The Filter-Kruskal Minimum Spanning Tree Algorithm*

Vitaly Osipov, Peter Sanders, and Johannes Singler†

Abstract

We present Filter-Kruskal – a simple modification of Kruskal's algorithm that avoids sorting edges that are "obviously" not in the MST. For arbitrary graphs with random edge weights, Filter-Kruskal runs in time $\mathcal{O}\big(m + n \log n \log \frac{m}{n}\big)$, i.e. in linear time for not too sparse graphs. Experiments indicate that the algorithm has very good practical performance over the entire range of edge densities. An equally simple parallelization seems to be the currently best practical algorithm on multi-core machines.

1 Introduction

A minimum spanning tree (MST) of a graph $G = (V, E)$ is a minimum total weight subset of E that forms a spanning tree of G. The MST problem has been intensively studied in the past since it is a fundamental network design problem with many applications and because it allows for elegant and multifaceted polynomial time algorithms. In practice (on sequential machines and in internal memory), two simple algorithms dating back at least half a century still perform best in most cases [14, 9, 5].

The Jarník–Prim algorithm [7, 17] grows a tree starting from an arbitrary node. Implemented using efficient priority queues, its running time is $\mathcal{O}(m + n \log n)$. Even with simpler priority queues, it performs well for random edge weights – time $\mathcal{O}\big(m + n \log n \log \frac{m}{n}\big)$ [15].

Kruskal's algorithm [11] grows a forest in time $\mathcal{O}((m + n) \log m)$ by scanning the edges in order of increasing weight and adding those that join two trees in the current forest. In practice, Kruskal outperforms Jarník–Prim for sparse graphs. For denser graphs, Kruskal suffers from the $\mathcal{O}(m \log m)$ time needed for sorting all the edges. Therefore it is a natural idea to avoid sorting heavy edges that cannot contribute to the MST. Moret and Shapiro [14] do this by building a priority queue of edges in linear time. Then Kruskal's algorithm subsequently removes the lightest edge until $n - 1$ tree edges have been found. For random graphs with random edge weights, the MST edges are expected to be among the $\mathcal{O}(n \log n)$ lightest edges. Hence, we get an average execution time of $\mathcal{O}\big(m + n \log^2 n\big)$. Unfortunately,

*Partially supported by DFG grant SA 933/3-1.

†Universität Karlsruhe (TH), Germany, {osipov,sanders,singler}@ira.uka.de.

the stopping idea fails already if the MST contains a single heavy edge. Note that this can even happen for random edge weights: Consider a "lollipop graph" consisting of a random graph and an additional path of length k attached to one of its nodes. The MST needs all the path edges, about half of which will belong to the heavier half of the edges for random edge weights. Navarro and Paredes [16] implement the stopping idea more cache efficiently by integrating Kruskal's algorithm with quicksort (*qKruskal*). Apply Kruskal to small inputs. Otherwise, as in quicksort, partition the edges into a light part and a heavy part. Recurse on the light part. If the MST is not complete yet, recurse on the heavy part.

Our key idea is to make qKruskal more robust by adding a filtering step that, before recursing on the heavy part, removes all heavy edges that are within a component of the current forest. In Section 3 we explain the algorithm Filter-Kruskal in more detail. The analysis shows that for arbitrary graphs with random edge weights, Filter-Kruskal runs in expected time $\mathcal{O}\big(m + n \log n \log \frac{m}{n}\big)$. Note that this is the same performance also achieved by Jarník–Prim using binary heaps [15]. The experiments reported in Section 4 confirm that Filter-Kruskal performs very well for both sparse and dense graphs. Moreover, Filter-Kruskal allows a more coarse-grained and hence more practical parallelization than Jarník–Prim.

More Related Work

MSTs can even be found in linear (expected) time [10, 8]. This algorithm can filter out edges without any sorting using sophisticated data structures that can check whether an edge e is the heaviest edge on the cycle defined by the minimum spanning forest (MSF) of an edge sample and e. However, such algorithms are complicated and large constant factors are involved. To check whether general edge filters are useful at all, [9] invests $\Theta(n \log n)$ preprocessing to allow for a better constant factor in filtering. This algorithm only significantly outperforms Jarník–Prim for rather dense graphs with weights that force m decreaseKey operations.

2 Preliminaries

Let $G = (V, E)$ denote an undirected, weighted graph with $|V| = 1..n$. Let $m = |E|$. Since we need Kruskal's algorithm as a subroutine, we outline it here for self-containedness. Figure 1 gives pseudocode that should be self-explaining. When Kruskal skips an edge $\{u, v\}$ that falls within a single component of T, this is safe because $\{u, v\}$ closes a cycle in T and is at least as heavy as all edges in T. In this situation, the *cycle* property of MSTs tells us that $\{u, v\}$ is not needed for an MST. The most sophisticated aspect of the algorithm is the Union-Find data structure P maintaining a partition of the nodes into components defined by the MST edges T found so far. P supports an operation union joining two partitions and an operation find(v) returning the node number of the *representative* of the partition of the node v. Indeed, the implementation will exploit that partitions are represented using *parent* references defining trees rooted at the representatives and that

```
Procedure kruskal(E, T : Sequence of Edge, P : UnionFind)
    sort E by increasing edge weight
    foreach {u, v} ∈ E do
        if u and v are in different components of P then
            add edge {u, v} to T
            join the partitions of u and v in P

Procedure filterKruskal(E, T : Sequence of Edge, P : UnionFind)
    if m ≤ kruskalThreshold(n, |E|, |T|) then kruskal(E, T, P)
    else
        pick a pivot p ∈ E
        E≤ := ⟨e ∈ E : e ≤ p⟩
        E> := ⟨e ∈ E : e > p⟩
        filterKruskal(E≤, T, P)
        E> := filter(E>, P)
        filterKruskal(E>, T, P)

Function filter(E)
    return ⟨{u, v} ∈ E : u, v are in different components of P⟩
```

Figure 1: Pseudocode for Kruskal and Filter-Kruskal. T is a set of MST edges already known and P is the partition of V induced by T. When used as a standalone method, the procedures are called with empty T and a trivial partition P. The result is output in T.

the paths leading to the roots are very short in an amortized sense (union-by-rank and path compression). In particular, if $m \gg n$, most path lengths will be one.

3 The Algorithm

Figure 1 gives pseudocode for Filter-Kruskal. Similar to [16], the basic approach is to use quicksort for sorting the edges and to move the edge scanning part of Kruskal into the quicksort code. Hence, the algorithm now calls Kruskal on small[1] inputs and it calls itself for the lighter part of the edges. The only new ingredient at this level of abstraction is that before recursing on the heavier edges $E_>$, they are filtered. Filtering removes those edges that fall within the same component of the current node partitioning. Note that these edges are heavier than all edges in T and close a cycle in T. Hence, the cycle property implies that the filtered edges are not needed for an MST. The advantage of filtering is that filtered edges need not be sorted.

[1] As far as asymptotic performance is concerned, any choice of the function kruskalThreshold works as long as $\text{kruskalThreshold}(n, |E|, |T|) = \mathcal{O}(n)$.

3.1 Analysis of Filter-Kruskal With Random Pivot Selection

We will first show that we can essentially restrict the analysis to counting comparisons since this quantity is indicative of the total execution time:

Lemma 1 *Let C denote the number of (edge weight) comparisons performed by Filter-Kruskal. Then Filter-Kruskal performs $\leq n-1$* union *operations, an expected number of $\leq 2m+C$* find *operations, and $\mathcal{O}(m+C)$ work outside* union *and* find *operations.*

Proof: See Appendix A. ∎

Lemma 2 ([18]) *Consider n* union *operations and M* find *operations on a union-find data structure with n elements using path compression and union by rank. Then the total execution time is $\mathcal{O}(M + n \log^* n)$ where $\log^* n$ denotes the iterated logarithm with $\log^* n = 0$ for $n \leq 1$ and $\log^* n = 1 + \log^* \log n$ otherwise.*

We now analyze Filter-Kruskal for *random edge weights* which we define to be edge weights that are all different and whose rank is a random permutation of some original fixed order of edge weights (wlog, we can assume that the edge weights are the set $1..m$).

Theorem 3 *Given an arbitrary graph and random edge weights, the expected running time of Filter-Kruskal is $\mathcal{O}\big(m + n \log(n) \log \frac{m}{n}\big)$.*

Proof: See Appendix A. ∎

We also give an informal argument why the complexity computed above is tight in the sense that using the sampling lemma from [4] we cannot expect a better result. Suppose, we had an algorithm that filters every edge with respect to all lighter edges "for free". Then, $\sum_{n<i\leq m} n/i = \Theta\big(n \log \frac{m}{n}\big)$ edges would survive this filtering (in expectation). Sorting those edges also yields the bound from Theorem 3.

Note that the term $n \log(n) \log \frac{m}{n}$ in the execution time of Filter-Kruskal can be simplified to $\mathcal{O}(n \log(n) \log\log n)$, i.e., for $m = \Omega\left(n \log(n) \log\log(n)\right)$ we get linear execution time. Note that this is up to a factor $\log n / \log\log n$ better than qKruskal for random graphs with random edge weights and recall that our result applies to *arbitrary* graphs with random edge weights.

Indeed, it seems that for random graphs with random edge weights we get an even better bound. Figure 3 in the appendix indicates that the number of edge comparisons executed by Filter-Kruskal for graphs with $n \log n$ edges[2] is proportional to $n \log n$ (at least the double-logarithmic upper bound from Theorem 3 is too pessimistic). This is quite strong evidence that the expected running time of Filter-Kruskal is $\mathcal{O}(m + n \log(n))$: First observe that by Lemmas 1 and 2, the comparisons are representative of the asymptotic

[2]Throughout this paper $\log x$ stands for $\log_2 x$.

execution time. Second, for instances with *less* than $n \log n$ edges, the running time cannot be larger. Finally, random graph theory tells us that the $n \log n$ lightest edges will define the MST with high probability[3]. Hence, all the heavier edges will be filtered out anyway. We believe that a formal proof would not be too difficult by looking even closer at the structure of random graphs. In particular, since the number of nodes outside the giant component shrinks geometrically with the average degree, the probability that an edge survives filtering will also shrink geometrically with its rank divided by n. We believe that the same bound also applies to many other classes of graphs. Indeed we do not know a family of graphs for wich Filter-Kruskal with random edge weights would take more than $\mathcal{O}(m + n \log(n))$ expected time.

3.2 Implementation

For $m \gg n \log(n) \log \log n$, most of the work is done in $\mathcal{O}(m)$ element comparisons performed using quicksort partitioning and the associated find operations in function filter. Therefore, it makes sense to think about the constant factors involved here and to compare them with the constant factors involved in the Jarník–Prim algorithm. The number of comparisons (and associated finds) can be reduced by a constant factor by choosing pivots more carefully. Therefore, for an input segment of size k, our pivot is the median of a random sample of size $\sqrt{k}$. For the find operations, observe that most of the find operations will follow two single parent references to a common representative. This common case can be made fast as follows: When filtering edge (u, v), we first load the parent references pu and pv of u and v respectively. When pu = pv, we can immediately discard (u, v). Otherwise, we complete the find operations for u and v and compare the results as usual.[4] All in all, when most edges are filtered out immediately in this way, the resulting $2m$ random memory references may dominate the running time for large, not too sparse graphs – the quicksort partitioning operations work cache efficiently.

Now let us compare this to the best case of the Jarník–Prim algorithm where for most edges (u, v), we perform one random memory access to the distance value of v, compare it with the edge weight and discard (u, v) without accessing the priority queue. Since all edges are stored in both directions, we also get $2m$ random memory accesses. Hence, we can expect Filter-Kruskal and Jarník–Prim to perform similarly for large, sufficiently dense instances.

3.3 Parallelization

Most parts of Filter-Kruskal are well suited for parallelization – we can parallelize partitioning, sorting, and filtering. It is interesting to note that the find-operations done for filtering are logically completely independent although, due to path compression, there may be simultaneous read and write accesses to the same parent references. However,

[3]The threshold for connectivity is at $n \ln(n)/2$ edges.

[4]We use the same trick within Kruskal's algorithm.

no matter how such operations are executed by the hardware, we will get correct results since we maintain the invariant that parent references eventually lead to the representative. Only the union-find operations in the Kruskal-call for the base case have to be executed sequentially. In the best case, these will only be called for a linear number of edges however.

We have implemented a multi-core parallel version of Filter-Kruskal. Sorting and partitioning uses a parallel implementation of the C++ standard template library [19]. Partitioning is done using the inplace parallel algorithm from [20].

3.4 More Sophisticated Variants

Besides *removing* edges during filtering that are not needed for an MST, we can also *identify* some MST edges for good. Consider the multigraph G_s whose nodes are the components currently represented in the union-find data structure and whose edges are $E_s := \{(\mathsf{find}(u), \mathsf{find}(v)) : (u, v) \in E_>\}$ where $E_>$ contains the edges which survived filtering. For an edge $(u, v) \in E_>$, if $\mathsf{find}(u)$ or $\mathsf{find}(v)$ have degree one in G_s, then (u, v) is an MST edge. More generally, all edges outside the two-core[5] of G_s correspond to MST edges and can be found in time linear in the size of G_s.

We have implemented the first variant of the algorithm (henceforth called Filter-Kruskal+) since degree-one nodes of G_s can be identified easily by maintaining a counter for each representative. Indeed, counter values 0, 1, and "> 1" suffice and can be stored together with the rank information in the unused parent references of component representatives (see also [5]). Figure 4 in the appendix indicates that we get a near linear number of comparisons for sparse random graphs. Unfortunately, we will see that the overhead even for this simple measure is such that we see no improvement with respect to running time. Therefore we refrain from implementing the two-core refinement because this would even require building a proper adjacency-list representation of G_s which would probably be even slower. For the convertion time, refer to the Figure 13.

If we would take the time to build G_s explicitly, it would probably be even better to solve the MST problem recursively for G_s. With this measure we would move into the direction of a variant of the linear time randomized algorithm [8]. The difference would be that by taking advantage of random edge weights, we do not need a complicated data structure for filtering out heaviest edges on a cycle. Instead, we recurse on the lighter half of the edges and use a simple union-find data structure (which for this application could be made to run in linear time). The only missing ingredient to a linear time algorithm would be a node reduction algorithm. We could use the traditional Boruvka algorithm [3], or the sequential node reduction from [5]. The latter algorithm has an interesting deterministic variant where in each step, we remove the node with minimum degree. We have not implemented any of these algorithms because we do not think they could be competetive to our simple Filter-Kruskal algorithm in practice.

[5]The k-core of a graph G is the maximal vertex induced subgraph of G with minimal degree k. All k-cores can be found in linear time by subsequently removing small degree nodes.

4 Experiments

Algorithms. We have experimented with Kruskal, qKruskal – the Kruskal modification from [16], Filter-Kruskal, Filter-Kruskal+ from Section 3.4, and several variants of Jarník–Prim (JP). For JP we only show results for the best implementations: pJP for Irit Katriels implementation with pairing heaps [9] and qJP, our own implementation combined with Paredes quickHeap priority queue [16]. qJP is considerably faster than Paredes own code since we use a faster graph representation (adjacency arrays rather than adjacency lists). We also use a multi-core implementation of Filter-Kruskal (Filter-Kruskal$P$ for P cores) and a version of Kruskal with parallel sorting of edges (Kruskal$P$). Our graph data structure implements the interface of the Boost Graph Library [12], but uses a graph representation that is specific to the algorithm. For the variants of Kruskals algorithm this is simply an array of edges, for the JP algorithm, we use an adjacency array representation. We have also measured the time needed for converting between these representations.
Implementation. The implementation uses C++ with the GNU compiler version 4.3.1 and optimization level O3. The experiments are run on machine with two AMD Opteron 2.0 GHz quad-core CPUs.
Instances. Unfortunately, there is no established suite of real world instances for MST problems. Mostly, synthetic graphs families from the study [14] were used in the past. From these we use random graphs with random edge weights, graphs that force a decreaseKey for every edge (and, incidentally, make filtering completely ineffective), and random geometric graphs where n random points in the unit square are connected with their k closest neighbors wrt. Euclidean distance. Note that the resulting edge weights are not independent random numbers. We also use lollipop graphs with random edge weights where a path of length $n/2$ is appended to a random graph with $n/2$ nodes. Perhaps most interestingly, we have obtained a few instances generated by the image segmentation application by Jan Wassenberg (see also [6]), that was ran on satelite images, [1].

Figure 2 shows the running time of the algorithms discussed above for random graphs with random edge weights and $n = 2^{16}$. Kruskal's algorithm performs well for up to $8n$ edges where it is also well parallelizable. For more dense graphs, JP is better. None of the two priority queue variants is a clear winner. Quickheaps are a bit better for very sparse graphs whereas pairing heaps win for more dense graph. qKruskal does improve on Kruskal and outperforms qJP. Filter-Kruskal shows uniformly good performance over the entire range of densities. It is clearly better than qKruskal and only for rather dense graph it is still outperformed by JP. On 8 cores, Filter-Kruskal becomes the clear winner. Note that a parallel implementation of JP does not look promising except for very large, very dense graphs where parallelizing the innermost loop becomes interesting.

A more direct comparison to the sophisticated parallel MST implementations by Bader and Cong [2] would be interesting. However, they only report speedups for at least one million nodes and our codes are considerably faster than their codes if one simply scales the clock frequency of the machines. Hence, it currently looks like our algorithms are better at least for a small number of cores and in particular for small inputs.

Somewhat disappointingly, the "improved" Algorithm Filter-Kruskal+ is always slightly

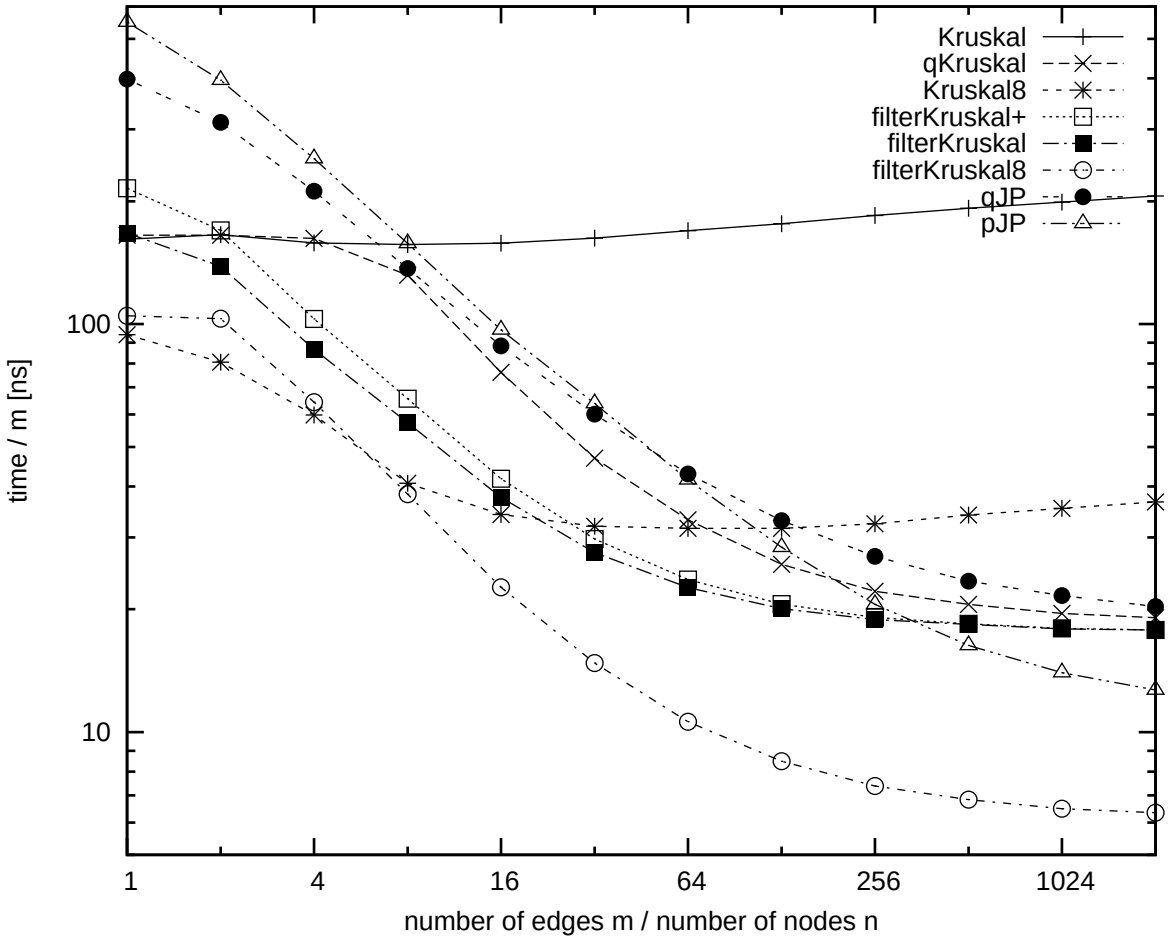

Figure 2: Time per edge for random graphs with random edge weight and 2^{16} nodes.

slower than Filter-Kruskal – even the moderate additional effort for including degree-one edges never really pays off. We view this as an indication that even more complicated algorithms like [8] are even more far from practical than we thought.

We have performed analogous experiments for smaller and larger random graphs with $n = 1024$ and $n = 2^{22}$ (see Figures 5 and 6 in the appendix). The ranking of the algorithms is similar as before, except that for very large graphs, Filter-Kruskal consistently outperforms JP. For small graphs, it is not astonishing that parallelization is not worthwile. Here, pJP is the best algorithm throughout whereas qJP performs worse than both pJP and Filter-Kruskal.

For the difficult instances, we see in Figure 7 in the appendix that qJP becomes extremely slow, pJP and filterKruskal are now somewhat worse than Kruskal, qKruskal yields a slight improvement over Kruskal and parallel Kruskal is the best algorithm.

For random geometric graphs (see Figure 8 in the appendix) with 2^{16} nodes, we again

have similar ranking as for random graphs, except that this time Filter-Kruskal outperforms the JP variants.

For lollipop graphs (see Figures 9–11 in the appendix) we see similar result as for random graphs. The biggest difference is that qKruskal is now no better than Kruskal. JP ourperforms sequential Filter-Kruskal for sufficiently dense graphs but not by a large margin.

Finally, for the image segmentation instances shown in Figure 12, we see few surprises. Filter-Kruskal is again the best algorithm. As for lollipop graphs, qKruskal performs no better than Kruskal which is a confirmation of the intuition that this heuristics is not very robust.

The bottom line is that Kruskal remains a good algorithm for very sparse graphs and Filter-Kruskal and pJP contend for the best performance on more dense instances. We tend to give preference for Filter-Kruskal for three reasons. First, it shows good performance also for sparse graphs. Second, it is easily parallelizable, yielding a speedup of above two already on low cost servers. Third, it only requires a list of edges as its input whereas JP needs a full fledged adjacency array. Figure 13 in the appendix shows that building an adjacency array from a list of edges can take an order of magnitude longer than computing th MST![6] This means that in cases where the adjacency array is not available, Filter-Kruskal will be much faster than JP. In contrast, building an edge list from an adjacency array is very fast, indeed, the times given in Figure 13 are overestimates because we could fuse the loops for conversion and for the top level partitioning – scan the adjacency array and output to a partitioned array of edges.

5 Conclusions

Enhancing Kruskal's algorithm with a simple filtering step leads to considerably improved performance. In particular, for arbitrary graphs with random edge weights, we obtain linear expected execution time for all but rather sparse graphs. It seems that this also applies to edge weights occurring in some real world applications. The resulting Filter-Kruskal algorithm not only outperforms Kruskals algorithm but is also competitive with the Jarník–Prim algorithm even for dense graph. Filter-Kruskal considerably outperforms Jarník–Prim if multiple cores are available or if the adjacency array is not given as part of the input.

The more sophisticated Filter-Kruskal+ algorithm that also includes some edges into the MST without sorting, is interesting because it seems to yield a practical and relatively simple algorithm with average case linear execution time. Its somewhat disappointing practical performance might be offset in the future by more opportunities for parallelization. First experiments indicate that its nonparallelizable component grows sublinearly with n (something like $n^{0.6}$).

[6]We use the standard conversion algorithm that essentially performs a bucket sort on the endpoints of the edges [13, page 169], causing $\approx 10m$ cache faults. For large instances, this could be somewhat accelerated by using multi-pass algorithms but it is unlikely that the general picture gets reversed.

Acknowledgements

We would like to thank Irit Katriel, Rodrigo Paredes, Dominik Schultes and Jan Wassenberg for providing graph generators, instances, and source codes. We also thank for fruitful discussions with Martin Dietzfelbinger, Gonzalo Navarro, and Rodrigo Paredes.

References

[1] Space imaging gallery. `http://www.spaceimagingme.com/content/Gallery/index.asp`.

[2] D.A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (smps). *Parallel and Distributed Computing*, 65(9):994–1006, 2005.

[3] O. Boruvka. O jistém problému minimálním. *Pràce, Moravské Prirodovedecké Spolecnosti*, pages 1–58, 1926.

[4] T. M. Chan. Backwards analysis of the Karger–Klein–Tarjan algorithm for minimum spanning trees. *Information Processing Letters*, 67(6):303–304, 1998.

[5] R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *IFIP TCS*, Toulouse, 2004.

[6] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59(2):167–181, 2004.

[7] V. Jarník. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 6:57–63, 1930. In Czech.

[8] D. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *Journal of the ACM*, 42:321–329, 1995.

[9] I. Katriel, P. Sanders, and J. L. Träff. A practical minimum spanning tree algorithm using the cycle property. In *11th European Symposium on Algorithms (ESA)*, number 2832 in LNCS, pages 679–690. Springer, 2003.

[10] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:263–270, 1997.

[11] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[12] L. Q. Lee, A. Lumsdaine, and J. G. Siek. *The Boost graph library: user guide and reference manual*. Addison-Wesley, 2002.

[13] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures — The Basic Toolbox*. Springer, 2008.

[14] B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 15:99–117, 1994.

[15] K. Noshita. A theorem on the expected complexity of Dijkstra's shortest path algorithm. *Journal of Algorithms*, 6:400–408, 1985.

[16] R. Paredes and G. Navarro. Optimal incremental sorting. In *8th Workshop on Algorithm Engineering & Experiments*, pages 171–182. SIAM, 2006.

[17] R. C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, pages 1389–1401, November 1957.

[18] R. Seidel and M. Sharir. Top-down analysis of path compression. *SIAM Journal of Computing*, 34(3):515–525, 2005.

[19] J. Singler, P. Sanders, and F. Putze. MCSTL: The multi-core standard template library. In *13th International Euro-Par Conference*, volume 4641 of *LNCS*, pages 682–694. Springer, 2007.

[20] P. Tsigas and Y. Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In *PDP*, pages 372–381. IEEE Computer Society, 2003.

A Omitted Proofs (With Repeated Theorems)

Lemma 1 Let C denote the number of (edge weight) comparisons performed by Filter-Kruskal. Then Filter-Kruskal performs $\leq n-1$ union operations, an expected number of $\leq 2m+C$ find operations, and $\mathcal{O}(m+C)$ work outside union and find operations.

Proof: Since Filter-Kruskal performs a union operation whenever it finds an MST edge, the claim on the number of union operations is obvious. Recall that Kruskal performs $2m$ find operation – one for each endpoint of an edge. In the worst case, Filter-Kruskal performs the same number of find operations in calls of Kruskal for subsets of edges. In addition, Filter-Kruskal performs some find operations in calls of filter. More precisely, when a call of Filter-Kruskal picks a pivot of rank r, the corresponding call of filter will perform $|E|$ comparisons and $2(|E|-r)$ find operations. The expected value of r is $|E|/2$ end hence, $\mathbb{E}[2(|E|-r)] = |E|$ – the same as the number of comparisons. All the remaining operations can be charged to comparisons or find operations proving the claim on the execution time. ■

Theorem 3 Given an arbitrary graph and random edge weights, the expected running time of Filter-Kruskal is $\mathcal{O}\big(m + n\log(n)\log\frac{m}{n}\big)$.

Proof: By Lemmas 1 and 2, it suffices to account for the number of edge weight comparisons. Suppose the MSF of the r lightest edges is known. This situation is analogous to the one in the linear time algorithm [8] when we filter against a random sample of r edges. Our approach is to integrate the backward analysis of the linear algorithm from [4] and a textbook analysis of quicksort [13].

Let edge i denote the edge with the i-th smallest weight. Edges i and j are compared at most once. Hence, we can count comparisons by looking at the indicator random variables X_{ij}, $i<j$, where $X_{ij}=1$ if edges i and j are compared and $X_{ij}=0$ otherwise. Hence, the expected number of comparison is

$$\mathrm{E}\left[\sum_{i\leq m}\sum_{i<j\leq m} X_{ij}\right] = \sum_{i\leq m}\sum_{i<j\leq m}\mathrm{E}[X_{ij}] = \sum_{i\leq m}\sum_{i<j\leq m}\mathbb{P}\left[X_{ij}=1\right] \quad .$$

Edges i and j are compared only if one of them is picked as a pivot and neither of them is filtered out. If we ignore the second condition, the probability is $2/(j-i+1)$ (see [13] for more details). This bound already suffices to bound the expected comparisons for the case $i \leq n$:

$$\sum_{i\leq n}\sum_{i<j\leq m}\frac{2}{j-i+1} \leq 2n\sum_{j\leq m}\frac{1}{j} = 2nH_m \leq 2n\left(\ln m+1\right) = \mathcal{O}(n\log n)$$

where $H_m = \sum_{i\leq m} 1/m \leq \ln m + 1$ denotes the harmonic sum. We will need the last estimate several times – $\ln m \leq 2\ln n = \mathcal{O}(\log n)$. For the case $i > n$ we do take the second condition into account but only for the first pivot with rank $r \leq j$ that we encounter. (Pivots initially chosen that have rank $r > j$ will not lead to comparisons of edges i and j

but they pass both edges to the same recursive subproblem.) If $r = i$ or $r = j$ (probability is $2/j$), edges i and j are compared. If $i < r < j$, edges i and j are not compared. If $r \leq n$ (probability n/j), we assume filtering to be ineffective and only multiply the probability $2/(j-i+1)$ that at some point i or j are chosen as a pivot. Finally, if $r > n$ (probability $1/j$ for each such choice of r), Chan [4] has proven that i (or, independently, j) will survive filtering with probability at most n/r. Hence we get comparison probability

$$\sum_{n<r<i} \frac{1}{j} \left(\frac{n}{r}\right)^2 \frac{2}{j-i+1} \leq \frac{2n}{j(j-i+1)}$$

where the latter estimate stems from the relation $\sum_{n<r<i} 1/r^2 \leq \sum_{r>n} 1/r^2 \leq 1/n$. Overall, we get

$$\mathbb{P}\left[X_{ij} = 1\right] \leq \frac{2}{j} + \frac{n}{j} \cdot \frac{2}{j-i+1} + \frac{2n}{j(j-i+1)} = \frac{2}{j} + \frac{4n}{j(j-i+1)} \ .$$

For the $2/j$ term and the case $i > n$ we get the total contribution

$$\sum_{n<i\leq m} \sum_{i<j\leq m} \frac{2}{j} = 2 \sum_{n<i\leq m} H_m - H_i \leq 2 \sum_{1<i\leq m} H_m - H_i = 2(m-1) \ .$$

The latter sum can for example be evaluated with a computer algebra system like Maple or you write the sums into a triangular form such that each of $m-1$ columns sums to 2.

For the term $4n/j(j-i+1)$, we estimate

$$\begin{aligned} 4n \sum_{i=n+1}^{m} \sum_{j=i+1}^{m} \frac{1}{j(j-i+1)} &= 4n \sum_{j=n+2}^{m} \frac{1}{j} \sum_{\Delta=2}^{j-n+1} \frac{1}{\Delta} \leq 4nH_m \sum_{j=n+2}^{m} \frac{1}{j} = H_m(H_m - H_{n+2}) \\ &= \mathcal{O}\Big(\log n \log \frac{m}{n}\Big) \ . \end{aligned}$$

■

B Additional Figures

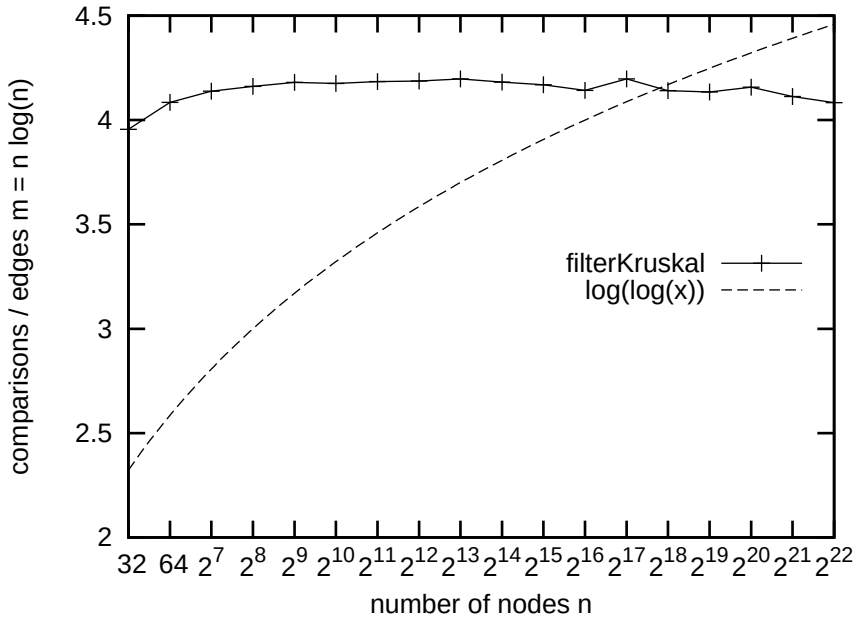

Figure 3: Number of edge comparisons performed by algorithm Filter-Kruskal for random graphs with $n \log n$ edges.

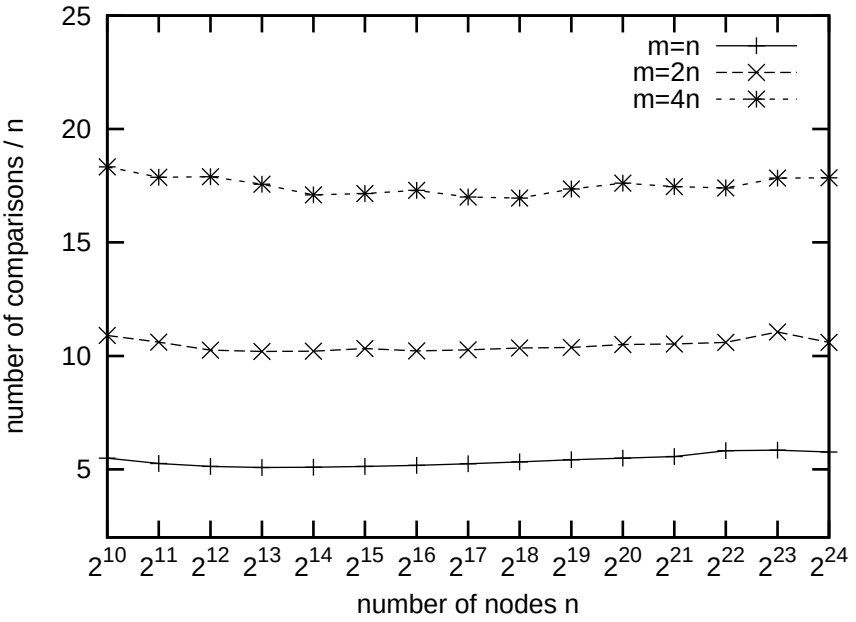

Figure 4: Number of edge comparisons performed by algorithm Filter-Kruskal+ for random graphs.

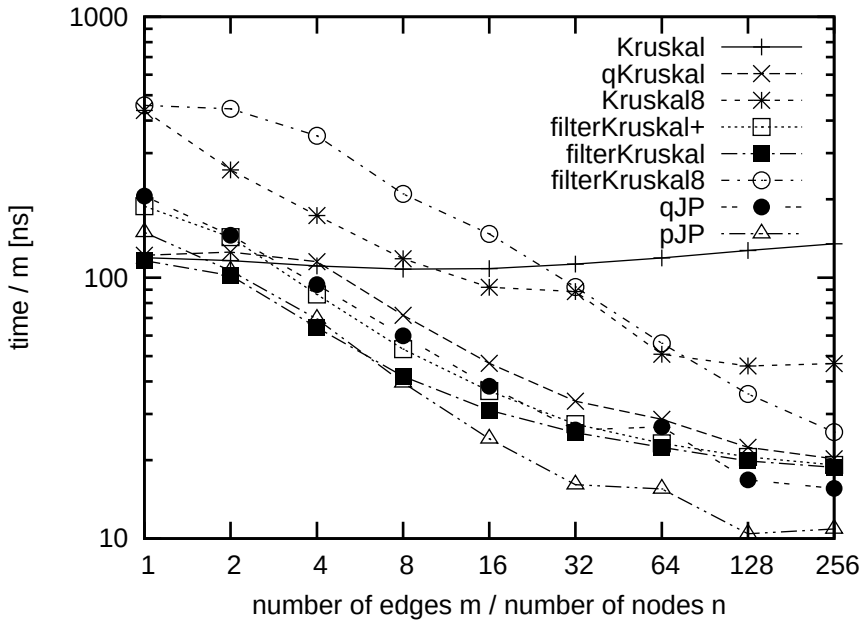

Figure 5: Time per edge for random graphs with random edge weights and 1024 nodes.

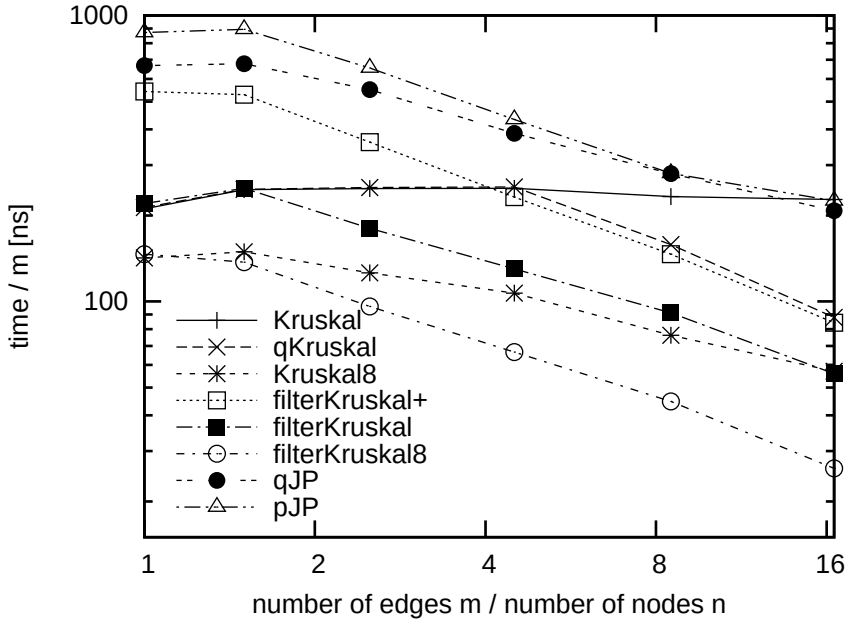

Figure 6: Time per edge for random graphs with random edge weights and 2^{22} nodes.

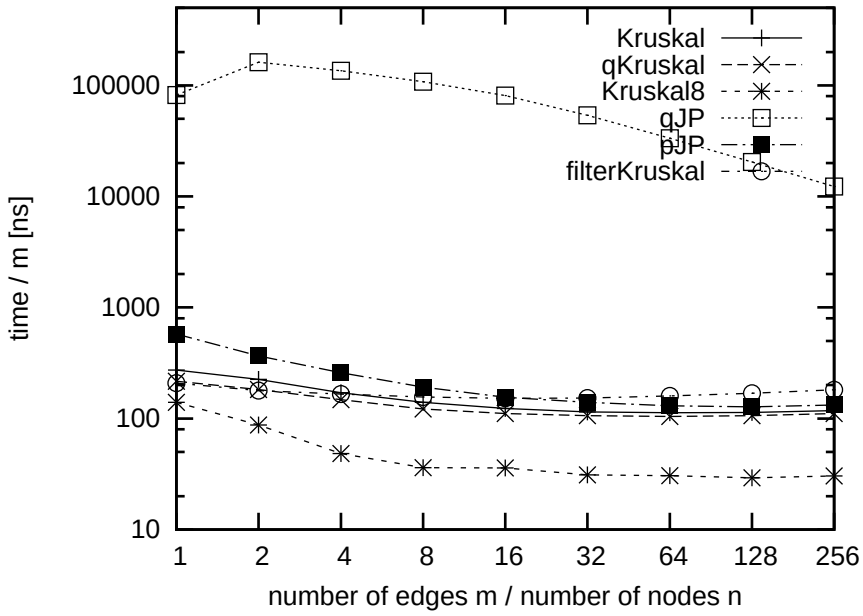

Figure 7: Time per edge for graphs that are bad for JP and Filter-Kruskal with 2^{16} nodes.

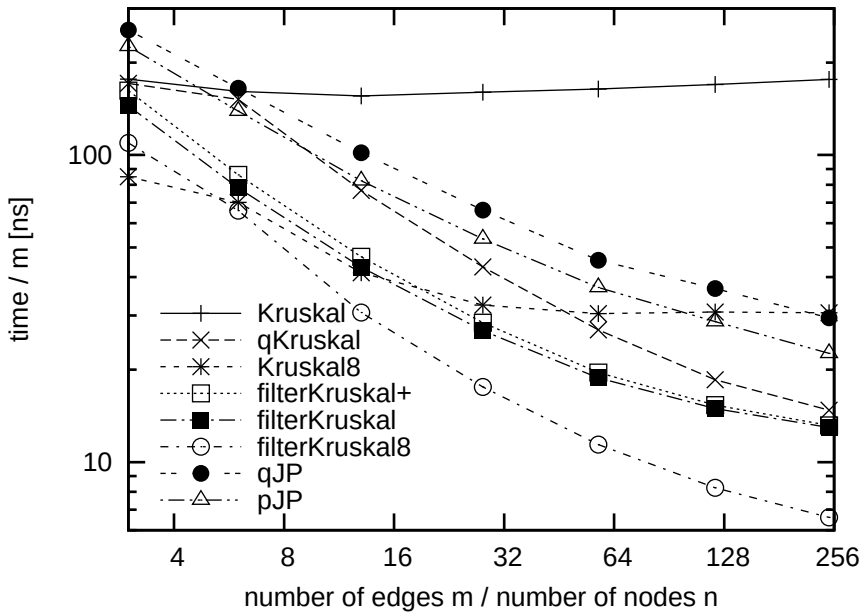

Figure 8: Time per edge for random geometric graphs with 2^{16} nodes.

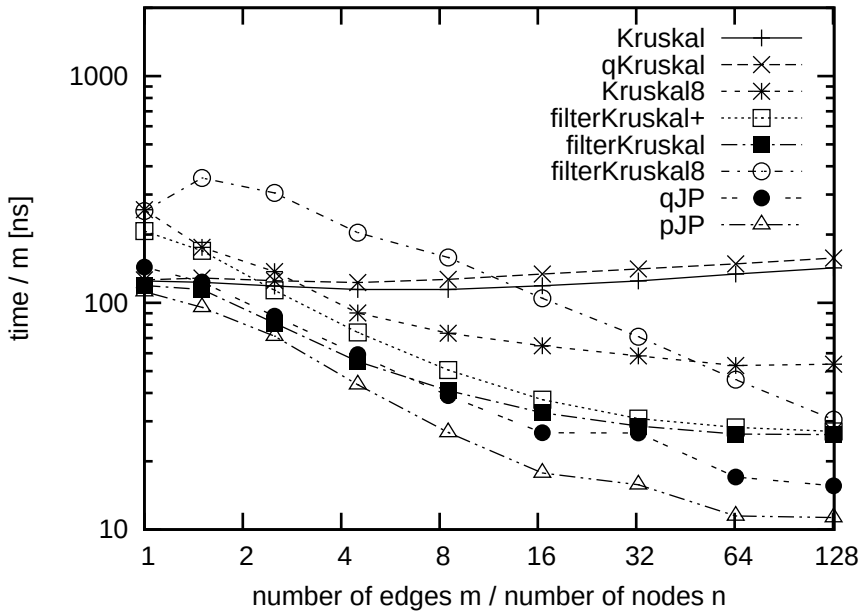

Figure 9: Time per edge for lollipop graphs with random edge weights and 1024 nodes.

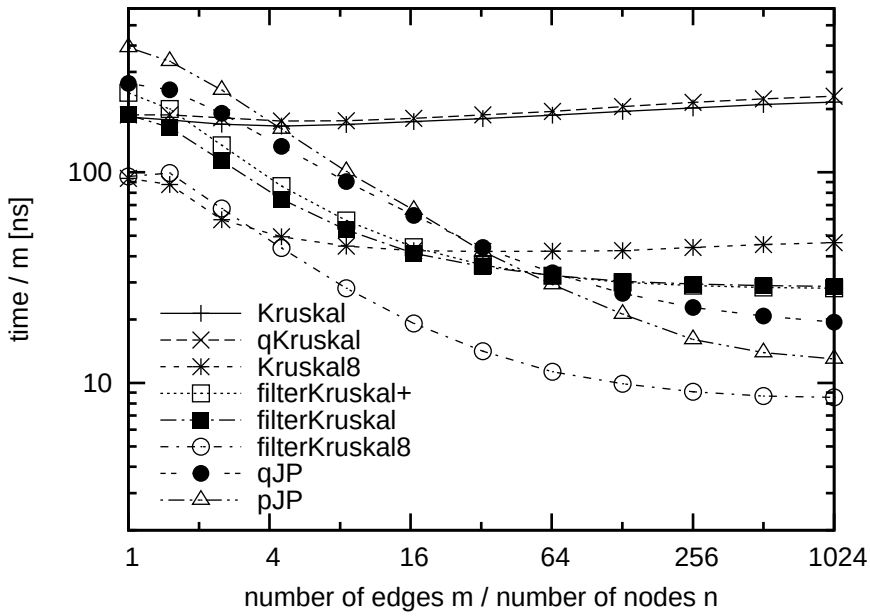

Figure 10: Time per edge for lollipop graphs with random edge weights and 2^{17} nodes.

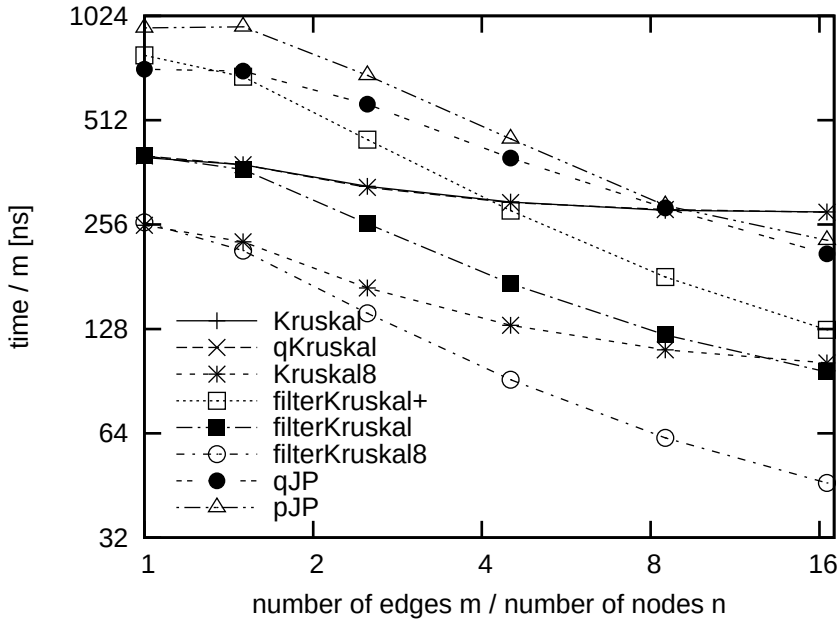

Figure 11: Time per edge for lollipop graphs with random edge weights and 2^{23} nodes.

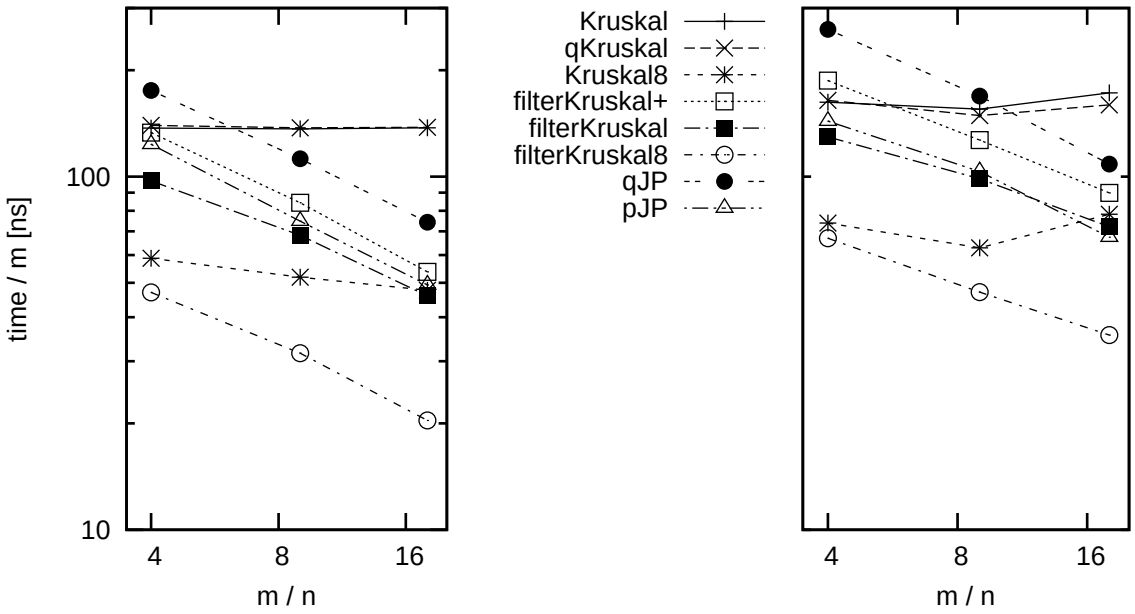

Figure 12: Time per edge for two families of image segmentation problems with 800 000 nodes (left) and 4 163 616 nodes (right) respectively.

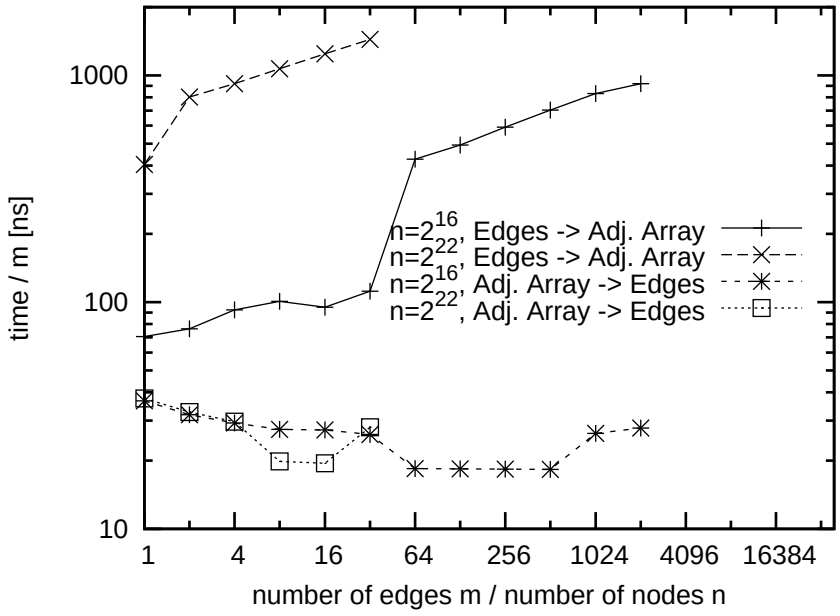

Figure 13: Time per edge for converting between edge sequences and adjacency arrays.