $\{\mathcal{A}_\varepsilon\}$ such that, for each fixed $\varepsilon > 0$, $\mathcal{A}_\varepsilon$ runs in polynomial time in the size of the input and produces a $(1 + \varepsilon)$-approximation.

Related Work

The classical book by Lawler et al. [16] provides extensive information about the TSP. Also, the survey exposition of Bern and Eppstein [8] presents the state of the art for geometric TSP until 1995, and the survey of Arora [2] discusses the research after 1995.

Key Results

We begin with the hardness results. The TSP in general graphs is well known to be $\mathcal{NP}$-hard, and the same claim holds for the Euclidean TSP [14, 18].

Theorem 1 *The Euclidean TSP is $\mathcal{NP}$-hard.*

Perhaps rather surprisingly, it is still not known if the decision version of the problem is $\mathcal{NP}$-complete [14]. (The decision version of the Euclidean TSP: given a point set in the Euclidean space $\mathbb{R}^d$ and a number t, verify if there is a simple path of length smaller than t that visits each point exactly once.)

The approximability of TSP has been studied extensively over the last few decades. It is not hard to see that TSP is not approximable in polynomial time (unless $\mathcal{P} = \mathcal{NP}$) for arbitrary graphs with arbitrary edge costs. When the weights satisfy the triangle inequality (the so-called metric TSP), there is a polynomial-time 3/2-approximation algorithm due to Christofides [9], and it is known that no PTAS exists (unless $\mathcal{P} = \mathcal{NP}$). This result has been strengthened by Trevisan [21] to include Euclidean graphs in high dimensions (the same result holds also for any ℓ_p metric).

Theorem 2 (Trevisan [21]) *If $d \geq \log n$, then there exists a constant $\varepsilon > 0$ such that the Euclidean TSP in $\mathbb{R}^d$ is $\mathcal{NP}$-hard to approximate within a factor of $1 + \varepsilon$.*

In particular, this result implies that if $d \geq \log n$, then the Euclidean TSP in $\mathbb{R}^d$ has no PTAS unless $\mathcal{P} = \mathcal{NP}$.

The same result holds also for any ℓ_p metric. Furthermore, Theorem 2 implies that Euclidean TSP in $\mathbb{R}^{\log n}$ is *APX PB*-hard under E-reductions and *APX*-complete under AP-reductions.

It has been believed for some time that Theorem 2 might hold for smaller values of d, in particular even for $d = 2$, but this has been disproved independently by Arora [1] and Mitchell [17].

Theorem 3 (Arora [1] and Mitchell [17]) *The Euclidean TSP on the plane has a PTAS.*

The main idea of the algorithms of Arora and Mitchell is rather simple, but the details of the analysis are quite complicated. Both algorithms follow the same approach. One first proves a so-called structure theorem, which demonstrates that there is a $(1 + \varepsilon)$-approximation that has some local properties (in the case of the Euclidean TSP, there is a quadtree partition of the space containing all the points such that there is a $(1 + \varepsilon)$-approximation in which each cell of the quadtree is crossed by the tour at most a constant number of times and only in some prespecified locations). Then, one uses dynamic programming to find an optimal (or almost optimal) solution that obeys the local properties specified in the structure theorem.

The original algorithms presented in the first conference version of [1] and in the early version of [17] have the running times of the form $\mathcal{O}(n^{1/\epsilon})$ to obtain a $(1 + \varepsilon)$-approximation, but this has been subsequently improved. In particular, Arora's randomized algorithm in [1] runs in time $\mathcal{O}(n(\log n)^{1/\epsilon})$, and it can be derandomized with a slowdown of $\mathcal{O}(n)$. The result from Theorem 3 can be also extended to higher dimensions. Arora shows the following result.

Theorem 4 (Arora [1]) *For every constant d, the Euclidean TSP in $\mathbb{R}^d$ has a PTAS.*

For every fixed $c > 1$ and given any n points in $\mathbb{R}^d$, there is a randomized algorithm that finds a $\left(1 + \frac{1}{c}\right)$-approximation of the optimum traveling salesman tour in $\mathcal{O}\left(n(\log n)^{(\mathcal{O}(\sqrt{d}c))^{d-1}}\right)$ time. In particular, for any constant d and c, the running time is $\mathcal{O}\left(n(\log n)^{\mathcal{O}(1)}\right)$. The algorithm can be derandomized by increasing the running time by a factor of $\mathcal{O}(n^d)$.

This has been later extended by Rao and Smith [19], who proved the following.

Theorem 5 (Rao and Smith [19]) *There is a deterministic algorithm that computes a $\left(1+\frac{1}{c}\right)$-approximation of the optimum traveling salesman tour in $\mathcal{O}\left(2^{(cd)^{\mathcal{O}(d)}} n + (cd)^{\mathcal{O}(d)} n \log n\right)$ time.*

There is a randomized algorithm that succeeds with probability at least $\frac{1}{2}$ and that computes a $\left(1+\frac{1}{c}\right)$-approximation of the optimum traveling salesman tour in expected $\left(c\sqrt{d}\right)^{\mathcal{O}(d(c\sqrt{d})^{d-1})} n + \mathcal{O}(d\, n \log n)$ time.

These results are essentially asymptotically optimal in the decision tree model thanks to a lower bound of $\Omega(n \log n)$ for any sublinear approximation for 1-dimensional Euclidean TSP due to Das et al. [12]. In the *real RAM* model, one can further improve the randomized results.

Theorem 6 (Bartal and Gottlieb [6]) *Given a set S of n points in d-dimensional grid $\{0,\ldots,\Delta\}^d$ with $\Delta = 2^{(cd)^{\mathcal{O}(d)}} n$, there is a randomized algorithm that with probability $1 - e^{-\mathcal{O}_d(n^{1/3d})}$ computes a $\left(1+\frac{1}{c}\right)$-approximation of the optimum traveling salesman tour for S in time $2^{(cd)^{\mathcal{O}(d)}} n$ in the integer RAM model.*

If the data is not given in the integral form, then one may round the data into this form using the floor or mod functions, and assuming these functions are atomic operations, the rounding can be done in $\mathcal{O}(dn)$ total time, leading to the following theorem.

Theorem 7 (Bartal and Gottlieb [6]) *Given a set of n points in $\mathbb{R}^d$, there is a randomized algorithm that with probability $1 - e^{-\mathcal{O}_d(n^{1/3d})}$ computes a $\left(1+\frac{1}{c}\right)$-approximation of the optimum traveling salesman tour in time $2^{(cd)^{\mathcal{O}(d)}} n$ in the real RAM model with atomic floor or mod operations.*

Applications

The techniques developed by Arora [1] and Mitchell [17] found numerous applications in the design of polynomial-time approximation schemes for geometric optimization problems.

Euclidean Minimum Steiner Tree

For a given set S of n points in the Euclidean space $\mathbb{R}^d$, find the minimum-cost network connecting all the points in S (where the cost of a network is equal to the sum of the lengths of the edges defining it).

Euclidean k-median

For a given set S of n points in the Euclidean space $\mathbb{R}^d$ and an integer k, find k-medians among the points in S so that the sum of the distances from each point in S to its closest median is minimized.

Euclidean k-TSP

For a given set S of n points in the Euclidean space $\mathbb{R}^d$ and an integer k, find the shortest tour that visits at least k points in S.

Euclidean k-MST

For a given set S of n points in the Euclidean space $\mathbb{R}^d$ and an integer k, find the shortest tree that visits at least k points in S.

Euclidean Minimum-Cost k-Connected Subgraph

For a given set S of n points in the Euclidean space $\mathbb{R}^d$ and an integer k, find the minimum-cost subgraph (of the complete graph on S) that is k-connected.

Theorem 8 *For every constant d, the following problems have a PTAS:*

- *Euclidean minimum Steiner tree problem in $\mathbb{R}^d$ [1, 19]*
- *Euclidean k-median problem in $\mathbb{R}^d$ [5]*
- *Euclidean k-TSP and the Euclidean k-MST problems in $\mathbb{R}^d$ [1]*
- *Euclidean minimum-cost k-connected subgraph problem in $\mathbb{R}^d$ (constant k) [10]*

The technique developed by Arora [1] and Mitchell [17] led also to some quasi-polynomial-time approximation schemes, that is, the algorithms with the running time of $n^{\mathcal{O}(\log n)}$. For example, Arora and Karokostas [4] gave a quasi-polynomial-time approximation scheme for the Euclidean minimum latency problem, Das and Mathieu [13] gave a quasi-polynomial-time approximation scheme for the Euclidean capacitated vehicle routing problem, and Remy and Steger [20] gave a quasi-polynomial-time approximation scheme for the minimum-weight triangulation problem.

For more discussion, see the survey by Arora [2] and Czumaj and Lingas [11].

Extensions to Planar Graphs and Metric Spaces with Bounded Doubling Dimension

The dynamic programming approach used by Arora [1] and Mitchell [17] is also related to the recent advances for a number of optimization problems for planar graphs and in graphs in metric spaces with bounded doubling dimension. For example, Arora et al. [3] designed a PTAS for the TSP in weighted planar graphs (cf. [15] for a linear-time PTAS), and there is a PTAS for metric spaces with bounded doubling dimension [7].

Open Problems

An interesting open problem is if the quasi-polynomial-time approximation schemes mentioned above (for the minimum latency, the capacitated vehicle routing, and the minimum-weight triangulation problems) can be extended to obtain PTAS. For more open problems, see Arora [2].

Experimental Results

The Web page of the 8th DIMACS Implementation Challenge, http://dimacs.rutgers.edu/Challenges/TSP/, contains a lot of instances.

URLs to Code and Data Sets

The Web page of the 8th DIMACS Implementation Challenge, http://dimacs.rutgers.edu/Challenges/TSP/, contains a lot of instances.

Cross-References

- ▸ Approximation Schemes for Geometric Network Optimization Problems
- ▸ Metric TSP
- ▸ Minimum k-Connected Geometric Networks
- ▸ Minimum Weight Triangulation

Recommended Reading

1. Arora S (1998) Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. J Assoc Comput Mach 45(5):753–782
2. Arora S (2003) Approximation schemes for NP-hard geometric optimization problems: a survey. Math Program Ser B 97:43–69
3. Arora S, Grigni M, Karger D, Klein P, Woloszyn A (1998) A polynomial time approximation scheme for weighted planar graph TSP. In: Proceedings of the 9th annual ACM-SIAM symposium on discrete algorithms (SODA), San Francisco, pp 33–41
4. Arora S, Karakostas G (1999) Approximation schemes for minimum latency problems. In: Proceedings of the 31st annual ACM symposium on theory of computing (STOC), Atlanta, pp 688–693
5. Arora S, Raghavan P, Rao S (1998) Approximation schemes for Euclidean k-medians and related problems. In: Proceedings of the 30th annual ACM symposium on theory of computing (STOC), Dallas, pp 106–113
6. Bartal Y, Gottlieb LA (2013) A linear time approximation scheme for Euclidean TSP. In: Proceedings of the 54th IEEE symposium on foundations of computer science (FOCS), Berkeley, pp 698–706
7. Bartal Y, Gottlieb LA, Krauthgamer R (2012) The traveling salesman problem: low-dimensionality implies a polynomial time approximation scheme. In: Proceedings of the 44th annual ACM symposium on theory of computing (STOC), New York, pp 663–672
8. Bern M, Eppstein D (1996) Approximation algorithms for geometric problems. In: Hochbaum D (ed) Approximation algorithms for NP-hard problems. PWS Publishing, Boston
9. Christofides N (1976) Worst-case analysis of a new heuristic for the traveling salesman problem. Technical report, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh
10. Czumaj A, Lingas A (1999) On approximability of the minimum-cost k-connected spanning subgraph problem. In: Proceedings of the 10th annual ACM-SIAM symposium on discrete algorithms (SODA), Baltimore, pp 281–290
11. Czumaj A, Lingas A (2007) Approximation schemes for minimum-cost k-connectivity problems in geometric graphs. In: Gonzalez TF (ed) Handbook of

approximation algorithms and metaheuristics. CRC, Boca Raton

12. Das G, Kapoor S, Smid M (1997) On the complexity of approximating Euclidean traveling salesman tours and minimum spanning trees. Algorithmica 19(4):447–462
13. Das A, Mathieu C (2010) A quasi-polynomial time approximation scheme for Euclidean capacitated vehicle routing. In: Proceedings of the 21st annual ACM-SIAM symposium on discrete algorithms (SODA), Austin, pp 390–403
14. Garey MR, Graham RL, Johnson DS (1976) Some NP-complete geometric problems. In: Proceedings of the 8th annual ACM symposium on theory of computing (STOC), Hershey, pp 10–22
15. Klein P (2008) A linear-time approximation scheme for TSP in undirected planar graphs with edge-weights. SIAM J Comput 37(6):1926–1952
16. Lawler EL, Lenstra JK, Rinnooy Kan AHG, Shmoys DB (1985) The traveling salesman problem: a guided tour of combinatorial optimization. Wiley, Chichester/New York
17. Mitchell JSB (1999) Guillotine subdivisions approximate polygonal subdivisions: a simple polynomial-time approximation scheme for geometric TSP, k-MST, and related problems. SIAM J Comput 28(4):1298–1309
18. Papadimitriou CH (1977) Euclidean TSP is NP-complete. Theor Comput Sci 4:237–244
19. Rao SB, Smith WD (1998) Approximating geometrical graphs via "spanners" and "banyans." In: Proceedings of the 30th annual ACM symposium on theory of computing (STOC), Dallas, pp 540–550
20. Remy J, Steger A (2006) A quasi-polynomial time approximation scheme for minimum weight triangulation. In: Proceedings of the 38th annual ACM symposium on theory of computing (STOC), Seattle, pp 316–325
21. Trevisan L (2000) When Hamming meets Euclid: the approximability of geometric TSP and Steiner tree. SIAM J Comput 30(2):475–485

Exact Algorithms and Strong Exponential Time Hypothesis

Joshua R. Wang and Ryan Williams
Department of Computer Science, Stanford University, Stanford, CA, USA

Keywords

Exact algorithms; Exponential-time hypothesis; Satisfiability; Treewidth

Years and Authors of Summarized Original Work

2010; Păatraşscu, Williams
2011; Lokshtanov, Marx, Saurabh
2012; Cygan, Dell, Lokshtanov, Marx, Nederlof, Okamoto, Paturi, Saurabh, Wahlström

Problem Definition

All problems in NP can be exactly solved in $2^{\text{poly}(n)}$ time via exhaustive search, but research has yielded faster exponential-time algorithms for many NP-hard problems. However, some key problems have not seen improved algorithms, and problems with improvements seem to converge toward $O(C^n)$ for some unknown constant $C > 1$.

The satisfiability problem for Boolean formulas in conjunctive normal form, CNF-SAT, is a central problem that has resisted significant improvements. The complexity of CNF-SAT and its special case k-SAT, where each clause has k literals, is the canonical starting point for the development of NP-completeness theory.

Similarly, in the last 20 years, two hypotheses have emerged as powerful starting points for understanding exponential-time complexity. In 1999, Impagliazzo and Paturi [5] defined the *exponential-time hypothesis* (ETH), which asserts that 3-*SAT cannot be solved in subexponential time*. Namely, it asserts there is an $\epsilon > 0$ such that 3-SAT cannot be solved in $O((1+\epsilon)^n)$ time. ETH has been a surprisingly useful assumption for ruling out subexponential-time algorithms for other problems [2, 6]. A stronger hypothesis has led to more fine-grained lower bounds, which is the focus of this article. Many NP-hard problems are solvable in C^n time via exhaustive search (for some $C > 1$) but are not known to be solvable in $(C - \epsilon)^n$ time, for any $\epsilon > 0$. The *strong exponential-time hypothesis* (SETH) [1,5] asserts that for every $\epsilon > 0$, there exists a k such that k-SAT cannot be solved in time $O((2-\epsilon)^n)$. SETH has been very useful in establishing tight (and exact) lower bounds for many problems. Here we survey some of these tight results.

Key Results

The following results are reductions from k-SAT to other problems. They can be seen either as new attacks on the complexity of SAT or as lower bounds for exact algorithms that are conditional on SETH.

Lower Bounds on General Problems

The following problems have lower bounds conditional on SETH. The reduction for the first problem is given to illustrate the technique.

k-Dominating Set

A *dominating set* of a graph $G = (V, E)$ is a subset $S \subseteq V$ such that every vertex is either in S or is a neighbor of a vertex in S. The k-Dominating Set problem asks to find a dominating set of size k. Assuming SETH, for any $k \geq 3$ and $\epsilon > 0$, k-Dominating Set cannot be solved in $O(n^{k-\epsilon})$ time [8].

The reduction from SAT to k-Dominating Set proceeds as follows. Fix some $k \geq 3$ and let F be a CNF formula on n variables; we build a corresponding graph G_F. Partition its variables into k equally sized parts of n/k variables. For each part, take all $2^{n/k}$ partial assignments and make a node for each partial assignment. Make each of the k parts into a clique (disjoint from the others). Add a dummy node for each partial assignment clique that is connected to every node in that clique but has no other edges. Add m more nodes, one for each clause. Finally, make an edge from a partial assignment node to a clause node iff the partial assignment satisfies the clause. We observe that there is a k-dominating set in G_F if F is satisfiable.

2Sat+2Clauses

The 2Sat+2Clauses problem asks whether a Boolean formula is satisfiable, given that it is a 2-CNF with two additional clauses of arbitrary length. Assuming SETH, for any $m = n^{1+o(1)}$ and $\epsilon > 0$, 2Sat+2Clauses cannot be solved in $O(n^{2-\epsilon})$ time [8]. It is known that 2Sat+2Clauses can be solved in $O(mn + n^2)$ time [8].

HornSat+kClauses

The HornSat+kClauses problem asks whether a Boolean formula is satisfiable, given that it is a CNF of clauses that contain at most one nonnegative literal per clause (a Horn CNF), conjoined with k additional clauses of arbitrary length but only positive literals. Assuming SETH, for any $k \geq 2$ and $\epsilon > 0$, HornSat+kClauses cannot be solved in $O((n + m)^{k-\epsilon})$ time [8]. It can be trivially solved in $O(n^k \cdot (m + n))$ time by guessing a variable to set to true for each of the k additional clauses and checking if the remaining Horn CNF is satisfiable in linear time.

3-Party Set Disjointness

The 3-Party Set Disjointness problem is a communication problem with three parties and three subsets $S_1, S_2, S_3 \subseteq [m]$, where the ith party has access to all sets except for S_i. The parties wish to determine if $S_1 \cap \cdots \cap S_3 = \varnothing$. Clearly this can be done with $O(m)$ bits of communication. Assuming SETH, 3-Party Set Disjointness cannot be solved using protocols running in $2^{o(n)}$ time and communicating only $o(m)$ bits [8].

k-SUM

The k-Sum problem asks whether a set of n numbers contains a k-tuple that sums to zero. Assuming SETH, k-Sum on n numbers cannot be solved in $n^{o(k)}$ time for any $k < n^{0.99}$. (It is well known that k-Sum is in $O(n^{\lceil k/2 \rceil})$ time) [8].

For all the problems below, we can solve in $2^n n^O(1)$ time via exhaustive search.

k-Hitting Set

Given a set system $\mathcal{F} \subseteq 2^U$ in some universe U, a *hitting set* is a subset $H \subseteq U$ such that $H \cap S \neq \varnothing$ for every $S \in \mathcal{F}$. The k-HittingSet problem asks whether there is a hitting set of size at most t, given that each set $S \in \mathcal{F}$ has at most k elements. SETH is equivalent to the claim that for all $\epsilon > 0$, there is a k for which k-HITTINGSET cannot be solved in time $O((2-\epsilon)^n)$ [3].

k-Set Splitting

Given a set system $\mathcal{F} \subseteq 2^U$ in some universe U, a set splitting is a subset $X \subseteq U$ such that the first element of the universe is in X and for every $S \in \mathcal{F}$, neither $S \subseteq X$ nor $S \subseteq (U \setminus X)$. The k-SETSPLITTING problem asks whether there is a set splitting, given that each set $S \in \mathcal{F}$ has at most k elements. SETH is equivalent to the claim that for all $\epsilon > 0$, there is a k for which k-SETSPLITTING cannot be solved in time $O((2-\epsilon)^n)$ [3].

k-NAE-Sat

The k-NAE-SAT problem asks whether a k-CNF has an assignment where the first variable is set to true and each clause has both a true literal and a false literal. SETH is equivalent to the claim that for all $\epsilon > 0$, there is a k for which k-NAE-SAT cannot be solved in time $O((2-\epsilon)^n)$ [3].

c-VSP-Circuit-SAT

The c-VSP-CIRCUIT-SAT problem asks whether a cn-size Valiant series-parallel circuit over n variables has a satisfying assignment. SETH is equivalent to the claim that for all $\epsilon > 0$, there is a k for which c-VSP-CIRCUIT-SAT cannot be solved in time $O((2-\epsilon)^n)$ [3].

Problems Parameterized by Treewidth

A variety of NP-complete problems have been shown to be much easier on graphs of bounded treewidth. Reductions starting from SETH given by Lokshtanov, Marx, and Saurabh [7] can also prove lower bounds that depend on the treewidth of an input graph, $\mathrm{tw}(G)$. The following are proven via analyzing the pathwidth of a graph, $\mathrm{pw}(G)$, and the fact that $\mathrm{tw}(G) \leq \mathrm{pw}(G)$.

Independent Set

An *independent set* of a graph $G = (V, E)$ is a subset $S \subseteq V$ such that the subgraph induced by S contains no edges. The INDEPENDENT SET problem asks to find an independent set of maximum size. Assuming SETH, for any $\epsilon > 0$, INDEPENDENT SET cannot be solved in $(2-\epsilon)^{\mathrm{tw}(G)} n^{O(1)}$ time.

Dominating Set

A *dominating set* of a graph $G = (V, E)$ is a subset $S \subseteq V$ such that every vertex is either in S or is a neighbor of a vertex in S. The DOMINATING SET problem asks to find a dominating set of minimum size. Assuming SETH, for any $\epsilon > 0$, DOMINATING SET cannot be solved in $(3-\epsilon)^{\mathrm{tw}(G)} n^{O(1)}$ time.

Max Cut

A *cut* of a graph $G = (V, E)$ is a partition of V into S and $V \setminus S$. The size of a cut is the number of edges that have one endpoint in S and the other in $V \setminus S$. The MAX CUT problem asks to find a cut of maximum size. Assuming SETH, for any $\epsilon > 0$, MAX CUT cannot be solved in $(2-\epsilon)^{\mathrm{tw}(G)} n^{O(1)}$ time.

Odd Cycle Transversal

An *odd cycle transversal* of a graph $G = (V, E)$ is a subset $S \subseteq V$ such that the subgraph induced by $V \setminus S$ is bipartite. The ODD CYCLE TRANSVERSAL problem asks to, given an integer k, determine whether there is an odd cycle transversal of size k. Assuming SETH, for any $\epsilon > 0$, ODD CYCLE TRANSVERSAL cannot be solved in $(3-\epsilon)^{\mathrm{tw}(G)} n^{O(1)}$ time.

Graph Coloring

A *q-coloring* of a graph $G = (V, E)$ is a function $\mu : V \to [q]$. A q-coloring is *proper* if for all edges $(u, v) \in E$, $\mu(u) \neq \mu(v)$. The q-COLORING problem asks to decide whether the graph has a proper q-coloring. Assuming SETH, for any $q \geq 3$ and $\epsilon > 0$, q-COLORING cannot be solved in $(q-\epsilon)^{\mathrm{tw}(G)} n^{O(1)}$ time.

Partition Into Triangles

A graph $G = (V, E)$ can be partitioned into triangles if there is a partition of the vertices into $S_1, S_2, \ldots, S_{n/3}$ such that each S_i induces a triangle in G. The PARTITION INTO TRIANGLES problem asks to decide whether the graph can be partitioned into triangles. Assuming SETH, for any $\epsilon > 0$, PARTITION INTO TRIANGLES cannot be solved in $(2-\epsilon)^{\mathrm{tw}(G)} n^{O(1)}$ time.

All of the above results are tight, in the sense that when $\epsilon = 0$, there is an algorithm for each of them.

Showing Difficulty Via Set Cover

Given a set system $\mathcal{F} \subseteq 2^U$ in some universe U, a set cover is a subset $\mathcal{C} \subseteq \mathcal{F}$ such that $\bigcup_{S\in\mathcal{C}} S = U$. The SET COVER problem asks whether there is a set cover of size at most t.

Cygan et al. [3] also gave reductions from SET COVER to several other problems, showing lower bounds conditional on the assumption that for all $\epsilon > 0$, there is a k such that SET COVER where sets in $\mathcal{F}$ have size at most k cannot be computed in time $O^*((2-\epsilon)^n)$.

It is currently unknown how SET COVER is related to SETH; if there is a reduction from CNF-SAT to SET COVER, then all of these problems would have conditional lower bounds as well.

Steiner Tree

Given a graph $G = (V, E)$ and a set of terminals $T \subseteq V$, a *Steiner Tree* is a subset $X \subseteq V$ such that the graph induced by X is connected and $T \subseteq X$. The STEINER TREE problem asks whether G has a Steiner tree of size at most t. With the above SET COVER assumption, for all $\epsilon > 0$, STEINER TREE cannot be solved in $O^*((2-\epsilon)^t)$ time.

Connected Vertex Cover

A *connected vertex cover* of a graph $G = (V, E)$ is a subset $X \subseteq V$ such that the subgraph induced by X is connected and every edge contains at least one endpoint in X. The CONNECTED VERTEX COVER problem asks whether G has a connected vertex cover of size at most t. With the above SET COVER assumption, for all $\epsilon > 0$, CONNECTED VERTEX COVER cannot be solved in $O^*((2-\epsilon)^t)$ time.

Set Partitioning

Given a set system $\mathcal{F} \subseteq 2^U$ in some universe U, a set partitioning is a set cover $\mathcal{C}$ where pairwise disjoint elements have an empty intersection. The SET PARTITIONING problem asks whether there is a set partitioning of size at most t. With the above SET COVER assumption, for all $\epsilon > 0$, SET PARTITIONING cannot be solved in $O^*((2-\epsilon)^n)$ time.

Subset Sum

The SUBSET SUM problem asks whether a set of n positive numbers contains a subset that sums to a target t. With the above SET COVER assumption, for all $\delta < 1$, SUBSET SUM cannot be solved in $O^*(t^\delta)$ time. Note that there is a dynamic programming solution that runs in $O(nt)$ time.

Open Problems

- Does ETH imply SETH?
- Does SETH imply SET COVER requires $O^*((2-\epsilon)^n)$ time for all $\epsilon > 0$?
- Does SETH imply that the Traveling Salesman Problem in its most general, weighted form requires $O^*((2-\epsilon)^n)$ time for all $\epsilon > 0$?
- Given two graphs F and G, on k and n nodes, respectively, the SUBGRAPH ISOMORPHISM problem asks whether a (noninduced) subgraph of G is isomorphic to F. Does SETH imply that SUBGRAPH ISOMORPHISM cannot be solved in $2^{O(n)}$?

Cross-References

▸ Backtracking Based k-SAT Algorithms
▸ Exact Algorithms for General CNF SAT
▸ Exact Algorithms for Treewidth

Recommended Reading

1. Calabro C, Impagliazzo R, Paturi R (2009) The complexity of satisfiability of small depth circuits. In: Chen J, Fomin F (eds) Parameterized and exact computation. Lecture notes in computer science, vol 5917. Springer, Berlin/Heidelberg, pp 75–85. doi:10.1007/978-3-642-11269-0_6. http://dx.doi.org/10.1007/978-3-642-11269-0_6
2. Chen J, Chor B, Fellows M, Huang X, Juedes D, Kanj IA, Xia G (2005) Tight lower bounds for certain parameterized np-hard problems. Inf Comput 201(2):216–231. doi:http://dx.doi.org/10.1016/j.ic.

2005.05.001. http://www.sciencedirect.com/science/article/pii/S0890540105000763
3. Cygan M, Dell H, Lokshtanov D, Marx D, Nederlof J, Okamoto Y, Paturi R, Saurabh S, Wahlstrom M (2012) On problems as hard as cnf-sat. In: Proceedings of the 2012 IEEE conference on computational complexity (CCC '12), Washington, DC. IEEE Computer Society, pp 74–84. doi:10.1109/CCC.2012.36. http://dx.doi.org/10.1109/CCC.2012.36
4. Fomin F, Kratsch D (2010) Exact exponential algorithms. Texts in theoretical computer science, an EATCS series. Springer, Berlin/Heidelberg
5. Impagliazzo R, Paturi R (2001) On the complexity of k-sat. J Comput Syst Sci 62(2):367–375. doi:http://dx.doi.org/10.1006/jcss.2000.1727. http://www.sciencedirect.com/science/article/pii/S0022000000917276
6. Impagliazzo R, Paturi R, Zane F (2001) Which problems have strongly exponential complexity? J Comput Syst Sci 63(4):512–530. doi:http://dx.doi.org/10.1006/jcss.2001.1774. http://www.sciencedirect.com/science/article/pii/S002200000191774X
7. Lokshtanov D, Marx D, Saurabh S (2011) Known algorithms on graphs of bounded treewidth are probably optimal. In: Proceedings of the twenty-second Annual ACM-SIAM symposium on discrete algorithms (SODA '11), San Francisco. SIAM, pp 777–789. http://dl.acm.org/citation.cfm?id=2133036.2133097
8. Pătraşcu M, Williams R (2010) On the possibility of faster sat algorithms. In: Proceedings of the twenty-first annual ACM-SIAM symposium on discrete algorithms (SODA '10), Philadelphia. Society for Industrial and Applied Mathematics, pp 1065–1075. http://dl.acm.org/citation.cfm?id=1873601.1873687
9. Woeginger G (2003) Exact algorithms for np-hard problems: a survey. In: Jünger M, Reinelt G, Rinaldi G (eds) Combinatorial optimization Eureka, you shrink! Lecture notes in computer science, vol 2570. Springer, Berlin/Heidelberg, pp 185–207. doi:10.1007/3-540-36478-1_17. http://dx.doi.org/10.1007/3-540-36478-1_17

Exact Algorithms and Time/Space Tradeoffs

Jesper Nederlof
Technical University of Eindhoven, Eindhoven, The Netherlands

Keywords

Dynamic programming; Knapsack; Space efficiency; Steiner tree; Subset discrete Fourier transform

Years and Authors of Summarized Original Work

2010; Lokshtanov, Nederlof

Problem Definition

In the *subset sum problem*, we are given integers $a_1, \dots, a_n, t$ and are asked to find a subset $X \subseteq \{1, \dots, n\}$ such that $\sum_{i \in X} a_i = t$. In the *Knapsack problem*, we are given $a_1, \dots, a_n, b_1, \dots, b_n, t, u$ and are asked to find a subset $X \subseteq \{1, \dots, n\}$ such that $\sum_{i \in X} a_i \leq t$ and $\sum_{i \in X} b_i \geq u$. It is well known that both problems can be solved in $O(nt)$ time using dynamic programming. However, as is typical for dynamic programming, these algorithms require a lot of working memory and are relatively hard to execute in parallel on several processors: the above algorithms use $O(t)$ space which may be *exponential* in the input size.

This raises the question: when can we avoid these disadvantages and still be (approximately) as fast as dynamic programming algorithms? It appears that by (slightly) loosening the time budget, space usage and parallelization can be significantly improved in many dynamic programs.

Key Results

A Space Efficient Algorithm for Subset Sum

In this article, we will use $\tilde{O}(\cdot)$ to suppress factors that are poly-logarithmic in the input size. In what follows, we will discuss how to prove the following theorem:

Theorem 1 (Lokshtanov and Nederlof, [7]) *There is an algorithm counting the number of solutions of a subset sum instance in* $\tilde{O}(n^2 t(n + \log t))$ *time and* $(n + lg(t))(\lg nt)$ *space.*

The Discrete Fourier Transform

We use Iverson's bracket notation: given a Boolean predicate b, $[b]$ denotes 1 if b is true and 0 otherwise. Let $P(x)$ be a polynomial of degree $N-1$, and let $p_0, \dots, p_{N-1}$ be its coefficients. Thus, $P(x) = \sum_{i=0}^{N-1} p_i x^i$. Let ω_N denote the N'th root of unity, that is, $\omega_N = e^{\frac{2\pi\iota}{N}}$. Let k, t be integers such that $k \neq t$. By the summation formula for geometric progressions $\left(\sum_{\ell=0}^{N-1} r^\ell = \frac{1-r^N}{1-r} \text{for } r \neq 1\right)$, we have:

$$\sum_{\ell=0}^{N-1} \omega_N^{\ell(k-t)} = \frac{1-\omega_N^{(k-t)N}}{1-\omega_N^{k-t}} = \frac{1-\left(\omega_N^N\right)^{k-t}}{1-\omega_N^{k-t}}$$
$$= \frac{1-(1)^{k-t}}{1-\omega_N^{k-t}} = 0.$$

On the other hand, if $k = t$, then $\sum_{\ell=0}^{N-1} \omega_N^{\ell(k-t)} = \sum_{\ell=0}^{N-1} 1 = N$. Thus, both cases can be compactly summarized as $\sum_{\ell=0}^{N-1} \omega_N^{\ell(k-t)} = [k = t]N$. As a consequence, we can express a coefficient p_t of $P(x)$ directly in terms of its evaluations:

$$\begin{aligned} p_t &= \sum_{k=0}^{N-1} [k=t] p_k \\ &= \sum_{k=0}^{N-1} \frac{1}{N} \sum_{\ell=0}^{N-1} \omega_N^{\ell(k-t)} \sum p_k \\ &= \frac{1}{N} \sum_{\ell=0}^{N-1} \omega_N^{-\ell t} \sum_{k=0}^{N-1} p_k \left(\omega_N^\ell\right)^k \\ &= \frac{1}{N} \sum_{\ell=0}^{N-1} \omega_N^{-\ell t} P(\omega_N^\ell) \end{aligned} \tag{1}$$

Using the Discrete Fourier Transform for Subset Sum

Given an instance $a_1, \dots, a_n, t$ of subset sum, define the polynomial $P(x)$ to be $P(x) = \prod_{i=1}^n (1 + x^{a_i})$. Clearly, we can discard integers a_i larger than t, and assume that $P(x)$ has degree at most $N = nt$. If we expand the products in this polynomial to get rid of the parentheses, we get a sum of 2^n products and each of these products is of the type x^k and corresponds to a subset $X \subseteq \{1, \dots, n\}$ such that $\sum_{i \in X} a_i = k$. Thus, if we aggregate these products, we obtained the normal form $P(x) = \sum_{k=0}^{N-1} p_k x^k$, where p_k equals the number of subsets $X \subseteq [n]$ such that $\sum_{i \in X} a_i = k$. Plugging this into Eq. 1, we have that the number of subset sum solutions equals

$$p_t = \frac{1}{N} \sum_{\ell=0}^{N-1} \omega_N^{-\ell t} \prod_{i=1}^{n} \left(1 + \omega_N^{\ell a_i}\right). \tag{2}$$

Given Eq. 2, the algorithm suggests itself: evaluation of the right-hand side gives the number of solutions of the subset sum instance. Given ω_N, this would be a straightforward on the unit-cost RAM model (recall that in this model arithmetic instructions as $+, -, *$ and $/$ are assumed to take constant time): the required powering operations are performed in $\log(N)$ arithmetic operations so an overall upper bound would be $O(n^2 t \log(nt))$ time.

However, still the value of this algorithm is not clear yet: for example, ω_N may be irrational, so it is not clear how to perform the arithmetic efficiently. This is an issue that also arises for the folklore fast Fourier transform (see, e.g., [3] for a nice exposition), and this issue is usually not addressed (a nice exception is Knuth [6]). Moreover in our case we should also be careful on the space budget: for example, we cannot even store 2^t in the usual way within our space budget. But, as we will now see, it turns out that we can simply evaluate Eq. 2 with finite precision and round to the nearest integer within the resource bounds claimed in Theorem 1.

Evaluating Equation 2 with Finite Precision

The algorithm establishing Theorem 1 is presented in Algorithm 1. Here, ρ represents the amount of precision the algorithm works with. The procedure $\mathtt{tr}_\rho$ truncates ρ bits after the decimal point. The procedure $\mathtt{apxr}_\rho(z)$ returns an estimate of ω_N^z. In order to do this, estimates of ω_N^z with z being powers of 2 are precomputed in Lines 3–4. We omit an explicit implementation of the right-hand side of Line 4 since this is very standard; for example, one can use an approximation of π together with a binary splitting approach

Algorithm 1 Approximate evaluation of Eq. 2

```
Algorithm: SSS($a_1, \ldots, a_n, t$)
Require: for every $1 \le i \le n, a_i < t$.
 1: $\rho \leftarrow 3n + 6 \log nt$
 2: $s \leftarrow 0$
 3: for $0 \le q \le \log N - 1$ do
 4:     //store roots of unity for powers of two
 5:     $r_q \leftarrow \mathtt{tr}_\rho(e^{\frac{2\pi 2^q}{N\iota}})$
 6: end for
 7: for $0 \le \ell \le N - 1$ do
 8:     $p \leftarrow \mathtt{apxr}_\rho(-\ell t \ \% \ N)$
 9:     for $1 \le i \le n$ do
10:         $p \leftarrow \mathtt{tr}_\rho(p * (1 + \mathtt{apxr}_\rho(\ell a_i \ \% \ N)))$
11:     end for
12:     $s \leftarrow s + p$
13: end for
14: return rnd($\mathtt{tr}_\rho(\frac{s}{N})$) //round to nearest int

Algorithm: $\mathtt{apxr}_\rho(z)$
Require: $z < N$
15: $p' \leftarrow 1$
16: for $1 \le q \le \log N - 1$ do
17:     if $2^q$ divides $z$ then
18:         $p \leftarrow \mathtt{tr}_\rho(p' * r_q)$
19:     end if
20: end for
21: return $p'$
```

(see [1, Section 4.9.1]) or a Taylor expansion-based approach (see [2]). Crude upper bounds on the time and space usage of both approaches are $O(\rho^2 \log N)$ time and $O(n \log nt + \log nt)$ space.

Let us proceed with verifying whether Algorithm 1 satisfies the resource bounds of Theorem 1. It is easy to see that all intermediate numbers have modulus at most $2^n N$, so their estimates can be represented with $O(\rho)$ bits. For all multiplications we will use an asymptotically fast algorithm running in $\tilde{O}(n)$ time (e.g., [5]). Then, Line 3–4 take $\tilde{O}(\rho^2 \log N)$ time. Line 6 takes $\tilde{O}(\rho \lg N)$ time; Lines 7–8 take $n\rho \lg N$ time, which is the bottleneck. So overall, the algorithm uses $\tilde{O}(Nn\rho) = \tilde{O}(n^2 t(n + \log t))$ time. The space usage is dominated by the precomputed values which use $O(\log N\rho) = O(n + \log t(\log nt))$ space.

For the correctness of Algorithm 1, let us first study what happens if we work with infinite precision (i.e., $\rho = \infty$). Note that $\mathtt{apxr}_\infty(z) = \omega_N^z$ since it computes

$$\prod_{q=1}^{\log N-1} [2^q \text{ divides } z] r_q$$

$$= \prod_{q=1}^{\log N-1} [2^q \text{ divides } z] \omega_N^{2^q}$$

$$= \omega_N^{\sum_{q=1}^{\log N-1} [2^q \text{ divides } z] 2^q} = \omega^z.$$

Moreover, note that on iteration ℓ of the for-loop of Line 5, we will have on Line 9 that $p = P(\omega_N^\ell)$ by the definition of $P(x)$. Then, it is easy to see that Algorithm 1 indeed evaluates the right-hand side of Eq. 2.

Now, let us focus on the finite precision. The algorithm computes an N-sized sum of $(n + 2\log N)$-sized products of precomputed values, (increased by one). Note that it is sufficient to guarantee that on Line 9 in every iteration ℓ, $|p - \omega_N^{-\ell t} \prod_{i=1}^n (1 + \omega_N^{\ell a_i})| \le 0.4$, since then the total error of s on Line 10 is at most $0.4N$ and the total error of s/N is 0.4, which guarantees rounding to the correct integer. Recall that p is the result of an $(n + 2\log N)$-sized product, so let us analyze how the approximation error propagates in this situation. If $\hat{a}, \hat{b}$ are approximations of a, b and we approximate c by $\mathtt{tr}_\rho \hat{a} * \hat{b}$, we have

$$|c - \hat{c}| \le |a - \hat{a}||b| + |b - \hat{b}||a| + |a - \hat{a}||b - \hat{b}| + 2^{-\rho}.$$

Thus, if a is the result of a (i-1)-sized product, and using an upper bound of 2 for the modulus of any of the product terms in the algorithm, we can upper bound the error of E_i estimating an i-length product as follows: $E_1 \le 2^{-\rho}$ and for $i > 1$:

$$E_i \le 2E_{i-1} + 2^{-\rho} 2^{i-1} + 2^{-\rho} E_{i-1} + 2^{-\rho}$$
$$\le 3E_{i-1} + 2^{-\rho} 2^i.$$

Using straightforward induction we have that $E_i \le 6^i 2^{-\rho}$ So indeed, setting $\rho = 3n + 6 \log nt$ suffices.

A Generic Framework

For Theorem 1, we only used that the to be determined value is a coefficient of a (relatively) small degree polynomial that we can evaluate efficiently. Whether this is the case for other problems solved by dynamic programming can be seen from the structure of the used recurrence: when the recurrence can be formulated over a polynomial ring where the polynomials have small degree, we can evaluate it fast and interpolate with the same technique as above to find a required coefficient. For example, for Knapsack, one can use the polynomial $P(x, y) = \prod_{i=1}^{n}(x^{a_i} y^{b_i})$ and look for a nonzero coefficient of $x^{t'} y^{u'}$ where $t' \leq t$ and $u' \geq u$ to obtain a pseudo-polynomial time and polynomial space algorithm as well.

Naturally, this technique does not only apply to the polynomial ring. In general, if the ring would be $R \subseteq \mathcal{C}^{N\times N}$ equipped with matrix addition and multiplication, we just need a matrix that simultaneously diagonalizes all matrices of R (in the above case, R are all circulant matrices which are simultaneously diagonalized by the Fourier matrix).

Applications

The framework applies to many dynamic programming algorithms. A nice additional example is the algorithm of Dreyfus and Wagner for Steiner tree [4, 7, 8].

Cross-References

▶ Knapsack

Recommended Reading

1. Brent R, Zimmermann P (2010) Modern computer arithmetic. Cambridge University Press, New York
2. Chudnovsky DV, Chudnovsky GV (1997) Approximations and complex multiplication according to ramanujan. In: Pi: a source book. Springer, New York, pp 596–622
3. Dasgupta S, Papadimitriou CH, Vazirani U (2006) Algorithms. McGraw-Hill, Boston
4. Dreyfus SE, Wagner RA (1972) The Steiner problem in graphs. Networks 1:195–207
5. Fürer M (2009) Faster integer multiplication. SIAM J Comput 39(3):979–1005
6. Knuth DE (1997) The art of computer programming, vol 2 (3rd edn.): seminumerical algorithms. Addison-Wesley Longman, Boston
7. Lokshtanov D, Nederlof J (2010) Saving space by algebraization. In: Proceedings of the forty-second ACM symposium on theory of computing, STOC '10, Cambridge. ACM, New York, pp 321–330
8. Nederlof J (2013) Fast polynomial-space algorithms using inclusion-exclusion. Algorithmica 65(4):868–884

Exact Algorithms for Bandwidth

Marek Cygan
Institute of Informatics, University of Warsaw, Warsaw, Poland

Keywords

Bandwidth; Exponential time algorithms; Graph ordering

Years and Authors of Summarized Original Work

2000; Feige
2008; Cygan, Pilipczuk
2010; Cygan, Pilipczuk
2012; Cygan, Pilipczuk

Problem Definition

Given a graph G with n vertices, an *ordering* is a bijective function $\pi : V(G) \to \{1, 2, \ldots, n\}$. The bandwidth of π is a maximal length of an edge, i.e.,

$$\mathrm{bw}(\pi) = \max_{uv \in E(G)} |\pi(u) - \pi(v)|.$$

The bandwidth problem, given a graph G and a positive integer b, asks if there exists an ordering of bandwidth at most b.

Key Results

An exhaustive search for the bandwidth problem enumerates all the $n!$ orderings, trying to find one of bandwidth at most b. The first single exponential time algorithm is due to Feige and Kilian [6], which we are going to describe now.

Bucketing

Definition 1 For a positive integer k, let $\mathcal{I}_k$ be the collection of $\lceil n/k \rceil$ sets obtained by splitting the set $\{1, \ldots, n\}$ into equal parts (except the last one), i.e., $\mathcal{I}_k = \{\{1, \ldots, k\}, \{k+1, \ldots, 2k\}, \ldots\}$. A function $f : V(G) \to \mathcal{I}_k$ is called a k-bucket assignment, if for every edge $uv \in E(G)$ at least one of the following conditions is satisfied:

- $f(u) = f(v)$,
- $|\max f(u) - \min f(v)| \leq b$,
- $|\min f(u) - \max f(v)| \leq b$.

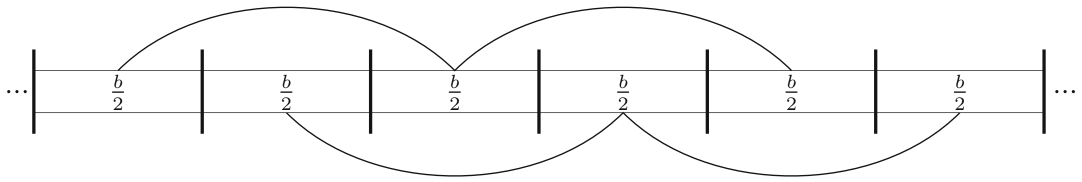
Clearly, if a function $f : V(G) \to \mathcal{I}_k$ is not a k-bucket assignment, then there is no ordering π of bandwidth at most b consistent with f, where π is *consistent* with f iff $\pi(v) \in f(v)$ for each $v \in V(G)$. A bucket function can be seen as a rough assignment – instead of assigning vertices to their final positions in the ordering, we assign them to intervals.

The $\mathcal{O}(10^n \text{poly}(n))$ time algorithm of [6] is based on two ideas, both related to the notion of bucket assignments. For the sake of presentation, let us assume that n is divisible by b, whereas b is a power of two. Moreover, we assume that G is connected, as otherwise it is enough to consider each connected component of G separately.

First, one needs to show that there is a family of at most $n3^{n-1}$ b-bucket assignments $\mathcal{F}$, such that any ordering of bandwidth at most b is consistent with some b-bucket assignment from $\mathcal{F}$. We create $\mathcal{F}$ recursively by branching. First, fix an arbitrary vertex v_0, and assign it to some interval from $\mathcal{I}_k$ (there are at most n choices here). Next, consider any vertex v without assigned interval, which has a neighbor u with already assigned interval. By the assumption that G is connected, v always exists. Note that in order to create a valid bucket assignment, v has to be assigned either to the same interval as u or to one of its two neighboring intervals. This gives at most three branches to be explored.

In the second phase, consider some b-bucket assignment $f \in \mathcal{F}$. We want to check whether there exists some ordering of bandwidth at most b consistent with f. To do this, for each vertex v, we branch into two choices, deciding whether v should be assigned to the left half of $f(v)$ or to the right half of $f(v)$. This leads to at most 2^n $b/2$-bucket assignments to be processed. The key observation is that each of those assignments can be naturally split into two independent subproblems. This is because each edge within an interval of length $b/2$ and each edge between two neighboring intervals of length $b/2$ will be of length at most $b - 1$. Additionally, each edge connecting two vertices with at least two intervals of length $b/2$ in between would lead to violating the constraint of being a valid $b/2$-bucket assignment. Therefore, it is enough to consider vertices in even and odd intervals separately (see Fig. 1). Such routine of creating more and more refined bucket assignments can be continued, where the running time used for n vertices satisfies

$$T(n) = 2^n \cdot 2 \cdot T(\frac{n}{2})$$



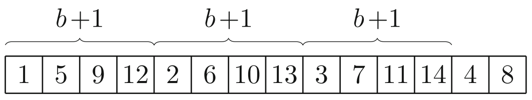
Exact Algorithms for Bandwidth, Fig. 1 *Thick vertical lines* separate subsequent intervals from $\mathcal{I}_{b/2}$. Meaningful edges connect vertices with exactly one interval in between which in turn gives $T(n) = 4^n \text{poly}(n)$. Since we have $|\mathcal{F}| \leq n3^{n-1}$, we end up with $\mathcal{O}(12^n \text{poly}(n))$ time algorithm. If instead of generating b-bucket assignments one uses $b/2$-bucket assignments (there are at most $n5^{n-1}$ of them), then the running time can be improved to $10^n \text{poly}(n)$.

Dynamic Programming

In [2, 5], Cygan and Pilipczuk have shown that for a single $(b+1)$-bucket assignment, one can check in time and space $\mathcal{O}(2^n \text{poly}(n))$ whether there exists an ordering of bandwidth at most b consistent with it. Since there are at most $n3^{n-1}$ $(b+1)$-bucket assignments, this leads to $\mathcal{O}(6^n \text{poly}(n))$ time algorithm.

The key idea is to assign vertices to their final positions consistent with some $f \in \mathcal{F}$ in a very specific order. Let us color the set of positions $\{1, \ldots, n\}$ with $\text{color}(i) = (i-1) \bmod (b+1)$. Define a *color order* of positions, where positions from $\{1, \ldots, n\}$ are sorted by their color values, breaking ties with position values (see Fig. 2).

A lemma that proves usefulness of the color order shows that if we assign vertices to positions in the color order, then we can use the standard Held-Karp dynamic programming over subsets approach. In particular, in a state of dynamic programming, it is enough to store the subset $S \subseteq V(G)$ of vertices already assigned to the first $|S|$ positions in the color order (see Fig. 3).



Exact Algorithms for Bandwidth, Fig. 2 An index of each position in a color order for $n = 14$ and $b = 3$

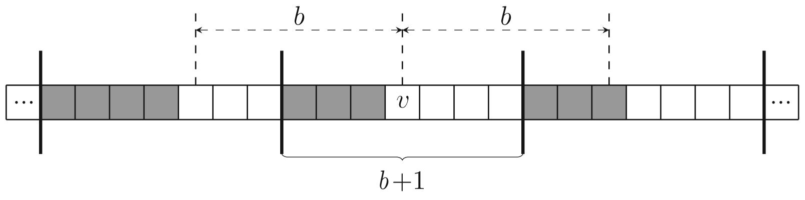
Further Improvements

Instead of upper bounding running time of the algorithm for each $(b+1)$-bucket assignment separately, one can count the number of states of the dynamic programming routine used by the algorithm throughout the processing of all the bucket assignments. As shown in [2], this leads to $\mathcal{O}(5^n \text{poly}(n))$ running time, which with more insights and more technical analysis can be further improved to $O(4.83^n)$ [5] and $O(4.383^n)$ [3]. If only polynomial space is allowed, then the best known algorithm needs $O(9.363^n)$ running time [4].

Related Work

Concerning small values of b, Saxe [8] presented a nontrivial $O(n^{b+1})$ time and space dynamic programming, consequently proving the problem to be in XP. However, Bodlaender et al. [1] have shown that bandwidth is hard for any fixed level of the W hierarchy.

For a related problem of minimum distortion embedding, Fomin et al. [7] obtained a $\mathcal{O}(5^n \text{poly}(n))$ time algorithm, improved by Cygan and Pilipczuk [4] to running times same as for the best known bandwidth algorithms.



Exact Algorithms for Bandwidth, Fig. 3 When a vertex v is to be assigned to the next position in the color order, then all its neighbors from the left interval cannot be yet assigned a position, whereas all its neighbors from the right interval have to be already assigned in order to obtain an ordering of bandwidth at most b

Open Problems

Many vertex ordering problems admit $\mathcal{O}(2^n \text{poly}(n))$ time and space algorithms, like Hamiltonicity, cutwidth, pathwidth, optimal linear arrangement, etc. In [2], Cygan and Pilipczuk have shown that a dynamic programming routine with such a running time is possible, provided a $(b + 1)$-bucket assignment is given. A natural question to ask is whether it is possible to obtain $\mathcal{O}(2^n \text{poly}(n))$ without the assumption of having a fixed assignment to be extended.

Cross-References

▶ Graph Bandwidth

Recommended Reading

1. Bodlaender HL, Fellows MR, Hallett MT (1994) Beyond np-completeness for problems of bounded width: hardness for the W hierarchy. In: Leighton FT, Goodrich MT (eds) Proceedings of the twenty-sixth annual ACM symposium on theory of computing, Montréal, 23–25 May 1994. ACM, pp 449–458, doi:10.1145/195058.195229. http://doi.acm.org/10.1145/195058.195229
2. Cygan M, Pilipczuk M (2008) Faster exact bandwidth. In: Broersma H, Erlebach T, Friedetzky T, Paulusma D (eds) Graph-theoretic concepts in computer science, 34th international workshop, WG 2008, Durham, June 30–July 2, 2008. Revised papers, Lecture notes in computer science, vol 5344, pp 101–109. doi:10.1007/978-3-540-92248-3_10. http://dx.doi.org/10.1007/978-3-540-92248-3_10
3. Cygan M, Pilipczuk M (2010) Exact and approximate bandwidth. Theor Comput Sci 411(40–42):3701–3713. doi:10.1016/j.tcs.2010.06.018. http://dx.doi.org/10.1016/j.tcs.2010.06.018
4. Cygan M, Pilipczuk M (2012) Bandwidth and distortion revisited. Discret Appl Math 160(4–5):494–504. doi:10.1016/j.dam.2011.10.032. http://dx.doi.org/10.1016/j.dam.2011.10.032
5. Cygan M, Pilipczuk M (2012) Even faster exact bandwidth. ACM Trans Algorithms 8(1):8. doi:10.1145/2071379.2071387. http://doi.acm.org/10.1145/2071379.2071387
6. Feige U (2000) Coping with the np-hardness of the graph bandwidth problem. In: Halldórsson MM (ed) Proceedings of the 7th Scandinavian workshop on algorithm theory (SWAT), Bergen. Lecture notes in computer science, vol 1851. Springer, pp 10–19
7. Fomin FV, Lokshtanov D, Saurabh S (2011) An exact algorithm for minimum distortion embedding. Theor Comput Sci 412(29):3530–3536. doi:10.1016/j.tcs.2011.02.043. http://dx.doi.org/10.1016/j.tcs.2011.02.043
8. Saxe J (1980) Dynamic programming algorithms for recognizing small-bandwidth graphs in polynomial time. SIAM J Algebr Methods 1:363–369

E

Exact Algorithms for Dominating Set

Dieter Kratsch
UFM MIM – LITA, Université de Lorraine,
Metz, France

Keywords

Branch and reduce; Dominating set; Exact exponential algorithm; Measure and conquer

Years and Authors of Summarized Original Work

2005; Fomin, Grandoni, Kratsch
2008; van Rooij, Bodlaender
2011; Iwata

Problem Definition

The dominating set problem is a classical NP-hard optimization problem which fits into the broader class of covering problems. Hundreds of papers have been written on this problem that has a natural motivation in facility location.

Definition 1 For a given undirected, simple graph $G = (V, E)$, a subset of vertices $D \subseteq V$ is called a *dominating set* if every vertex $u \in V - D$ has a neighbor in D. The minimum dominating set problem (abbr. MDS) is to find a *minimum dominating set* of G, i.e., a dominating set of G of minimum cardinality.

Problem 1 (MDS)

INPUT: Undirected simple graph $G = (V, E)$.
OUTPUT: A minimum dominating set D of G.

Various modifications of the dominating set problem are of interest, some of them obtained by putting additional constraints on the dominating set as, e.g., requesting it to be an independent set or to be connected. In graph theory, there is a huge literature on domination dealing with the problem and its many modifications. In graph algorithms, the MDS problem and some of its modifications like independent dominating set and connected dominating set have been studied as benchmark problems for attacking NP-hard problems under various algorithmic approaches.

Known Results

The algorithmic complexity of MDS and its modifications when restricted to inputs from a particular graph class has been studied extensively. Among others, it is known that MDS remains NP-hard on bipartite graphs, split graphs, planar graphs, and graphs of maximum degree 3. Polynomial time algorithms to compute a minimum dominating set are known, e.g., for permutation, interval, and k-polygon graphs. There is also a $O(3^k\, n^{O(1)})$ time algorithm to solve MDS on graphs of treewidth at most k.

The dominating set problem is one of the basic problems in parameterized complexity; it is W[2]-complete and thus it is unlikely that the problem is fixed parameter tractable. On the other hand, the problem is fixed parameter tractable on planar graphs. Concerning approximation, MDS is equivalent to MINIMUM SET COVER under L-reductions. There is an approximation algorithm solving MDS within a factor of $1 + \log|V|$, and it cannot be approximated within a factor of $(1-\epsilon)\ln|V|$ for any $\epsilon > 0$, unless NP $\subset$ DTIME($n^{\log\log n}$).

Exact Exponential Algorithms

If P $\neq$ NP, then no polynomial time algorithm can solve MDS. Even worse, it has been observed in [5] that unless SNP$\subseteq$ SUBEXP (which is considered to be highly unlikely), there is not even a subexponential time algorithm solving the dominating set problem.

The trivial $O(2^n\,(n+m))$ algorithm, which simply checks all the 2^n vertex subsets whether they are dominating, clearly solves MDS. Three faster algorithms have been established in 2004. The algorithm of Fomin et al. [5] uses a deep graph-theoretic result due to B. Reed, stating that every graph on n vertices with minimum degree at least three has a dominating set of size at most $3n/8$, to establish an $O(2^{0.955n})$ time algorithm solving MDS. The $O(2^{0.919n})$ time algorithm of Randerath and Schiermeyer [9] uses very nice ideas including matching techniques to restrict the search space. Finally, Grandoni [6] established an $O(2^{0.850n})$ time algorithm to solve MDS.

Key Results

Branch and Reduce and Measure and Conquer

The work of Fomin, Grandoni, and Kratsch presents a simple and easy way to implement a recursive branch and reduce algorithm to solve MDS. It was first presented at ICALP 2005 [2] and later published in 2009 in [3]. The running time of the algorithm is significantly faster than the ones stated for the previous algorithms. This is heavily based on the analysis of the running time by measure and conquer, which is a method to analyze the worst case running time of (simple) branch and reduce algorithms based on a sophisticated choice of the measure of a problem instance.

Theorem 1 *There is a branch and reduce algorithm solving* MDS *in time* $O(2^{0.610n})$ *using polynomial space.*

Theorem 2 *There is an algorithm solving* MDS *in time* $O(2^{0.598n})$ *using exponential space.*

The algorithms of Theorems 1 and 2 are simple consequences of a transformation from MDS to MINIMUM SET COVER (abbr. MSC) combined with new exact exponential time algorithms for MSC.

Problem 2 (MSC)

INPUT: Finite set $\mathcal{U}$ and a collection $\mathcal{S}$ of subsets $S_1, S_2, \ldots S_t$ of $\mathcal{U}$.

OUTPUT: A minimum set cover $\mathcal{S}'$, where $\mathcal{S}' \subseteq \mathcal{S}$ is a set cover of $(\mathcal{U}, \mathcal{S})$ if $\bigcup_{S_i \in \mathcal{S}'} S_i = \mathcal{U}$.

Theorem 3 *There is a branch and reduce algorithm solving* MSC *in time* $O(2^{0.305(|\mathcal{U}|+|\mathcal{S}|)})$ *using polynomial space.*

Applying memorization to the polynomial space algorithm of Theorem 3, the running time can be improved as follows.

Theorem 4 *There is an algorithm solving* MSC *in time* $O(2^{0.299(|\mathcal{S}|+|\mathcal{U}|)})$ *needing exponential space.*

The analysis of the worst case running time of the simple branch and reduce algorithm solving MSC (of Theorem 3) is done by a careful choice of the measure of a problem instance which allows to obtain an upper bound that is significantly smaller than the one that could be obtained using the standard measure. The refined analysis leads to a collection of recurrences. Then, random local search was used to compute the weights, used in the definition of the measure, aiming at the best achievable upper bound of the worst case running time. By now various other methods to do these time-consuming computations are available; see, e.g., [1].

Getting Faster MDS Algorithms

There is a lot of interest in exact exponential algorithms for solving MDS and in improving their best known running times. Two important improvements on the running times of the original algorithm stated in Theorems 1 and 2 have been achieved. To simplify the comparison, let us mention that in [4] those running times are stated as $O(1.5259^n)$ using polynomial space and $O(1.5132^n)$ needing exponential space.

Van Rooij and Bodlaender presented faster exact exponential algorithms solving MDS that are strongly based on the algorithms of Fomin et al. and the methods of their analysis. By introducing new reduction rules in the algorithm and a refined analysis, they achieved running time $O(1.5134^n)$ using polynomial space and time $O(1.5063^n)$, presented at STACS 2008. This analysis has been further improved in [11] to achieve a running time of $O(1.4969^n)$ using polynomial space, which was published in 2011. It should be emphasized that memorization cannot be applied to the latter algorithm.

The currently best known algorithms solving MDS have been obtained by Ywata [7] and presented at IPEC 2011.

Theorem 5 *There is a branch and reduce algorithm solving* MDS *in time* $O(1.4864^n)$ *using polynomial space.*

Theorem 6 *There is an algorithm solving* MDS *in time* $O(1.4689^n)$ *needing exponential space.*

Ywata's polynomial space branch and reduce algorithm is also strongly related to the algorithm of Fomin et al. and its analysis. The improvement in the running time is achieved by some crucial change in the order of branchings in the algorithm solving MSC, i.e., the algorithm branches on the same element consecutively. These consecutive branchings can then be exploited by a refined analysis using global weights called potentials. Thus, such an analysis is dubbed "potential method." By a variant of memorization where dynamic programming memorizes only solutions of subproblems with small number of elements, an algorithm of running time $O(1.4689^n)$ needing exponential space has been obtained.

Counting Dominating Sets

A strongly related problem is #DS that asks to determine for a given graph G the number of dominating sets of size k, for any k. In [8], Nederlof, van Rooij, and van Dijk show how to combine inclusion/exclusion and a branch and reduce algorithm while using measure and conquer, as to obtain an algorithm (needing exponential space) of running time $O(1.5002^n)$. Clearly, this also solves MDS.

Applications

There are various other NP-hard domination-type problems that can be solved by exact exponential algorithms based on an algorithm solving

MINIMUM SET COVER: any instance of the initial problem is transformed to an instance of MSC, and then an algorithm solving MSC is applied and thus the initial problem is solved. Examples of such problems are TOTAL DOMINATING SET, k-DOMINATING SET, k-CENTER, and MDS on split graphs. Measure and conquer and the strongly related quasiconvex analysis of Eppstein [1] have been used to design and analyze a variety of exact exponential branch and reduce algorithms for NP-hard problems, optimization, counting, and enumeration problems; see [4].

Open Problems

While for many algorithms it is easy to show that the worst case analysis is tight, this is not the case for the nowadays time analysis of branch and reduce algorithms. For example, the worst case running times of the branch and reduce algorithms of Fomin et al. [3] solving MDS and MSC remain unknown; a lower bound of $\Omega(3^{n/4})$ for the MDS algorithm is known. The situation is similar for many other branch and reduce algorithms. Consequently, there is a strong need for new and better tools to analyze the worst case running time of branch and reduce algorithms.

Cross-References

- ▶ Connected Dominating Set
- ▶ Exact Algorithms for Induced Subgraph Problems
- ▶ Exact Algorithms for Maximum Independent Set
- ▶ Minimal Dominating Set Enumeration

Recommended Reading

1. Eppstein D (2006) Quasiconvex analysis of backtracking algorithms. ACM Trans Algorithms 2(4):492–509
2. Fomin FV, Grandoni F, Kratsch D (2005) Measure and conquer: domination – a case study. In: Proceedings of ICALP 2005, Lisbon
3. Fomin FV, Grandoni F, Kratsch D (2009) A measure & conquer approach for the analysis of exact algorithms. J ACM 56(5)
4. Fomin FV, Kratsch D (2010) Exact exponential algorithms. Springer, Heidelberg
5. Fomin FV, Kratsch D, Woeginger GJ (2004) Exact (exponential) algorithms for the dominating set problem. In: Proceedings of WG 2004, Bonn. LNCS, vol 3353. Springer, pp 245–256
6. Grandoni F (2004) Exact algorithms for hard graph problems. PhD thesis, Università di Roma "Tor Vergata", Roma, Mar 2004
7. Iwata Y (2011) A faster algorithm for Dominating Set analyzed by the potential method. In: Proceedings of IPEC 2011, Saarbrücken. LNCS, vol 7112. Springer, pp 41–54
8. Nederlof J, van Rooij JMM, van Dijk TC (2014) Inclusion/exclusion meets measure and conquer. Algorithmica 69(3):685–740
9. Randerath B, Schiermeyer I (2004) Exact algorithms for MINIMUM DOMINATING SET. Technical Report, zaik-469, Zentrum für Angewandte Informatik Köln, Apr 2004
10. van Rooij JMM (2011) Exact exponential-time algorithms for domination problems in graphs. PhD thesis, University Utrecht
11. van Rooij JMM, Bodlaender HL (2011) Exact algorithms for dominating set. Discret Appl Math 159(17):2147–2164
12. Woeginger GJ (2003) Exact algorithms for NP-hard problems: a survey. Combinatorial optimization – Eureka, you shrink. LNCS, vol 2570. Springer, Berlin/Heidelberg, pp 185–207

Exact Algorithms for General CNF SAT

Edward A. Hirsch
Laboratory of Mathematical Logic, Steklov Institute of Mathematics, St. Petersburg, Russia

Keywords

Boolean satisfiability; Exponential-time algorithms; SAT

Years and Authors of Summarized Original Work

1998; Hirsch
2003; Schuler

Problem Definition

The satisfiability problem (*SAT*) for Boolean formulas in conjunctive normal form (*CNF*) is one of the first **NP**-complete problems [2, 13]. Since its **NP**-completeness currently leaves no hope for polynomial-time algorithms, the progress goes by decreasing the exponent. There are several versions of this parametrized problem that differ in the parameter used for the estimation of the running time.

Problem 1 (SAT)

INPUT: Formula F in CNF containing n variables, m clauses, and l literals in total.
OUTPUT: "Yes" if F has a *satisfying assignment*, i.e., a substitution of Boolean values for the variables that makes F true. "No" otherwise.

The bounds on the running time of SAT algorithms can be thus given in the form $|F|^{O(1)} \cdot \alpha^n$, $|F|^{O(1)} \cdot \beta^m$, or $|F|^{O(1)} \cdot \gamma^l$, where $|F|$ is the length of a reasonable bit representation of F (i.e., the formal input to the algorithm). In fact, for the present algorithms, the bases β and γ are constants, while α is a function $\alpha(n, m)$ of the formula parameters (because no better constant than $\alpha = 2$ is known).

Notation

A formula in conjunctive normal form is a set of clauses (understood as the conjunction of these clauses), a clause is a set of literals (understood as the disjunction of these literals), and a literal is either a Boolean variable or the negation of a Boolean variable. A truth assignment assigns Boolean values (*false* or *true*) to one or more variables. An assignment is abbreviated as the list of literals that are made true under this assignment (e.g., assigning *false* to x and *true* to y is denoted by $\neg x, y$). The result of the application of an assignment A to a formula F (denoted $F[A]$) is the formula obtained by removing the clauses containing the true literals from F and removing the falsified literals from the remaining clauses. For example, if $F = (x \vee \neg y \vee z) \wedge (y \vee \neg z)$, then $F[\neg x, y] = (z)$. A *satisfying* assignment for F is an assignment A such that $F[A] =$ true. If such an assignment exists, F is called *satisfiable*.

Key Results

Bounds for β and γ

General Approach and a Bound for β

The trivial brute-force algorithm enumerating all possible assignments to the n variables runs in 2^n polynomial-time steps. Thus $\alpha \leq 2$, and by trivial reasons also $\beta, \gamma \leq 2$. In the early 1980s, Monien and Speckenmeyer noticed that β could be made smaller. (They and other researchers also noticed that α could be made smaller for a special case of the problem where the length of each clause is bounded by a constant; the reader is referred to another entry (*Local search algorithms for k-SAT*) of the *Encyclopedia* for relevant references and algorithms.) Then Kullmann and Luckhardt [12] set up a framework for divide-and-conquer (Also called *DPLL* due to the papers of Davis and Putnam [6] and Davis, Logemann, and Loveland [7].) algorithms for SAT that split the original problem into several (yet usually a constant number of) subproblems by substituting the values of some variables and simplifying the obtained formulas. This line of research resulted in the following upper bounds for β and γ:

Theorem 1 (Hirsch [8]) *SAT can be solved in time*

1. $|F|^{O(1)} \cdot 2^{0.30897m}$;
2. $|F|^{O(1)} \cdot 2^{0.10299l}$.

A typical divide-and-conquer algorithm for SAT consists of two phases: splitting of the original problem into several subproblems (e.g., reducing SAT(F) to SAT(F[x]) and $SAT(F[\neg x])$) and simplification of the obtained subproblems using polynomial-time transformation rules that do not affect the satisfiability of the subproblems (i.e., they replace a formula by an equi-satisfiable one). The subproblems $F_1, \ldots, F_k$ for splitting are chosen so that the corresponding recurrent inequality using the simplified problems

$F_1', \ldots, F_k'$,

$$T(F) \le \sum_{i=1}^{k} T\left(F_i'\right) + const,$$

gives a desired upper bound on the number of leaves in the recurrence tree and, hence, on the running time of the algorithm. In particular, in order to obtain the bound $|F|^{O(1)} \cdot 2^{0.30897m}$ one takes either two subproblems $F[x], F[\neg x]$ with recurrent inequality

$$t_m \le t_{m-3} + t_{m-4}$$

or four subproblems $F[x, y], F[x, \neg y], F[\neg x, y], F[\neg x, \neg y]$ with recurrent inequality

$$t_m \le 2t_{m-6} + 2t_{m-7}$$

where $t_i = \max_{m(G) \le i} T(G)$. The simplification rules used in the $|F|^{O(1)} \cdot 2^{0.30897m}$-time and the $|F|^{O(1)} \cdot 2^{0.10299l}$-time algorithms are as follows:

Simplification Rules

Elimination of 1-Clauses If F contains a 1-clause (a), replace F by $F[a]$.

Subsumption If F contains two clauses C and D such that $C \subseteq D$, replace F by $F \backslash \{D\}$.

Resolution with Subsumption Suppose a literal a and clauses C and D are such that a is the only literal satisfying both conditions $a \in C$ and $\neg a \in D$. In this case, the clause $(C \cup D) \backslash \{a, \neg a\}$ is called the *resolvent by the literal* a of the clauses C and D and denoted by $R(C, D)$.

The rule is: if $R(C, D) \subseteq D$, replace F by $(F \backslash \{D\}) \cup \{R(C, D)\}$.

Elimination of a Variable by Resolution [6] Given a literal a, construct the formula DP$_a(F)$ by

1. Adding to F all resolvents by a
2. Removing from F all clauses containing a or $\neg a$

The rule is: if DP$_a(F)$ is not larger in m (resp., in l) than F, then replace F by DP$_a(F)$.

Elimination of Blocked Clauses A clause C is *blocked* for a literal a w.r.t. F if C contains the literal a, and the literal $\neg a$ occurs only in the clauses of F that contain the negation of at least one of the literals occurring in $C \backslash \{a\}$. For a CNF-formula F and a literal a occurring in it, the assignment $I(a, F)$ is defined as

$$\{a\} \cup \Big\{\text{literals } x \notin \{a, \neg a\} \mid \text{the clause } \{\neg a, x\} \text{ is blocked for } \neg a \text{ w.r.t. } F\Big\}.$$

Lemma 2 (Kullmann [11])

(1) If a clause C is blocked for a literal a w.r.t. F, then F and $F \backslash \{C\}$ are equi-satisfiable.
(2) Given a literal a, the formula F is satisfiable iff at least one of the formulas $F[\neg a]$ and $F[I(a, F)]$ is satisfiable.

The first claim of the lemma is employed as a simplification rule.

Application of the Black and White Literals Principle Let P be a binary relation between literals and formulas in CNF such that for a variable v and a formula F, at most one of $P(v, F)$ and $P(\neg v, F)$ holds.

Lemma 3 *Suppose that each clause of F that contains a literal w satisfying $P(w, F)$ contains also at least one literal b satisfying $P(\neg b, F)$. Then F and $F[\{l \mid P(\neg l, F)\}]$ are equi-satisfiable.*

A Bound for γ

To obtain the bound $|F|^{O(1)} \cdot 2^{0.10299l}$, it is enough to use a pair $F[\neg a], F[I(a, F)]$ of subproblems (see Lemma 2(2)) achieving the desired recurrent inequality $t_l \le t_{l-5} + t_{l-17}$ and to switch to the $|F|^{O(1)} \cdot 2^{0.30897m}$-time algorithm if there are none. A recent (much more technically involved) improvement to this algorithm [16] achieves the bound $|F|^{O(1)} \cdot 2^{0.0926l}$.

A Bound for α

Currently, no non-trivial constant upper bound for α is known. However, starting with [14] there was an interest to non-constant bounds. A series of randomized and deterministic algorithms showing successive improvements was developed, and at the moment the best possible bound is achieved by a deterministic divide-and-conquer algorithm employing the following recursive procedure. The idea behind it is a dichotomy: either each clause of the input formula can be shortened to its first k literals (then a k-CNF algorithm can be applied), or all these literals in one of the clauses can be assumed false. (This clause-shortening approach can be attributed to Schuler [15], who used it in a randomized fashion. The following version of the deterministic algorithm achieving the best known bound both for deterministic and randomized algorithms appears in [5].)

Procedure S

```
Input: a CNF formula $F$ and a positive integer $k$.

  1. Assume $F$ consists of clauses $C_1, \dots, C_m$.
     Change each clause $C_i$ to a clause $D_i$ as follows: If $|C_i| > k$ then choose any $k$
     literals in $C_i$ and drop the other literals; otherwise leave $C_i$ as is, i.e., $D_i = C_i$. Let
     $F'$ denote the resulting formula.
  2. Test satisfiability of $F'$ using the $m \cdot \text{poly}(n) \cdot (2 - 2/(k+1))^n$-time $k$-CNF
     algorithm defined in [4].
  3. If $F'$ is satisfiable, output "satisfiable" and halt. Otherwise, for each $i$, do the following:
     1. Convert $F$ to $F_i$ as follows:
        1. Replace $C_j$ by $D_j$ for all $j < i$.
        2. Assign false to all literals in $D_i$.
     2. Recursively invoke Procedure S on $(F_i, k)$.
  4. Return "unsatisfiable".
```

The algorithm just invokes *Procedure S* on the original formula and the integer parameter $k = k*(m, n)$. The most accurate analysis of this family of algorithms by Calabro, Impagliazzo, and Paturi [1] implies that, assuming that $m > n$, one can obtain the following bound by taking $k(m, n) = 2\log(m/n) + \text{const}$. (This explicit bound is not stated in [1] and is inferred in [3].)

Theorem 4 (Dantsin, Hirsch [3]) *Assuming $m > n$, SAT can be solved in time*

$$|F|^{O(1)} \cdot 2^n \left(1 - \frac{1}{O(\log(m/n))}\right).$$

Applications

While SAT has numerous applications, the presented algorithms have no direct effect on them.

Open Problems

Proving a constant upper bound on $\alpha < 2$ remains a major open problem in the field, as well as the hypothetic existence of $(1+\varepsilon)^l$-time algorithms for arbitrary small $\varepsilon > 0$.

It is possible to perform the analysis of a divide-and-conquer algorithm and even to generate simplification rules automatically [10]. However, this approach so far led to new bounds only for the (NP-complete) optimization version of 2-SAT [9].

Experimental Results

Jun Wang has implemented the algorithm yielding the bound on β and collected some statistics regarding the number of applications of the simplification rules [17].

Cross-References

▸ Exact Algorithms for k SAT Based on Local Search

Recommended Reading

1. Calabro C, Impagliazzo R, Paturi R (2006) A duality between clause width and clause density for SAT. In: Proceedings of the 21st annual IEEE conference on computational complexity (CCC 2006), Prague. IEEE Computer Society, pp 252–260
2. Cook SA (2006) The complexity of theorem proving procedures. In: Proceedings of the third annual ACM symposium on theory of computing, Shaker Heights, May 1971. ACM, pp 151–158
3. Dantsin E, Hirsch EA (2008) Worst-case upper bounds. In: Biere A, van Maaren H, Walsh T (eds) Handbook of satisfiability. IOS. In: Biere A, Heule MJH, van Maaren H, Walsh T (eds) Handbook of satisfiability. Frontiers in artificial intelligence and applications, vol 185. IOS Press, Amsterdam/Washington, DC, p 980. ISBN:978-1-58603-929-5, ISSN:0922-6389
4. Dantsin E, Goerdt A, Hirsch EA, Kannan R, Kleinberg J, Papadimitriou C, Raghavan P, Schöning U (2002) A deterministic $(2 - 2/(k+1))^n$ algorithm for k-SAT based on local search. Theor Comput Sci 289(1):69–83
5. Dantsin E, Hirsch EA, Wolpert A (2006) Clause shortening combined with pruning yields a new upper bound for deterministic SAT algorithms. In: Proceedings of CIAC-2006, Rome. Lecture notes in computer science, vol 3998. Springer, Berlin, pp 60–68
6. Davis M, Putnam H (1960) A computing procedure for quantification theory. J ACM 7: 201–215
7. Davis M, Logemann G, Loveland D (1962) A machine program for theorem-proving. Commun ACM 5:394–397
8. Hirsch EA (2000) New worst-case upper bounds for SAT. J Autom Reason 24(4):397–420
9. Kojevnikov A, Kulikov A (2006) A new approach to proving upper bounds for MAX-2-SAT. In: Proceedings of the seventeenth annual ACM-SIAM symposium on discrete algorithms (SODA 2006), Miami. ACM/SIAM, pp 11–17
10. Kulikov A (2005) Automated generation of simplification rules for SAT and MAXSAT. In: Proceedings of the eighth international conference on theory and applications of satisfiability testing (SAT 2005), St. Andrews. Lecture notes in computer science, vol 3569. Springer, Berlin, pp 430–436
11. Kullmann O (1999) New methods for 3-{SAT} decision and worst-case analysis. Theor Comput Sci 223(1–2):1–72
12. Kullmann O, Luckhardt H (1998) Algorithms for SAT/TAUT decision based on various measures, 71pp. Preprint, http://cs-svr1.swan.ac.uk/csoliver/papers.html
13. Levin LA (1973) Universal search problems. Probl Inf Trans 9(3):265–266. In Russian. English translation Trakhtenbrot BA (1984) A survey of russian approaches to perebor (Brute-force Search) algorithms. Ann Hist Comput 6(4):384–400
14. Pudlák P (1998) Satisfiability – algorithms and logic. In: Proceedings of the 23rd international symposium on mathematical foundations of computer science, MFCS'98. Lecture notes in computer science, Brno, vol 1450. Springer, Berlin, pp 129–141
15. Schuler R (2005) An algorithm for the satisfiability problem of formulas in conjunctive normal form. J Algorithms **54**(1), 40–44
16. Wahlström M (2005) An algorithm for the SAT problem for formulae of linear length. In: Proceedings of the 13th annual European symposium on algorithms, ESA 2005. Lecture notes in computer science, Mallorca, vol 3669. Springer, Berlin, pp 107–118
17. Wang J (2002) Generating and solving 3-SAT. MSc thesis, Rochester Institute of Technology, Rochester

Exact Algorithms for Induced Subgraph Problems

Michał Pilipczuk
Institute of Informatics, University of Warsaw, Warsaw, Poland
Institute of Informatics, University of Bergen, Bergen, Norway

Keywords

Exact algorithms; Hereditary property; Induced subgraph; 2^n barrier

Years and Authors of Summarized Original Work

2010; Fomin, Villanger
2013; Bliznets, Fomin, Pilipczuk, Villanger

Problem Definition

A graph class Π is a set of simple graphs. One can also think of Π as a *property*: Π comprises all the graphs that satisfy a certain condition. We say that class (property) Π is *hereditary* if it is closed under taking induced subgraphs. More precisely, whenever $G \in \Pi$ and H is an induced subgraph of G, then also $H \in \Pi$.

We shall consider the MAXIMUM INDUCED Π-SUBGRAPH problem: given a graph G, find the largest (in terms of the number of vertices) induced subgraph of G that belongs to Π. Suppose now that class Π is polynomial-time recognizable: there exists an algorithm that decides whether a given graph H belongs to Π in polynomial time. Then MAXIMUM INDUCED Π-SUBGRAPH on an n-vertex graph G can be solved by brute force in time (The $\mathcal{O}^{\star}(\cdot)$ notation hides factors polynomial in the input size.) $\mathcal{O}^{\star}(2^n)$: we iterate through all the induced subgraphs of G, and on each of them, we run a polynomial-time test deciding whether it belongs to Π.

Can we do anything smarter? Of course, this very much depends on the class Π we are working with. MAXIMUM INDUCED Π-SUBGRAPH is a generic problem that encompasses many other problems as special cases; examples include CLIQUE (Π = complete graphs), INDEPENDENT SET (Π = edgeless graphs), or FEEDBACK VERTEX SET (Π = forests). It is convenient to assume that Π is also hereditary; this assumption is satisfied in many important examples, including the aforementioned special cases.

So far, the MAXIMUM INDUCED Π-SUBGRAPH problem has been studied for many graph classes Π, and basically in all the cases it turned out that it is possible to find an algorithm with running time $\mathcal{O}(c^n)$ for some $c < 2$. Obtaining a result of this type is often informally called *breaking the* 2^n *barrier*. While the algorithms share a common general methodology, vital details differ depending on the structural properties of the class Π. This makes each and every algorithm of this type contrived to a particular scenario. However, it is tempting to formulate the following general conjecture.

Conjecture 1 ([1]) *For every hereditary, polynomial-time recognizable class of graphs Π, there exists a constant $c_\Pi < 2$ for which there is an algorithm solving* MAXIMUM INDUCED Π-SUBGRAPH *in time $\mathcal{O}(c_\Pi^n)$.*

On one hand, current partial progress on this conjecture consists of scattered results exploiting different properties of particular classes Π, without much hope for proving more general statements. On the other hand, finding a counterexample refuting Conjecture 1 based, e.g., on the Strong Exponential Time Hypothesis seems problematic: the input to MAXIMUM INDUCED Π-SUBGRAPH consists only of $\binom{n}{2}$ bits of information about adjacencies between the vertices, and it seems difficult to model the search space of a general k-SAT using such input under the constraint that Π has to be hereditary and polynomial-time recognizable.

It can be that Conjecture 1 is either false or very difficult to prove, and therefore, one can postulate investigating its certain subcases connected to well-studied classes of graphs. For instance, one could assume that graphs from Π have constant treewidth or that Π is a subclass of chordal or interval graphs. Another direction is to strengthen the assumption about the description of the class Π by requiring that belonging to Π can be expressed in some formalism (e.g., some variant of logic). Finally, one can investigate the algorithms for MAXIMUM INDUCED Π-SUBGRAPH where Π is not required to be hereditary; here, natural nonhereditary properties are connectivity and regularity.

Key Results

Table 1 presents a selection of results on the MAXIMUM INDUCED Π-SUBGRAPH problem. Since the algorithms are usually quite technical when it comes to details, we now present an overview of the general methodology and most important techniques. In the following, we assume that Π is hereditary and polynomial-time recognizable.

Most often, the general approach is to examine the structure of the input instance and of a fixed, unknown optimum solution. The goal is to identify as broad spectrum of situations as possible where the solution can be found by examining $\mathcal{O}((2-\varepsilon)^n)$ candidates, for some $\varepsilon > 0$. By checking the occurrence of each of these situations, we eventually narrow down our investigations to the case where we have a well-defined structure of the input instance and

Exact Algorithms for Induced Subgraph Problems, Table 1 Known results for MAXIMUM INDUCED Π-SUBGRAPH. The first part of the table presents results for problems for which breaking the 2^n barrier follows directly from branching on forbidden subgraphs. The second part contains results for which breaking the barrier requires a nontrivial insight into the structure of Π. Finally, the last part contains results for nonhereditary classes Π. Here, ε denotes a small, positive constant, and its index specifies a parameter on which the value of this constant depends

Property	Time complexity	Reference
Edgeless	$\mathcal{O}(1.2109^n)$	Robson [10]
Biclique	$\mathcal{O}(1.3642^n)$	Gaspers et al. [6]
Cluster graph	$\mathcal{O}(1.6181^n)$	Fomin et al. [3]
Bipartite	$\mathcal{O}(1.62^n)$	Raman et al. [9]
Acyclic	$\mathcal{O}(1.7347^n)$	Fomin et al. [2]
Constant treewidth	$\mathcal{O}(1.7347^n)$	Fomin et al. [2]
Planar	$\mathcal{O}(1.7347^n)$	Fomin et al. [4]
d-degenerate	$\mathcal{O}((2-\varepsilon_d)^n)$	Pilipczuk×2 [8]
Chordal	$\mathcal{O}((2-\varepsilon)^n)$	Bliznets et al. [1]
Interval	$\mathcal{O}((2-\varepsilon)^n)$	Bliznets et al. [1]
r-regular	$\mathcal{O}((2-\varepsilon_r)^n)$	Gupta et al. [7]
Matching	$\mathcal{O}(1.6957^n)$	Gupta et al. [7]

a number of assumptions about how the solution looks like. Then, hopefully, a direct algorithm can be devised.

Let us consider a very simple example of this principle, which is also a technique used in many algorithms for breaking the 2^n barrier. Suppose the input graph has n vertices and assume the optimum solution is of size larger that $(1/2+\delta)n$, for some $\delta > 0$. Then, as candidates for the optimum solution, we can consider all the vertex subsets of at least this size: there is only $(2-\varepsilon)^n$ of them, where $\varepsilon > 0$ depends on δ. Similarly, if the optimum solution has size smaller than $(1/2-\delta)n$, then we can identify this situation by iterating through all the vertex subsets of size $(1/2-\delta)n$ (whose number is again $(2-\varepsilon)^n$ for some $\varepsilon > 0$) and verifying that none of them induces a graph belonging to Π; note that here we use the assumption that Π is hereditary. In this case we can solve the problem by looking at all vertex subsets of size at most $(1/2-\delta)n$. All in all, we can solve the problem faster than $\mathcal{O}^\star(2^n)$ provided that the number of vertices in the optimum solution differs by at least δn from $n/2$, for some $\varepsilon > 0$. More precisely, for every $\delta > 0$ we will obtain a running time of the form $\mathcal{O}((2-\varepsilon)^n)$, where ε tends to 0 when δ tends to 0. Hence, we can focus only on the situation when the number of vertices in the optimum solution is very close to $n/2$.

We now give an overview of some other important techniques.

Branching on Forbidden Induced Subgraphs

Every hereditary graph class Π can be characterized by giving a minimal set of forbidden induced subgraphs $\mathcal{F}$: a graph belongs to Π if and only if it does not contain any graph from $\mathcal{F}$ as an induced subgraph, and $\mathcal{F}$ is inclusion-wise minimal with this property. For instance, the class of forests is characterized by $\mathcal{F}$ being the family of all the cycles, whereas taking $\mathcal{F}$ to be the family of all the cycles of length at least 4 gives the class of chordal graphs. For many important classes the family $\mathcal{F}$ is infinite, but there are notable examples where it is finite, like cluster, trivially perfect, or split graphs.

If Π is characterized by a finite set of forbidden subgraphs $\mathcal{F}$, then already a simple branching strategy yields an algorithm working in time $\mathcal{O}((2-\varepsilon)^n)$, for some $\varepsilon > 0$ depending on $\mathcal{F}$. Without going into details, we iteratively find a forbidden induced subgraph that is not yet removed by the previous choices and branch on the fate of all the undecided vertices in this subgraph, omitting the branch where all of them are included in the solution. Since this forbidden induced subgraph is of constant size, a standard analysis shows that the running time of this algorithm is $\mathcal{O}((2-\varepsilon)^n)$ for some $\varepsilon > 0$ depending on $\max_{H\in\mathcal{F}} |V(H)|$. This simple observation can be combined with more sophisticated techniques in case when $\mathcal{F}$ is infinite. We can namely start the algorithm by branching on forbidden induced subgraphs that are of constant size and, when their supply is exhausted, turn to some other algorithms. The following lemma provides a formalization of this concept; a graph is called $\mathcal{F}$-*free* if it does not contain any graph from $\mathcal{F}$ as an induced subgraph.

Lemma 1 ([1]) *Let $\mathcal{F}$ be a finite set of graphs and let ℓ be the maximum number of vertices in a graph from $\mathcal{F}$. Let Π be a hereditary graph class that is polynomial-time recognizable. Assume that there exists an algorithm $\mathcal{A}$ that for a given $\mathcal{F}$-free graph G on n vertices, in time $\mathcal{O}((2-\varepsilon)^n)$ finds a maximum induced subgraph of G that belongs to Π, for some $\varepsilon > 0$. Then there exists an algorithm $\mathcal{A}'$ that for a given graph G on n vertices, in time $\mathcal{O}((2-\varepsilon')^n)$ finds a maximum induced subgraph of G that is $\mathcal{F}$-free and belongs to Π, where $\varepsilon' > 0$ is a constant depending on ε and ℓ.*

Thus, for the purpose of breaking the 2^n barrier, it is sufficient to focus on the case when no constant-size forbidden induced subgraph is present in the input graph.

Exploiting a Large Substructure

Here, the general idea is to look for a large substructure in the graph that can be leveraged to design an algorithm breaking the barrier. Let us take as an example the Maximum Induced Chordal Subgraph problem, considered by Bliznets et al. [1]. Suppose that in the input graph G one can find a clique Q of size δn, for some $\delta > 0$; recall that the largest clique in a graph can be found as fast as in time $\mathcal{O}(1.2109^n)$ [10]. Then consider the following algorithm: guess, by considering $2^{n-|Q|}$ possibilities, the intersection of the optimum solution with $V(G) \setminus Q$. Then observe that, since Q is a clique, every induced cycle in G can have only at most two vertices in common with Q. Hence, the problem of optimally extending the choice on $V(G) \setminus Q$ to Q essentially boils down to solving a Vertex Cover instance on $|Q|$ vertices, which can be done in time $\mathcal{O}(1.2109^{|Q|})$. As Q constitutes a linear fraction of all the vertices, the overall running time is $\mathcal{O}(1.2109^{|Q|} \cdot 2^{n-|Q|})$, which is $\mathcal{O}((2-\varepsilon)^n)$ for some $\varepsilon > 0$ depending on δ. Thus, one can focus on the case where the largest clique in the input graph, and hence also in any maximum-sized induced chordal subgraph, has less than δn vertices.

Potential Maximal Cliques

A *potential maximal clique* (*PMC*) in a graph G is a subset of vertices that becomes a clique in some inclusion-wise minimal triangulation (By a triangulation of a graph we mean any its chordal supergraph.) of G. Fomin and Villanger in [2] observed two facts. Firstly, whenever H is an induced subgraph of G of treewidth t, then there exists a minimal triangulation TG of G that captures H in the following sense: every clique of TG intersects $V(H)$ only at a subset of some bag of a fixed width-t tree decomposition of H. Secondly, a graph G on n vertices can have only $\mathcal{O}(1.734601^n)$ PMCs, which can be enumerated in time $\mathcal{O}(1.734601^n)$. Intuitively, this means that we can effectively search the space of treewidth-t induced subgraphs of G in time $\mathcal{O}(1.734601^n \cdot n^{\mathcal{O}(t)})$ using dynamic programming. Slightly more precisely, treewidth-t induced subgraphs of G can be assembled in a dynamic programming manner using states of the form (Ω, X), where Ω is a PMC in G and X is a subset of Ω of size at most $t+1$, corresponding to $\Omega \cap V(H)$. In this manner one can obtain an algorithm with running time $\mathcal{O}(1.734601^n \cdot n^{\mathcal{O}(t)})$ for finding the maximum induced treewidth-t subgraph, which in particular implies a $\mathcal{O}(1.734601^n)$-time algorithm for Maximum Induced Forest, equivalent to Feedback Vertex Set. Recently, Fomin et al. [5] extended this framework to encapsulate also problems where the induced subgraph H is in addition required to satisfy a property expressible in *Monadic Second-Order Logic*.

Recommended Reading

1. Bliznets I, Fomin FV, Pilipczuk M, Villanger Y (2013) Largest chordal and interval subgraphs faster than 2^n. In: Bodlaender HL, Italiano GF (eds) ESA, Sophia Antipolis. Lecture Notes in Computer Science, vol 8125. Springer, pp 193–204
2. Fomin FV, Villanger Y (2010) Finding induced subgraphs via minimal triangulations. In: Marion JY, Schwentick T (eds) STACS, Nancy. LIPIcs, vol 5. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, pp 383–394
3. Fomin FV, Gaspers S, Kratsch D, Liedloff M, Saurabh S (2010) Iterative compression and exact algorithms. Theor Comput Sci 411(7–9):1045–1053
4. Fomin FV, Todinca I, Villanger Y (2011) Exact algorithm for the maximum induced planar subgraph problem. In: Demetrescu C, Halldórsson MM (eds) ESA, Saarbrücken. Lecture notes in computer science, vol 6942. Springer, pp 287–298
5. Fomin FV, Todinca I, Villanger Y (2014) Large induced subgraphs via triangulations and CMSO. In: Chekuri C (ed) SODA, Portland. SIAM, pp 582–583
6. Gaspers S, Kratsch D, Liedloff M (2012) On independent sets and bicliques in graphs. Algorithmica 62(3–4):637–658
7. Gupta S, Raman V, Saurabh S (2012) Maximum r-regular induced subgraph problem: fast exponential algorithms and combinatorial bounds. SIAM J Discr Math 26(4):1758–1780
8. Pilipczuk M, Pilipczuk M (2012) Finding a maximum induced degenerate subgraph faster than 2^n. In: Thilikos DM, Woeginger GJ (eds) IPEC, Ljubljana. Lecture notes in computer science, vol 7535. Springer, pp 3–12
9. Raman V, Saurabh S, Sikdar S (2007) Efficient exact algorithms through enumerating maximal independent sets and other techniques. Theory Comput Syst 41(3):563–587
10. Robson JM (1986) Algorithms for maximum independent sets. J Algorithms 7(3):425–440

Exact Algorithms for *k* SAT Based on Local Search

Kazuo Iwama
Computer Engineering, Kyoto University, Sakyo, Kyoto, Japan
School of Informatics, Kyoto University, Sakyo, Kyoto, Japan

Keywords

CNF satisfiability; Exponential-time algorithm; Local search

Years and Authors of Summarized Original Work

1999; Schöning

Problem Definition

The CNF satisfiability problem is to determine, given a CNF formula F with n variables, whether or not there exists a satisfying assignment for F. If each clause of F contains at most k literals, then F is called a k-CNF formula and the problem is called k-SAT, which is one of the most fundamental NP-complete problems. The trivial algorithm is to search 2^n 0/1-assignments for the n variables. But since [6], several algorithms which run significantly faster than this $O(2^n)$ bound have been developed. As a simple exercise, consider the following straightforward algorithm for 3-SAT, which gives us an upper bound of 1.913^n: choose an arbitrary clause in F, say, $(x_1 \vee \overline{x_2} \vee x_3)$. Then generate seven new formulas by substituting to these x_1, x_2, and x_3 all the possible values except $(x_1, x_2, x_3) = (0, 1, 0)$ which obviously unsatisfies F. Now one can check the satisfiability of these seven formulas and conclude that F is satisfiable iff at least one of them is satisfiable. (Let $T(n)$ denote the time complexity of this algorithm. Then one can get the recurrence $T(n) \leq 7 \times T(n-3)$ and the above bound follows.)

Key Results

In the long history of k-SAT algorithms, the one by Schöning [11] is an important breakthrough. It is a standard local search and the algorithm itself is not new (see, e.g., [7]). Suppose that y is the current assignment (its initial value is selected uniformly at random). If y is a satisfying assignment, then the algorithm answers yes and terminates. Otherwise, there is at least one clause whose three literals are all false under y. Pick an arbitrary such clause and select one of the three literals in it at random. Then flip (true to false and vice versa) the value of that variable, replace y with that new assignment, and then repeat the same procedure. More formally:

```
SCH(CNF formula $F$, integer $I$)
repeat $I$ times
$y$ = uniformly random vector $\in \{0,1\}^n$
$z$ = RandomWalk($F, y$);
if $z$ satisfies $F$
then output($z$); exit;
end
output('Unsatisfiable');
RandomWalk(CNF formula $G(x_1, x_2, \ldots, x_n)$,
  assignment $y$);
$y' = y$;
for $3n$ times
if $y'$ satisfies $G$
then return $y'$; exit;
$C \leftarrow$ an arbitrary clause of $G$ that is not satisfied
  by $y'$;
Modify $y'$ as follows:
select one literal of $C$ uniformly at random and
flip the assignment to this literal;
end
return $y'$
```

Schöning's analysis of this algorithm is very elegant. Let $d(a,b)$ denote the Hamming distance between two binary vectors (assignments) a and b. For simplicity, suppose that the formula F has only one satisfying assignment y^* and the current assignment y is far from y^* by Hamming distance d. Suppose also that the currently false clause C includes three variables, x_i, x_j, and x_k. Then y and y^* must differ in at least one of these three variables. This means that if the value of x_i, x_j, or x_k is flipped, then the new assignment gets closer to y^* by Hamming distance one with probability at least 1/3. Also, the new assignment gets farther by Hamming distance one with probability at most 2/3. The argument can be generalized to the case that F has multiple satisfying assignments. Now here comes the key lemma:

Lemma 1 *Let F be a satisfiable formula and y^* be a satisfying assignment for F. For each assignment y, the probability that a satisfying assignment (that may be different from y^*) is found by **RandomWalk** (F, y) is at least $(1/(k-1))^{d(y,y*)}/p(n)$, where $p(n)$ is a polynomial in n.*

By taking the average over random initial assignments, the following theorem follows:

Theorem 1 *For any satisfiable formula F on n variables, the success probability of **RandomWalk** (F, y) is at least $(k/2(k-1))^n/p(n)$ for some polynomial p. Thus, by setting $I = (2(k-1)/k)^n \cdot p(n)$, **SCH** finds a satisfying assignment with high probability. When $k = 3$, this value of I is $O(1.334^n)$.*

Applications

The Schöning's result has been improved by a series of papers [1, 3, 9] based on the idea of [3]. Namely, RandomWalk is combined with the (polynomial time) 2SAT algorithm, which makes it possible to choose better initial assignments. For derandomization of **SCH**, see [2]. Iwama and Tamaki [4] developed a nontrivial combination of **SCH** with another famous, backtrack-type algorithm by [8], resulting in the then fastest algorithm with $O(1.324^n)$ running time. The current fastest algorithm is due to [10], which is based on the same approach as [4] and runs in time $O(1.32216^n)$.

Open Problems

k-SAT is probably the most popular NP-complete problem for which numerous researchers are competing for its fastest algorithm. Thus, improving its time bound is always a good research target.

Experimental Results

AI researchers have also been very active in SAT algorithms including local search; see, e.g., [5].

Cross-References

▶ Exact Algorithms for General CNF SAT
▶ Random Planted 3-SAT

Recommended Reading

1. Baumer S, Schuler R (2003) Improving a probabilistic 3-SAT algorithm by dynamic search and independent clause pairs. ECCC TR03-010. Also presented at SAT
2. Dantsin E, Goerdt A, Hirsch EA, Kannan R, Kleinberg J, Papadimitriou C, Raghavan P, Schöning U (2002) A deterministic $(2 - 2/(k + 1))^n$ algorithm for k-SAT based on local search. Theor Comput Sci 289(1):69–83
3. Hofmeister T, Schöning U, Schuler R, Watanabe O (2002) Probabilistic 3-SAT algorithm further improved. In: Proceedings 19th symposium on theoretical aspects of computer science, Juan-les-Pins. LNCS, vol 2285, pp 193–202
4. Iwama K, Tamaki S (2004) Improved upper bounds for 3-SA T. In: Proceedings 15th annual ACM-SIAM symposium on discrete algorithms, New Orleans, pp 321–322
5. Kautz H, Selman B (2003) Ten challenges redux: recent progress in propositional reasoning and search. In: Proceedings 9th international conference on principles and practice of constraint programming, Kinsale, pp 1–18
6. Monien B, Speckenmeyer E (1985) Solving satisfiability in less than 2^n steps. Discret Appl Math 10:287–295
7. Papadimitriou CH (1991) On selecting a satisfying truth assignment. In: Proceedings 32nd annual symposium on foundations of computer science, San Juan, pp 163–169
8. Paturi R, Pudlák P, Saks ME, Zane F (1998) An improved exponential-time algorithm for k-SAT. In: Proceedings 39th annual symposium on foundations of computer science, Palo Alto, pp 628–637; J ACM 52(3):337–364 (2006)
9. Rolf D (2003) 3-SAT $\in$ *RTIME*$(O(1.32793^n))$. ECCC TR03-054
10. Rolf D (2006) Improved bound for the PPSZ/Schöning-algorithm for 3-SAT. J Satisf Boolean Model Comput 1:111–122
11. Schöning U (1999) A probabilistic algorithm for k-SAT and constraint satisfaction problems. In: Proceedings 40th annual symposium on foundations of computer science, New York, pp 410–414

Exact Algorithms for Maximum Independent Set

Fabrizio Grandoni
IDSIA, USI-SUPSI, University of Lugano, Lugano, Switzerland

Keywords

Exact exponential algorithms; Maximum independent set

Years and Authors of Summarized Original Work

1977; Tarjan, Trojanowski
1985; Robson
1999; Beigel
2009; Fomin, Grandoni, Kratsch

Problem Definition

Let $G = (V, E)$ be an n-node undirected, simple graph without loops. A set $I \subseteq V$ is called an *independent set* of G if the nodes of I are pairwise not adjacent. The *maximum independent set* (MIS) problem asks to determine the maximum cardinality $\alpha(G)$ of an independent set of G. MIS is one of the best studied NP-hard problems.

We will need the following notation. The (open) *neighborhood* of a vertex v is $N(v) = \{u \in V : uv \in E\}$, and its *closed neighborhood* is $N[v] = N(v) \cup \{v\}$. The degree $\deg(v)$ of v is $|N(v)|$. For $W \subseteq V$, $G[W] = (W, E \cap \binom{W}{2})$ is the *graph induced* by W. We let $G - W = G[V - W]$.

Key Results

A very simple algorithm solves MIS (exactly) in $O^*(2^n)$ time: it is sufficient to enumerate all the subsets of nodes, check in polynomial time whether each subset is an independent set or not, and return the maximum cardinality independent set. We recall that the O^* notation suppresses polynomial factors in the input size. However, much faster (though still exponential-time) algorithms are known. In more detail, there exist algorithms that solve MIS in worst-case time $O^*(c^n)$ for some constant $c \in (1,2)$. In this section, we will illustrate some of the most relevant techniques that have been used in the design and analysis of exact MIS algorithms. Due to space constraints, our description will be slightly informal (please see the references for formal details).

Bounding the Size of the Search Tree

All the nontrivial exact MIS algorithms, starting with [7], are recursive branching algorithms. As an illustration, consider the following simple MIS algorithm `Alg1`. If the graph is empty, output $\alpha(G) = 0$ (base instance). Otherwise, choose any node v of maximum degree, and output

$$\alpha(G) = \max\{\alpha(G - \{v\}), 1 + \alpha(G - N[v])\}.$$

Intuitively, the subgraph $G - \{v\}$ corresponds to the choice of not including v in the independent set (v is *discarded*), while the subgraph $G - N[v]$ to the choice of including v in the independent set (v is *selected*). Observe that, when v is selected, the neighbors of v have to be discarded. We will later refer to this branching as a *standard branching*.

The running time of the above algorithm, and of branching algorithms more in general, can be bounded as follows. The recursive calls induce a *search tree*, where the root is the input instance and the leaves are base instances (that can be solved in polynomial time). Observe that each branching step can be performed in polynomial time (excluding the time needed to solve subproblems). Furthermore, the height of the search tree is bounded by a polynomial. Therefore, the running time of the algorithm is bounded by $O^*(L(n))$, where $L(n)$ is the maximum number of leaves of any search tree that can be generated by the considered algorithm on an input instance with n nodes. Let us assume that $L(n) \leq c^n$ for some constant $c \geq 1$. When we branch at node v, we generate two subproblems containing $n-1$ and $n - |N[v]|$ nodes, respectively. Therefore, c has to satisfy $c^n \geq c^{n-1} + c^{n-|N[v]|}$. Assuming pessimistically $|N[v]| = 1$, one obtains $c^n \geq 2c^{n-1}$ and therefore $c \geq 2$. We can conclude that the running time of the algorithm is $O^*(2^n)$. Though the running time of `Alg1` does not improve on exhaustive search, much faster algorithms can be obtained by branching in a more careful way and using a similar type of analysis. This will be discussed in the next subsections.

Refined Branching Rules

Several refined branching rules have been developed for MIS. Let us start with some *reduction rules*, which reduce the problem without branching (alternatively, by branching on a single subproblem). An isolated node v can be selected w.l.o.g.:

$$\alpha(G) = 1 + \alpha(G - N[v]).$$

Observe that if $N[u] \subseteq N[v]$, then node v can be discarded w.l.o.g. (*dominance*):

$$\alpha(G) = \alpha(G - \{v\}).$$

This rule implies that nodes of degree 1 can always be selected.

Suppose that we branch at a node v, and in the branch where we discard v we select exactly one of its neighbors, say w. Then by replacing w with v, we obtain a solution of the same cardinality including v: this means that the branch where we select v has to provide the optimal solution. Therefore, we can assume w.l.o.g. that the optimal solution either contains v or at least 2 of its neighbors. This idea is exploited in the *folding* operation [1], which we next illustrate only in the case of degree-2 nodes. Let $N[v] = \{w_1, w_2\}$. Remove $N[v]$. If $w_1w_2 \notin E$, create a node v' and add edges between v' and nodes in $N(w_1) \cup N(w_2) - \{v\}$. Let $G_{\text{fold}}(v)$ be the resulting graph. Then, one has

$$\alpha(G) = 1 + \alpha(G_{\text{fold}}(v)).$$

Intuitively, including node v' in the optimal solution to $G_{\text{fold}}(v)$ corresponds to selecting both w_1 and w_2, while discarding v' corresponds to selecting v.

Let `Alg2` be the algorithm that exhaustively applies the mentioned reduction rules and then performs a standard branching on a node of maximum degree. Reduction rules reduce the number of nodes at least by 1; hence, we have the constraint $c^n \geq c^{n-1}$. If we branch at node v, $\deg(v) \geq 3$. This gives $c^n \geq c^{n-1} + c^{n-4}$, which is satisfied by $c \geq 1.380\ldots$. Hence, the running time is in $O^*(1.381^n)$.

Let us next briefly sketch some other useful ideas that lead to refined branchings. A *mirror* [3] of a node v is a node u at distance 2 from v such that $N(v) - N(u)$ induces a clique. By the above discussion, if we branch by discarding v, we can assume that we select at least two neighbors of v and therefore we have also to discard the mirrors $M(v)$ of v. In other terms, we can use the refined branching

$$\alpha(G) = \max\{\alpha(G - \{v\} - M(v)), 1 + \alpha(G - N[v])\}.$$

A *satellite* [5] of a node v is a node u at distance 2 from v such that there exists a node $u' \in N(v) \cap N(u)$ that satisfies $N[u'] - N[v] = \{u\}$. Observe that if an optimal solution discards u, then we can discard v as well by dominance since $N[u'] \subseteq N[v]$ in $G - \{u\}$. Therefore, we can assume that in the branch where we select v, we also select its satellites $S(v)$. In other terms,

$$\alpha(G) = \max\{\alpha(G - \{v\}), 1 + |S(v)| + \alpha(G - N[v] - \cup_{u \in S(v)} N[u])\}.$$

Another useful trick [4] is to branch on nodes that form a *small separator* (of size 1 or 2 in the graph), hence isolating two or more connected components that can be solved independently (see also [2, 5]).

Measure and Conquer

Above we always used the number n of nodes as a *measure* of the size of subproblems. As observed in [3], much tighter running time bounds can be achieved by using smarted measures. As an illustration, we will present a refined bound on the running time of `Alg2`.

Let us measure the size of subproblems with the number n_3 of nodes of degree at least 3 (*large* nodes). Observe that, when $n_3 = 0$, G is a collection of isolated nodes, paths, and cycles. Therefore, in that case, `Alg2` only applies reduction rules, hence solving the problem in polynomial time. In other terms, $L(n_3) = L(0) = 1$ in this case. If the algorithm applies any reduction rule, the number of large nodes cannot increase and we obtain the trivial inequality $c^{n_3} \geq c^{n_3}$. Suppose next that `Alg2` performs a standard branching at a node v. Note that at this point all nodes in the graph are large. If $\deg(v) \geq 4$, then we obtain the inequality $c^{n_3} \geq c^{n_3-1} + c^{n_3-5}$ which is satisfied by $c \geq 1.324\ldots$. Otherwise ($\deg(v) = 3$), observe that the neighbors of v have degree 3 in G and at most 2 in $G - \{v\}$. Therefore, the number of large nodes is at most $n_3 - 4$ in both subproblems $G - \{v\}$ and $G - N[v]$. This gives the inequality $c^{n_3} \geq 2c^{n_3-4}$ which is satisfied by $c \geq 2^{1/4} < 1.1893$. We can conclude that the running time of the algorithm is in $O^*(1.325^n)$. In [3], each node is assigned a weight which is a growing function of its degree, and the measure is the sum of node weights (a similar measure is used also in [2, 5]).

In [2], it is shown how to use a fast MIS algorithm for graphs of maximum degree Δ to derive faster MIS algorithms for graphs of maximum degree $\Delta + 1$. Here the measure used in the analysis is a combination of the number of nodes and edges.

Memorization

So far we described algorithms with polynomial space complexity. *Memorization* [6] is a technique to speed up exponential-time branching algorithms at the cost of an exponential space complexity. The basic idea is to store the optimal solution to subproblems in a proper (exponential size) data structure. Each time a new subproblem is generated, one first checks (in polynomial time) whether that subproblem was already solved before. This way one avoids to solve the same subproblem several times.

In order to illustrate this technique, it is convenient to consider the variant `Alg3` of `Alg2` where we do not apply folding. This way, each subproblem corresponds to some induced subgraph $G[W]$ of the input graph. We will also use the standard measure though memorization is compatible with measure and conquer. By adapting the analysis of `Alg2`, one obtains the constraint $c^n \geq c^{n-1} + c^{n-3}$ and hence a running time of $O^*(1.466^n)$. Next, consider the variant `Alg3mem` of `Alg3` where we apply memorization. Let $L_k(n)$ be the maximum number of subproblems on k nodes generated by `Alg3mem` starting from an instance with n nodes. A slight adaptation of the standard analysis shows that $L_k(n) \leq 1.466^{n-k}$. However, since there are at most $\binom{n}{k}$ induced subgraphs on k nodes and we never solve the same subproblem twice, one also has $L_k(n) \leq \binom{n}{k}$. Using Stirling's formula, one obtains that the two upper bounds are roughly equal for $k = \alpha n$ and $\alpha = 0.107\ldots$. We can conclude that the running time of `Alg3mem` is in $O^*(\sum_{k=0}^{n} L_k(n)) = O^*(\sum_{k=0}^{n} \min\{1.466^{n-k}, \binom{n}{k}\}) = O^*(\max_{k=0}^{n} \min\{1.466^{n-k}, \binom{n}{k}\}) = O^*(1.466^{(1-0.107)n}) = O^*(1.408^n)$. The analysis can be refined [6] by bounding the number of *connected* induced subgraphs with k nodes in graphs of small maximum degree.

Cross-References

▸ Exact Algorithms for Dominating Set

Recommended Reading

1. Beigel R (1999) Finding maximum independent sets in sparse and general graphs. In: ACM-SIAM symposium on discrete algorithms (SODA), Baltimore, pp 856–857
2. Bourgeois N, Escoffier B, Paschos VT, van Rooij JMM (2012) Fast algorithms for max independent set. Algorithmica 62(1–2):382–415
3. Fomin FV, Grandoni F, Kratsch D (2009) A measure & conquer approach for the analysis of exact algorithms. J ACM 56(5):Article no. 25
4. Fürer M (2006) A faster algorithm for finding maximum independent sets in sparse graphs. In: Latin American theoretical informatics symposium (LATIN), Valdivia, pp 491–501
5. Kneis J, Langer A, Rossmanith P (2009) A fine-grained analysis of a simple independent set algorithm. In: Foundations of software technology and theoretical computer science (FSTTCS), Kanpur, pp 287–298
6. Robson JM (1986) Algorithms for maximum independent sets. J Algorithms 7(3):425–440
7. Tarjan R, Trojanowski A (1977) Finding a maximum independent set. SIAM J Comput 6(3):537–546

Exact Algorithms for Maximum Two-Satisfiability

Ryan Williams
Department of Computer Science, Stanford University, Stanford, CA, USA

Keywords

Max 2-SAT

Years and Authors of Summarized Original Work

2004; Williams

Problem Definition

In the maximum 2-satisfiability problem (abbreviated as MAX 2-SAT), one is given a Boolean formula in conjunctive normal form, such that each clause contains at most two literals. The task is to find an assignment to the variables of the formula such that a maximum number of clauses are satisfied.

MAX 2-SAT is a classic optimization problem. Its decision version was proved *NP*-complete by Garey, Johnson, and Stockmeyer [7], in stark contrast with 2-SAT which is solvable in linear time [2]. To get a feeling for the difficulty of the problem, the *NP*-completeness reduction is sketched here. One can transform any 3-SAT instance F into a MAX 2-SAT instance F', by replacing each clause of F such as

$$c_i = (\ell_1 \vee \ell_2 \vee \ell_3),$$

where ℓ_1, ℓ_2, and ℓ_3 are arbitrary literals, with the collection of 2-CNF clauses

$$(\ell_1), (\ell_2), (\ell_3), (c_i), (\neg\ell_1 \vee \neg\ell_2), (\neg\ell_2 \vee \neg\ell_3), (\neg\ell_1 \vee \neg\ell_3), (\ell_1 \vee c_i), (\ell_2 \vee c_i), (\ell_3 \vee c_i),$$

where c_i is a new variable. The following are true:

- If an assignment satisfies c_i, then exactly seven of the ten clauses in the 2-CNF collection can be satisfied.
- If an assignment does not satisfy c_i, then exactly six of the ten clauses can be satisfied.

If F is satisfiable then there is an assignment satisfying 7/10 of the clauses in F', and if F is not satisfiable, then no assignment satisfies more than 7/10 of the clauses in F'. Since 3-SAT reduces to MAX 2-SAT, it follows that MAX 2-SAT (as a decision problem) is *NP*-complete.

Notation

A CNF formula is represented as a set of clauses.

The letter ω denotes the smallest real number such that for all $\epsilon > 0$, n by n matrix multiplication over a field can be performed in $O(n^{\omega+\epsilon})$ field operations. Currently, it is known that $\omega < 2.373$ [4, 16]. The field matrix product of two matrices A and B is denoted by $A \times B$.

Let A and B be matrices with entries from $\mathbb{R} \cup \{\infty\}$. The *distance product* of A and B (written in shorthand as $A \circledast B$) is the matrix C defined by the formula

$$C[i,j] = \min_{k=1,\ldots,n} \{A[i,k] + B[k,j]\}.$$

A word on m's and n's: in reference to graphs, m and n denote the number of edges and the number of nodes in the graph, respectively. In reference to CNF formulas, m and n denote the number of clauses and the number of variables, respectively.

Key Result

The primary result of this entry is a procedure solving Max 2-Sat in $O(m \cdot 2^{\omega n/3})$ time. The method can be generalized to *count* the number of solutions to *any* constraint optimization problem with at most two variables per constraint. Indeed, in the same running time, one can find a Boolean assignment that maximizes any given degree-two polynomial in n variables [18, 19]. In this entry, we shall restrict attention to be Max 2-Sat, for simplicity. There are several other known exact algorithms for Max 2-Sat that are more effective in special cases, such as sparse instances [3, 8, 9, 11–13, 15, 17]. The procedure described below is the only one known (to date) that runs in c^n steps for a constant $c < 2$.

Key Idea

The algorithm gives a reduction from MAX 2-SAT to the problem MAX TRIANGLE, in which one is given a graph with integer weights on its nodes and edges, and the goal is to output a 3-cycle of maximum weight. At first, the existence of such a reduction sounds strange, as MAX TRIANGLE can be trivially solved in $O(n^3)$ time by trying all possible 3-cycles. The key is that the reduction exponentially increases the problem size, from a MAX 2-SAT instance with m clauses and n variables to a MAX TRIANGLE instance having $O(2^{2n/3})$ edges, $O(2^{n/3})$ nodes, and weights in the range $\{-m, \ldots, m\}$.

Note that if MAX TRIANGLE required $\Theta(n^3)$ time to solve, then the resulting MAX 2-SAT algorithm would take $\Theta(2^n)$ time, rendering the above reduction pointless. However, it turns out that the brute-force search of $O(n^3)$ for MAX TRIANGLE is not the best one can do: using fast matrix multiplication, there is an algorithm for MAX TRIANGLE that runs in $O(Wn^{\omega})$ time on graphs with weights in the range $\{-W, \ldots, W\}$.

Main Algorithm

First, a reduction from MAX 2-SAT to MAX TRIANGLE is described, arguing that each triangle of weight K in the resulting graph is in one-to-one correspondence with an assignment that satisfies K clauses of the MAX 2-SAT instance. Let a, b be reals, and let $\mathbb{Z}[a,b] := [a,b] \cap \mathbb{Z}$.

Lemma 1 *If* MAX TRIANGLE *on graphs with n nodes and weights in $\mathbb{Z}[-W,W]$ is solvable in $O(f(W) \cdot g(n))$ time, for polynomials f and g, then* MAX 2-SAT *is solvable in $O(f(m) \cdot g(2^{n/3}))$ time, where m is the number of clauses and n is the number of variables.*

Proof Let C be a given 2-CNF formula. Assume without loss of generality that n is divisible by 3. Let F be an instance of MAX 2-SAT. Arbitrarily partition the n variables of F into three sets P_1, P_2, P_3, each having $n/3$ variables. For each P_i, make a list L_i of all $2^{n/3}$ assignments to the variables of P_i.

Define a graph $G = (V, E)$ with $V = L_1 \cup L_2 \cup L_3$ and $E = \{(u,v) | u \in P_i, v \in P_j, i \neq j\}$. That is, G is a complete tripartite graph with $2^{n/3}$ nodes in each part, and each node in G corresponds to an assignment to $n/3$ variables in C. Weights are placed on the nodes and edges of G as follows. For a node v, define $w(v)$ to be the number of clauses that are satisfied by the partial assignment denoted by v. For each edge $\{u, v\}$, define $w(\{u,v\}) = -W_{uv}$, where W_{uv} is the number of clauses that are satisfied by *both* u and v.

Define the weight of a triangle in G to be the total sum of all weights and nodes in the triangle.

Claim 1 There is a one-to-one correspondence between the triangles of weight K in G and the variable assignments satisfying exactly K clauses in F.

Proof Let a be a variable assignment. Then there exist unique nodes $v_1 \in L_1, v_2 \in L_2$, and $v_3 \in L_3$ such that a is precisely the concatenation of v_1, v_2, v_3 as assignments. Moreover, any triple of nodes $v_1 \in L_1, v_2 \in L_2$, and $v_3 \in L_3$ corresponds to an assignment. Thus, there is a one-to-one correspondence between triangles in G and assignments to F.

The number of clauses satisfied by an assignment is exactly the weight of its corresponding triangle. To see this, let $T_a = \{v_1, v_2, v_3\}$ be the triangle in G corresponding to assignment a. Then

$$\begin{aligned} w(T_a) &= w(v_1) + w(v_2) + w(v_3) + w(\{v_1, v_2\}) \\ &\quad + w(\{v_2, v_3\}) + w(\{v_1, v_3\}) \\ &= \sum_{i=1}^{3} |\{c \in F | v_i \text{ satisfies } F\}| \\ &\quad - \sum_{i,j:i \neq j} |\{c \in F | v_i \text{ and } v_j \text{ satisfy } F\}| \\ &= |\{c \in F | a \text{ satisfies } F\}|, \end{aligned}$$

where the last equality follows from the inclusion-exclusion principle.

Notice that the number of nodes in G is $3 \cdot 2^{n/3}$, and the absolute value of any node and edge weight is m. Therefore, running a MAX TRIANGLE algorithm on G, a solution to MAX 2-SAT, is obtained in $O(f(m) \cdot g(3 \cdot 2^{n/3}))$, which is $O(f(m) \cdot g(2^{n/3}))$ since g is a polynomial. This completes the proof of Lemma 1.

Next, a procedure is described for finding a maximum triangle faster than brute-force search, using fast matrix multiplication. Alon, Galil, and Margalit [1] (following Yuval [22]) showed that the distance product for matrices with entries drawn from $\mathbb{Z}[-W,W]$ can be computed using fast matrix multiplication as a subroutine.

Theorem 1 (Alon, Galil, Margalit [1]) *Let A and B be $n \times n$ matrices with entries from $\mathbb{Z}[-W,W] \cup \{\infty\}$. Then $A \circledast B$ can be computed in $O(Wn^{\omega} \log n)$ time.*

Proof (Sketch) One can replace ∞ entries in A and B with $2W + 1$ in the following. Define matrices A' and B', where

$$A'[i,j] = x^{3W - A[i,j]}, \; B'[i,j] = x^{3W - B[i,j]},$$

and x is a variable. Let $C = A' \times B'$. Then

$$C[i,j] = \sum_{k=1}^{n} x^{6W - A[i,k] - B[k,j]}.$$

The next step is to pick a number x that makes it easy to determine, from the sum of arbitrary powers of x, the largest power of x appearing in the sum; this largest power immediately gives the minimum $A[i,k] + B[k,j]$. Each $C[i,j]$ is a polynomial in x with coefficients from $\mathbb{Z}[0,n]$. Suppose each $C[i,j]$ is evaluated at $x = (n+1)$. Then each entry of $C[i,j]$ can be seen as an $(n+1)$-ary number, and the position of this number's most significant digit gives the minimum $A[i,k] + B[k,j]$.

In summary, $A \otimes_d B$ can be computed by constructing

$$A'[i,j] = (n+1)^{3W - A[i,j]},$$

$$B'[i,j] = (n+1)^{3W - B[i,j]}$$

in $O(W \log n)$ time per entry, computing $C = A' \times B'$ in $O(n^{\omega} \cdot (W \log n))$ time (as the sizes of the entries are $O(W \log n)$), then extracting the minimum from each entry of C, in $O(n^2 \cdot W \log n)$ time. Note if the minimum for an entry $C[i,j]$ is at least $2W+1$, then $C[i,j] = \infty$.

Using the fast distance product algorithm, one can solve Max Triangle faster than brute force. The following is based on an algorithm by Itai and Rodeh [10] for detecting if an unweighted graph has a triangle in less than n^3 steps. The result can be generalized to *counting* the number of k-*cliques*, for arbitrary $k \geq 3$. (To keep the presentation simple, the counting result is omitted. Concerning the k-clique result, there is unfortunately no asymptotic runtime benefit from using a k-clique algorithm instead of a triangle algorithm, given the current best algorithms for these problems.)

Theorem 2 Max Triangle *can be solved in* $O(Wn^{\omega} \log n)$, *for graphs with weights drawn from* $\mathbb{Z}[-W,W]$.

Proof First, it is shown that a weight function on nodes and edges can be converted into an equivalent weight function with weights on only edges. Let w be the weight function of G, and redefine the weights to be:

$$w'(\{u,v\}) = \frac{w(u)+w(v)}{2} + w(\{u,v\}),$$

$$w'(u) = 0.$$

Note the weight of a triangle is unchanged by this reduction.

The next step is to use a fast distance product to find a maximum weight triangle in an edge-weighted graph of n nodes. Construe the vertex set of G as the set $\{1,\ldots,n\}$. Define A to be the $n \times n$ matrix such that $A[i,j] = -w(\{i,j\})$ if there is an edge $\{i,j\}$, and $A[i,j] = \infty$ otherwise. The claim is that there is a triangle through node i of weight at least K if and only if $(A \circledast A \circledast A)[i,i] \leq -K$. This is because $(A \circledast A \circledast A)[i,i] \leq -K$ if and only if there are distinct j and k such that $\{i,j\},\{j,k\},\{k,i\}$ are edges and $A[i,j] + A[j,k] + A[k,i] \leq -K$, i.e., $w(\{i,j\}) + w(\{j,k\}) + w(\{k,i\}) \geq K$.

Therefore, by finding an i such that $(A \circledast A \circledast A)[i,i]$ is minimized, one obtains a node i contained in a maximum triangle. To obtain the actual triangle, check all m edges $\{j,k\}$ to see if $\{i,j,k\}$ is a triangle.

Theorem 3 Max 2-Sat *can be solved in* $O(m \cdot 1.732^n)$ *time.*

Proof Given a set of clauses C, apply the reduction from Lemma 1 to get a graph G with $O(2^{n/3})$ nodes and weights from $\mathbb{Z}[-m,m]$. Apply the algorithm of Theorem 2 to output a max triangle in G in $O(m \cdot 2^{\omega n/3} \log(2^{n/3})) = O(m \cdot 1.732^n)$ time, using the $O(n^{2.376})$ matrix multiplication of Coppersmith and Winograd [4].

Applications

By modifying the graph construction, one can solve other problems in $O(1.732^n)$ time, such as Max Cut, Minimum Bisection, and Sparsest Cut. In general, any constraint optimization problem for which each constraint has at most two variables can be solved faster using the above approach. For more details, see [18] and the survey by Woeginger [21]. Techniques similar to the above algorithm have also been used by Dorn

[6] to speed up dynamic programming for some problems on planar graphs (and in general, graphs of bounded branchwidth).

Open Problems

- Improve the space usage of the above algorithm. Currently, $\Theta(2^{2n/3})$ space is needed. A very interesting open question is if there is a $O(1.99^n)$ time algorithm for MAX 2-SAT that uses only *polynomial* space. This question would have a positive answer if one could find an algorithm for solving the k-CLIQUE problem that uses polylogarithmic space and $n^{k-\delta}$ time for some $\delta > 0$ and $k \geq 3$.
- Find a faster-than-2^n algorithm for MAX 2-SAT that does not require fast matrix multiplication. The fast matrix multiplication algorithms have the unfortunate reputation of being impractical.
- Generalize the above algorithm to work for MAX k-SAT, where k is any positive integer. The current formulation would require one to give an efficient algorithm for finding a small hyperclique in a hypergraph. However, no general results are known for this problem. It is conjectured that for all $k \geq 2$, MAX k-SAT is in $\tilde{O}(2^{n(1-\frac{1}{k+1})})$ time, based on the conjecture that matrix multiplication is in $n^{2+o(1)}$ time [17].

Cross-References

► All Pairs Shortest Paths via Matrix Multiplication

Recommended Reading

1. Alon N, Galil Z, Margalit O (1997) On the exponent of the all-pairs shortest path problem. J Comput Syst Sci 54:255–262
2. Aspvall B, Plass MF, Tarjan RE (1979) A linear-time algorithm for testing the truth of certain quantified boolean formulas. Inf Proc Lett 8(3):121–123
3. Bansal N, Raman V (1999) Upper bounds for Max Sat: further improved. In: Proceedings of ISAAC, Chennai. LNCS, vol 1741. Springer, Berlin, pp 247–258
4. Coppersmith D, Winograd S (1990) Matrix multiplication via arithmetic progressions. JSC 9(3):251–280
5. Dantsin E, Wolpert A (2006) Max SAT for formulas with constant clause density can be solved faster than in $O(2^n)$ time. In: Proceedings of the 9th international conference on theory and applications of satisfiability testing, Seattle. LNCS, vol 4121. Springer, Berlin, pp 266–276
6. Dorn F (2006) Dynamic programming and fast matrix multiplication. In: Proceedings of 14th annual European symposium on algorithms, Zurich. LNCS, vol 4168. Springer, Berlin, pp 280–291
7. Garey M, Johnson D, Stockmeyer L (1976) Some simplified NP-complete graph problems. Theor Comput Sci 1:237–267
8. Gramm J, Niedermeier R (2000) Faster exact solutions for Max2Sat. In: Proceedings of CIAC. LNCS, vol 1767, Rome. Springer, Berlin, pp 174–186
9. Hirsch EA (2000) A $2^{m/4}$-time algorithm for Max 2-SAT: corrected version. Electronic colloquium on computational complexity report TR99-036
10. Itai A, Rodeh M (1978) Finding a minimum circuit in a graph. SIAM J Comput 7(4):413–423
11. Kneis J, Mölle D, Richter S, Rossmanith P (2005) Algorithms based on the treewidth of sparse graphs. In: Proceedings of workshop on graph theoretic concepts in computer science, Metz. LNCS, vol 3787. Springer, Berlin, pp 385–396
12. Kojevnikov A, Kulikov AS (2006) A new approach to proving upper bounds for Max 2-SAT. In: Proceedings of the seventeenth annual ACM-SIAM symposium on discrete algorithms, Miami, pp 11–17
13. Mahajan M, Raman V (1999) Parameterizing above guaranteed values: MAXSAT and MAXCUT. J Algorithms 31(2):335–354
14. Niedermeier R, Rossmanith P (2000) New upper bounds for maximum satisfiability. J Algorithms 26:63–88
15. Scott A, Sorkin G (2003) Faster algorithms for MAX CUT and MAX CSP, with polynomial expected time for sparse instances. In: Proceedings of RANDOM-APPROX 2003, Princeton. LNCS, vol 2764. Springer, Berlin, pp 382–395
16. Vassilevska Williams V (2012) Multiplying matrices faster than Coppersmith-Winograd. In: Proceedings of the 44th annual ACM symposium on theory of computing, New York, pp 887–898
17. Williams R (2004) On computing k-CNF formula properties. In: Theory and applications of satisfiability testing. LNCS, vol 2919. Springer, Berlin, pp 330–340
18. Williams R (2005) A new algorithm for optimal 2-constraint satisfaction and its implications. Theor Comput Sci 348(2–3):357–365
19. Williams R (2007) Algorithms and resource requirements for fundamental problems. PhD thesis, Carnegie Mellon University

20. Woeginger GJ (2003) Exact algorithms for NP-hard problems: a survey. In: Combinatorial optimization – Eureka! You shrink! LNCS, vol 2570. Springer, Berlin, pp 185–207
21. Woeginger GJ (2004) Space and time complexity of exact algorithms: some open problems. In: Proceedings of 1st international workshop on parameterized and exact computation (IWPEC 2004), Bergen. LNCS, vol 3162. Springer, Berlin, pp 281–290
22. Yuval G (1976) An algorithm for finding all shortest paths using $N^{2.81}$ infinite-precision multiplications. Inf Process Lett 4(6):155–156

Exact Algorithms for Treewidth

Ioan Todinca
INSA Centre Val de Loire, Universite d'Orleans, Orléans, France

Keywords

Extremal combinatorics; Potential maximal cliques; Treewidth

Years and Authors of Summarized Original Work

2008; Fomin, Kratsch, Todinca, Villanger
2012; Bodlaender, Fomin, Koster, Kratsch, Thilikos
2012; Fomin, Villanger

Problem Definition

The *treewidth* parameter intuitively measures whether the graph has a "treelike" structure. Given an undirected graph $G = (V, E)$, a *tree decomposition* of G is a pair $(\mathcal{X}, T)$, where $T = (I, F)$ is a tree and $\mathcal{X} = \{X_i \mid i \in I\}$ is a collection of subsets of V called *bags* satisfying:

1. $\bigcup_{i\in I} X_i = V$,
2. For each edge uv of G, there is a bag X_i containing both endpoints,
3. For all $v \in V$, the set $\{i \in I \mid v \in X_i\}$ induces a connected subtree of T.

The *width* of a tree decomposition $(\mathcal{X}, T)$ is the size of its largest bag, minus one. The *treewidth* of G, denoted by $\text{tw}(G)$, is the minimum width over all possible tree decompositions. One can easily observe that n-vertex graphs have treewidth at most $n - 1$ and that the graphs of treewidth at most one are exactly the forests.

Given a graph G and a number k, the TREEWIDTH problem consists in deciding if $\text{tw}(G) \leq k$. Arnborg, Corneil, and Proskurowski show that the problem is NP-hard [1]. On the positive side, Bodlaender [2] gives an algorithm solving the problem in time $2^{\mathcal{O}(k^3)}n$. Bouchitté and Todinca [4,5] prove that the problem is polynomial on classes of graphs with polynomially many minimal separators, with an algorithm based on the notion of *potential maximal clique*. This latter technique is also employed by several exact, moderately exponential algorithms for TREEWIDTH.

Key Results

TREEWIDTH can be solved in $\mathcal{O}^*(2^n)$ time by adapting the $\mathcal{O}(n^k)$ algorithm of Arnborg et al. [1] or the Held-Karp technique initially designed for the TRAVELING SALESMAN problem [12]. (We use here the $\mathcal{O}^*$ notation that suppresses polynomial factors.) Fomin et al. [9] break this "natural" 2^n barrier with an algorithm running in time $\mathcal{O}^*(1.8135^n)$, using the same space complexity. Bodlaender et al. [3] present a polynomial-space algorithm running in $\mathcal{O}^*(2.9512^n)$ time. A major improvement for both results is due to Fomin and Villanger [8].

Theorem 1 ([8]) *The* TREEWIDTH *problem can be solved in $\mathcal{O}^*(1.7549^n)$ time using exponential space and in $\mathcal{O}^*(2.6151^n)$ time using polynomial space.*

These algorithms use an alternative definition for treewidth. A graph $H = (V, E)$ is *chordal* or *triangulated* if it has no induced cycle with four or more vertices. It is well-known that a chordal graph has tree decompositions whose bags are exactly its maximal cliques. Given an arbitrary graph $G = (V, E)$, a chordal graph $H = (V, F)$ on the same vertex set is called a *minimal triangulation* of G if H contains G as a subgraph and no chordal subgraph of H contains G. The treewidth of G can be defined as the minimum clique size of H minus one, over all minimal triangulations H of G.

A vertex subset S of graph G is a *minimal separator* if there are two distinct components $G[C]$ and $G[D]$ of the graph $G[V \setminus S]$ such that $N_G(C) = N_G(D) = S$ ($N_G(C)$ denotes the neighborhood of C in graph G).

A vertex subset Ω of G is a *potential maximal clique* if there exists some minimal triangulation H of G such that Ω induces a maximal clique in H. Potential maximal cliques are characterized as follows [4]: Ω is a potential maximal clique of G if and only if (i) for each pair of vertices $u, v \in \Omega$, u and v are adjacent or see a same component of $G[V \setminus \Omega]$, and (ii) no component of $G[V \setminus \Omega]$ sees the whole set Ω. As an example, when G is a cycle, its minimal separators are exactly the pairs of nonadjacent vertices, and the potential maximal cliques are exactly the triples of vertices.

A *block* is pair (S, C) such that S is a minimal separator of G and $G[C]$ is a component of $G[V \setminus S]$. Denote by $R_G(S, C)$ the graph obtained from $G[S \cup C]$ by turning S into a clique, i.e., by adding all missing edges with both endpoints in S. The treewidth of G can be obtained as follows:

$$\mathrm{tw}(G) = \min_S \left(\max_C \mathrm{tw}(R(S, C)) \right) \qquad (1)$$

where the minimum is taken over all minimal separators S and the maximum is taken over all connected components $G[C]$ of $G[V \setminus S]$.

All quantities $\mathrm{tw}(R_G(S, C))$ can be computed by dynamic programming over blocks (S, C), by increasing the size of $S \cup C$. We only consider here blocks (S, C) such that $S = N_G(C)$ (see [4] for more details).

$$\mathrm{tw}(R_G(S, C)) = \min_{S \subset \Omega \subseteq S \cup C} \left(\max_{1 \le i \le p} (|\Omega| - 1, \mathrm{tw}(R_G(S_i, C_i))) \right) \qquad (2)$$

where the minimum is taken over all potential maximal cliques Ω with $S \subset \Omega \subseteq S \cup C$ and the maximum is taken over all pairs (S_i, C_i), where $G[C_i]$ is a component of $G[C \setminus \Omega]$ and $S_i = N_G(C_i)$. Let Π_G denote the set of all potential maximal cliques of graph G. It was pointed in [9] that the number of triples (S, Ω, C) like in Eq. 2 is at most $n|\Pi_G|$, which proves that TREEWIDTH can be computed in $\mathcal{O}^*(|\Pi_G|)$ time and space, if Π_G is given in the input.

Therefore, it remains to give a good upper bound for the number $|\Pi_G|$ of potential maximal cliques of G, together with efficient algorithms for listing these objects. Based on the previously mentioned characterization of potential maximal cliques, Kratsch et al. provide an algorithm listing them in time $\mathcal{O}^*(1.8135^n)$. Fomin and Villanger [8] improve this result, thanks to the following combinatorial theorem:

Theorem 2 ([8]) *Let $G = (V, E)$ be an n-vertex graph, let v be a vertex of G, and b, f be two integers. The number of vertex subsets B containing v such that $G[B]$ is connected, $|B| = b + 1$, and $|N_G(B)| = f$ is at most $\binom{b+f}{f}$.*

The elegant inductive proof also leads to an $\mathcal{O}^*(\binom{b+f}{f})$ time algorithm listing all such sets B. Eventually, the potential maximal cliques of an input graph G can be listed in $\mathcal{O}^*(1.7549^n)$ time [8]. This bound was further improved to $\mathcal{O}^*(1.7347^n)$ in [7].

In order to obtain polynomial-space algorithms for TREEWIDTH, Bodlaender et al. [3] provide a relatively simple divide-and-conquer algorithm, based on the Held-Karp approach, running in $\mathcal{O}^*(4^n)$ time. They also observe that Eq. 1 can be used for recursive, polynomial-space algorithms, by replacing the minimal separators S by *balanced* separators, in the sense that each component of $G[V \setminus S]$ contains at most $n/2$ vertices. This leads to polynomial-space algorithm with $\mathcal{O}^*(2.9512^n)$ running time.

Fomin and Villanger [8] restrict the balanced separators to a subset of the potential maximal cliques, and based on Theorem 2 they obtain, still using polynomial space, a running time of $\mathcal{O}^*(2.6151^n)$.

We refer to the book of Fomin and Kratsch [6] for more details on the TREEWIDTH problem and more generally on exact algorithms.

Applications

Exact algorithms based on potential maximal cliques have been extended to many other problems like FEEDBACK VERTEX SET, LONGEST INDUCED PATH, or MAXIMUM INDUCED SUBGRAPH WITH A FORBIDDEN PLANAR MINOR. More generally, for any constant t and any property $\mathcal{P}$ definable in counting monadic second-order logic, consider the problem of finding, in an arbitrary graph G, a maximum-size induced subgraph $G[F]$ of treewidth at most t and with property $\mathcal{P}$. This generic problem can be solved in $\mathcal{O}^*(|\Pi_G|)$ time, if Π_G is part of the input [7, 10]. Therefore, there is an algorithm in $\mathcal{O}^*(1.7347^n)$ time for the problem, significantly improving the $\mathcal{O}^*(2^n)$ time for exhaustive search.

Open Problems

Currently, the best known upper bound on the number of potential maximal cliques in n-vertex graphs is of $\mathcal{O}^*(1.7347^n)$ and does not seem to be tight [7]. Simple examples show that this bound is of at least $3^{n/3} \sim 1.4425^n$. A challenging question is to find a tight upper bound and efficient algorithms enumerating all potential maximal cliques of arbitrary graphs.

Experimental Results

Several experimental results are reported in [3], especially on an "engineered" version of the $\mathcal{O}^*(2^n)$ time and space algorithm based on the Held-Karp approach. This dynamic programming algorithm is compared with the branch and bound approach of Gogate and Dechter [11] on instances of up to 50 vertices. The results are relatively similar. Bodlaender et al. [3] also observe that the polynomial-space algorithms become too slow even for small instances.

Cross-References

▶ Kernelization, Preprocessing for Treewidth

Recommended Reading

1. Arnborg S, Corneil DG, Proskurowski A (1987) Complexity of finding embeddings in a k-tree. SIAM J Algebr Discret Methods 8(2):277–284
2. Bodlaender HL (1996) A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J Comput 25(6):1305–1317
3. Bodlaender HL, Fomin FV, Koster AMCA, Kratsch D, Thilikos DM (2012) On exact algorithms for treewidth. ACM Trans Algorithms 9(1):12
4. Bouchitté V, Todinca I (2001) Treewidth and minimum fill-in: grouping the minimal separators. SIAM J Comput 31(1):212–232
5. Bouchitté V, Todinca I (2002) Listing all potential maximal cliques of a graph. Theor Comput Sci 276(1–2):17–32
6. Fomin FV, Kratsch D (2010) Exact exponential algorithms, 1st edn. Springer, New York
7. Fomin FV, Villanger Y (2010) Finding induced subgraphs via minimal triangulations. In: Marion JY, Schwentick T (eds) STACS, Nancy. LIPIcs, vol 5. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, pp 383–394
8. Fomin FV, Villanger Y (2012) Treewidth computation and extremal combinatorics. Combinatorica 32(3):289–308
9. Fomin FV, Kratsch D, Todinca I, Villanger Y (2008) Exact algorithms for treewidth and minimum fill-in. SIAM J Comput 38(3):1058–1079
10. Fomin FV, Todinca I, Villanger Y (2014) Large induced subgraphs via triangulations and cmso. In: Chekuri C (ed) SODA, Portland. SIAM, pp 582–583
11. Gogate V, Dechter R (2004) A complete anytime algorithm for treewidth. In: Chickering DM, Halpern JY (eds) UAI, Banff. AUAI Press, pp 201–208
12. Held M, Karp RM (1962) A dynamic programming approach to the sequencing problem. J Soc Ind Appl Math 10(1):196–210

Exact Algorithms on Graphs of Bounded Average Degree

Marcin Pilipczuk
Institute of Informatics, University of Bergen, Bergen, Norway
Institute of Informatics, University of Warsaw, Warsaw, Poland

Keywords

Bounded average degree graphs; Bounded degree graphs; Chromatic number; Counting perfect matchings; Traveling salesman problem; TSP

Years and Authors of Summarized Original Work

2010; Björklund, Husfeldt, Kaski, Koivisto
2012; Björklund, Husfeldt, Kaski, Koivisto
2013; Cygan, Pilipczuk
2014; Golovnev, Kulikov, Mihajlin

Problem Definition

We focus on the following question: how an assumption on the sparsity of an input graph, such as bounded (average) degree, can help in designing exact (exponential-time) algorithms for NP-hard problems. The following classic problems are studied:

Traveling Salesman Problem Find a minimum-length Hamiltonian cycle in an input graph with edge weights.

Chromatic Number Find a minimum number k for which the vertices of an input graph can be colored with k colors such that no two adjacent vertices receive the same color.

Counting Perfect Matchings Find the number of perfect matchings in an input graph.

Key Results

The classic algorithms of Bellman [1] and Held and Karp [10] for traveling salesman problem run in $2^n n^{\mathcal{O}(1)}$ time for n-vertex graphs. Using the inclusion-exclusion principle, the chromatic number of an input graph can be determined within the same running time bound [4]. Finally, as long as counting perfect matchings is concerned, a half-century-old $2^{n/2} n^{\mathcal{O}(1)}$-time algorithm of Ryser for bipartite graphs [12] has only recently been transferred to arbitrary graphs by Björklund [2].

In all three aforementioned cases, it is widely open whether the 2^n or $2^{n/2}$ factor in the running time bound can be improved. In 2008, Björklund, Husfeldt, Kaski, and Koivisto [5,6] observed that such an improvement can be made if we restrict ourselves to bounded degree graphs. Further work of Cygan and Pilipczuk [8] and Golovnev, Kulikov, and Mihajlin [9] extended these results to graphs of bounded average degree.

Bounded Degree Graphs

Traveling Salesman Problem

Let us present the approach of Björklund, Husfeldt, Kaski, and Koivisto on the example of traveling salesman problem. Assume we are given an n-vertex edge-weighted graph G. The classic dynamic programming algorithm picks a root vertex r and then, for every vertex $v \in V(G)$ and every set $X \subseteq V(G)$ containing v and r, computes $T[X, v]$: the minimum possible length of a path in G with vertex set X that starts in r and ends in v. The running time bound $2^n n^{\mathcal{O}(1)}$ is dominated by the number of choices of the set X.

The simple, but crucial, observation is as follows: if a set X satisfies $X \cap N_G[u] = \{u\}$ for some $u \in V(G) \setminus \{r\}$, then the values $T[X, v]$ are essentially useless, as no path starting in r can visit the vertex u without visiting any neighbor of u (here $N_G[u] = N_G(u) \cup \{u\}$ stands for the closed neighborhood of u). Let us call a set $X \subseteq V(G)$ *useful* if $X \cap N_G[u] \neq \{u\}$ for every $u \in V(G) \setminus \{r\}$. The argumentation so far proved that we may skip the computation of $T[X, v]$ for all sets X that are not useful. The natural question is how many different useful sets may exist in an n-vertex graph?

Consider the following greedy procedure: initiate $A = \emptyset$ and, as long as there exists a vertex $u \in V(G)$ such that $N_G[u] \cap N_G[A] = \emptyset$, add an arbitrarily chosen vertex u to the set A. By construction, the set A satisfies the following property: for every $u_1, u_2 \in A$, we have $N_G[u_1] \cap N_G[u_2] = \emptyset$. An interesting fact is that $|A| = \Omega(n)$ for graphs of bounded degree: whenever we insert a vertex u into the set A, we cannot later insert into A any neighbor of u nor any neighbor of a neighbor of u. However, if the maximum degree of G is bounded by d, then there are at most d neighbors of u, and every such neighbor has at most $d-1$ further neighbors. Consequently, when we insert a vertex u into A, we prohibit at most $d + d(d-1) = d^2$ other vertices from being inserted into A, and $|A| \geq n/(1+d^2)$.

It is easy to adjust the above procedure such that the root vertex r does not belong to A. Observe that for every useful set X and every $u \in A$, we have $X \cap N_G[u] \neq \{u\}$ and, furthermore, the sets $N_G[u]$ for $u \in A$ are pairwise disjoint. We can think of choosing a useful set X as follows: first, for every $u \in A$, we choose the intersection $X \cap N_G[u]$ (there are $2^{|N_G[u]|} - 1$ choices, as the choice $\{u\}$ is forbidden), and, second, we choose the set $X \setminus N_G[A]$. Hence, the number of useful sets is bounded by

$$\begin{aligned}\left(\prod_{u\in A} 2^{|N_G[u]|} - 1\right) \cdot 2^{n-|N_G[A]|} &= 2^n \cdot \prod_{u\in A}\left(1 - 2^{-|N_G[u]|}\right) \leq 2^n \cdot \prod_{u\in A}(1 - 2^{-d-1}) \\ &= 2^n \cdot (1-2^{-d-1})^{|A|} \leq 2^n \cdot (1-2^{-d-1})^{\frac{n}{1+d^2}} \\ &= \left(2 \cdot \sqrt[1+d^2]{1-2^{-d-1}}\right)^n .\end{aligned}$$

Thus, for every degree bound d, there exists a constant $\varepsilon_d > 0$ such that the number of useful sets in an n-vertex graph of maximum degree d is bounded by $(2-\varepsilon_d)^n$, yielding a $(2-\varepsilon_d)^n n^{\mathcal{O}(1)}$-time algorithm for traveling salesman problem. A better dependency on d in the formula for ε_d can be obtained using a projection theorem of Chung, Frankl, Graham, and Shearer [7] (see [5]).

Chromatic Number

A similar reasoning can be performed for the problem of determining the chromatic number of an input graph. Here, it is useful to rephrase the problem as follows: find a minimum number k such that the vertex set of an input graph can be covered by k maximal independent sets; note that we do not insist that the independent sets are disjoint. Observe that if X is a set of vertices covered by one or more such maximal independent sets, we have $X \cap N_G[u] \neq \emptyset$ for every $u \in V(G)$, as otherwise the vertex u should have been included into one of the covering sets. Hence, we can call a set $X \subseteq V(G)$ *useful* if it intersects every closed neighborhood in G, and we obtain again a $(2-\varepsilon_d)^n$ bound on the number of useful sets. An important contribution of Björklund, Husfeldt, Kaski, and Koivisto [5] can be summarized as follows: using the fact that the useful sets are upward-closed (any superset of a useful set is useful as well), we can trim the fast subset convolution algorithm of [3] to consider useful sets only. Consequently, we obtain a $(2-\varepsilon_d)^n n^{\mathcal{O}(1)}$-time algorithm for computing the chromatic number of an input graph of maximum degree bounded by d.

Bounded Average Degree

Generalizing Algorithms for Bounded Degree Graphs

The above approach for traveling salesman problem has been generalized to graphs of bounded average degree by Cygan and Pilipczuk [8] using the following observation. Assume a graph G has n vertices and average degree bounded by d. Then, a simple Markov-type inequality implies that for every $\zeta > 1$ there are at most n/ζ vertices of degree larger than ζd. However, this bound cannot be tight for all values of ζ at once, and one can prove the following: if we want at most $n/(\alpha\zeta)$ vertices of degree larger than ζd for some $\alpha > 1$, then we can always find such a constant ζ of order roughly exponential in α.

An appropriate choice of α and the corresponding value of ζ allow us to partition the vertex set of an input graph into a large part of bounded degree and a very small part of unbounded degree. The extra multiplicative gap of α in the size bound allows us to hide the cost of extensive branching on the part with unbounded degree in the gains obtained by considering only (appropriately defined) useful sets in the bounded degree part.

With this line of reasoning, Cygan and Pilipczuk [8] showed that for every degree bound d, there exists a constant $\varepsilon_d > 0$ such that traveling salesman problem in graphs of bounded average degree by d can be solved in $(2 - \varepsilon_d)^n n^{\mathcal{O}(1)}$ time. It should be noted that the constant ε_d depends here doubly exponentially on d, as opposed to single-exponential dependency in the works for bounded degree graphs.

Furthermore, Cygan and Pilipczuk showed how to express the problem of counting perfect matchings in an n-vertex graph as a specific variant of a problem of counting Hamiltonian cycles in an $n/2$-vertex graph. This reduction not only gives a simpler $2^{n/2}n^{\mathcal{O}(1)}$-time algorithm for counting perfect matchings, as compared to the original algorithm of Björklund [2], but since the reduction does not increase the number of edges in a graph, it also provides a $(2 - \varepsilon_d)^{n/2}n^{\mathcal{O}(1)}$-time algorithm in the case of bounded average degree.

In a subsequent work, Golovnev, Kulikov, and Mihajlin [9] showed how to use the aforementioned multiplicative gap of α to obtain a $(2 - \varepsilon_d)^n n^{\mathcal{O}(1)}$-time algorithm for computing the chromatic number of a graph with average degree bounded by d. Furthermore, they expressed all previous algorithms as the task of determining one coefficient in a carefully chosen polynomial, obtaining polynomial space complexity without any significant loss in time complexity.

Counting Perfect Matchings in Bipartite Graphs

A somewhat different line of research concerns counting perfect matchings in bipartite graphs. Here, a $2^{n/2}n^{\mathcal{O}(1)}$-time algorithm is known for several decades [12]. Cygan and Pilipczuk presented a very simple $2^{(1-1/(3.55d))n/2}n^{\mathcal{O}(1)}$-time algorithm for this problem in graphs of average degree at most d, improving upon the previous works of Servedio and Wan [13] and Izumi and Wadayama [11]. Furthermore, this result generalizes to the problem of computing the permanent of a matrix over an arbitrary commutative ring with the number of nonzero entries linear in the dimension of the matrix.

Cross-References

▶ Exact Graph Coloring Using Inclusion-Exclusion
▶ Fast Subset Convolution

Recommended Reading

1. Bellman R (1962) Dynamic programming treatment of the travelling salesman problem. J ACM 9:61–63
2. Björklund A (2012) Counting perfect matchings as fast as ryser. In: Rabani Y (ed) SODA, Kyoto. SIAM, pp 914–921
3. Björklund A, Husfeldt T, Kaski P, Koivisto M (2007) Fourier meets möbius: fast subset convolution. In: Johnson DS, Feige U (eds) STOC, San Diego. ACM, pp 67–74
4. Björklund A, Husfeldt T, Koivisto M (2009) Set partitioning via inclusion-exclusion. SIAM J Comput 39(2):546–563
5. Björklund A, Husfeldt T, Kaski P, Koivisto M (2010) Trimmed moebius inversion and graphs of bounded degree. Theory Comput Syst 47(3):637–654
6. Björklund A, Husfeldt T, Kaski P, Koivisto M (2012) The traveling salesman problem in bounded degree graphs. ACM Trans Algorithms 8(2):18
7. Chung FRK, Frankl P, Graham RL, Shearer JB (1986) Some intersection theorems for ordered sets and graphs. J Comb Theory Ser A 43(1):23–37
8. Cygan M, Pilipczuk M (2013) Faster exponential-time algorithms in graphs of bounded average degree. In: Fomin FV, Freivalds R, Kwiatkowska MZ, Peleg D (eds) ICALP (1). Lecture notes in computer science, vol 7965. Springer, Berlin/Heidelberg, pp 364–375

9. Golovnev A, Kulikov AS, Mihajlin I (2014) Families with infants: a general approach to solve hard partition problems. In: ICALP (1). Lecture notes in computer science. Springer, Berlin/Heidelberg, pp 551–562. Available at http://arxiv.org/abs/1311.2456
10. Held M, Karp RM (1962) A dynamic programming approach to sequencing problems. J Soc Ind Appl Math 10:196–210
11. Izumi T, Wadayama T (2012) A new direction for counting perfect matchings. In: FOCS, New Brunswick. IEEE Computer Society, pp 591–598
12. Ryser H (1963) Combinatorial mathematics. The Carus mathematical monographs. Mathematical Association of America, Buffalo
13. Servedio RA, Wan A (2005) Computing sparse permanents faster. Inf Process Lett 96(3):89–92

Exact Graph Coloring Using Inclusion-Exclusion

Andreas Björklund and Thore Husfeldt
Department of Computer Science, Lund University, Lund, Sweden

Keywords

Vertex coloring

Years and Authors of Summarized Original Work

2006; Björklund, Husfeldt

Problem Definition

A *k-coloring* of a graph $G = (V, E)$ assigns one of k colors to each vertex such that neighboring vertices have different colors. This is sometimes called *vertex coloring*.

The smallest integer k for which the graph G admits a k-coloring is denoted $\chi(G)$ and called the *chromatic number*. The number of k-colorings of G is denoted $P(G;k)$ and called the *chromatic polynomial*.

Key Results

The central observation is that $\chi(G)$ and $P(G;k)$ can be expressed by an inclusion-exclusion formula whose terms are determined by the number of independent sets of induced subgraphs of G. For $X \subseteq V$, let $s(X)$ denote the number of nonempty independent vertex subsets disjoint from X, and let $s_r(X)$ denote the number of ways to choose r nonempty independent vertex subsets $S_1, \ldots, S_r$ (possibly overlapping and with repetitions), all disjoint from X, such that $|S_1| + \cdots + |S_r| = |V|$.

Theorem 1 ([1]) *Let G be a graph on n vertices.*

1.

$$\chi(G) = \min_{k\in\{1,\ldots,n\}} \left\{ k : \sum_{X\subseteq V} (-1)^{|X|} s(X)^k > 0 \right\}.$$

2. For $k = 1, \ldots, n$,

$$P(G;k) = \sum_{r=1}^{k} \binom{k}{r} \left(\sum_{X\subseteq V} (-1)^{|X|} s_r(X) \right).$$

The time needed to evaluate these expressions is dominated by the 2^n evaluations of $s(X)$ and $s_r(X)$, respectively. These values can be precomputed in time and space within a polynomial factor of 2^n because they satisfy

$$s(X) = \begin{cases} 0, & \text{if } X = V, \\ s(X \cup \{v\}) + s(X \cup \{v\} \cup N(v)) + 1, & \text{for } v \notin X, \end{cases}$$

where $N(v)$ are the neighbors of v in G. Alternatively, the values can be computed using exponential-time, polynomial-space algorithms from the literature.

This leads to the following bounds:

Theorem 2 ([3]) *For a graph G on n vertices, $\chi(G)$ and $P(G;k)$ can be computed in*

1. *Time and space $2^n n^{O(1)}$.*
2. *Time $O(2.2461^n)$ and polynomial space*

The space requirement can be reduced to $O(1.292^n)$ *[4].*

The techniques generalize to arbitrary families of subsets over a universe of size n, provided membership in the family can be decided in polynomial time [3, 4], and to the Tutte polynomial and the Potts model [2].

Applications

In addition to being a fundamental problem in combinatorial optimization, graph coloring also arises in many applications, including register allocation and scheduling.

Recommended Reading

1. Björklund A, Husfeldt T (2008) Exact algorithms for exact satisfiability and number of perfect matchings. Algorithmica 52(2):226–249
2. Björklund A, Husfeldt T, Kaski P, Koivisto M (2007) Fourier meets Möbius: fast subset convolution. In: Proceedings of the 39th annual ACM symposium on theory of computing (STOC), San Diego, 11–13 June 2007. Association for Computing Machinery, New York, pp 67–74
3. Björklund A, Husfeldt T, Koivisto M (2009) Set partitioning via inclusion-exclusion. SIAM J Comput 39(2):546–563
4. Björklund A, Husfeldt T, Kaski P, Koivisto M (2011) Covering and packing in linear space. Inf Process Lett 111(21–22):1033–1036

Exact Quantum Algorithms

Ashley Montanaro
Department of Computer Science, University of Bristol, Bristol, UK

Keywords

Exact algorithms; Quantum algorithms; Quantum query complexity

Years and Authors of Summarized Original Work

2013; Ambainis

Problem Definition

Many of the most important known quantum algorithms operate in the query complexity model. In the simplest variant of this model, the goal is to compute some Boolean function of n input bits by making the minimal number of queries to the bits. All other resources (such as time and space) are considered to be free. In the model of *exact* quantum query complexity, one insists that the algorithm succeeds with certainty on every allowed input. The aim is then to find quantum algorithms which satisfy this constraint and still outperform any possible classical algorithm. This can be a challenging task, as achieving a probability of error equal to zero requires delicate cancellations between the amplitudes in the quantum algorithm. Nevertheless, efficient exact quantum algorithms are now known for certain functions.

Some basic Boolean functions which we will consider below are:

- Parity^n: $f(x_1,\dots,x_n) = x_1 \oplus x_2 \oplus \cdots \oplus x_n$.
- Threshold$_k^n$: $f(x_1,\dots,x_n) = 1$ if $|x| \geq k$, and $f(x) = 0$ otherwise, where $|x| := \sum_i x_i$

is the Hamming weight of x. The special case $k = n/2$ is called the majority function.

- Exact$_k^n$: $f(x_1,\dots,x_n) = 1$ if $|x| = k$, and $f(x) = 0$ otherwise.
- NE ("not-all-equal") on 3 bits: $f(x_1,x_2,x_3) = 0$ if $x_1 = x_2 = x_3$, and $f(x_1,x_2,x_3) = 1$ otherwise.

Key Results

Early Results

One of the earliest results in quantum computation was that the parity of 2 bits can be computed with certainty using only 1 quantum query [6], implying that Parity^n can be computed using $\lceil n/2 \rceil$ quantum queries. By contrast, any classical algorithm which computes this function must make n queries. The quantum algorithm for Parity^n can be used as a subroutine to obtain speedups over classical computation for other problems. For example, based on this algorithm the majority function on n bits can be computed exactly using $n + 1 - w(n)$ quantum queries, where $w(n)$ is the number of 1s in the binary expansion of n [8]; this result has recently been improved (see below).

If the function to be computed is partial, i.e., some possible inputs are disallowed, the separation between exact quantum and classical query complexity can be exponential. For example, in the Deutsch-Jozsa problem we are given query access to an n-bit string x (with n even) such that either all the bits of x are equal or exactly half of them are equal to 1. Our task is to determine which is the case. Any exact classical algorithm must make at least $n/2 + 1$ queries to bits of x to solve this problem, but it can be solved with only one quantum query [7]. An exponential separation is even known between exact quantum and *bounded-error* classical query complexity for a different partial function [5].

Recent Developments

For some years, the best known separation between exact quantum and classical query complexity of a total Boolean function (i.e., a function $f : \{0,1\}^n \to \{0,1\}$ with all possible n-bit strings allowed as input) was the factor of 2 discussed above. However, recently the first example has been presented of an exact quantum algorithm for a family of total Boolean functions which achieves a lower asymptotic query complexity than the best possible classical algorithm [1].

The family of functions used can be summarized as a "not-all-equal tree of depth d." It is based around the recursive use of the NE function. Define the function NE$^0(x_1) = x_1$ and then for $d > 0$

$$\begin{aligned}&\mathrm{NE}^d(x_1,\dots,x_{3^d})\\ &\quad = \mathrm{NE}(\mathrm{NE}^{d-1}(x_1,\dots,x_{3^{d-1}}), \mathrm{NE}^{d-1}(x_{3^{d-1}+1},\dots,x_{2\cdot 3^{d-1}}), \mathrm{NE}^{d-1}(x_{2\cdot 3^{d-1}+1},\dots,x_{3^d})).\end{aligned}$$

Then the following separation is known:

Theorem 1 (Ambainis [1]) *There is an exact quantum algorithm which computes* NE^d *using* $O(2.593\ldots^d)$ *queries. Any classical algorithm which computes* NE^d *must make* $\Omega(3^d)$ *queries, even if it is allowed probability of failure* $1/3$.

In addition, Theorem 1 implies the first known asymptotic separation between exact quantum and classical communication complexity for a total function. Improvements over the best possible classical algorithms are also known for the other basic Boolean functions previously mentioned.

Theorem 2 (Ambainis, Iraids, and Smotrovs [2]) *There is an exact quantum algorithm which computes* Exact$_k^n$ *using* $\max\{k, n-k\}$ *queries and an exact quantum algorithm which computes* Threshold$_k^n$ *using* $\max\{k, n-k+1\}$ *queries. Both of these complexities are optimal.*

By contrast, it is easy to see that any exact classical algorithm for these functions must make

n queries. An optimal exact quantum algorithm for the special case Exact_2^4 had already been found prior to this, in work which also gave optimal exact quantum query algorithms for all Boolean functions on up to 3 bits [9].

Methods

We briefly describe the main ingredients of the efficient quantum algorithm for NE^d [1]. The basic idea is to fix some small d_0, start with an exact quantum algorithm which computes NE^{d_0} using fewer queries than the best possible classical algorithm, and then amplify the separation by using the algorithm recursively. A difficulty with this approach is that the standard approach for using a quantum algorithm recursively incurs a factor of 2 penalty in the number of queries with each recursive call. This factor of 2 is required to "uncompute" information left over after the algorithm has completed. Therefore, a query complexity separation by a factor of 2 or less does not immediately give an asymptotic separation.

This problem can be addressed by introducing the notion of p-computation. Let $p \in [-1, 1]$. A quantum algorithm $\mathcal{A}$ is said to p-compute a function $f(x_1, \ldots, x_n)$ if, for some state $|\psi_{\text{start}}\rangle$:

- Whenever $f(x_1, \ldots, x_n) = 0$, $\mathcal{A}|\psi_{\text{start}}\rangle = |\psi_{\text{start}}\rangle$.
- Whenever $f(x_1, \ldots, x_n) = 1$, $\mathcal{A}|\psi_{\text{start}}\rangle = p|\psi_{\text{start}}\rangle + \sqrt{1-p^2}|\psi\rangle$ for some $|\psi\rangle$, which may depend on x, such that $\langle\psi|\psi_{\text{start}}\rangle = 0$.

It can be shown that if there exists an algorithm which p-computes some function f for some $p \leq 0$, there exists an exact quantum algorithm which computes f using the same number of queries. Further, if an algorithm (-1)-computes some function f, the same algorithm can immediately be used recursively, without needing any additional queries at each level of recursion. Thus, to obtain an asymptotic quantum-classical separation for NE^d, it suffices to obtain an algorithm which (-1)-computes NE^{d_0} using strictly fewer than 3^{d_0} queries, for some d_0.

The NE^d problem also behaves particularly well with respect to p-computation for general values of p:

Lemma 1 *If there is an algorithm $\mathcal{A}$ which p-computes NE^{d-1} using k queries, there is an algorithm $\mathcal{A}'$ which p'-computes NE^d with $2k$ queries, for $p' = 1 - 4(1-p)^2/9$.*

This lemma allows algorithms for NE^{d-1} to be lifted to algorithms for NE^d, at the expense of making the value of p worse. Nevertheless, given that it is easy to write down an algorithm which (-1)-computes NE^0 using one query, the lemma is sufficient to obtain an exact quantum algorithm for NE^2 using 4 queries. This is already enough to prove an asymptotic quantum-classical separation, but this separation can be improved using the following lemma (a corollary of a variant of amplitude amplification):

Lemma 2 *If there is an algorithm $\mathcal{A}$ which p-computes NE^d using k queries, there is an algorithm $\mathcal{A}'$ which p'-computes NE^d with $2k$ queries, for $p' = 2p^2 - 1$.*

Interleaving Lemmas 1 and 2 allows one to derive an algorithm which (-1)-computes NE^8 using 2,048 queries, which implies an exact quantum algorithm for NE^d using $O(2{,}048^{d/8}) = O(2.593\ldots^d)$ queries.

Experimental Results

It is a difficult task to design exact quantum query algorithms, even for small functions, as these algorithms require precise cancellations between amplitudes. One way to gain numerical evidence for what the exact quantum query complexity of a function should be is to use the formulation of quantum query complexity as a semidefinite programming (SDP) problem [4]. This allows one to estimate the optimal success probability of any quantum algorithm using a given number of queries to compute a given function. If this success probability is very close to 1, this gives numerical evidence that there exists an exact quantum algorithm using that number of queries.

This approach has been applied for all Boolean functions on up to 4 bits, giving strong evidence that the only function on 4 bits which requires 4 quantum queries is the AND function and functions equivalent to it [9]. This has led to the conjecture that, for any n, the only function on n bits which requires n quantum queries to be computed exactly is the AND function and functions equivalent to it. This would be an interesting contrast with the classical case where most functions on n bits require n queries. This conjecture has recently been proven for various special cases: symmetric functions, monotone functions, and functions with formula size n [3].

Cross-References

▸ Quantum Algorithm for the Parity Problem
▸ Quantum Search

Recommended Reading

1. Ambainis A (2013) Superlinear advantage for exact quantum algorithms. In: Proceedings of the 45th annual ACM symposium on theory of computing, pp 891–900. arXiv:1211.0721
2. Ambainis A, Iraids J, Smotrovs J (2013) Exact quantum query complexity of EXACT and THRESHOLD. In: Proceedings of the 8th conference on the theory of quantum computation, communication, and cryptography (TQC'13), pp 263–269. arXiv:1302.1235
3. Ambainis A, Gruska J, Zheng S (2014) Exact query complexity of some special classes of Boolean functions. arXiv:1404.1684
4. Barnum H, Saks M, Szegedy M (2003) Quantum query complexity and semi-definite programming. In: Proceedings of the 18th annual IEEE conference on computational complexity, Aarhus, pp 179–193
5. Brassard G, Høyer P (1997) An exact quantum polynomial-time algorithm for Simon's problem. In: Proceedings of the fifth Israeli symposium on theory of computing and systems, Aarhus, Denmark pp 12–23. quant-ph/9704027
6. Cleve R, Ekert A, Macchiavello C, Mosca M (1998) Quantum algorithms revisited. Proc R Soc Lond A 454(1969):339–354. quant-ph/9708016
7. Deutsch D, Jozsa R (1992) Rapid solution of problems by quantum computation. Proc R Soc Lond Ser A 439(1907):553–558
8. Hayes T, Kutin S, van Melkebeek D (2002) The quantum black-box complexity of majority. Algorithmica 34(4):480–501. quant-ph/0109101
9. Montanaro A, Jozsa R, Mitchison G (2013) On exact quantum query complexity. Algorithmica 71(4):775–796

Experimental Implementation of Tile Assembly

Constantine G. Evans
Division of Biology and Bioengineering, California Institute of Technology, Pasadena, CA, USA

Keywords

DNA tiles; Experimental tile self-assembly

Years and Authors of Summarized Original Work

2007; Schulman, Winfree
2008; Fujibayashi, Hariadi, Park, Winfree, Murata
2009; Barish, Schulman, Rothemund, Winfree
2012; Schulman, Yurke, Winfree

Problem Definition

From the earliest works on tile self-assembly, abstract theoretical models and experimental implementations have been linked. In 1998, in addition to developing the abstract and kinetic Tile Assembly Models (aTAM and kTAM) [14], Winfree et al. demonstrated the use of DNA tiles to construct a simple, periodic lattice [16]. Periodic lattices and "uniquely addressed" assemblies, where each tile type appears once in each assembly, have been widely studied, with systems employing up to a thousand unique tiles in three dimensions [8, 13]. While these systems provide insight into the behavior of DNA tile systems, *algorithmic* tile systems of more theoretical interest pose specific challenges for experimental implementation.

In the aTAM, abstract tiles attach individually to empty lattice sites if bonds of a sufficient total strength b (at least abstract "temperature" τ) can be made, and once attached, never detach. Experimentally, free tiles and assemblies of bound tiles are in solution. Tiles have short single-stranded "sticky ends" regions that form bonds with complementary regions on other tiles. Tiles attach to assemblies at rates dependent only upon their concentrations, regardless of the strength of bonds that can be made. Once attached, tiles can detach and do so at a rate that is exponentially dependent upon the total strength of the bonds [6]. Thus, for a tile t_i with concentration $[t_i]$ binding by a total abstract bond strength b, we have attachment and detachment rates of

$$r_f = k_f[t_i] \qquad r_b = k_f e^{-b\Delta G^\circ_{se}/RT+\alpha} \tag{1}$$

where k_f is an experimentally determined rate constant, α is a constant binding free energy change (e.g., from entropic considerations), ΔG°_{se} is the free energy change of a single-strength bond, and T is the (physical) temperature. Using the substitutions $[t_i] = e^{-G_{mc}+\alpha}$, $G_{se} = -\Delta G^\circ_{se}/RT$, and $\hat{k}_f = k_f e^{\alpha}$, these can be simplified to

$$r_f = \hat{k}_f e^{-G_{mc}} \qquad r_b = \hat{k}_f e^{-bG_{se}} \tag{2}$$

where G_{se} is a (positive) unitless free energy for a single-strength bond (larger values correspond to stronger bonds), G_{mc} is a free energy analogue of concentration (larger values correspond to lower concentrations), and $\hat{k}_f$ is an adjusted rate constant.

These rates are the basis of the kinetic Tile Assembly Model (kTAM), which is widely used as a physical model of tile assembly [14]. Tiles that attach faster than they detach will tend to remain attached and allow further growth: for example, if $G_{mc} < 2G_{se}$, tile attachments by $b \geq 2$ will be favorable. Tiles that detach faster than they attach will tend to remain detached and not allow further growth. Since G_{mc} is dependent upon tile concentration, and G_{se} is dependent upon physical temperature (lower temperatures result in larger G_{se} values), the attachment and detachment rates can be tuned such that attachment is slightly more favorable than detachment for tiles attaching by a certain total bond strength and less favorable for less strongly bound tiles. In this way, in the limit of low concentrations and slow growth, the kTAM approximates the aTAM at a given abstract temperature τ. When moving away from this limit and toward experimentally feasible conditions, however, the kTAM provides insight into many of the challenges faced in experimental implementation of algorithmic tile assembly:

Growth errors: While tile assembly in the aTAM is error-free, tiles can attach in erroneous locations in experiments. Even ignoring the possibility of lattice defects, malformed tiles, and other experimental peculiarities, errors can arise in the kTAM via tiles that attach by less than the required bond strength (e.g., one single-strength bond for a $\tau = 2$ system) and are then "frozen" in place by further attachments [4]. As the further growth of algorithmic systems depends on the tiles already present in an assembly, a single erroneously incorporated tile can propagate undesired growth via further, valid attachments. These errors can arise both in growth sites where another tile could attach correctly ("growth errors") and lattice sites where no correct tile could attach ("facet nucleation errors") [3, 14].

Seeding: Tile assembly in the aTAM is usually initiated from a designated "seed" tile. In solution, however, tiles are free to attach to all other tiles and can form assemblies without starting from a seed, even if this requires several unfavorable attachments to form a stable structure that can allow further growth. Depending upon the tile system, these "spuriously nucleated" structures can potentially form easily. For example, a $T = 2$ system with boundaries of identical tiles that attach by double bonds on both sides can readily form long strings of boundary tiles [10, 11].

Tile depletion: As free tiles in solution are incorporated into assemblies, their concentrations are correspondingly reduced. This depletion lowers the attachment rates for those tiles and in turn changes the favorability of growth. If different tile types are incorporated in different quantities, their attachment rates will become unequal, and at some point in assembly, attachment by two single-strength bonds may be favorable for one tile type and unfavorable for another.

Tile design: While theoretical constructions may employ an arbitrary number of sticky ends types, this number is limited by tile designs in practice. Most tiles use short single-stranded DNA regions of 5–10 nucleotides (nt), limiting the number of possible sticky ends to 4^5–4^{10} at best. However, since partial bonds can form between subsequences of the sticky ends, sequences with sufficient orthogonality are required, and since DNA binding strength is sequence dependent, sequences with similar binding energies are required [5]. Both of these effects place considerably more stringent limits on the number of sticky ends and change the behavior of experimental systems.

Key Results

Winfree and Bekbolatov developed a tileset transformation, "uniform proofreading," that reduced per-site *growth* error rates from $r_{\mathrm{err}} \approx me^{-G_{se}}$ (where m is the number of possible errors) to $\approx me^{-KG_{se}}$ by scaling each tile into a $K \times K$ block of individually attaching tiles with unique internal bonds [15]. However, this transformation did not reduce facet nucleation errors. Chen and Goel later created a modified transformation, "snaked proofreading," that reduced both growth and facet nucleation errors by changing the strengths of the internal bonds used [3]. These and other proofreading methods have the potential to drastically reduce error rates in experimental systems.

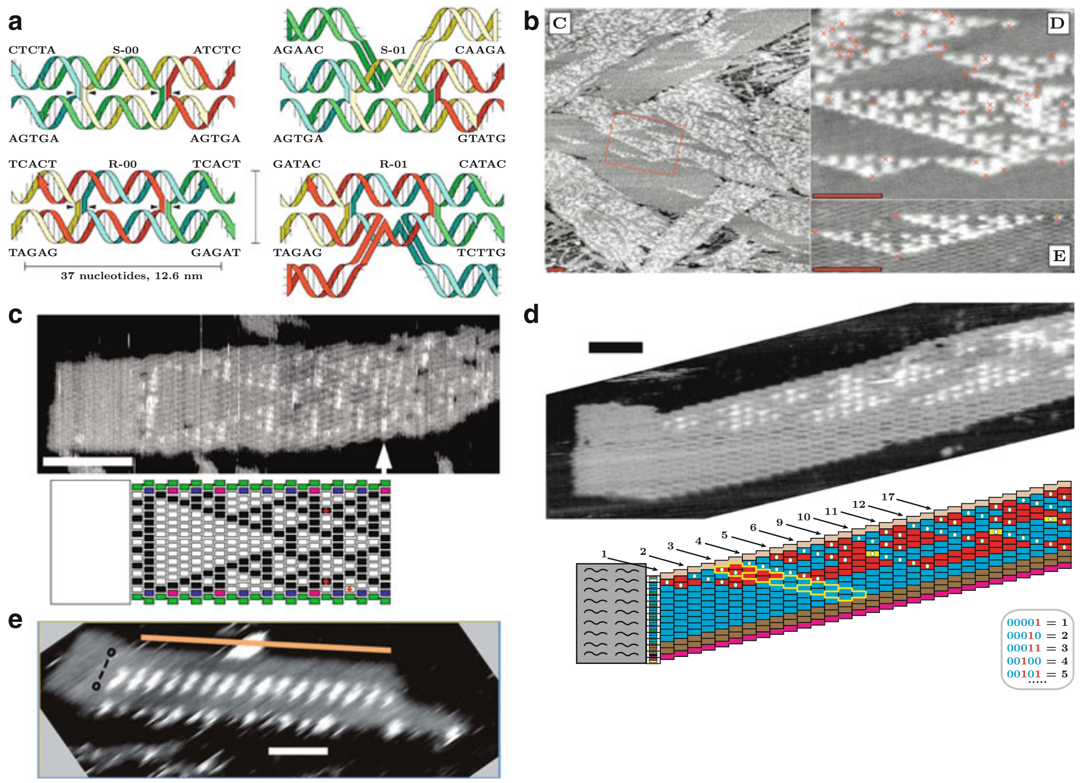
Schulman et al. analyzed tile system nucleation through the consideration of "critical nuclei," tile assemblies where melting and further growth are equally favorable, and showed that by ensuring a sufficient number of unfavorable attachments would be required for a critical nucleus to form, the rate of spurious nucleation can be kept arbitrarily low [11]. Using this analysis, Schulman et al. constructed the "zigzag" ribbon system, which forms a ribbon where each row must assemble completely before the next can begin growth, as an example of a system where spurious nucleation can be made arbitrarily low by increasing ribbon width. To nucleate desired structures, this system makes use of a large, preformed seed structure to allow the growth of the first ribbon row.

Schulman et al. also devised a "constant-temperature" growth technique where the concentrations of assemblies, controlled by the concentration of initial seeds in a nucleation-controlled system, are kept small enough in comparison to the concentrations of free tiles that growth does not significantly deplete tile concentrations, which thus remain approximately constant [12]. After growth is completed, the remaining free tiles are "deactivated" by adding an excess of DNA strands complementary to specific sticky ends sequences.

In analyzing the effects of DNA sequences on tile assembly, Evans and Winfree showed an exponential increase of error rates in the kTAM for partial binding between different sticky ends sequences and for differing sequence-dependent binding energies and developed algorithms for sequence design and assignment to reduce these effects [5]. With reasonable design constraints, their algorithms suggested limits of around 80 sticky ends types for tiles using 5 nt sticky ends and around 360 for tiles using 10 nt sticky ends before significant sequence effects begin to become unavoidable and must be incorporated into tile system design.

Experimental Results

While numerous designs exist for tile structures, experimental implementations have usually used either double-crossover (DX) tiles with 5 or 6 nt sticky ends [16] or single-stranded tiles (SST) with 10 and 11 nt sticky ends [17]. SSTs potentially offer a significantly larger sequence



Experimental Implementation of Tile Assembly, Fig. 1 Experimental results for algorithmic tile assembly. (**a**) and (**b**) show the Rothemund et al. XOR system's DX tiles and resulting structures, with (b) illustrating the high error rates and seeding problems of the system [9]. (**c**) shows the Fujibayashi et al. fixed-width XOR ribbon [7], while (**d**) shows the Barish et al. binary counter ribbon with partial 2×2 proofreading [2]; the rectangular structures on the left of both systems are preformed DNA origami seeds. (**e**) shows an example bit-copying ribbon from Schulman et al. [12]

space and have been employed in large, non-algorithmic systems [8, 13] but have not yet been used for complex algorithmic systems.

Early experiments in algorithmic tile assembly using DX tiles did not employ any of the key results discussed above. Rothemund et al. implemented a simple XOR system of four logical tiles (eight tiles were needed owing to structural considerations), using DNA hairpins on "one-valued" tiles as labels [9] and flexible, one-dimensional seeds (Fig. 1a,b). While assemblies grew, and Sierpinski triangle patterns were visible, error rates were between 1 and 10 % per tile. Barish et al. implemented more complex bit-copying and binary counting systems in a similar way, finding per-tile error rates of around 10 % [1].

More recently, Fujibayashi et al. used rigid DNA origami structures to serve as seeds for the growth of a fixed-width XOR ribbon system and, in doing so, reduced error rates to 1.4 % per tile without incorporating proofreading [7] (Fig. 1c). This seeding mechanism was also used by Barish et al. to seed zigzag bit-copying and binary counting ribbon systems that implemented 2×2 uniform proofreading [2]. With nucleation control and proofreading, these systems resulted in dramatically reduced error rates of 0.26 % per proofreading block for copying and 4.1 % for the more algorithmically complex binary counting, which only partially implemented uniform proofreading (Fig. 1d).

A similar bit-copying ribbon was later implemented by Schulman et al., with the addition of the constant-temperature, constant-concentration growth method and the use of biotin-streptavidin labels rather than DNA hairpins. The result was a decrease in error rates by almost a factor of ten to 0.034 % per block [12] (Fig. 1e). At this error rate, structures of around 2,500 error-free blocks, or 10,000 individual tiles, could be grown with reasonable yields, suggesting that with the incorporation of proofreading, nucleation control and constant-concentration growth methods, low-error experimental implementations of increasingly complex algorithmic tile systems may be feasible up to sequence space limitations.

Cross-References

▶ Robustness in Self-Assembly

Recommended Reading

1. Barish RD, Rothemund PWK, Winfree E (2005) Two computational primitives for algorithmic self-assembly: copying and counting. Nano Lett 5(12):2586–2592. doi:10.1021/nl052038l
2. Barish RD, Schulman R, Rothemund PWK, Winfree E (2009) An information-bearing seed for nucleating algorithmic self-assembly. PNAS 106:6054–6059. doi:10.1073/pnas.0808736106
3. Chen HL, Goel A (2005) Error free self-assembly using error prone tiles. In: DNA 10, Milan. LNCS, vol 3384. Springer, pp 702–707
4. Doty D (2012) Theory of algorithmic self-assembly. Commun ACM 55(12):78–88. doi:10.1145/2380656.2380675
5. Evans CG, Winfree E (2013) DNA sticky end design and assignment for robust algorithmic self-assembly. In: DNA 19, Tempe. LNCS, vol 8141. Springer, pp 61–75. doi:10.1007/978-3-319-01928-4_5
6. Evans CG, Hariadi RF, Winfree E (2012) Direct atomic force microscopy observation of DNA tile crystal growth at the single-molecule level. J Am Chem Soc 134:10,485–10,492. doi:10.1021/ja301026z
7. Fujibayashi K, Hariadi R, Park SH, Winfree E, Murata S (2008) Toward reliable algorithmic self-assembly of DNA tiles: a fixed-width cellular automaton pattern. Nano Lett 8(7):1791–1797. doi:10.1021/nl0722830
8. Ke Y, Ong LL, Shih WM, Yin P (2012) Three-dimensional structures self-assembled from DNA bricks. Science 338(6111):1177–1183. doi:10.1126/science.1227268
9. 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
10. Schulman R, Winfree E (2007) Synthesis of crystals with a programmable kinetic barrier to nucleation. PNAS 104(39):15,236–15,241. doi:10.1073/pnas.0701467104
11. Schulman R, Winfree E (2010) Programmable control of nucleation for algorithmic self-assembly. SIAM J Comput 39(4):1581–1616. doi:10.1137/070680266
12. Schulman R, Yurke B, Winfree E (2012) Robust self-replication of combinatorial information via crystal growth and scission. PNAS 109(17):6405–6410. doi:10.1073/pnas.1117813109
13. Wei B, Dai M, Yin P (2012) Complex shapes self-assembled from single-stranded DNA tiles. Nature 485(7400):623–626
14. Winfree E (1998) Simulations of computing by self-assembly. Technical report CaltechCSTR:1998.22, Pasadena
15. Winfree E, Bekbolatov R (2004) Proofreading tile sets: error correction for algorithmic self-assembly. In: DNA 9, Madison. Wisconson, LNCS, vol 2943. Springer, pp 126–144
16. Winfree E, Liu F, Wenzler LA, Seeman NC (1998) Design and self-assembly of two-dimensional DNA crystals. Nature 394(6693):539–544
17. Yin P, Hariadi RF, Sahu S, Choi HMT, Park SH, LaBean TH, Reif JH (2008) Programming DNA tube circumferences. Science 321(5890):824–826. doi:10.1126/science.1157312

Experimental Methods for Algorithm Analysis

Catherine C. McGeoch
Department of Mathematics and Computer Science, Amherst College, Amherst, MA, USA

Keywords

Algorithm engineering; Empirical algorithmics; Empirical analysis of algorithms; Experimental algorithmics

Years and Authors of Summarized Original Work

2001; McGeoch

Problem Definition

Experimental analysis of algorithms describes not a specific algorithmic problem, but rather an approach to algorithm design and analysis. It complements, and forms a bridge between, traditional *theoretical analysis*, and the application-driven methodology used in *empirical analysis*.

The traditional theoretical approach to algorithm analysis defines algorithm efficiency in terms of counts of dominant operations, under some abstract model of computation such as a RAM; the input model is typically either worst-case or average-case. Theoretical results are usually expressed in terms of asymptotic bounds on the function relating input size to number of dominant operations performed.

This contrasts with the tradition of empirical analysis that has developed primarily in fields such as operations research, scientific computing, and artificial intelligence. In this tradition, the efficiency of implemented programs is typically evaluated according to CPU or wall-clock times; inputs are drawn from real-world applications or collections of benchmark test sets, and experimental results are usually expressed in comparative terms using tables and charts.

Experimental analysis of algorithms spans these two approaches by combining the sensibilities of the theoretician with the tools of the empiricist. Algorithm and program performance can be measured experimentally according to a wide variety of *performance indicators*, including the dominant cost traditional to theory, bottleneck operations that tend to dominate running time, data structure updates, instruction counts, and memory access costs. A researcher in experimental analysis selects performance indicators most appropriate to the scale and scope of the specific research question at hand. (Of course time is not the only metric of interest in algorithm studies; this approach can be used to analyze other properties such as solution quality or space use.)

Input instances for experimental algorithm analysis may be randomly generated or derived from application instances. In either case, they typically are described in terms of a small- to medium-sized collection of *controlled parameters*. A primary goal of experimentation is to investigate the cause-and-effect relationship between input parameters and algorithm/program performance indicators.

Research goals of experimental algorithmics may include discovering functions (not necessarily asymptotic) that describe the relationship between input and performance, assessing the strengths and weaknesses of different algorithm/data structures/programming strategies, and finding best algorithmic strategies for different input categories. Results are typically presented and illustrated with graphs showing comparisons and trends discovered in the data.

The two terms "empirical" and "experimental", are often used interchangeably in the literature. Sometimes the terms "old style" and "new style" are used to describe, respectively, the empirical and experimental approaches to this type of research. The related term "algorithm engineering" refers to a systematic design process that takes an abstract algorithm all the way to an implemented program, with an emphasis on program efficiency. Experimental and empirical analysis is often used to guide the algorithm engineering process. The general term *algorithmics* can refer to both design and analysis in algorithm research.

Key Results

None

Applications

Experimental analysis of algorithms has been used to investigate research problems originating in theoretical computer science. One example arises in the average-case analysis of algorithms for the One-Dimensional Bin Packing problem. Experimental analyses have led to new theorems about the performance of the optimal algorithm; new asymptotic bounds on average-case performance of approximation algorithms; extensions of theoretical results to new models of inputs; and to new algorithms with tighter approximation guarantees. Another example is the experimental discovery of a type of phase-transition behavior for random instances of the 3CNF-Satisfiabilty problem, which has led to new ways to characterize the difficulty of problem instances.

A second application of experimental algorithmics is to find more realistic models of computation, and to design new algorithms that perform better on these models. One example is found in the development of new memory-based models of computation that give more accurate time predictions than traditional unit-cost models. Using these models, researchers have found new cache-efficient and I/O-efficient algorithms that exploit properties of the memory hierarchy to achieve significant reductions in running time.

Experimental analysis is also used to design and select algorithms that work best in practice, algorithms that work best on specific categories of inputs, and algorithms that are most robust with respect to bad inputs.

Data Sets

Many repositories for data sets and instance generators to support experimental research are available on the Internet. They are usually organized according to specific combinatorial problems or classes of problems.

URL to Code

Many code repositories to support experimental research are available on the Internet. They are usually organized according to specific combinatorial problems or classes of problems. Skiena's *Stony Brook Algorithm Repository* (www.cs.sunysb.edu/~algorith/) provides a comprehensive collection of problem definitions and algorithm descriptions, with numerous links to implemented algorithms.

Recommended Reading

The algorithmic literature containing examples of experimental research is much too large to list here. Some articles containing advice and commentary on experimental methodology in the context of algorithm research appear in the list below.

The workshops and journals listed below are specifically intended to support research in experimental analysis of algorithms. Experimental work also appears in more general algorithm research venues such as SODA (ACM/IEEE Symposium on Data Structures and Algorithms), *Algorithmica*, and *ACM Transactions on Algorithms*.

1. ACM Journal of Experimental Algorithmics. Launched in 1996, this journal publishes contributed articles as well as special sections containing selected papers from ALENEX and WEA. Visit www.jea.acm.org, or visit portal.acm.org and click on ACM Digital Library/Journals/Journal of Experimental Algorithmics
2. ALENEX. Beginning in 1999, the annual workshop on Algorithm Engineering and Experimentation is sponsored by SIAM and ACM. It is co-located with SODA, the SIAM Symposium on Data Structures and Algorithms. Workshop proceedings are published in the Springer LNCS series. Visit www.siam.org/meetings/ for more information
3. Barr RS, Golden BL, Kelly JP, Resende MGC, Stewart WR (1995) Designing and reporting on computational experiments with heuristic methods. J Heuristics 1(1):9–32
4. Cohen PR (1995) Empirical methods for artificial intelligence. MIT, Cambridge
5. DIMACS Implementation Challenges. Each DIMACS Implementation Challenge is a year-long cooperative research event in which researchers cooperate to find the most efficient algorithms and strategies for selected algorithmic problems. The DIMACS Challenges since 1991 have targeted a variety of optimization problems on graphs; advanced data structures; and scientific application areas involving computational biology and parallel computation. The DIMACS Challenge proceedings are published by AMS as part of the DIMACS Series in Discrete Mathematics and Theoretical Computer Science. Visit dimacs.rutgers.edu/Challenges for more information
6. Johnson DS (2002) A theoretician's guide to the experimental analysis of algorithms. In: Goodrich MH, Johnson DS, McGeoch CC (eds) Data structures, near neighbors searches, and methodology: fifth and sixth DIMACS implementation challenges, vol 59, DIMACS series in discrete mathematics and theoretical

computer science. American Mathematical Society, Providence

7. McGeoch CC (1996) Toward an experimental method for algorithm simulation. INFORMS J Comput 1(1):1–15
8. WEA. Beginning in 2001, the annual Workshop on Experimental and Efficient Algorithms is sponsored by EATCS. Workshop proceedings are published in the Springer LNCS series

Exponential Lower Bounds for *k*-SAT Algorithms

Dominik Scheder
Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China
Institute for Computer Science, Shanghai Jiaotong University, Shanghai, China

Keywords

Exponential algorithms; k-SAT; Lower bounds; PPSZ algorithm; Proof complexity

Years and Authors of Summarized Original Work

2013; Shiteng Chen, Dominik Scheder, Navid Talebanfard, Bangsheng Tang

Problem Definition

Given a propositional formula in conjunctive normal form, such as $(x \vee y) \wedge (\bar{x} \vee \bar{y} \vee z) \wedge (\bar{z})$, one wants to find an assignment of truth values to the variables that makes the formula evaluate to true. Here, $[x \mapsto 1, y \mapsto 0, z \mapsto 0]$ does the job. We call such formulas *CNF formulas* and such assignments *satisfying assignments*. SAT is the problem of deciding whether a given CNF formula is satisfiable. If every clause (such as $(\bar{x} \vee \bar{y} \vee z)$ above) has at most k literals, we call this a k-CNF formula. The above example is a 3-CNF formula. The problem of deciding whether a given k-CNF formula is satisfiable is called k-SAT. This is one of the most fundamental NP-complete problems.

Several clever algorithms have been developed for k-SAT. In this note we are mostly concerned with the PPSZ algorithm [3]. This is itself an improved version of the older PPZ algorithm [4]. Another prominent SAT algorithm is Schöning's random walk algorithm [6], which is slower than PPSZ, but has the benefit that it can be turned into a deterministic algorithm [5].

Given that we currently cannot prove P $\neq$ NP, all super-polynomial lower bounds on the running time of k-SAT algorithms must be either conditional, that is, rest on widely believed but yet unproven assumptions, or must be for a particular family of algorithms. In this note we sketch exponential lower bounds for the PPSZ algorithm, which is the currently fastest algorithm for k-SAT. We measure the running time of a SAT algorithm in terms of n, the number of variables. Often probabilistic algorithms for k-SAT (like PPSZ) have polynomial running time and success probability p^n for some $p < 1$. One can turn this into a Monte Carlo algorithm with success probability at least $1/2$ by repeating it $(1/p)^n$ times. We prefer the formulation of PPSZ as having polynomial running time, and we are interested in the worst-case success probability p^n.

Key Results

The worst-case behavior of PPSZ is exponential. That is, there are satisfiable k-CNF formulas on n variables, for which PPSZ finds a satisfying assignment with probability at most $2^{-\Omega(n)}$. More precisely, there is a constant C and a sequence $\epsilon_k \leq \frac{C \log^2 k}{k}$ such that the worst-case success probability of PPSZ for k-SAT is at most $2^{-(1-\epsilon_k)n}$. See Theorem 3 below for a formal statement.

The PPSZ Algorithm

The PPSZ algorithm, named after its inventors Paturi, Pudlák, Saks, and Zane [3], is the fastest known algorithm for k-SAT. We now give a brief description of it: Choose a random ordering σ of the n variables $x_1, \ldots, x_n$ of F. Choose random truth values $b = (b_1, \ldots, b_n) \in \{0, 1\}$. Iterate through the variables in the ordering given by σ. When processing x_i check whether it is "obvious" what the correct value of x_i should be. If so, fix x_i to that value. Otherwise, fix x_i to b_i. By fixing we mean replacing each occurrence of x_i in F by that value (and each occurrence of $\bar{x}_i$ by the negation of that value). After all variables have been processed, the algorithm returns the satisfying assignment it has found or returns `failure` if it has run into a contradiction.

It remains to specify what "obvious" means: Given a CNF formula F and a variable x_i, we say that the correct value of x_i is *obviously* b if the statement $x_i = b$ can be derived from F by width-w resolution, where w is some large constant (think of $w = 1{,}000$). This can be checked in time $O(n^w)$, which is polynomial.

Let ppsz(F, σ, b) be the return value of ppsz. That is, ppsz$(F, \sigma, b) \in \text{sat}(F) \cup \{\texttt{failure}\}$, where sat$(F)$ is the set of satisfying assignments of F.

A Very Brief Sketch of the Analysis of PPSZ

Let σ be a permutation of $x_1, \ldots, x_n$ and let $b = (b_1, \ldots, b_n) \in \{0, 1\}^n$. Suppose we run PPSZ on F using this permutation σ and the truth values b. For $1 \le i \le n$, define Z_i to be 1 if PPSZ did not find it obvious what the correct value of x_i should be. Let $Z = Z_1 + \cdots + Z_n$. To underline the dependence on F, σ, and b, we sometimes write $Z(F, \sigma, b)$. It is not difficult to show the following lemma.

Lemma 1 ([3]) *Let F be a satisfiable CNF formula over n variables. Let σ be a random permutation of its variables and let $a \in \{0, 1\}^n$ be satisfying assignment of F. Then*

$$\Pr_{\sigma,b}[\text{ppsz}(F, \sigma, b) = a] = \mathbb{E}_\sigma[2^{-Z(F,\sigma,a)}] \qquad (1)$$

Since $x \mapsto 2^x$ is a convex function, Jensen's inequality implies that $\mathbb{E}_\sigma[2^{-Z}] \ge 2^{-\mathbb{E}[Z]}$, and by linearity of expectation, it holds that $\mathbb{E}[Z] = \sum_{i=1}^n \mathbb{E}[Z_i]$.

Lemma 2 ([3]) *There are numbers $c_k \in [0, 1]$ such that the following holds: If F is a k-CNF formula over n variables with a unique satisfying assignment a, then $\mathbb{E}_\sigma[Z_i(F, \sigma, a)] \le c_k$ for all $1 \le i \le n$. Furthermore, for large k we have $c_k \approx 1 - \frac{\pi^2}{6k}$, and in particular $c_3 = 2\ln(2) - 1 \approx 0.38$.*

Combining everything, Paturi, Pudlák, Saks, and Zane obtain their main result:

Theorem 1 ([3]) *Let F be a k-CNF formula with a unique satisfying assignment a. Then PPSZ finds this satisfying assignment with probability at least $2^{-c_k n}$.*

It takes a considerable additional effort to show that the same bound holds also if F has multiple satisfying assignments:

Theorem 2 ([2]) *Let F be a satisfiable k-CNF formula. Then PPSZ finds a satisfying assignment with probability at least $2^{-c_k n}$.*

We sketch the intuition behind the proof of Lemma 2. It turns out that in the worst case the event $Z_i = 1$ can be described by the following random experiment: Let $T = (V, E)$ be the infinite rooted $(k-1)$-ary tree. For each node $v \in V$ choose $\tau(v) \in [0, 1]$ randomly and independently. Call a node v *alive* if $\tau(v) \ge \tau(\text{root})$. Then $\Pr[Z_i = 1]$ is (roughly) equal to the probability that T contains an infinite path of alive vertices, starting with the root. Call this probability c_k. A simple calculation shows that $c_3 = 2\ln(2) - 1$. For larger values of c_k, there is not necessarily a closed form, but Paturi, Pudlák, Saks, and Zane show that $c_k \approx 1 - \frac{\pi^2}{6k}$ for large k.

Hard Instances for the PPSZ Algorithm

One can construct instances on which the success probability of PPSZ is exponentially small. The construction is probabilistic and rather simple. Its analysis is quite technical, so we can only sketch it here. We start with some easy estimates. By Lemma 1 we can write the success probability of PPSZ as

$$\Pr_{\sigma,b}[\text{ppsz}(F,\sigma,b) \in \text{sat}(F)] = \sum_{a \in \text{sat}(F)} \mathbb{E}_\sigma[2^{-Z(F,\sigma,a)}]. \quad (2)$$

Above we used Jensen's inequality to prove $\mathbb{E}[2^{-Z}] \geq 2^{-\mathbb{E}[Z]}$. In this section we want to construct *hard instances*, that is, instances on which the success probability (2) is exponentially small. Thus, we cannot use Jensen's inequality, as it gives a lower bound, not an upper. Instead, we use the following trivial estimate:

$$\sum_{a \in \text{sat}(F)} \mathbb{E}_\sigma[2^{-Z(F,\sigma,a)}] \leq \sum_{a \in \text{sat}(F)} \max_\sigma 2^{-Z(F,\sigma,a)}$$
$$\leq |\text{sat}(F)| \cdot \max_{a \in \text{sat}(F),\sigma} 2^{-Z(F,\sigma,a)}. \quad (3)$$

We would like to construct a satisfiable k-CNF formula F for which (i) $|\text{sat}(F)|$ is small, i.e., F has few satisfying assignments, and (ii) $Z(F,\sigma,a)$ is large for every permutation and every satisfying assignment a. It turns out there are formulas satisfying both requirements:

Theorem 3 *There are numbers ϵ_k converging to* 0 *such that the following holds: For every k, there is a family $(F_n)_{n\geq 1}$, where each F_n is a satisfiable k-CNF formula over n variables such that*

1. $|\text{sat}(F_n)| \leq 2^{\epsilon_k n}$.
2. $Z(F,\sigma,a) \geq (1-\epsilon_k)n$ *for all* σ *and all* $a \in \text{sat}(F_n)$.

Thus, the probability of PPSZ finding a satisfying assignment of F_n is at most $2^{-(1-2\epsilon_k)n}$. Furthermore, $\epsilon_k \leq \frac{C\log^2(k)}{k}$ for some universal constant C.

This theorem shows that PPSZ has exponentially small success probability. Also, it shows that the *strong exponential time hypothesis* (SETH) holds for PPSZ: As k grows, the advantage over the trivial success probability 2^{-n} becomes negligible.

The Probabilistic Construction

Let $A \in \mathbb{F}_2^{n\times n}$. The system $Ax = 0$ defines a Boolean function $f_A : \{0,1\}^n \to \{0,1\}$ as follows: $f_A(x) = 1$ if and only if $A \cdot x = 0$. Say A is *k-sparse* if every row of A has at most k nonzero entries. If A is k-sparse, then f_A can be written as a k-CNF formula with n variables and $2^{k-1}n$ clauses. Our construction will be probabilistic. For this, we define a distribution over k-sparse matrices in $\mathbb{F}_2^{n\times n}$. Our distribution will have the form $\mathcal{D}^n$, where $\mathcal{D}$ is a distribution over row vectors from $\mathbb{F}_2^n$. That is, we sample each row of A independently from $\mathcal{D}$. Let us describe $\mathcal{D}$. Define $\mathbf{e}_i \in \mathbb{F}_2^n$ to be the vector with a 1 at the i^{th} position and 0 elsewhere. Sample $i_1,\ldots,i_k \in \{1,\ldots,n\}$ uniformly and independently and let $X = \mathbf{e}_{i_1} + \cdots + \mathbf{e}_{i_k}$. Clearly, $X \in \mathbb{F}_2^n$ has at most k nonzero entries. This is our distribution $\mathcal{D}$.

Let A be a random matrix sampled as described, and write f_A as a k-CNF formula F. Note that $\text{sat}(F) = \ker A$. The challenge is to show that F satisfies the two conditions of Theorem 3.

Lemma 3 **(A has high rank)** *With probability* $1 - o(1)$, $|\ker(A)| \leq 2^{\epsilon_k n}$.

This shows that F satisfies the first condition of the theorem, i.e., it has few satisfying assignments. Lemma 3 is quite straightforward to prove, though not trivial. The next lemma shows that $Z(F,\sigma,a)$ is large.

Lemma 4 *With probability* $1-o(1)$, *it holds that* $Z(F,\sigma,a) \geq (1-\epsilon_k)n$ *for all permutations* σ *and all* $a \in \text{sat}(F)$.

Proving this lemma is the main technical challenge. The proof uses ideas from proof complexity (indeed, the above construction is inspired by constructions in proof complexity).

Open Problems

Suppose the true worst-case success probability of PPSZ on k-CNF formulas is $2^{-r_k n}$. Paturi, Pudlák, Saks, and Zane have proved that $r_k \leq 1 - \Omega(1k)$. Chen, Scheder, Talebanfard, and

Tang showed that $r_k \geq 1 - O\left(\frac{\log^2 k}{k}\right)$. Can one close this gap by construction harder instances or maybe even improve the analysis of PPSZ?

What is the average-case success probability of PPSZ on F when we sample A from $\mathcal{D}^n$? Note that F is exponentially hard with probability $1 - o(1)$, but this might leave a $1/n$ probability that F is very easy for PPSZ.

The construction of [1] is probabilistic. Can one make it explicit? The proof of Lemma 4 uses (implicit in [1]) a nonstandard notion of expansion. We do not know of explicit construction of those expanders.

Cross-References

- ▶ Backtracking Based k-SAT Algorithms
- ▶ Derandomization of k-SAT Algorithm
- ▶ Exact Algorithms and Strong Exponential Time Hypothesis
- ▶ Exact Algorithms and Time/Space Tradeoffs
- ▶ Exact Algorithms for General CNF SAT
- ▶ Exact Algorithms for k-SAT Based on Local Search
- ▶ Exact Algorithms for Maximum Two-Satisfiability
- ▶ Exact Graph Coloring Using Inclusion-Exclusion
- ▶ Random Planted 3-SAT
- ▶ Thresholds of Random k-Sat
- ▶ Unique k-SAT and General k-SAT

Recommended Reading

1. Chen S, Scheder D, Tang B, Talebanfard N (2013) Exponential lower bounds for the PPSZ k-SAT algorithm. In Khanna S (ed) SODA, New Orleans. SIAM, pp 1253–1263
2. Hertli T (2011) 3-SAT faster and simpler – unique-SAT bounds for PPSZ hold in general. In Ostrovsky R (ed) FOCS, Palm Springs. IEEE, pp 277–284
3. Paturi R , Pudlák P, Saks ME, Zane F (2005) An improved exponential-time algorithm for k-SAT. J ACM 52(3):337–364
4. Paturi R, Pudlák P, Zane F (1999) Satisfiability coding lemma. Chicago J Theor Comput Sci 11(19). (electronic)
5. Robin RA, Scheder D (2011) A full derandomization of Schöning's k-SAT algorithm. In Fortnow L Salil P, Vadhan (eds) STOC, San Jose. ACM, pp 245–252
6. Schöning U (1999) A probabilistic algorithm for k-SAT and constraint satisfaction problems. In FOCS '99: Proceedings of the 40th annual symposium on foundations of computer science, New York. IEEE Computer Society, Washington, DC, p 410

External Sorting and Permuting

Jeffrey Scott Vitter
University of Kansas, Lawrence, KS, USA

Keywords

Disk; External memory; I/O; Out-of-core; Permuting; Secondary storage; Sorting

Synonyms

Out-of-core sorting

Years and Authors of Summarized Original Work

1988; Aggarwal, Vitter

Problem Definition

Notations The main properties of magnetic disks and multiple disk systems can be captured by the commonly used *parallel disk model* (PDM), which is summarized below in its current form as developed by Vitter and Shriver [22]:

$$\begin{aligned}
N &= \text{problem size (in units of data items);}\\
M &= \text{internal memory size(inunitsofdata items);}\\
B &= \text{block transfer size (in units of data items);}\\
D &= \text{number of independent disk drives;}\\
P &= \text{number of CPUs,}
\end{aligned}$$

where $M < N$, and $1 \leq DB \leq M/2$. The data items are assumed to be of fixed length. In a single I/O, each of the D disks can simultaneously transfer a block of B contiguous data items. (In the original 1988 article [2], the D blocks per I/O were allowed to come from the same disk, which is not realistic.) If $P \leq D$, each of the P processors can drive about D/P disks; if $D < P$, each disk is shared by about P/D processors. The internal memory size is M/P per processor, and the P processors are connected by an interconnection network.

It is convenient to refer to some of the above PDM parameters in units of disk blocks rather than in units of data items; the resulting formulas are often simplified. We define the lowercase notation

$$n = \frac{N}{B}, \qquad m = \frac{M}{B}, \qquad q = \frac{Q}{B}, \qquad z = \frac{Z}{B} \tag{1}$$

to be the problem input size, internal memory size, query specification size, and query output size, respectively, in units of disk blocks.

The primary measures of performance in PDM are:

1. The number of I/O operations performed
2. The amount of disk space used
3. The internal (sequential or parallel) computation time

For reasons of brevity in this survey, focus is restricted onto only the first two measures. Most of the algorithms run in $O(N \log N)$ CPU time with one processor, which is optimal in the comparison model, and in many cases are optimal for parallel CPUs. In the word-based RAM model, sorting can be done more quickly in $O(N \log\log N)$ CPU time. Arge and Thorup [5] provide sorting algorithms that are theoretically optimal in terms of both I/Os and time in the word-based RAM model. In terms of auxiliary storage in external memory, algorithms and data structures should ideally use linear space, which means $O(N/B) = O(n)$ disk blocks of storage. Vitter [20] gives further details about the PDM model and provides optimal algorithms and data structures for a variety of problems. The content of this chapter comes largely from an abbreviated form of [19].

Problem 1 External sorting
INPUT: The input data records $R_0, R_1, R_2, \ldots$ are initially "striped" across the D disks, in units of blocks, so that record R_i is in block $\lfloor i/B \rfloor$ and block j is stored on disk $j \bmod D$.
OUTPUT: A striped representation of a permuted ordering $R_{\sigma(0)}, R_{\sigma(1)}, R_{\sigma(2)}, \ldots$ of the input records with the property that $key(R_{\sigma(i)} \leq key(R_{\sigma(i+1)}$ for all $i \geq 0$.

Permuting is the special case of sorting in which the permutation that describes the final position of the records is given explicitly and does not have to be discovered, for example, by comparing keys.

Problem 2 Permuting
INPUT: Same input assumptions as in external sorting. In addition, a permutation σ of the integers $\{0, 1, 2, \ldots, N-1\}$ is specified.
OUTPUT: A striped representation of a permuted ordering $R_{\sigma(0)}, R_{\sigma(1)}, R_{\sigma(2)}, \ldots$ of the input records.

Key Results

Theorem 1 ([2, 15]) *The average-case and worst-case number of I/Os required for sorting $N = nB$ data items using D disks is*

$$Sort(N) = \Theta\left(\frac{n}{D} \log_m n\right). \tag{2}$$

Theorem 2 ([2]) *The average-case and worst-case number of I/Os required for permuting N data items using D disks is*

$$\Theta\left(\min\left\{\frac{N}{D}, Sort(N)\right\}\right). \tag{3}$$

A more detailed lower bound is provided in (9) in section "Lower Bounds on I/O."

Matrix transposition is the special case of permuting in which the permutation can be represented as a transposition of a matrix from row-major order into column-major order.

Theorem 3 ([2]) *With D disks, the number of I/Os required to transpose a $p \times q$ matrix from row-major order to column-major order is*

$$\Theta\left(\frac{n}{D}\log_m \min\{M, p, q, n\}\right), \qquad (4)$$

where $N = pq$ and $n = N/B$.

Matrix transposition is a special case of a more general class of permutations called *bit-permute/complement* (BPC) permutations, which in turn is a subset of the class of *bit-matrix-multiply/complement* (BMMC) permutations. BMMC permutations are defined by a $\log N \times \log N$ nonsingular 0-1 matrix A and a $(\log N)$-length 0-1 vector c. An item with binary address x is mapped by the permutation to the binary address given by $Ax \oplus c$, where $\oplus$ denotes bitwise exclusive-or. BPC permutations are the special case of BMMC permutations in which A is a permutation matrix, that is, each row and each column of A contain a single 1. BPC permutations include matrix transposition, bit-reversal permutations (which arise in the FFT), vector-reversal permutations, hypercube permutations, and matrix re-blocking. Cormen et al. [8] characterize the optimal number of I/Os needed to perform any given BMMC permutation solely as a function of the associated matrix A, and they give an optimal algorithm for implementing it.

Theorem 4 ([8]) *With D disks, the number of I/Os required to perform the BMMC permutation defined by matrix A and vector c is*

$$\Theta\left(\frac{n}{D}\left(1 + \frac{\operatorname{rank}(\gamma)}{\log m}\right)\right), \qquad (5)$$

where γ is the lower-left $\log n \times \log B$ submatrix of A.

The two main paradigms for external sorting are *distribution* and *merging*, which are discussed in the following sections for the PDM model.

Sorting by Distribution

Distribution sort [12] is a recursive process that uses a set of $S - 1$ partitioning elements to partition the items into S disjoint buckets. All the items in one bucket precede all the items in the next bucket. The sort is completed by recursively sorting the individual buckets and concatenating them together to form a single fully sorted list.

One requirement is to choose the $S - 1$ partitioning elements so that the buckets are of roughly equal size. When that is the case, the bucket sizes decrease from one level of recursion to the next by a relative factor of $\Theta(S)$, and thus there are $O(\log_S n)$ levels of recursion. During each level of recursion, the data are scanned. As the items stream through internal memory, they are partitioned into S buckets in an online manner. When a buffer of size B fills for one of the buckets, its block is written to the disks in the next I/O, and another buffer is used to store the next set of incoming items for the bucket. Therefore, the maximum number of buckets (and partitioning elements) is $S = \Theta(M/B) = \Theta(m)$, and the resulting number of levels of recursion is $\Theta(\log_m n)$. How to perform each level of recursion in a linear number I/Os is discussed in [2, 14, 22].

An even better way to do distribution sort, and deterministically at that, is the BalanceSort method developed by Nodine and Vitter [14]. During the partitioning process, the algorithm keeps track of how evenly each bucket has been distributed so far among the disks. It maintains an invariant that guarantees good distribution across the disks for each bucket.

The distribution sort methods mentioned above for parallel disks perform write operations in complete stripes, which make it easy to write parity information for use in error correction and recovery. But since the blocks written in each stripe typically belong to multiple buckets, the buckets themselves will not be striped on the disks, and thus the disks must be used independently during read operations. In the write phase, each bucket must therefore keep track of the last block written to each disk so that the blocks for the bucket can be linked together.

An orthogonal approach is to stripe the contents of each bucket across the disks so that read operations can be done in a striped manner. As a result, the write operations must use disks independently, since during each write, multiple buckets will be writing to multiple stripes. Error correction and recovery can still be handled efficiently by devoting to each bucket one block-sized buffer in internal memory. The buffer is continuously updated to contain the exclusive-or (parity) of the blocks written to the current stripe, and after $D - 1$ blocks have been written, the parity information in the buffer can be written to the final (Dth) block in the stripe.

Under this new scenario, the basic loop of the distribution sort algorithm is, as before, to read one memory load at a time and partition the items into S buckets. However, unlike before, the blocks for each individual bucket will reside on the disks in contiguous stripes. Each block therefore has a predefined place where it must be written. With the normal round-robin ordering for the stripes (namely, $\ldots, 1, 2, 3, \ldots, D, 1, 2, 3, \ldots, D, \ldots$), the blocks of different buckets may "collide," meaning that they need to be written to the same disk, and subsequent blocks in those same buckets will also tend to collide. Vitter and Hutchinson [21] solve this problem by the technique of *randomized cycling*. For each of the S buckets, they determine the ordering of the disks in the stripe for that bucket via a random permutation of $\{1, 2, \ldots, D\}$. The S random permutations are chosen independently. If two blocks (from different buckets) happen to collide during a write to the same disk, one block is written to the disk and the other is kept on a write queue. With high probability, subsequent blocks in those two buckets will be written to different disks and thus will not collide. As long as there is a small pool of available buffer space to temporarily cache the blocks in the write queues, Vitter and Hutchinson [21] show that with high probability the writing proceeds optimally.

The randomized cycling method or the related merge sort methods discussed at the end of section "Sorting by Merging" are the methods of choice for sorting with parallel disks. Distribution sort algorithms may have an advantage over the merge approaches presented in section "Sorting by Merging" in that they typically make better use of lower levels of cache in the memory hierarchy of real systems, based upon analysis of distribution sort and merge sort algorithms on models of hierarchical memory.

Sorting by Merging

The *merge* paradigm is somewhat orthogonal to the distribution paradigm of the previous section. A typical merge sort algorithm works as follows [12]: In the "run formation" phase, the n blocks of data are scanned, one memory load at a time; each memory load is sorted into a single "run," which is then output onto a series of stripes on the disks. At the end of the run formation phase, there are $N/M = n/m$ (sorted) runs, each striped across the disks. (In actual implementations, "replacement selection" can be used to get runs of $2M$ data items, on the average, when $M \gg B$ [12].) After the initial runs are formed, the merging phase begins. In each pass of the merging phase, R runs are merged at a time. For each merge, the R runs are scanned and its items merged in an online manner as they stream through internal memory. Double buffering is used to overlap I/O and computation. At most $R = \Theta(m)$ runs can be merged at a time, and the resulting number of passes is $O(\log_m n)$.

To achieve the optimal sorting bound (2), each merging pass must be done in $O(n/D)$ I/Os, which is easy to do for the single-disk case. In the more general multiple-disk case, each parallel read operation during the merging must on the average bring in the next $\Theta(D)$ blocks needed for the merging. The challenge is to ensure that those blocks reside on different disks so that they can be read in a single I/O (or a small constant number of I/Os). The difficulty lies in the fact that the runs being merged were themselves formed during the previous merge pass. Their blocks were written to the disks in the previous pass without knowledge of how they would interact with other runs in later merges.

The Greed Sort method of Nodine and Vitter [15] was the first optimal deterministic EM algorithm for sorting with multiple disks. It works by relaxing the merging process with a final pass to fix the merging. Aggarwal and Plaxton [1] developed an optimal deterministic merge sort based upon the Sharesort hypercube parallel sorting algorithm. To guarantee even distribution during the merging, it employs two high-level merging schemes in which the scheduling is almost oblivious. Like Greed Sort, the Sharesort algorithm is theoretically optimal (i.e., within a constant factor of optimal), but the constant factor is larger than the distribution sort methods.

One of the most practical methods for sorting is based upon the *simple randomized merge sort* (SRM) algorithm of Barve et al. [7], referred to as "randomized striping" by Knuth [12]. Each run is striped across the disks, but with a random starting point (the only place in the algorithm where randomness is utilized). During the merging process, the next block needed from each disk is read into memory, and if there is not enough room, the least needed blocks are "flushed" (without any I/Os required) to free up space.

Further improvements in merge sort are possible by a more careful prefetching schedule for the runs. Barve et al. [6], Kallahalla and Varman [11], Shah et al. [17], and others have developed competitive and optimal methods for prefetching blocks in parallel I/O systems.

Hutchinson et al. [10] have demonstrated a powerful duality between parallel writing and parallel prefetching, which gives an easy way to compute optimal prefetching and caching schedules for multiple disks. More significantly, they show that the same duality exists between distribution and merging, which they exploit to get a provably optimal and very practical parallel disk merge sort. Rather than use random starting points and round-robin stripes as in SRM, Hutchinson et al. [10] order the stripes for each run independently, based upon the randomized cycling strategy discussed in section "Sorting by Distribution" for distribution sort. These approaches have led to successfully faster external memory sorting algorithms [9]. Clever algorithm engineering optimizations on multicore architectures have won recent big data sorting competitions [16].

Handling Duplicates: Bundle Sorting

For the problem of *duplicate removal*, in which there are a total of K distinct items among the N items, Arge et al. [4] use a modification of merge sort to solve the problem in $O\big(n \max\{1, \log_m (K/B)\}\big)$ I/Os, which is optimal in the comparison model. When duplicates get grouped together during a merge, they are replaced by a single copy of the item and a count of the occurrences. The algorithm can be used to sort the file, assuming that a group of equal items can be represented by a single item and a count.

A harder instance of sorting called *bundle sorting* arises when there are K distinct key values among the N items, but all the items have different secondary information that must be maintained, and therefore items cannot be aggregated with a count. Matias et al. [13] develop optimal distribution sort algorithms for bundle sorting using

$$O\big(n \max\{1, \log_m \min\{K, n\}\}\big) \tag{6}$$

I/Os and prove the matching lower bound. They also show how to do bundle sorting (and sorting in general) *in place* (i.e., without extra disk space).

Permuting and Transposition

Permuting is the special case of sorting in which the key values of the N data items form a permutation of $\{1, 2, \ldots, N\}$. The I/O bound (3) for permuting can be realized by one of the optimal sorting algorithms except in the extreme case $B \log m = o(\log n)$, where it is faster to move the data items one by one in a nonblocked way. The one-by-one method is trivial if $D = 1$, but with multiple disks, there may be bottlenecks on individual disks; one solution for doing the permuting in $O(N/D)$ I/Os is to apply the randomized balancing strategies of [22].

Matrix transposition can be as hard as general permuting when B is relatively large (say, $\frac{1}{2}M$) and N is $O(M^2)$, but for smaller B, the special structure of the transposition permutation makes transposition easier. In particular, the matrix can be broken up into square submatrices of B^2 elements such that each submatrix contains B

blocks of the matrix in row-major order and also B blocks of the matrix in column-major order. Thus, if $B^2 < M$, the transpositions can be done in a simple one-pass operation by transposing the submatrices one at a time in internal memory. Thonangi and Yang [18] discuss other types of permutations realizable with fewer I/Os than sorting.

Fast Fourier Transform and Permutation Networks

Computing the fast Fourier transform (FFT) in external memory consists of a series of I/Os that permit each computation implied by the FFT directed graph (or butterfly) to be done while its arguments are in internal memory. A permutation network computation consists of an oblivious (fixed) pattern of I/Os such that any of the $N!$ possible permutations can be realized; data items can only be reordered when they are in internal memory. A permutation network can be realized by a series of three FFTs.

The algorithms for FFT are faster and simpler than for sorting because the computation is nonadaptive in nature, and thus the communication pattern is fixed in advance [22].

Lower Bounds on I/O

The following proof of the permutation lower bound (3) of Theorem 2 is due to Aggarwal and Vitter [2]. The idea of the proof is to calculate, for each $t \geq 0$, the number of distinct orderings that are realizable by sequences of t I/Os. The value of t for which the number of distinct orderings first exceeds $N!/2$ is a lower bound on the average number of I/Os (and hence the worst-case number of I/Os) needed for permuting.

Assuming for the moment that there is only one disk, $D = 1$, consider how the number of realizable orderings can change as a result of an I/O. In terms of increasing the number of realizable orderings, the effect of reading a disk block is considerably more than that of writing a disk block, so it suffices to consider only the effect of read operations. During a read operation, there are at most B data items in the read block, and they can be interspersed among the M items in internal memory in at most $\binom{M}{B}$ ways, so the number of realizable orderings increases by a factor of $\binom{M}{B}$. If the block has never before resided in internal memory, the number of realizable orderings increases by an extra $B!$ factor, since the items in the block can be permuted among themselves. (This extra contribution of $B!$ can only happen once for each of the N/B original blocks.) There are at most $n + t \leq N \log N$ ways to choose which disk block is involved in the tth I/O (allowing an arbitrary amount of disk space). Hence, the number of distinct orderings that can be realized by all possible sequences of t I/Os is at most

$$(B!)^{N/B} \left(N(\log N) \binom{M}{B} \right)^t . \qquad (7)$$

Setting the expression in (7) to be at least $N!/2$, and simplifying by taking the logarithm, the result is

$$N \log B + t \left(\log N + B \log \frac{M}{B} \right) = \Omega(N \log N). \qquad (8)$$

Solving for t gives the matching lower bound $\Omega(n \log_m n)$ for permuting for the case $D = 1$. The general lower bound (3) of Theorem 2 follows by dividing by D.

Hutchinson et al. [10] derive an asymptotic lower bound (i.e., one that accounts for constant factors) from a more refined argument that analyzes both input operations and output operations. Assuming that $m = M/B$ is an increasing function, the number of I/Os required to sort or permute n indivisible items, up to lower-order terms, is at least

$$\frac{2N}{D} \frac{\log n}{B \log m + 2 \log N} \sim \begin{cases} \dfrac{2n}{D} \log_m n & \text{if } B \log m = \omega(\log N); \\ \dfrac{N}{D} & \text{if } B \log m = o(\log N). \end{cases} \qquad (9)$$

For the typical case in which $B \log m = \omega(\log N)$, the lower bound, up to lower order terms, is $2n \log_m n$ I/Os. For the pathological in which $B \log m = o(\log N)$, the I/O lower bound is asymptotically N/D.

Permuting is a special case of sorting, and hence the permuting lower bound applies also to sorting. In the unlikely case that $B \log m = o(\log n)$, the permuting bound is only $\Omega(N/D)$, and in that case the comparison model must be used to get the full lower bound (2) of Theorem 1 [2]. In the typical case in which $B \log m = \Omega(\log n)$, the comparison model is not needed to prove the sorting lower bound; the difficulty of sorting in that case arises not from determining the order of the data but from permuting (or routing) the data.

The proof used above for permuting also works for permutation networks, in which the communication pattern is oblivious (fixed). Since the choice of disk block is fixed for each t, there is no $N \log N$ term as there is in (7), and correspondingly there is no additive $\log N$ term in the inner expression as there is in (8). Hence, solving for t gives the lower bound (2) rather than (3). The lower bound follows directly from the counting argument; unlike the sorting derivation, it does not require the comparison model for the case $B \log m = o(\log n)$. The lower bound also applies directly to FFT, since permutation networks can be formed from three FFTs in sequence. The transposition lower bound involves a potential argument based upon a togetherness relation [2].

For the problem of bundle sorting, in which the N items have a total of K distinct key values (but the secondary information of each item is different), Matias et al. [13] derive the matching lower bound.

The lower bounds mentioned above assume that the data items are in some sense "indivisible," in that they are not split up and reassembled in some magic way to get the desired output. It is conjectured that the sorting lower bound (2) remains valid even if the indivisibility assumption is lifted. However, for an artificial problem related to transposition, removing the indivisibility assumption can lead to faster algorithms. Whether the conjecture is true is a challenging theoretical open problem.

Applications

Sorting and sorting-like operations account for a significant percentage of computer use [12], with numerous database applications. In addition, sorting is an important paradigm in the design of efficient EM algorithms, as shown in [20], where several applications can be found. With some technical qualifications, many problems that can be solved easily in linear time in internal memory, such as permuting, list ranking, expression tree evaluation, and finding connected components in a sparse graph, require the same number of I/Os in PDM as does sorting.

Open Problems

Several interesting challenges remain. One difficult theoretical problem is to prove lower bounds for permuting and sorting without the indivisibility assumption. Another question is to determine the I/O cost for each individual permutation, as a function of some simple characterization of the permutation, such as number of inversions. A continuing goal is to develop optimal EM algorithms and to translate theoretical gains into observable improvements in practice.

Many interesting challenges and opportunities in algorithm design and analysis arise from new architectures being developed. For example, Arge et al. [3] propose the *parallel external memory (PEM)* model for the design of efficient algorithms for chip multiprocessors, in which each processor has a private cache and shares a larger main memory with the other processors. The paradigms described earlier form the basis for efficient algorithms for sorting, selection, and prefix sums. Further architectures to explore include other forms of multicore architectures, networks of workstations, hierarchical storage devices, disk drives with processing capabilities, and storage devices based upon microelectromechanical systems (MEMS). Active (or intelligent) disks, in which disk drives have some processing capability and can filter information sent to the host, have been proposed to further reduce the I/O bottleneck, especially in large database applications. MEMS-based nonvolatile storage has the potential to serve as an intermediate level in the memory hierarchy between DRAM and disks. It could ultimately provide better latency and bandwidth than disks, at less cost per bit than DRAM.

URL to Code

Two systems for developing external memory algorithms are TPIE and STXXL, which can be downloaded from http://www.madalgo.au.dk/tpie/ and http://stxxl.sourceforge.net/, respectively. Both systems include subroutines for sorting and permuting and facilitate development of more advanced algorithms.

Cross-References

▸ I/O-Model

Recommended Reading

1. Aggarwal A, Plaxton CG (1994) Optimal parallel sorting in multi-level storage. In: Proceedings of the 5th ACM-SIAM symposium on discrete algorithms, Arlington, vol 5, pp 659–668
2. Aggarwal A, Vitter JS (1988) The input/output complexity of sorting and related problems. Commun ACM 31:1116–1127
3. Arge L, Goodrich MT, Nelson M, Sitchinava N (2008) Fundamental parallel algorithms for private-cache chip multiprocessors. In: Proceedings of the 20th symposium on parallelism in algorithms and architectures, Munich, pp 197–206
4. Arge L, Knudsen M, Larsen K (1993) A general lower bound on the I/O-complexity of comparison-based algorithms. In: Proceedings of the workshop on algorithms and data structures, Montréal. Lecture notes in computer science, vol 709, pp 83–94
5. Arge L, Thorup M (2013) RAM-efficient external memory sorting. In: Proceedings of the 24th international symposium on algorithms and computation, Hong Kong. Lecture notes in computer science, vol 8283, pp 491–501
6. Barve RD, Kallahalla M, Varman PJ, Vitter JS (2000) Competitive analysis of buffer management algorithms. J Algorithms 36:152–181
7. Barve RD, Vitter JS (2002) A simple and efficient parallel disk mergesort. ACM Trans Comput Syst 35:189–215
8. Cormen TH, Sundquist T, Wisniewski LF (1999) Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. SIAM J Comput 28:105–136
9. Dementiev R, Sanders P (2003) Asynchronous parallel disk sorting. In: Proceedings of the 15th ACM symposium on parallelism in algorithms and architectures, San Diego, pp 138–148
10. Hutchinson DA, Sanders P, Vitter JS (2005) Duality between prefetching and queued writing with parallel disks. SIAM J Comput 34:1443–1463
11. Kallahalla M, Varman PJ (2005) Optimal read-once parallel disk scheduling. Algorithmica 43:309–343
12. Knuth DE (1998) Sorting and searching. The art of computer programming, vol 3, 2nd edn. Addison-Wesley, Reading
13. Matias Y, Segal E, Vitter JS (2006) Efficient bundle sorting. SIAM J Comput 36(2):394–410
14. Nodine MH, Vitter JS (1993) Deterministic distribution sort in shared and distributed memory multiprocessors. In: Proceedings of the 5th ACM symposium on parallel algorithms and architectures, Velen, vol 5. ACM, pp 120–129
15. Nodine MH, Vitter JS (1995) Greed sort: an optimal sorting algorithm for multiple disks. J ACM 42:919–933
16. Rahn M, Sanders P, Singler J (2010) Scalable distributed-memory external sorting. In: Proceedings of the 26th IEEE international conference on data engineering, Long Beach, pp 685–688
17. Shah R, Varman PJ, Vitter JS (2004) Online algorithms for prefetching and caching on parallel disks. In: Proceedings of the 16th ACM symposium on parallel algorithms and architectures, Barcelona, pp 255–264
18. Thonangi R, Yang J (2013) Permuting data on random-access block storage. Proc VLDB Endow 6(9):721–732
19. Vitter JS (2001) External memory algorithms and data structures: dealing with massive data. ACM Comput Surv 33(2):209–271
20. Vitter JS (2008) Algorithms and data structures for external memory. Series on foundations and trends in theoretical computer science. Now Publishers, Hanover. (Also referenced as Volume 2, Issue 4 of Foundations and trends in theoretical computer science, Now Publishers)
21. Vitter JS, Hutchinson DA (2006) Distribution sort with randomized cycling. J ACM 53:656–680
22. Vitter JS, Shriver EAM (1994) Algorithms for parallel memory I: two-level memories. Algorithmica 12:110–147