An empirical analysis of algorithms for constructing a minimum spanning tree

Bernard M.E. Moret and Henry D. Shapiro[1]

Department of Computer Science
University of New Mexico
Albuquerque, NM 87131

Abstract

We compare algorithms for the construction of a minimum spanning tree through large-scale experimentation on randomly generated graphs of different structures and different densities. In order to extrapolate with confidence, we use graphs with up to 130,000 nodes (sparse) or 750,000 edges (dense). Algorithms included in our experiments are Prim's algorithm (implemented with a variety of priority queues), Kruskal's algorithm (using presorting or demand sorting), Cheriton and Tarjan's algorithm, and Fredman and Tarjan's algorithm. We also ran a large variety of tests to investigate low-level implementation decisions for the data structures, as well as to enable us to eliminate the effect of compilers and architectures.

Within the range of sizes used, Prim's algorithm, using pairing heaps or sometimes binary heaps, is clearly preferable. While versions of Prim's algorithm using efficient implementations of Fibonacci heaps or rank-relaxed heaps often approach and (on the densest graphs) sometimes exceed the speed of the simpler implementations, the code for binary or pairing heaps is much simpler, so that these two heaps appear to be the implementation of choice.

Some conclusions regarding implementation of priority queues also emerge from our study: in the context of a greedy algorithm, pairing heaps appear faster than other implementations, closely followed by binary, rank-relaxed and Fibonacci heaps, the latter two implemented with sacks, while splay trees finish a decided last.

1 Introduction

Finding spanning trees of minimum weight (often simply called "minimum spanning trees" or MSTs) is one of the best known graph problems; algorithms for this problem have a long history, for which see the article of Graham and Hell [7]. The best comparison-based algorithm to date, due to Gabow *et al.* [6], runs in almost linear time—its running time is $O(|E| \log \beta(|E|, |V|))$, where $\beta(m,n) = \min\{\, i \mid \log^{(i)} n \leq m/n \,\}$; very recently, Fredman and Willard [5] have described a linear-time algorithm that uses address arithmetic and bit manipulations. Classical algorithms have slower asymptotic running times, but also tend to be simpler. Kruskal's algorithm runs in $O(|E| \log |V|)$ time; Prim's algorithm runs in $O(|E| \log |V|)$ time with simple implementations and in $O(|E| + |V| \log |V|)$ with more sophisticated priority queues; Cheriton and Tarjan's algorithm [1] runs in $O(|E| \log \log |V|)$ time. Because both Kruskal's and Prim's algorithms rarely attain their worst-case bounds and because their data structures can be maintained with very low overhead, one cannot determine the best algorithm solely on the basis of asymptotic analysis and so must resort to experimentation.

We studied four comparison-based algorithms in numerous implementations: Prim's algorithm, implemented with each of binary heaps, d-heaps, pairing heaps (for a description of which see [3]), Fibonacci heaps (see [4]), rank-relaxed heaps (see [2]), and splay trees (see [11]); Kruskal's algorithm, implemented with a first sorting pass and with demand sorting; Cheriton and Tarjan's algorithm (see [13]); and Fredman and Tarjan's algorithm (see [4]), implemented with Fibonacci heaps—we chose this last algorithm, which runs in $O(|E| \cdot \beta(|E|, |V|))$ time, rather than the very slightly faster algorithm of Gabow *et al.*, because the additional overhead of the latter seems certain to overwhelm its slight asymptotic gain. (We did not include the linear-time algorithm of Fredman and Willard

[1] This author did his work while a visiting professor at the Institut für Grundlagen der Informationsverarbeitung und Computergestützte neue Medien, Technische Universität Graz, A-8010 Graz, Austria.

in our study, for two reasons. First, it is so recent that little is yet known about implementation alternatives—the implementation described by its authors appears to suffer from significant overhead; and secondly, as it is closely related to the algorithm of Fredman and Tarjan, which we found not to be competitive, we believe that it would not be competitive either.)

Moreover, we experimented with both low- and intermediate-level implementations of the various priority queues used with Prim's algorithm, so that our study provides insights into the behavior of priority queue implementations when used within the setting of a typical greedy algorithm—thereby complementing the studies of Jones [8], who experimented with various implementations of priority queues under random distributions of operations (not including DECREASEKEY), and of Stasko and Vitter [12], who experimented with a variety of pairing heap designs under a variety of distributions of operations (including DECREASEKEY).

In order to avoid biassing our study because of a particular language, compiler, or architecture, we ran our tests on five different machines using two different languages. We also ran tests to evaluate the effect of paging and caching.

2 Methodology

As the actual performance of these implementations depends in various ways on the structure and density of the graphs (for instance, MSTs can be constructed in linear-time with a comparison-based algorithm for, among others, planar graphs), we decided to run experiments with graphs from four different families: (i) randomly generated graphs of specified densities; (ii) graphs of specified densities that represent worst cases for Prim's algorithm; (iii) sparse graphs induced by points placed at random in the Euclidean plane; and (iv) randomly generated trees. For the first two families, we chose densities corresponding to $|E| = |V|^{3/2}$ (dense graphs) and to $|E| = |V| \log_2 |V|$ (a density that represents a breakpoint, at least asymptotically, for Prim's algorithm). For each family and density of graphs, we generated random graphs in a range of sizes (up to 2^{17} nodes on sparse graphs and up to 2^{13} nodes and about 750,000 edges on dense graphs) and gathered running time data on 40 samples of each size.

We generate our random graphs of specified density by proceeding vertex by vertex. Let d be the density, $d = 2|E|/(|V| \cdot (|V| - 1))$; for vertex i, we generate exactly $(|V| - i) \cdot d$ edges connecting it to randomly chosen vertices of higher index and assign edge weights chosen from a uniform distribution. The worst-case graphs for Prim's method are constructed by connecting the first vertex to all other vertices (to foil the dynamic insertion versions), then using the same distribution of edges as for the first family, but running Prim's algorithm in adversary mode to determine connections and edge weights in order to ensure that every potential DECREASEKEY operation actually takes place and that, for binary and d-heaps, each DECREASEKEY operation makes its argument bubble all the way to the root (from the very bottom in the case of binary heaps). To generate graphs in the third family, points are randomly placed in the unit square, and, for each vertex, arcs are drawn to its five nearest neighbors; finally the graph is made undirected and duplicate edges are removed. We experimented with trees in the form of star graphs and line graphs, and with randomly generated trees; the latter are constructed by generating a random degree distribution and building the corresponding tree, then assigning edge weights by running Prim's algorithm in adversary mode in order to force each DECREASEKEY or INSERT operation to move all the way to the root in binary and d-heaps. As the lower density graphs thus generated in the first three families may not be connected, we simply reiterate the generation algorithm until a connected graph is generated.

In order to allow extrapolation and comparison with the asymptotic analysis of each algorithm, we computed the ratio of the running time of each implementation to that of a program that represents the ideal running time—i.e., a program that runs in $O(|E|)$ time with low coefficients (our actual baseline is a program that simply counts the number of edges in the graph). Our timing comparisons exclude the overhead of reading in or generating the data and of writing out the solution.

We ran our programs on five different machines: two CISC architectures (VaxStation 2000 and SUN 3/60) and three RISC architectures (DEC 5000/200, SUN SparcStation II, and Silicon Graphics

Iris 4D/340VGX), all running Unix (or Ultrix). Our programs are all coded extremely carefully and by the same author, so as to minimize any variations across implementations; in order to ascertain the effect of compilers on the running times, our programs were written in both Pascal and C—although we did all of our final testing in C. The code is written to be as efficient as possible, in terms of both time and space. Finally, we studied the effect of low-level implementation decisions on all architectures used in our study before settling on a standard implementation of each data structure: such considerations include the use of parallel arrays rather than records, the effect (in Pascal) of `with` statements on efficiency, the tradeoff between indirection and data moves, and so forth. In our final experiments, we used the Gnu compiler (`gcc`) on all machines except the MIPS-based machines (DEC 5000 and SG Iris), where we used the MIPS compiler (`cc`).

3 The Algorithms

We assume that the reader is already familiar with MST algorithms. Briefly stated, all algorithms operate greedily, choosing the next shortest edge that does not create a cycle and that may have to obey certain algorithmic constraints.

Kruskal's algorithm simply selects the next shortest edge that does not cause a cycle; it requires sorting the edges and a Union-Find data structure to detect cycles. Amortized over the execution of the algorithm, the latter data structure contributes a total cost of $O(|E| \cdot \alpha(|E| + |V|, |V|))$, where $\alpha(m, n)$ is the inverse Ackermann function. In the worst case, sorting takes $\Omega(|E| \log |V|)$ time and dominates the running time; demand sorting, either by using heaps or by using quicksort, has the same asymptotic worst case, but may be faster whenever Kruskal's algorithm produces a spanning tree without having to test each edge.

Prim's algorithm selects the the next shortest edge that touches the current subtree (effectively selecting a vertex to add to the tree) and requires a priority queue of vertices, with operations DELETEMIN and DECREASEKEY—the second operation being more common (especially in dense graphs) than the first. A binary heap or a d-heap both offer straightforward implementations, with logarithmic behavior for both operations; so do other heap structures, such as binomial queues, skew heaps, and leftist heaps. (For an overview of various implementations of priority queues, see [10], Section 3.1.) A splay tree or some other search tree, ordered by keys, also offers logarithmic behavior. The best data structure, in asymptotic terms, is a rank-relaxed or a Fibonacci heap, as both offer logarithmic DELETEMIN and amortized constant-time DECREASEKEY (Fredman *et al.* conjecture that pairing heaps offer the same asymptotic performance). Among data structures that we did not implement, finger search trees (balanced search trees with parent pointers and direct access to the leftmost leaf) offer amortized constant-time DELETEMIN and logarithmic DECREASEKEY, but, as the second operation is more common than the first, they give us a losing tradeoff over rank-relaxed and Fibonacci heaps. Regardless of the choice of priority queue, one is faced with a fundamental decision in the implementation of Prim's algorithm: whether to build the heap by preinserting all vertices with a key value of "infinity" or to build it dynamically, by inserting vertices when they become adjacent to the subtree under construction. The former immediately creates a priority queue of maximal size, but it can also take advantage of the special structure of the initial queue to build it in linear time; the latter spends additional time to ascertain whether an item is already in the heap, but generally keeps the priority queue much smaller. We experimented with both versions, although we used only dynamic insertion with pairing heaps, because preinsertion raises the unresolved issue of the ideal structure of a pairing heap when all keys are the same.

Fredman and Tarjan's algorithm may be viewed as a selective improvement of Prim's algorithm that eliminates the $|V| \log |V|$ expense of DELETEMIN operations by limiting the size of the priority queue, moving to a new subtree whenever the size of the current priority queue reaches a certain threshold; as such an approach eventually produce a spanning forest, each tree in the forest is then contracted (turned into a supervertex) and a new pass is started on the resulting graph. While any efficient heap structure can be adapted to work within this algorithm, we chose Fibonacci heaps without sacks; the algorithm also requires a Union-Find structure to maintain the supervertices.

Boruvka's algorithm, which we did not implement, proceeds in parallel by building a forest and adding to each tree of the forest the shortest edge with one endpoint in that tree; Cheriton and Tarjan's algorithm may be viewed as a selective serialization of that algorithm: it adds an edge to each subtree in the forest, but does so serially, using a queue-based scheduling and merging the pairs of subtrees as it goes. Like Kruskal's algorithm, Cheriton and Tarjan's algorithm selects edges and needs a Union-Find data structure to detect cycles; edges that are found to have both endpoints in the same subtree are not physically deleted (as the physical deletion would be too costly), deferring the actual deletion to further operations. A suitable implementation for meldable priority queues with this lazy deletion is the lazy leftist heap (see Tarjan [13]).

4 Results and Discussion

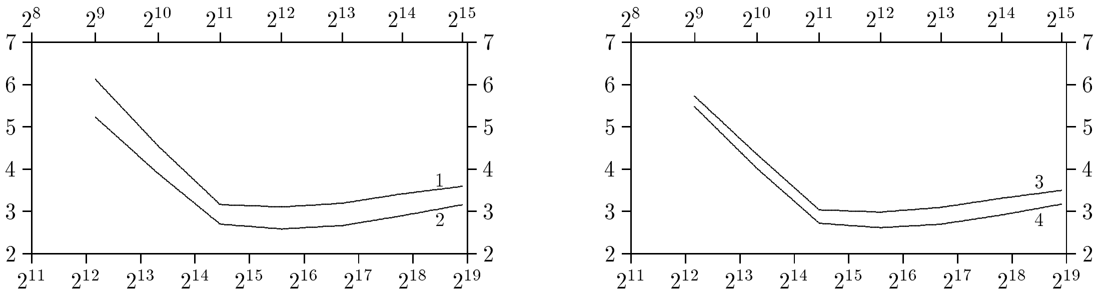
We begin with some observations concerning low-level implementation details and the effect of architectures. On the basis of these observations, we selected a style of implementation for each data structure and then ran tests (using Prim's algorithm) to compare various data structures used for implementing priority queues; we report the results of these comparisons in a second subsection. On the basis of these results, we chose one or two standard implementations for each of the four algorithms and ran tests to compare them; we present the results of these comparisons in the last subsection. (We collected far more data than are summarized here; further details will be found in a forthcoming technical report.)

4.1 Results on low-level implementation details

Apart from obvious differences in speed (in our experiments, the VaxStation 2000 is slowest; the SUN 3/60 runs about five to ten times faster; and the three RISC machines run yet faster, by a factor of five), we noted that the RISC machines were pretty much insensitive to low-level implementation choices, whereas such choices were very important for the CISC machines. In particular, the VAX implementations run faster when actually moving the data, while the SUN 3 implementations run faster when using indirection to avoid data moves (the SUN 3 is so efficient at pointer work that it executes our baseline routine almost as fast as the SparcStation II); in contrast, the RISC machines are essentially insensitive to these differences. Using C rather than Pascal attenuated many of the differences, both between machines (the code generated by the `gcc` compiler on the CISC machines was much faster than that generated by `pc`, whereas the code generated by either `gcc` or `cc` on the RISC machines was comparable to that generated by `pc`) and between choices of priority queues (for instance, rank-relaxed heaps consistently run faster than Fibonacci heaps when coded in Pascal, but not when coded in C, mostly as a result of the pointer arithmetic available in C.)

We also examined the influence of paging and caching. Our main conclusion regarding paging is the obvious one: if the various data structures occupy too much memory, the (wallclock) running times suffer enormously, because none of the algorithms exhibits strong locality of reference. The effectiveness of caching is strongly affected by the method used for storing graphs. Our baseline ran 4 times more slowly on graphs stored in the order in which they were generated than on the same graphs read in from a file where they had been stored in adjacency list order; the effect is far smaller (and more variable) on the MST algorithms, since they cause a lot of nonsequential memory references anyway. We used generation order for storage in our final experiments, since the effect of caching then tends to be cancelled when taking execution ratios.

Since our experimental results show that speed ratios as large as two can arise from seemingly arbitrary implementation choices on the VAX and SUN 3, it is clear that the RISC machines are preferable for the purpose of our experiments; since the Iris machine had the largest memory (65 Mbytes) of our various RISC platforms, we show below the results obtained on it.



(1) preinsertion, no sacks
(2) preinsertion, sacks

(3) dynamic insertion, no sacks
(4) dynamic insertion, sacks

Figure 1: The effect of sacks on Prim's algorithm with Fibonacci heaps, using random graphs of density $|V| \log |V|$

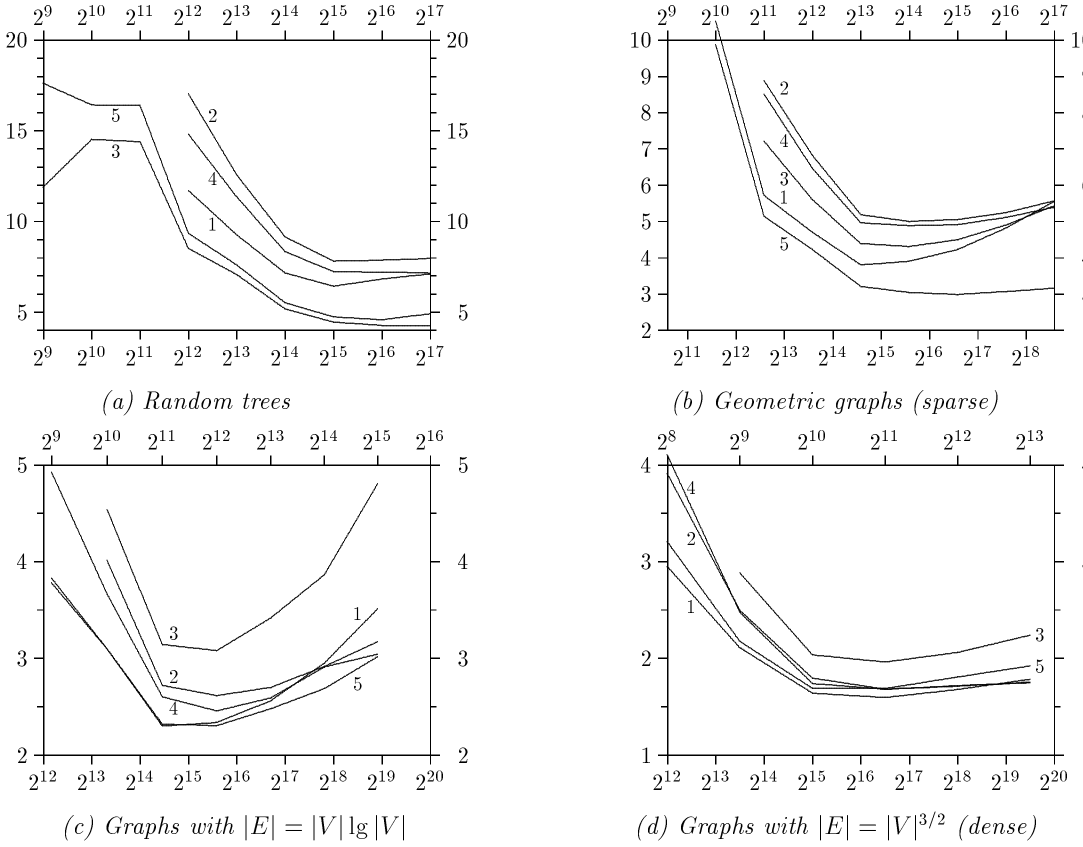
4.2 Results on priority queues implementations

Our findings here are of two types: first, given a specific data structure, what improvements can be made that will make it more efficient? And secondly, how do the best versions of various data structures compare?

Driscoll *et al.* suggested an improvement for rank-relaxed heaps:[2] as the normal pointer implementation suffers from high coefficients, they suggested storing in each node an array of child pointers of approximately $\log_2 |V|$ in length and, in order to keep the storage down to $O(|V|)$, grouping approximately $\log_2 |V|$ vertices together into one heap node—which we call a "sack." Such an implementation should speed up the DECREASEKEY operation on average, since the change in key affects one element of the sack, but need not affect the sack's position within the heap. We realized that such an improvement could also be wrought on any tree-based implementations of priority queues and so experimented with sacks in Fibonacci and binary heaps as well as in rank-relaxed heaps. (Sacks give rise to further questions: should they be kept packed? and should storage within them be reclaimed? We found that packing the children as tightly as possible within each sack is effective for sparse graphs, becoming increasingly less useful with increasing density—but it seems to make little difference in any case.) Sacks proved very important for Fibonacci and rank-relaxed heaps: for all graph families except trees (where, with dynamic insertion, DECREASEKEY operations never occur), the versions using sacks always run faster than the standard implementations. The speed-up is most noticeable when dynamic insertion is not used, but it remains significant in all cases. Figure 1 illustrates the behavior (as ratios to the baseline) of Prim's algorithm using Fibonacci heaps, in linked implementations, with and without sacks, and with static or dynamic insertion. (In this figure, as in all following ones, the size of the graph is shown on the logarithmic horizontal scale, with the number of vertices at the top and the number of edges at the bottom of the frame.) Another effect of sacks is a drastic reduction in storage requirements. In the case of Fibonacci heaps, and within the range of sizes we used. implementations with sacks used only half as much storage as implementations without; they oftne use less storage than conventional binary heaps.

We implemented binary heaps with the "bounce" heuristic of Floyd (see [10], Section 8.6); since this heuristic decreases the number of comparisons and does not significantly alter the number of moves, it routinely improves the behavior of the binary heaps. We tried various choices for storing the keys: in the heap; outside the heap with a pointer reference; both inside and outside. The last version is fastest, but requires additional $|V|$ storage; we chose to store the key outside the heap, a version that is barely slower and saves on storage. Although d-heaps offer better asymptotic performance than binary heaps for Prim's algorithm, their performance was uniformly worse in practice.

[2] They also suggested two strategies for deletion, which they call "reclaiming" and "nonreclaiming." We found that reclaiming versions are slightly better for dense graphs, becoming less useful with decreasing edge density; however, the overall effect is minimal. We used reclaiming versions in final testing.



(a) Random trees

(b) Geometric graphs (sparse)

(c) Graphs with $|E| = |V| \lg |V|$

(d) Graphs with $|E| = |V|^{3/2}$ (dense)

(1) Binary heaps
(2) Fibonacci heaps, packed sacks
(3) Splay trees, parent pointers
(4) Rank-relaxed heaps, sacks and storage reclamation
(5) Pairing heaps, auxiliary two-pass

Figure 2: Experimental results with priority queues for Prim's algorithm

The problem is due to the manner in which d was determined. Johnson [9] chose d so as to balance the work of the DELETEMIN operations ($|V| \cdot d \log_d |V|$) against the DECREASEKEY operations ($|E| \log_d |E|$), a choice that can be made when starting the algorithm. However, these two terms should be balanced based on the *actual* number of DECREASEKEY operations performed, call it m, not on the upper bound of $|E|$; but m cannot be computed ahead of time. Since m is much less than $|E|$ in most of our experiments, binary heaps actually yield a better balance than d-heaps; even for worst-case graphs, where m is maximized, d-heaps remain slower, due to larger overhead.

We implemented splay trees with top-down splaying—which offers a clearly better tradeoff between speed and storage than bottom-up splaying—and ran two versions: one with, and one without, parent pointers. The storage cost of the extra parent pointer is well justified, as the version with parent pointers consistently runs twice as fast as the version without. (In a splay tree, a DECREASEKEY is a DELETE followed by an INSERT and parent pointers allow most DELETE operations to remain local.)

Finally, we implemented four versions of pairing heaps: two-pass and multi-pass, with or without the auxiliary strategy proposed by Stasko and Vitter. We did not use sacks and used only insertion on demand. Unlike Stasko and Vitter, who reported that the auxiliary two-pass method consistently (if not always clearly) outperformed the others, we found it impossible to rank the four strategies. On the trees and the worst-case graphs for Prim's algorithm, the auxiliary methods ran slightly more slowly, while no difference could be discerned on the dense graphs.

Turning now to a comparison of the various priority queues, we conclude that pairing heaps are preferable. In our implementation (without sacks), they use a little more storage than binary heaps (25% more in the context of Prim's algorithm). As to speed, they are consistently the fastest or second fastest implementation, as evidenced by the data of Figure 2 (all methods illustrated in the figure use insertion on demand). Note that, even in the case of random trees, where (due to dynamic insertion) DECREASEKEY operations do not occur, pairing heaps remain very close to the best structure—which, in agreement with Jones' results, is the splay tree. An intriguing result is that pairing heaps do not seem to offer asymptotic performance equal to that of Fibonacci heaps, as the curves for the dense graphs, where Prim's algorithm runs in linear time, show; more testing on this issue is clearly necessary.

4.3 Results on the four algorithms

The remaining issue with Prim's algorithm is the desirability of dynamic insertion. We observed that the main advantage of dynamic insertion is that it keeps the priority queue small; this characteristic may not translate into smaller storage requirements, because our implementations allocate storage in large chunks (e.g., a whole array) in order to gain efficiency, but it may translate into smaller execution times. As a result, dynamic insertion should be more beneficial with sparse graphs, since these graphs generally give rise to small priority queues during execution. However, the tradeoff also depends on the cost of single insertions within the priority queue and on the cost and number of DECREASEKEY operations. We found that dynamic insertion is a clear winner on all sparse graphs (our geometric and tree families), for all priority queues; for the trees, in particular, the running times with preinsertion are often several times larger than those with dynamic insertion. On denser graphs, dynamic insertion remains the better choice for splay trees and rank-relaxed heaps (although the margin of gain decreases with increasing density), while the choice between it and preinsertion becomes indifferent for classical and Fibonacci heaps.

Turning to Kruskal's algorithm, we found, as expected, that sorting on demand is preferable to presorting in all graphs of moderate to high density; the difference is so marked that even the inefficient heaps yield better running times than the highly optimized quicksort presorting pass. In contrast, even the interruptible quicksort is slower than a presorting pass when run on our geometric and tree graphs. Overall, Kruskal's algorithm always runs significantly more slowly than Prim's and also uses considerably more storage (especially on dense graphs, because its storage requirements are proportional to $|E|$, whereas Prim's are proportional to $|V|$). On the other hand, using adjacency lists as our graph representation imposes a "preparation" penalty on Kruskal's algorithm, which must begin by scanning all edges and writing them into some auxiliary storage area; using an array of edges instead of adjacency lists alters the situation, assuming that the array can be overwritten during sorting. In order to get an estimate of the various costs, we recorded running times with and without the preparation time.

Cheriton and Tarjan's algorithm requires the most storage of all our implementations, closely followed by Fredman and Tarjan's algorithm. Note that the latter reduces to a standard Prim's algorithm with Fibonacci heaps whenever the density of the graph is at least $|E| = |V| \log |V|$ and also on line graphs; thus it cannot be competitive on these graphs, since it suffers from significant overhead, in both time and space, when compared to Prim's algorithm. On random trees, Fredman and Tarjan's algorithm used two passes; but it switched to new subtrees so often (on a tree of 2^{17} vertices, it built a total of over 10^5 new subtrees!) that the overhead involved in such switches overwhelmed any advantage inherent in the smaller heaps. Neither Cheriton and Tarjan's nor Fredman and Tarjan's algorithm ever approached the speed of Prim's algorithm on any of our graph families.

Figure 3 shows the comparative running times of the four algorithms on our three main graph families: Prim's algorithm with pairing heaps, Kruskal's algorithm with interrupted quicksort, with and without the time needed to set up the edge array, Cheriton and Tarjan's algorithm, and Fredman and Tarjan's algorithm. Note that Kruskal's algorithm, if given an (unsorted, modifiable) array of edges, actually runs very quickly—often much faster than the best version of Prim's algorithm: on

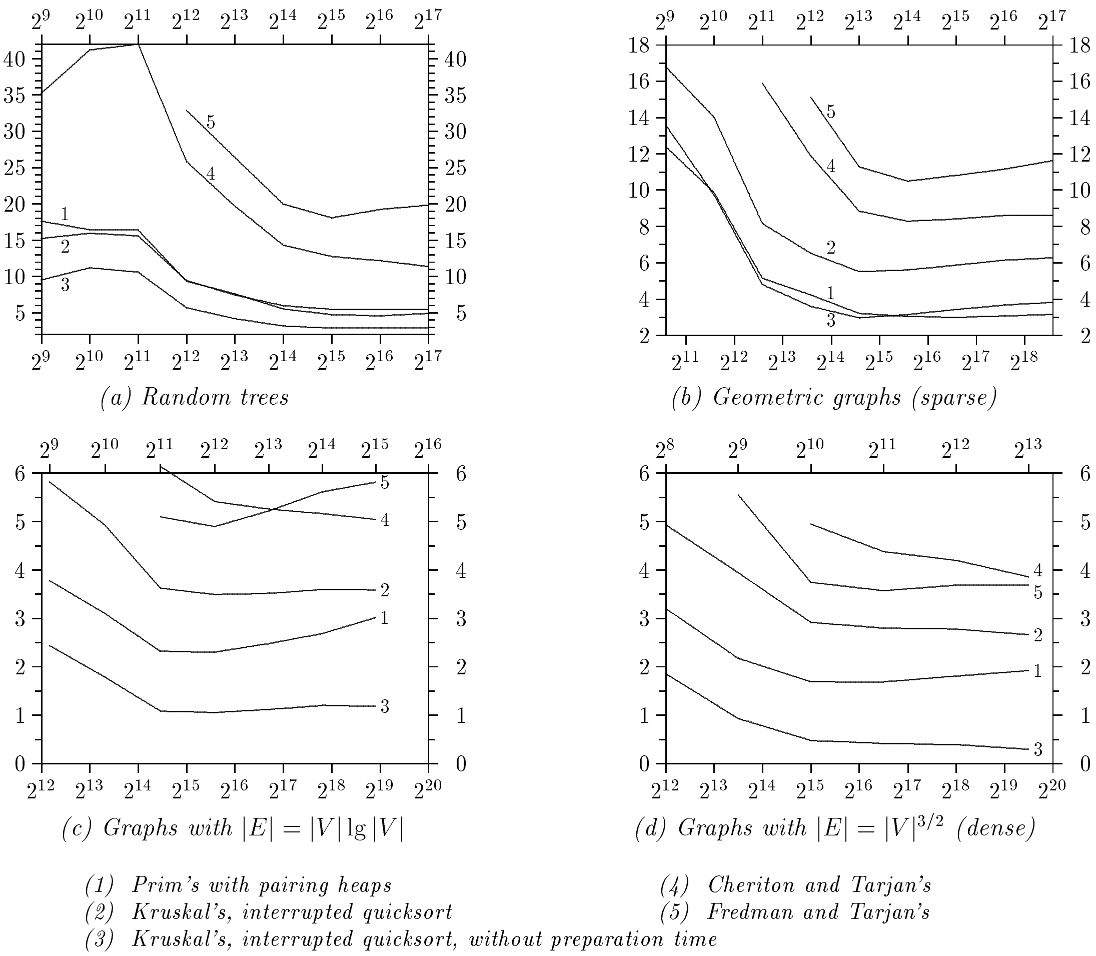
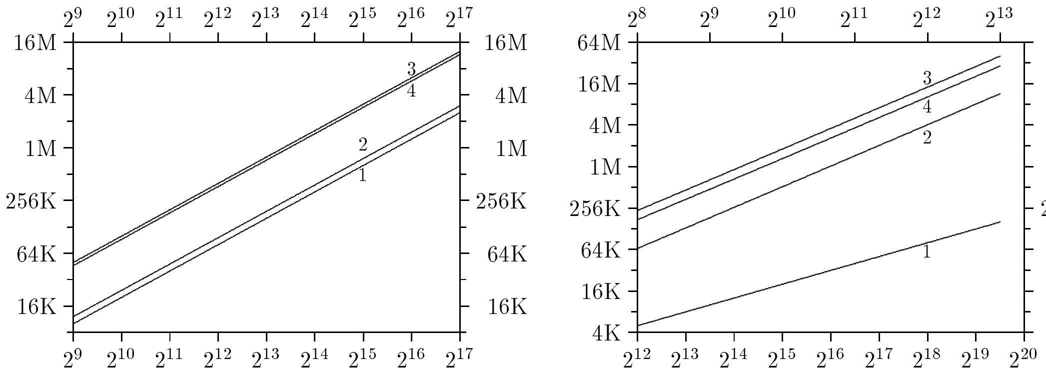


Figure 3: Experimental results with the four comparison-based algorithms

the dense graphs, it need only sort a fraction of the input before the spanning tree is complete, while on the sparse graphs, the very small overhead of quicksort allows the complete sorting to proceed very quickly. However, not only must the input be in this special array form, but the worst-case behavior remains as poor as ever: we ran experiments with worst-case graphs—where we force demand sorting to process all edges by generating graphs made of two distinct components connected by a single, very long bridge—and found that Kruskal's algorithm runs slowly with or without the preparation time. Versions of Kruskal's algorithm based on adjacency lists use much more additional storage, as shown in Figure 4—showing the relative storage used (in addition to the graph itself) by the four methods on trees and on graphs of density $|E| = |V|^{3/2}$.

We have not shown any actual times, but let us simply note that, on the Iris, the source for the figures shown here (but also the slowest of the three RISC machines we used), the baseline for a dense graph with 8,192 vertices and about 750,000 edges takes about 3.5s and Prim's algorithm takes about 7.1s, while the baseline for a geometric graph with about 130,000 vertices and 390,000 edges takes 2.0s and Prim's algorithm takes about 6.0s. Thus very large graphs can be processed very quickly; indeed, the major limiting factor with these efficient algorithms is not time, but space: processing a graph with about 750,000 edges calls for 16 to 50 Mbytes of memory, depending on the algorithm chosen.



(a) Random trees *(b) Graphs with $|E| = |V|^{3/2}$ (dense)*

(1) Prim's with pairing heaps
(2) Kruskal's, interrupted quicksort
(3) Cheriton and Tarjan's
(4) Fredman and Tarjan's

Figure 4: Storage requirements of the four algorithms

5 Conclusion and Future Work

We have presented the results of an exhaustive experimental investigation of comparison-based algorithms for the construction of a minimum spanning tree for a weighted graph. We conclude that the best algorithm for graphs given by their adjacency lists remains the simplest: Prim's algorithm implemented with dynamic insertion and with pairing or binary heaps. These implementations are least demanding of all in terms of storage and ran as fast as, or often much faster than, all other implementations.

Our study also uncovered new facts about the behavior of priority queues: since the operation of DECREASEKEY is of fundamental importance in the context of a greedy algorithm, the implementation that was reported as most efficient when restricted to INSERT and DELETEMIN operations—namely, the splay tree, actually turns out to be the least efficient of the various choices we examined. Pairing heaps appear best, but may not ofer the same asymptotic behavior as Fibonacci or rank-relaxed heaps.

Several issues remain. We want to compare these algorithms with the linear-time algorithm of Fredman and Willard. We also want to examine in more detail the amortized behavior of pairing heaps, the influence of sacks, and the tradeoffs involved in preinsertion. Most importantly, we want to look at the problem of constructing minimum spanning trees for truly enormous graphs (one or more orders of magnitude beyond those considered in this study), presumably in a geometric setting; and we want to experiment with real-world data.

References

[1] Cheriton, D., and R.E. Tarjan, "Finding minimum spanning trees," *SIAM J. Comput.* **5** (1976), pp. 724–742.

[2] Driscoll, J.R., H.N. Gabow, R. Shrairman, and R.E. Tarjan, "Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation," *Comm. ACM* **11** (1988), pp. 1343–1354.

[3] Fredman, M.L., R. Sedgewick, D.D. Sleator, and R.E. Tarjan, "The pairing heap: a new form of self-adjusting heap," *Algorithmica* **1** (1986), pp. 111—129.

[4] Fredman, M.L., and R.E. Tarjan, "Fibonacci heaps and their use in improved network optimization algorithms," Proc. 25th Ann. IEEE Symp. Foundations Comput. Sci. FOCS-84, pp. 338–346; also in final form in *J. ACM* **34** (1987), pp. 596–615.

[5] Fredman, M.L., and D.E. Willard, "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," Proc. 31st Ann. IEEE Symp. Foundations Comput. Sci. FOCS-90, pp. 719–725.

[6] Gabow, H.N., Z. Galil, T.H. Spencer, and R.E. Tarjan, "Efficient algorithms for minimum spanning trees on directed and undirected graphs," *Combinatorica* **6** (1986), pp. 109–122.

[7] Graham, R.L., and O. Hell, "On the history of the minimum spanning tree problem," *Ann. Hist. Comput.* **7** (1985), pp. 43–57.

[8] Jones, D.W. "An empirical comparison of priority-queue and event-set implementations," *Commun. ACM* **29** (1986), pp. 300–311.

[9] Johnson, D.B., "Priority queues with update and finding minimum spanning trees," *Inf. Process. Lett.* **4** (1975), pp. 53–57.

[10] Moret, B.M.E., and H.D. Shapiro. *Algorithms from P to NP. Volume I: Design and Efficiency.* Benjamin-Cummings, Menlo Park, CA, 1991.

[11] Sleator, D.D., and R.E. Tarjan, "Self-adjusting binary trees," Proc. 15th Ann. ACM Symp. Theory Comput. STOC-83, pp. 235–245; also in final form in *J. ACM* **32** (1985), pp. 652–686.

[12] Stasko, J.T., and J.S. Vitter, "Pairing heaps: experiments and analysis," *Commun. ACM* **30** (1987), pp. 234–249.

[13] Tarjan, R.E. *Data Structures and Network Algorithms.* SIAM, Philadelphia, 1983.