Agentic Algorithm Engineering: Improving Shared-Memory Exact Minimum Cuts

David A. Bader* Adil Chhabra† Ernestine Großmann† Monika Henzinger‡
Alexander Noe§ Christian Schulz†

Abstract

The minimum cut problem for an undirected edge-weighted graph asks us to divide its set of nodes into two blocks while minimizing the weighted sum of the cut edges. Over the last years, we engineered a range of fast algorithms for this problem. Our fastest *exact* algorithm uses an *inexact* algorithm to obtain a better bound for the problem, reductions that depend on this bound, improved data structures and parallel contraction routines. It is available in the open-source package `VieCut` and, on real-world instances, outperformed the previously fastest solvers by a factor of up to 2.5 sequentially and up to 12.9 when run in parallel.

We improve this algorithm using *agentic algorithm engineering (AAE)*, a methodology that we introduce here, in which autonomous large language model agents run the algorithm engineering cycle on an existing code base: they form hypotheses about where running time is lost, implement them, benchmark the result on a fixed instance set and keep or discard the change. Even though we had already tuned our algorithm by hand extensively, the agent finds significant optimizations, in particular on the DIMACS core instances: factors of 1.28 (sequential) and 1.63 (32 threads) on real-world k-cores, and 6.26 and 127 on the DIMACS core instances.

1 Introduction

Given an undirected graph with non-negative edge weights, the *minimum cut problem* asks for a partition of the vertices into two sets that minimizes the total weight of the edges between them; its value is also called the *edge connectivity* of the graph [36, 26]. Minimum cuts are used in network reliability [28, 43], VLSI design [31] and as a subroutine in branch-and-cut algorithms for the traveling salesman and other combinatorial problems [40], so they have to be computed quickly on large inputs. All exact algorithms have non-linear running time, and those of Hao et al. [20] and Nagamochi et al. [36, 37], which are orders of magnitude faster than the others in practice [9, 24, 27], do not parallelize easily. Appendix A surveys related work.

Over the last years, we engineered a range of algorithms for the minimum cut problem and its variants. We first gave a practical shared-memory parallel *heuristic* algorithm, available in the open-source package `VieCut` [24, 25], which computes near-optimal and often exact minimum cuts very quickly. We then used this algorithm to obtain a better bound $\hat{\lambda}$ and engineered a shared-memory parallel *exact* algorithm [23] on top of it. It outperformed the previously fastest solvers, the implementations of the algorithms of Nagamochi et al. and of Hao and Orlin, by a factor of up to 2.5 sequentially and up to 12.9 when run in parallel on real-world instances.

*New Jersey Institute of Technology, Newark, NJ, USA
†Heidelberg University, Heidelberg, Germany. Christian Schulz is the corresponding author.
‡Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria
§Amazon.com, New York, NY, USA. Work done prior to joining Amazon.

Recently, large language models (LLMs) have been used not only to generate code, but also to discover and improve algorithms [45, 38]. This includes low-level optimizations that so far required a large amount of manual work. We use the algorithm engineering methodology as a starting point and introduce *agentic algorithm engineering*, a methodology in which autonomous LLM agents run the algorithm engineering cycle on an existing code base. The agent receives the code and a fixed set of benchmark instances, and modifies the code to make it faster on the whole set rather than on a single instance. Unlike classical auto-tuning [2], which searches a fixed parameter space of an unchanged code, the agent modifies the code itself, reaching changes without a tuning knob, such as replacing a data structure or repartitioning parallel work.

Contribution. *First*, we summarize our shared-memory parallel *exact* minimum cut algorithm [23] in a self-contained way. *Second*, we introduce *agentic algorithm engineering* (AAE), in which an autonomous LLM agent runs the algorithm engineering cycle on an existing code base: it inspects the code and the experimental infrastructure, forms hypotheses about where running time is lost, implements candidate optimizations and benchmarks them on a fixed set of instances. Improvements are kept as the starting point of the next round; the rest is discarded together with the reason for its failure, so that later iterations can build on it. *Third*, we run the agent, Claude Opus 5 [3], on our exact algorithm on the challenge instances and on the instances of the original publication. It finds further optimizations, factors of 1.28 sequentially and 1.63 at 32 threads on the real-world k-cores, and 6.26 and 127 on the DIMACS core instances, for about $350 in API usage (1.1 million output tokens). The agent tunes on the same 53 instances on which we report these speedups; we ask how much an agent gains by tuning a hand-optimized code to a given workload, not how the result fares on unseen instances (Section 4.1).

2 Preliminaries

Basic Concepts. Let $G = (V, E, c)$ be an undirected graph with $n = |V|$ vertices, $m = |E|$ edges and non-negative edge weights $c : E \to \mathbb{N}$, extended to edge sets by summation. The *degree* of a vertex is the total weight of its incident edges. A *cut* $(A, V \setminus A)$ partitions V into two non-empty *sides*; its *capacity* is the total weight of the edges between them. A *minimum cut* has smallest capacity, denoted by $\lambda(G)$ or simply λ, and the *minimum s-t-cut* $\lambda(G, s, t)$ is the smallest cut that separates the vertices s and t. $\hat{\lambda}$ denotes the smallest upper bound on λ discovered so far; since the *trivial cut* $(\{u\}, V \setminus \{u\})$ has capacity equal to the degree of u, the minimum degree serves as an initial bound. *Contracting* an edge $\{u, v\}$ merges u and v into one vertex that inherits their other edges, summing the weights of edges that become parallel.

Related Work. Exact minimum cut algorithms are either flow-based, computing the global minimum cut from maximum flow computations [13, 19, 20], or contraction-based, shrinking the graph while preserving at least one minimum cut [39, 36, 37, 29]. Recent theoretical work has lowered both the randomized and the deterministic bounds to near-linear [15, 16, 33, 21, 1], but none of these algorithms have been implemented. Appendix A gives a detailed overview. Our `VieCut` algorithm is available as the open-source package `VieCut` [22] as well as in the `CHSZLabLib` [10] via an easy-to-use Python interface. Since its publication it has been used as a minimum cut engine in a range of applications such as well-connected community detection [41, 44, 11], weighted connectivity augmentation [12], analysis of coalitional games [5] or in data reduction rules [47].

2.1 The Exact Minimum Cut Algorithm `VieCut`

To be self-contained, we summarize the main components of our shared-memory parallel *exact* minimum cut algorithm [23], which is the algorithm that we later optimize using agentic algorithm engineering. Everything described in this section is prior work. The algorithm is based on the contraction-based exact algorithm of Nagamochi et al. [36, 37], which we call `NOI`. Their algorithm uses a routine called CAPFOREST, which traverses the graph in a modified breadth-first order and computes a lower bound $q(e)$ of the connectivity $\lambda(G, u, v)$ for each edge $e = \{u, v\}$. If the connectivity between two vertices is at least as large as the current upper bound for the minimum cut, then the edge between them can be contracted. Hence, edges with $q(e) \geq \hat{\lambda}$ can be safely contracted, and the algorithm is guaranteed to find at least one such edge per run. This is repeated until only two vertices remain. The running time therefore depends on the bound $\hat{\lambda}$, which determines how many edges can be contracted, and on the cost of the priority queue operations in CAPFOREST; `VieCut` improved both and parallelized the routine.

Lowering the upper bound. The algorithm first runs the *heuristic* shared-memory parallel algorithm to obtain a better upper bound $\hat{\lambda}$ than the minimum degree. This heuristic [24] repeatedly computes a clustering with parallel label propagation [42], contracts each cluster into a single vertex and applies the local contraction tests of Padberg and Rinaldi [39]; once only a constant number of vertices remain, the contracted graph is solved exactly with `NOI`. The resulting cut is a cut of the input graph and thus a valid upper bound, but a cluster may contain vertices from both sides of a minimum cut, so it need not be optimal. In practice the heuristic algorithm almost always returns the exact minimum cut. As the contraction tests depend on $\hat{\lambda}$, a smaller bound allows more edges to be contracted in each round. This shrinks the graph faster and lowers the number of CAPFOREST runs.

Bounded priority queues. CAPFOREST uses a priority queue $\mathcal{Q}$ in which the key of a vertex is its connectivity to the already scanned vertices. We showed that the algorithm remains correct if the priorities in the queue are limited to $\hat{\lambda}$, meaning that elements in the queue having a key larger than $\hat{\lambda}$ are not updated. As the priorities are then integers bounded by $\hat{\lambda}$, $\mathcal{Q}$ can be implemented as a bucket priority queue with one bucket per attainable key; the implementation offers two bucket variants (FIFO and LIFO bucket order) alongside a binary heap.

Parallel CAPFOREST and contraction. We then adapted the algorithm so that contractible edges can be detected in parallel (Algorithm 1). Every process starts at a random vertex and runs a modified CAPFOREST, where every vertex is visited by only one process. This way, no process has to scan the whole graph, while the vertices in sparse regions of the graph, which might otherwise not be scanned by any process, are still scanned. We showed that the values $q(e)$ are still valid lower bounds, so that no cut smaller than $\hat{\lambda}$ is lost. Contraction is parallelized as well.

Overall algorithm. Lastly, everything is put together (Algorithm 2). The algorithm first runs the *heuristic* available in `VieCut` to obtain $\hat{\lambda}$ and then alternates parallel CAPFOREST and parallel graph contraction until only two vertices are left. In the unlikely case that no contractible edge was found, CAPFOREST is run sequentially. Contraction merges several vertices of the input graph into a single vertex of the contracted graph, which we call a *collapsed* vertex. The degree of a collapsed vertex is the capacity of the cut that separates the vertices it represents from the rest of the graph, and is therefore a valid upper bound on the minimum cut. Whenever we encounter a collapsed vertex whose degree is smaller than $\hat{\lambda}$, we can thus update the upper bound.

3 Agentic Algorithm Engineering

Algorithm engineering, as described by Sanders [46], organizes design, analysis, implementation and experimental evaluation into a cycle instead of a one-directional pipeline. Realistic machine models and real-world inputs are used instead of idealized assumptions, and experiments test falsifiable hypotheses about an implementation. Their outcome then drives the next round of design decisions. Reusable implementations are outcomes of this process as well.

Overview. In *agentic algorithm engineering* (AAE), an autonomous LLM agent runs this cycle.[1] A session is given four inputs: a *target program* whose source the agent may modify, a *metric* to be improved,[2] a *benchmark* that produces reproducible measurements and a *time budget* per experiment. The agent then runs an endless loop in which each iteration is one pass through the cycle. The first experiment of a session is always the baseline, so that all later results are relative to a measurement taken with the same benchmark on the same machine. All changes are kept under version control, so that experiments can be traced and reverted. Since an agent that optimizes for running time alone can easily produce code that is fast but wrong, every change is checked against a set of correctness assertions before it is benchmarked. The agent derives these assertions from the code, and the user can supply further ones.

The cycle. Each iteration has six steps. In the *model* step, the agent maintains an explicit picture of the hot paths, the hardware and where time is spent, and updates it whenever a measurement contradicts it. In the *design* step, it formulates a falsifiable hypothesis of the form "changing X improves Y by roughly Z, because ...", grounded in the current picture or in earlier results and changing one thing at a time. In the *analysis* step, it predicts direction and magnitude of the effect, which filters out experiments not worth running. In the *implementation* step, it applies a minimal change and commits it, so that every experiment is exactly one git commit. In the *experiment* step, it runs the benchmark, with independent runs concurrently if specified, and aggregates the running times over the instances by the geometric mean. In the *evaluation* step, it compares the result to the current best: improvements are kept and become the new reference point, regressions are reverted, crashes are repaired or given up on, and a small improvement that adds much complexity is not kept. The agent may change everything inside the target files: data structures, parameters, memory layout, control flow, parallelization, up to replacing a subcomponent by a different algorithm. It may not change the benchmark, the metric, the measurement methodology, the input data or the time budget, and it may not add dependencies. These constraints ensure that the agent can only improve the specified metric.

Bookkeeping. Every experiment, including the failed ones, is logged with its commit hash, the metric value, the resource usage, the decision and the hypothesis that motivated it. A session is therefore fully documented. In addition, a community-maintained knowledge base of earlier sessions can be consulted when generating hypotheses. The sessions in this paper did not use it; they started from the code base alone. In particular, the agent is not seeded with a curated list of optimization ideas: the hypotheses come from the model's own knowledge of algorithm and performance engineering, combined with what it observes in the code, the profiles and the measurements.

[1] `https://github.com/CHSZLab/AgenticAlgorithmEngineering`

[2] In this paper, since we have an exact algorithm, we optimize for running time; however the metric could also be solution quality of a heuristic algorithm or memory footprint of an algorithm.

4 Experiments and Results

Baseline. Our baseline is the exact `VieCut` algorithm [23], used unchanged; its original evaluation, on a different machine and on real-world instances only, is summarized in the introduction.

Setup/Instances. All experiments, including the benchmark runs of the agent sessions, run on a machine with an AMD EPYC 7702P processor and 1 TiB of main memory in a single NUMA domain, running Ubuntu 24.04. All code is written in C++, compiled with `g++` 13.3 using `-O3 -march=native`, and uses OpenMP for shared-memory parallelism. The benchmark has 53 instances. The 33 generated ones, three from each of eleven families, come from the generators of the DIMACS challenge, which are described by Chekuri et al. [9] and Levine [32] but were never published, so we reimplemented them from these descriptions; as the challenge designates the families of Chekuri et al. as its core benchmark, we call them the *DIMACS core instances*. The 20 real-world ones are four k-cores of each of the five web and social graphs of the original evaluation [23]. Appendix C lists every instance.

4.1 Agentic Optimization of the Exact Algorithm

We instantiate the methodology of Section 3 for `VieCut` in two sessions: a single-threaded session starting from the unmodified implementation, and a session at 32 threads starting from the result of the first. Together they comprise 44 experiments, of which 23 were accepted. An experiment is accepted only if the geometric mean of the running time over the 53 instances improves, every instance runs to completion, and every instance reports the same minimum cut as the baseline. Each experiment times every instance once; the acceptance statistic is the geometric mean over all 53 instances, so a small factor such as the 1.027 below, which would be within run-to-run variability on a single instance, is accepted only because it improves the whole set.

Agent setup. The agent is Claude with the Opus 5 model [3] (Anthropic), driven by the execution loop of Section 3. Session state lives on disk, not in the model's context window, which is finite and volatile: one benchmark run, build or profile prints far more than the agent needs, and a long conversation is compacted. Output is therefore redirected to log files, and the agent reads back only a few extracted lines per run, such as the geometric mean of the running time and the result of the correctness checks. What to read back is the design decision, since the model can only reason about what is in its context: too much buries the signal and costs tokens on every iteration, too little loses the experience of earlier experiments. The durable record is a ledger with one row per experiment, holding its hypothesis and outcome, one git commit per experiment and file snapshots for reverting refuted changes; a compaction of the conversation therefore loses nothing essential. The bottleneck of a session is the evaluation time of the 53-instance benchmark, not the agent's reasoning or implementation time.

The implementation in outline. Some details of the unmodified implementation are needed below. We call one iteration of CAPFOREST and contraction in the exact phase (Section 2.1) a *round*. A round costs $\Theta(n + m)$ time, and the number of rounds is the decisive quantity for the running time: on most graphs, a round contracts a constant fraction of the vertices and $\mathcal{O}(\log n)$ rounds suffice, while on the DIMACS core instances of Appendix C, which are constructed to resist contraction, a round merges a single vertex pair and $\Theta(n)$ rounds are needed. In the parallel algorithm, each thread runs a CAPFOREST *pass* from its own random start vertex. In the unmodified implementation, the passes share the array that marks visited vertices, so a vertex taken by one pass is skipped by the others and the passes divide the traversal between them. The graph is stored as one adjacency vector per vertex with an edge record of 40 bytes across five fields, of

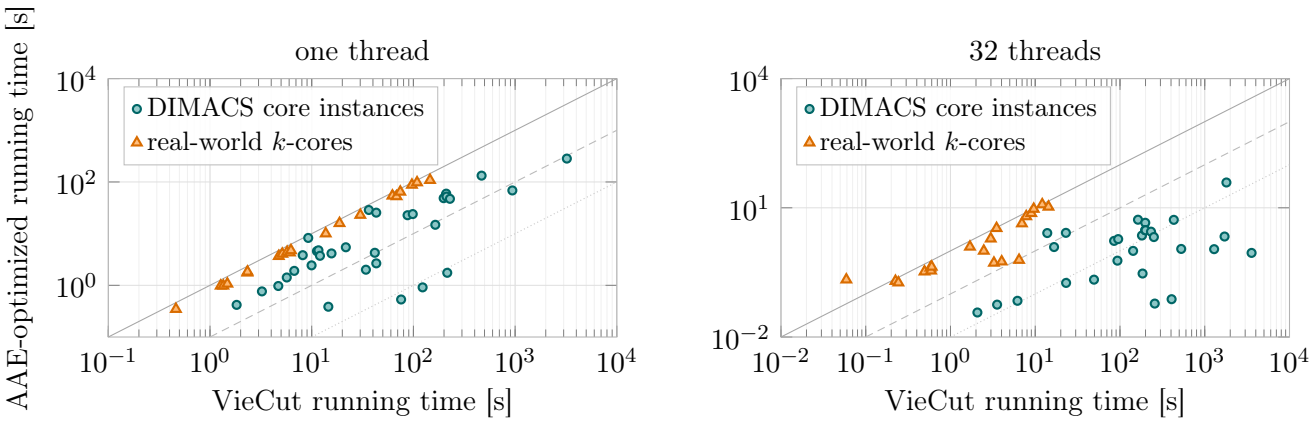

Figure 1: Running time of the AAE-optimized implementation against the unmodified one, one point per instance; the guides mark equality and speedups of 10 and 100. The 4 instances the unmodified implementation cannot solve at 32 threads are omitted from the right panel.

which CAPFOREST reads two, the target and the weight. Its inner loop runs once per directed edge per round and is the hottest path of the algorithm.

Overall improvement. On one thread, the accepted changes reduce the geometric mean of the running time over all 53 instances from 25.0 s to 7.28 s, a factor of 3.44, and the peak memory from 492 GiB to 193 GiB. At 32 threads, the unmodified implementation fails on 4 of the 53 instances even when granted the entire memory of the machine and two hours per instance, whereas the modified implementation solves all 53; over the 49 instances that all configurations solve, the accepted changes reduce the geometric mean of the running time from 23.0 s to 1.07 s, a factor of 21.49. Figure 1 shows the per-instance running times.

These factors have to be read against the instance set. Most DIMACS core instances are worst cases for contraction-based algorithms, where a round merges a single vertex pair so that any per-round cost is paid $\Theta(n)$ instead of $\mathcal{O}(\log n)$ times, and several have minimum cuts orders of magnitude larger than those of the original evaluation, for which data structures sized by the cut value were appropriate. Split accordingly, the accepted changes are worth factors of 1.28 on one thread and 1.63 at 32 threads on the 20 real-world k-cores, the kind of instance the implementation was tuned on, and 6.26 and 127 on the DIMACS core instances; all 4 instances that the unmodified implementation cannot solve are DIMACS core instances.

Single-threaded session. Of the 16 experiments of this session, 11 were accepted. The two largest improvements remove costs that grow with the minimum cut value and the graph size, respectively. Every speedup quoted for an individual change below is the improvement of the geometric mean of the running time over all 53 instances, measured by the agent against the code with all previously accepted changes. (1) CAPFOREST keys its priority queue by weights in $[0, \hat{\lambda}]$, so the bucket variants of the queue, which hold one bucket per attainable key, are sized by the value of the minimum cut rather than by the graph. Each bucket is a `std::deque` that allocates its first block on construction, about a kilobyte per attainable key: on a weighted instance with 8000 vertices and a minimum cut of $1.6 \cdot 10^8$, the queue alone requires roughly 200 GiB, rebuilt in every round. Selecting a binary heap, whose size is $\mathcal{O}(n)$, whenever the cut value exceeds both the number of vertices and 10 000 yields a speedup of 1.544; the instance above goes from 215 s and 202 GiB to 3.5 s and 2.2 GiB. (2) Both the sequential and the parallel exact loop rebuilt the contracted graph in every round and kept all intermediate graphs alive, a history that is only

ever read when the sides of the cut are requested rather than its value. On the families where a round merges a single pair, this costs $\Theta(n + m)$ time, and one live graph, per constant amount of progress. Contracting in place, keeping a single copy of the graph, yields speedups of 1.277 and 1.216 in the two loops. (3) The heuristic only coarsens graphs above 10 000 vertices and finishes with a sequential exact solve, so on inputs with $n < 10\,000$, the value it returns is not a bound but the exact minimum cut; returning it directly yields a speedup of 1.093. The remaining accepted changes contribute one to five percent each by removing work from the inner loop: (4) flags stored as bytes instead of bit-packed `std::vector<bool>`, whose probes cost a shift and a mask; (5) one lookup of a vertex's queue position per update instead of two; (6) resolving a vertex's adjacency vector once instead of once per edge; and (7) omitting, in the contraction, an allocation per vertex per round for group lists that are never read.

Parallel session. This session starts from the code produced by the single-threaded session, whose changes are effective in parallel as well: at 32 threads, that code already lowers the geometric mean of the running time over the 49 common instances from 23.0 s to 3.65 s, a factor of 6.29. The changes accepted here contribute a further factor of 3.42, composing to the 21.49 reported above. Of the 28 experiments in this session, 12 were accepted. The five changes described below are the largest ones. (1) One source of improvement is the treatment of the visited-vertex array in concurrent CAPFOREST passes, where the better strategy turns out to be instance dependent. Sharing the array lets a pass skip vertices already visited by another thread, reducing redundant work, but fragments what each pass sees and can therefore reduce the number of contractible edges found in a round. With private arrays, each pass performs a complete traversal, increasing work but potentially finding more contractions. Neither strategy consistently dominates: private arrays are about ten times faster on the random graphs, whereas sharing is several times faster on the real-world instances. Alternating the two settings over the first rounds and retaining the better one yields a speedup of 1.175.

(2) A second bottleneck occurs on small graphs. For instances with fewer than 10 000 vertices, the heuristic algorithm used by the original implementation computes the upper bound $\hat{\lambda}$ using a sequential solver rather than running label propagation contraction. Fifteen of the 53 instances fall below this threshold, so this sequential step becomes a bottleneck while the remaining threads are idle. For these instances, we instead skip the heuristic upper bound computation, initialize $\hat{\lambda}$ with the minimum vertex degree, and proceed directly with the parallel reductions and CAPFOREST. This yields a speedup of 1.474, with one instance improving from 22.6 s to 0.58 s. The dense families slow down by about five percent because they begin with a weaker upper bound.

(3) At 32 threads, contraction rather than traversal dominates the remaining running time: measured per round, it accounts for about 80 percent of the runtime and is largely sequential. The implementation partitions the work by vertices before contraction, so different threads can map different vertices to the same contracted vertex and generate the same contracted edge. Combining the weights of these duplicate edges requires a concurrent hash table keyed by the endpoint pair, while constructing the contracted graph from the table inserts edges one at a time inside a critical section, serializing this step. Partitioning instead by contracted vertex removes both bottlenecks: each thread owns the neighbor lists of its assigned contracted vertices, so no two threads write to the same entry, and parallel edges are summed in a thread-private dense array that fits into cache once only a few thousand vertices remain. Since each contracted vertex forms one unit of work regardless of how many pre-contraction vertices it contains, this approach is used only when the groups are of comparable size. On one social-network instance, for example, the first round reduces 13 million vertices to a few thousand, and one very large group would otherwise occupy a single thread for most of the round. Together, these changes yield a speedup of 1.294.

Two smaller optimizations provide further gains. (4) The edge record is reduced from 40 to 24 bytes. Two of its five fields are used only by the maximum flow algorithms that `VieCut` also contains, which are never called by the exact minimum cut path, so they are compiled out of the minimum cut binaries. Since each CAPFOREST traversal scans every edge once per round, the smaller record reduces memory traffic and yields a speedup of 1.131. (5) A third priority queue stores each bucket as an intrusive linked list and targets instances between the two existing queue variants, where the cut is small enough for buckets but most buckets remain empty, yielding a speedup of 1.027.

Correctness. Matching the baseline on all 53 instances is validation, not proof. Some accepted changes preserve the published invariants [23] by construction: the algorithm is correct for any valid upper bound $\hat{\lambda}$, the lower-bound argument holds however the passes divide the traversal, and the queue choice only reorders the search. The remaining changes are implementation-level and checked empirically, by the assertions of Section 3 and the identical cuts. A human read all 23 accepted diffs and checked each of them for correctness.

Tuning and the instance set. The thresholds in the accepted changes, such as the heap selection and the small-graph routing at 10 000 vertices, were fitted on the same 53 instances the speedups are reported on, since the methodology deliberately tunes for the workload it is given. The risk of overfitting is bounded: acceptance requires improving the geometric mean over the whole set, the accepted changes improve both the real-world k-cores and the DIMACS core instances, and the largest improvements remove asymptotic bottlenecks rather than fit constants. Users may re-tune the thresholds on their own workload; the effect on instances held out from tuning remains to be evaluated.

5 Discussion

We introduced *agentic algorithm engineering*, in which autonomous LLM agents run the algorithm engineering cycle under correctness assertions, and applied it to our extensively hand-tuned shared-memory parallel exact minimum cut algorithm. The accepted changes fall into three categories: the largest remove implicit assumptions of the original tuning that the DIMACS core instances violate, such as a priority queue sized by the cut value; a broad share is standard performance engineering that needs no insight into minimum cuts; and a few alter decisions rather than execution, such as probing whether the passes share the visited array. The agent did not invent a new algorithm, and the assertions fix its output to the baseline's cuts; the sessions show that the complete cycle can be run at a cost low enough to engineer regimes outside the original design envelope. Running the process was not effortless. An agent that optimizes running time alone produces code that is fast but wrong and chases changes that help one instance while slowing the set, so the correctness assertions, the geometric mean over the whole set as the only acceptance criterion and a human review of every accepted diff were necessary; 21 of the 44 experiments were discarded. The context window had to be managed explicitly, with all session state on disk, and the benchmark was the bottleneck: every experiment costs a full run over 53 instances, so a session is paced by machine time rather than by the agent. Future work includes applying the methodology to other well-engineered codes and to metrics beyond running time, such as the solution quality of heuristics or the memory footprint of an implementation, including trade-offs between them.

Acknowledgements. This research was funded in part by NSF grant numbers CCF-2109988, OAC-2402560, and CCF-2453324 (Bader), and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), project number 519626652 (Schulz).

References

[1] Daniel Anderson and Guy E. Blelloch. Parallel minimum cuts in $O(m \log^2 n)$ work and low depth. *ACM Transactions on Parallel Computing*, 2023. doi:10.1145/3565557.

[2] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman P. Amarasinghe. OpenTuner: an extensible framework for program autotuning. In *International Conference on Parallel Architectures and Compilation (PACT)*, pages 303–316. ACM, 2014. doi:10.1145/2628071.2628092.

[3] Anthropic. System Card: Claude Opus 5. https://www.anthropic.com/claude-opus-5-system-card, July 2026. Accessed 2026-08-26.

[4] D. Bader, A. Kappes, H. Meyerhenke, P. Sanders, C. Schulz, and D. Wagner. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*. Springer, 2014.

[5] Hodaya Barr, Yohai Trabelsi, Sarit Kraus, Liam Roditty, and Noam Hazon. The complexity of manipulation of k-coalitional games on graphs. In *27th European Conference on Artificial Intelligence (ECAI)*, volume 392 of *Frontiers in Artificial Intelligence and Applications*, pages 3388–3396, 2024. doi:10.3233/FAIA240889.

[6] Vladimir Batagelj and Matjaz Zaversnik. An $O(m)$ algorithm for cores decomposition of networks. *arXiv preprint cs/0310049*, 2003.

[7] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In Sadagopan Srinivasan, Krithi Ramamritham, Arun Kumar, M. P. Ravindra, Elisa Bertino, and Ravi Kumar, editors, *Proceedings of the 20th International Conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[8] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[9] Chandra S. Chekuri, Andrew V. Goldberg, David R. Karger, Matthew S. Levine, and Cliff Stein. Experimental study of minimum cut algorithms. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '97)*, pages 324–333. SIAM, 1997.

[10] CHSZ Lab. CHSZLabLib – algorithm library of the CHSZ Lab. https://github.com/CHSZLab/CHSZLabLib.

[11] Mohammad Dindoost, Oliver Alvarado Rodriguez, Bartosz Bryg, Minhyuk Park, George Chacko, Tandy Warnow, and David A. Bader. On the optimization of methods for establishing well-connected communities. *arXiv preprint arXiv:2509.02590*, 2025.

[12] Marcelo Fonseca Faraj, Ernestine Großmann, Felix Joos, Thomas Möller, and Christian Schulz. Engineering weighted connectivity augmentation algorithms. In *22nd International Symposium on Experimental Algorithms (SEA)*, volume 301 of *LIPIcs*, pages 11:1–11:22, 2024. doi:10.4230/LIPIcs.SEA.2024.11.

[13] Lester R. Ford and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8(3):399–404, 1956.

[14] Harold N Gabow. A matroid approach to finding edge connectivity and packing arborescences. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 112–122. ACM, 1991.

[15] Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Minimum cut in $O(m \log^2 n)$ time. *Theory of Computing Systems*, 68(4):814–834, 2024. doi:10.1007/s00224-024-10179-7.

[16] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1260–1279. SIAM, 2020. doi:10.1137/1.9781611975994.77.

[17] Lukas Gianinazzi, Pavel Kalvoda, Alessandro De Palma, Maciej Besta, and Torsten Hoefler. Communication-avoiding parallel minimum cuts and connected components. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 219–232. ACM, 2018.

[18] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[19] Ralph E. Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

[20] Jianxiu Hao and James B. Orlin. A faster algorithm for finding the minimum cut in a graph. In *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174. Society for Industrial and Applied Mathematics, 1992.

[21] Monika Henzinger, Jason Li, Satish Rao, and Di Wang. Deterministic near-linear time minimum cut in weighted graphs. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3089–3139. SIAM, 2024.

[22] Monika Henzinger, Alexander Noe, and Christian Schulz. VieCut – shared-memory parallel minimum cut algorithms. https://github.com/KaHIP/VieCut.

[23] Monika Henzinger, Alexander Noe, and Christian Schulz. Shared-memory exact minimum cuts. In *33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 13–22, 2019. doi:10.1109/IPDPS.2019.00013.

[24] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 48–61. SIAM, 2018.

[25] Monika Henzinger, Alexander Noe, Christian Schulz, and Darren Strash. Practical minimum cut algorithms. *ACM Journal of Experimental Algorithmics*, 23:1.8:1–1.8:22, 2018. doi:10.1145/3274662.

[26] Monika Henzinger, Satish Rao, and Di Wang. Local flow partitioning for faster edge connectivity. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1919–1938. SIAM, 2017.

[27] Michael Jünger, Giovanni Rinaldi, and Stefan Thienel. Practical performance of efficient minimum cut algorithms. *Algorithmica*, 26(1):172–195, 2000.

[28] David R. Karger. A randomized fully polynomial time approximation scheme for the all-terminal network reliability problem. *SIAM Review*, 43(3):499–522, 2001.

[29] David R Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, 1996.

[30] Ken-ichi Kawarabayashi and Mikkel Thorup. Deterministic global minimum cut of a simple graph in near-linear time. In *Proceedings of the 47th Annual ACM Symposium on Theory of Computing*, pages 665–674. ACM, 2015.

[31] Balakrishnan Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, 33(5):438–446, 1984.

[32] Matthew S. Levine. Experimental study of minimum cut algorithms. Master's thesis, Massachusetts Institute of Technology, 1997. URL: `http://dspace.mit.edu/bitstream/handle/1721.1/42665/37658972-MIT.pdf`.

[33] Jason Li. Deterministic mincut in almost-linear time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, pages 384–395. ACM, 2021.

[34] Jason Li and Debmalya Panigrahi. Deterministic minimum cut in poly-logarithmic maximum flows. *Journal of the ACM*, 72(4):29:1–29:18, 2025. `doi:10.1145/3744737`.

[35] David W. Matula. A linear time $2 + \varepsilon$ approximation algorithm for edge connectivity. In *Proceedings of the 4th annual ACM-SIAM Symposium on Discrete Algorithms*, pages 500–504. SIAM, 1993.

[36] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.

[37] Hiroshi Nagamochi, Tadashi Ono, and Toshihide Ibaraki. Implementing an efficient minimum capacity cut algorithm. *Mathematical Programming*, 67(1):325–341, 1994.

[38] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[39] Manfred Padberg and Giovanni Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47(1):19–36, 1990.

[40] Manfred Padberg and Giovanni Rinaldi. A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review*, 33(1):60–100, 1991.

[41] Minhyuk Park, Yasamin Tabatabaee, Vikram Ramavarapu, Baqiao Liu, Vidya Kamath Pailodi, Rajiv Ramachandran, Dmitriy Korobskiy, Fabio Ayres, George Chacko, and Tandy Warnow. Well-connectedness and community detection. *PLOS Complex Systems*, 1(3):e0000009, 2024. `doi:10.1371/journal.pcsy.0000009`.

[42] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.

[43] Aparna Ramanathan and Charles J. Colbourn. Counting almost minimum cutsets with reliability applications. *Mathematical Programming*, 39(3):253–261, 1987.

[44] Vikram Ramavarapu, Fábio José Ayres, Minhyuk Park, Vidya Kamath Pailodi, João Alfredo Cardoso Lamy, Tandy Warnow, and George Chacko. CM++ - a meta-method for well-connected community detection. *Journal of Open Source Software*, 9(93):6073, 2024. doi:10.21105/joss.06073.

[45] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi:10.1038/s41586-023-06924-6.

[46] Peter Sanders. Algorithm engineering – an attempt at a definition. In *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 321–340. Springer, 2009. doi:10.1007/978-3-642-03456-5_22.

[47] Hjalmar Schulz, André Nichterlein, Rolf Niedermeier, and Christopher Weyand. Applying a cut-based data reduction rule for weighted cluster editing in polynomial time. In *17th International Symposium on Parameterized and Exact Computation (IPEC)*, volume 249 of *LIPIcs*, pages 25:1–25:14, 2022. doi:10.4230/LIPIcs.IPEC.2022.25.

[48] Stephen B Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

[49] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997.

A Detailed Related Work

Ford and Fulkerson [13] proved that the minimum s-t-cut equals the maximum s-t-flow, and Gomory and Hu [19] observed that the global minimum cut can be computed with $n-1$ minimum s-t-cut computations. Hence, improved maximum flow algorithms such as push-relabel [18] were used to obtain better global minimum cut algorithms [29]. Hao and Orlin [20] adapt push-relabel to pass information to future flow computations, achieving a total running time of $O(mn \log \frac{n^2}{m})$. Gabow [14] gives an algorithm running in $O(m+\lambda^2 n \log(n/\lambda))$. Flow-free approaches contract edges instead. Padberg and Rinaldi [39] give four local tests that identify edges whose contraction preserves at least one minimum cut; applying them exhaustively costs $O(nm)$ time. Chekuri et al. [9] instead apply the tests in linear-time sweeps over the graph, which may miss some contractible edges, and repeat the sweeps only while they shrink the graph by a constant factor. This costs $O(m \log n)$ time in total and finds almost as many contractions in practice. Nagamochi et al. [36, 37] repeatedly use maximum spanning forests to find contractible edges, running in $O(mn + n^2 \log n)$. Stoer and Wagner [49] give a simpler variant of the same asymptotic complexity but significantly worse practical performance [27]. Among algorithms with better bounds, Kawarabayashi and Thorup [30] give a deterministic $O(m \log^{12} n)$ algorithm, later improved by Henzinger et al. [26] to $O(m \log^2 n \log\log^2 n)$. Matula [35] gives a linear time $(2+\varepsilon)$-approximation. Karger and Stein [29] give a randomized algorithm based on random edge contractions. The algorithms of Hao and Orlin, Nagamochi et al., Padberg and Rinaldi, Stoer and Wagner, and Karger and Stein have all been implemented and compared experimentally [9, 27, 32], whereas, to our knowledge, those of Kawarabayashi and Thorup and of Henzinger et al. have not.

Gawrychowski et al. [15] improve the randomized bound of Karger for weighted graphs to $\mathcal{O}\big(m \log^2 n\big)$, and for simple graphs Ghaffari et al. [16] give an algorithm with running time $\mathcal{O}(m \log n)$ that uses random 2-out contractions. Li and Panigrahi [34] introduce the *isolating cuts* technique, Li [33] then gives a deterministic algorithm with running time $m^{1+o(1)}$, and Henzinger et al. [21] give a deterministic $\tilde{\mathcal{O}}(m)$ algorithm for weighted graphs. Anderson and Blelloch [1] give a parallel algorithm with $\mathcal{O}\big(m \log^2 n\big)$ work and polylogarithmic depth, which matches the best sequential work bound. None of the algorithms of this paragraph have been implemented, so their performance in practice is unknown. Moreover, to the best of our knowledge, the only distributed implementation for the problem is still the one of Gianinazzi et al. [17], and there is no GPU implementation of the problem.

B Pseudocode

We give the pseudocode of the two routines described in Section 2.1. Algorithm 1 is the parallel variant of CAPFOREST that finds contractible edges and Algorithm 2 is the overall minimum cut algorithm. Both are taken from the original paper [23], which also contains the correctness proofs.

Algorithm 1 Parallel CAPFOREST

Input: $G = (V, E, c) \leftarrow$ undirected graph $\hat{\lambda} \leftarrow$ upper bound for minimum cut, $\mathcal{T} \leftarrow$ shared array of vertex visits
Output: $\mathcal{U} \leftarrow$ union-find data structure to mark contractible edges

Label all vertices $v \in V$ "unvisited", blacklist $\mathcal{B}$ empty
$\forall v \in V : r(v) \leftarrow 0$
$\forall e \in E : q(e) \leftarrow 0$, $\alpha \leftarrow 0$
$\mathcal{Q} \leftarrow$ empty priority queue
Insert random vertex into $\mathcal{Q}$
while $\mathcal{Q}$ not empty **do**
    $x \leftarrow \mathcal{Q}$.pop_max() ▷ Choose unvisited vertex with highest priority
    Mark x "visited"
    if $\mathcal{T}(x) =$ True **then** ▷ Every vertex is visited only once
        $\mathcal{B}(x) \leftarrow$ True
    else
        $\mathcal{T}(x) \leftarrow$ True
        $\alpha \leftarrow \alpha + c(x) - 2r(x)$
        $\hat{\lambda} \leftarrow min(\hat{\lambda}, \alpha)$
        for $e = \{x, y\} \leftarrow$ unscanned edge, where $y \notin \mathcal{B}$ **do**
            if $r(y) < \hat{\lambda} \leq r(y) + c(e)$ **then**
                $\mathcal{U}$.union(x,y) ▷ Mark edge e to contract
            end if
            $r(y) \leftarrow r(y) + c(e)$
            $q(e) \leftarrow r(y)$
            $\mathcal{Q}(y) \leftarrow min(r(y), \hat{\lambda})$
        end for
    end if
end while

Algorithm 2 Parallel Minimum Cut

Input: $G = (V, E, c)$

$\hat{\lambda} \leftarrow$ `VieCut`(G), $G_C \leftarrow G$
while G_C has more than 2 vertices **do**
    $\hat{\lambda} \leftarrow$ Parallel CAPFOREST$(G_C, \hat{\lambda})$
    if no edges marked contractible **then**
        $\hat{\lambda} \leftarrow$ CAPFOREST$(G_C, \hat{\lambda})$
    end if
    $G_C, \hat{\lambda} \leftarrow$ Parallel Graph Contract(G_C)
end while
return $\hat{\lambda}$

C Instance Set

Table 1 lists the instances we evaluate on: families we generate, the *DIMACS core instances*, and k-cores of real-world web and social graphs. For each one the table gives its size and its minimum cut λ.

Generated Families: The DIMACS Core Instances

The generated instances come from seven generators that reimplement the problem families of Chekuri et al. [9] and of Levine [32], and every family label in Table 1 maps to its generator as follows: unions of random cycles (`reg2`, `reg3` and `regsweep`, whose three instances arise from sweeping the graph size, the seed and the cut value, respectively), random graphs with heavy components (`noi1`, sweeping the size, and `noisweep`, sweeping the remaining parameters), regular (`random`) and irregular (`irreg`) random graphs, the generator of Padberg and Rinaldi [39] (`pr1` and `pr2`, its two types), bicycle wheels (`bikewheel`) and pairs of interleaved cycles (`dblcyc`). Those generators were never published, so ours follow the descriptions given there. Each takes the number of vertices, the parameters in the table and, where the generator is randomized, the seed s given in the table, and writes the METIS format; they are deterministic, so the set is reproducible from the table alone rather than from stored files.

At the sizes the study states, our algorithm finishes in milliseconds, so we enlarge every family until it takes seconds to minutes and keep the three slowest instances of each. Memory, not time, sets how far this goes: where the minimum cut is small the algorithm retains one contracted graph per round and needs $\Theta(n \cdot m)$, which stops `reg2` at $n = 12\,800$ and `irreg` at $n = 64\,000$; for `pr2` the cut itself grows like n^2 and the priority queue is sized by it, so 8 000 vertices already need hundreds of gigabytes. Regular random graphs are at the other extreme and scale to $n = 1\,000\,000$. These limits are limits of the *unmodified* implementation; the single-threaded session of Section 4.1 later removes exactly this per-round history and the cut-sized queue. The set was fixed before the sessions, so the instance sizes, and with them the reported factors, are capped by what the baseline could run at all.

Bicycle wheels and interleaved cycles are the adversarial cases: the first give all n trivial cuts the same value, the second hide one cut of 2000 below $\Theta(n^2)$ cuts of value 2006. Both defeat the tests of Padberg and Rinaldi, leaving a contraction-based algorithm no edge it may contract cheaply, and are therefore where such algorithms degrade first.

A dagger marks a minimum cut that the construction fixes in advance. Every other value was confirmed by two independent implementations: the Nagamochi-Ibaraki code of `VieCut` and, depending on the size, either a Stoer-Wagner implementation that shares no code with it or the parallel exact algorithm of Section 2.1. They agree on every instance, and with the predicted values.

Real-World Graphs

The second group are the web and social graphs [4, 7, 8] already used to evaluate the shared-memory exact algorithm [23], built the same way: as they are disconnected and full of low-degree vertices, we take four k-cores [48, 6] of each whose minimum cut differs from the minimum degree, and use the largest connected component. They are far larger than the DIMACS core instances, up to 68 141 228 vertices and 3 134 185 720 edges, yet are solved much faster: the algorithm contracts them aggressively, while the DIMACS core instances are built so that it cannot.

Table 1: The evaluation set. For every instance we give the number of vertices and edges and the value λ of the minimum cut. A dagger marks a minimum cut that the construction of the family fixes in advance; every other value was computed. The upper block is generated (the DIMACS core instances), the lower block consists of k-cores of real-world graphs. For the generated block, the parameters include the seed s where the generator is randomized, so each instance is reproduced by its row.

Instance	Parameters	n	m	λ
generated families (DIMACS core instances)				
`reg2`	$c = 50$, $s = 1$	3 200	157 664	$100^\dagger$
`reg2`	$c = 50$, $s = 1$	6 400	317 541	$100^\dagger$
`reg2`	$c = 50$, $s = 1$	12 800	637 542	$100^\dagger$
`reg3`	$c = 2$, $s = 2$	65 536	131 071	$4^\dagger$
`reg3`	$c = 2$, $s = 1$	65 536	131 072	$4^\dagger$
`reg3`	$c = 2$, $s = 3$	65 536	131 068	$4^\dagger$
`regsweep`	$c = 32$, $s = 1$	2 000	63 025	$64^\dagger$
`regsweep`	$c = 16$, $s = 1$	2 000	31 783	$32^\dagger$
`regsweep`	$c = 8$, $s = 1$	2 000	15 946	$16^\dagger$
`noi1`	$d = 20$, $k = 4$, $P = 100$, $s = 1$	8 000	6 399 200	1 608 874
`noi1`	$d = 20$, $k = 4$, $P = 100$, $s = 1$	12 000	14 398 800	2 542 217
`noi1`	$d = 10$, $k = 4$, $P = 100$, $s = 1$	16 000	12 799 200	1 646 880
`noisweep`	$d = 100$, $k = 4$, $P = 100$, $s = 1$	4 000	7 998 000	4 761 257
`noisweep`	$d = 20$, $k = 4$, $P = 1000$, $s = 1$	4 000	1 599 600	7 423 512
`noisweep`	$d = 20$, $k = 1$, $P = 100$, $s = 1$	8 000	6 399 200	7 261 322
`random`	$d = 32$, $s = 1$	400 000	6 399 736	30
`random`	$d = 16$, $s = 1$	1 000 000	7 999 939	14
`random`	$d = 8$, $s = 1$	1 000 000	3 999 989	6
`irreg`	$\lambda = 4$, $e = 32000$, $s = 1$	32 000	95 993	6
`irreg`	$\lambda = 4$, $e = 64000$, $s = 1$	64 000	191 995	6
`irreg`	$\lambda = 4$, $e = 0$, $s = 1$	64 000	127 999	$4^\dagger$
`pr1`	type 1, $d = 20$, $s = 1$	8 000	6 400 003	72 454
`pr1`	type 1, $d = 20$, $s = 1$	12 000	14 400 216	110 938
`pr1`	type 1, $d = 10$, $s = 1$	16 000	12 807 052	70 611
`pr2`	type 2, $d = 50$, $s = 1$	3 000	2 249 760	56 833 860
`pr2`	type 2, $d = 20$, $s = 1$	6 000	3 602 216	91 070 487
`pr2`	type 2, $d = 20$, $s = 1$	8 000	6 400 003	161 635 880
`bikewheel`	—	4 096	8 189	$8\,188^\dagger$
`bikewheel`	—	16 384	32 765	$32\,764^\dagger$
`bikewheel`	—	65 536	131 069	$131\,068^\dagger$
`dblcyc`	—	4 096	8 192	$2\,000^\dagger$
`dblcyc`	—	16 384	32 768	$2\,000^\dagger$
`dblcyc`	—	65 536	131 072	$2\,000^\dagger$
k-cores of real-world graphs				
`uk-2007-05`	$k = 10$	68 141 228	3 134 185 720	1
`uk-2007-05`	$k = 50$	16 759 375	1 770 324 244	1
`uk-2007-05`	$k = 100$	3 979 369	869 154 056	1
`uk-2007-05`	$k = 1000$	223 416	183 373 955	1
`twitter-2010`	$k = 25$	13 083 726	970 195 992	1
`twitter-2010`	$k = 30$	10 469 192	899 618 650	1
`twitter-2010`	$k = 50$	4 417 754	677 688 218	3

Table 1: The evaluation set *(continued)*.

Instance	Parameters	n	m	λ
`twitter-2010`	$k = 60$	3 529 047	629 640 787	3
`uk-2002`	$k = 10$	9 019 566	225 617 173	1
`uk-2002`	$k = 30$	2 553 366	114 612 142	1
`uk-2002`	$k = 50$	783 316	51 882 018	1
`uk-2002`	$k = 100$	98 275	10 689 202	1
`com-orkut`	$k = 16$	2 427 486	112 151 730	14
`com-orkut`	$k = 95$	114 190	17 970 565	89
`com-orkut`	$k = 98$	107 486	17 335 612	76
`com-orkut`	$k = 100$	103 911	16 992 328	70
`hollywood-2011`	$k = 20$	1 319 770	109 132 341	1
`hollywood-2011`	$k = 60$	576 111	86 835 657	6
`hollywood-2011`	$k = 100$	328 631	70 765 201	77
`hollywood-2011`	$k = 200$	138 536	47 103 731	27