How to Find a Minimum Spanning Tree in Practice

Bernard M.E. Moret and Henry D. Shapiro

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

Dr. Shapiro is also adjunct professor at the Institut für Grundlagen der Informationsverarbeitung und Computergestützte neue Medien, Technische Universität Graz, A-8010 Graz, Austria.

Abstract

We address the question of theoretical *vs.* practical behavior of algorithms for the minimum spanning tree problem. We review the factors that influence the actual running time of an algorithm, from choice of language, machine, and compiler, through low-level implementation choices, to purely algorithmic issues. We discuss how to design a careful experimental comparison between various alternatives. Finally, we present some results from an ongoing study in which we are using: multiple languages, compilers, and machines; all the major variants of the comparison-based algorithms; and eight varieties of graphs with sizes of up to 130,000 vertices (in sparse graphs) or 750,000 edges (in dense graphs).

1 Introduction

Finding spanning trees of minimum weight (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 [6]. The best comparison-based algorithm to date, due to Gabow *et al.* [5], runs in almost linear time—its running time is $O(|E| \log \beta(|E|, |V|))$, where $\beta(m, n) = \min\{\, i \mid \log^{(i)} n \leq m/n \,\}$ (in particular, $\beta(|E|, |V|) = O(1)$ for any moderately dense graph, in which case the algorithm runs in optimal linear time); very recently, Fredman and Willard [4] 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; for a review of these, consult [8], Section 5.3. Table 1 summarizes the asymptotic behavior of the principal comparison-based algorithms.

While the asymptotic behavior of the algorithms apparently indicates which algorithm is best, we must remember that the $O(\ )$ time bound given in the analysis is a poor descriptor of the actual efficiency of the algorithm, for two reasons. The first reason is that $O(\ )$ bounds are worst-case upper bounds; even if they are exact, these bounds may only be reachable with very unusual graph structures—so that, in practice, the algorithm runs much faster—or describe a worst case very different from the average case—so that the average performance of the algorithm is much better. The second reason is that the $O(\ )$ bounds are asymptotic and generally given without leading coefficients and lower-order terms; this relative independence from implementation decisions is, of course, what makes asymptotic analysis attractive, but it is also what makes it inadequate when

Table 1: The principal comparison-based MST algorithms

Algorithm	Variety	Running Time	Auxiliary Storage
Kruskal	all	$O(\|E\| \log \|V\|)$	$O(\|E\|)$
Prim	binary heap	$O(\|E\| \log \|V\|)$	$O(\|V\|)$
Prim	d-heap	$O((\|E\| + d\|V\|) \log_d \|V\|)$	$O(\|V\|)$
Prim	Fibonacci heap relaxed heap	$O(\|E\| + \|V\| \log \|V\|)$	$O(\|V\|)$
Cheriton and Tarjan [1]		$O(\|E\| \log\log \|V\|)$	$O(\|E\|)$
Fredman and Tarjan [3]		$O(\|E\| \cdot \beta(\|E\|, \|V\|))$	$O(\|E\|)$
Gabow *et al.*		$O(\|E\| \cdot \log \beta(\|E\|, \|V\|))$	$O(\|E\|)$

considering alternative designs for a problem such as the MST, where all algorithms are fast and thus where even modest leading coefficients play a large role.

While the algorithm of Gabow *et al.* is asymptotically fastest, there are good reasons to expect that simpler algorithms may in fact run faster, even on very large graphs. Both Kruskal's and Prim's algorithms have very pessimistic worst-cases: in practice, we do not expect them to attain such bounds often; moreover, their data structures can be maintained with very low overhead. On the other hand, the algorithm of Gabow *et al.* suffers from a large overhead for a comparatively small asymptotic gain in efficiency. Thus one cannot determine the best algorithm on the basis of asymptotic analysis and must resort to experimentation.

We begin our presentation by reviewing the various factors that affect the running time of an algorithm in practice. We then discuss the methodological problems that arise in experimental work with algorithms and our solutions to some of these problems. We conclude by presenting some of the results that we have observed in a large, on-going experimental study of MST algorithms. We focus here on the first two topics; a companion paper [9] offers a more detailed analysis of our results, while a forthcoming technical report will present all of our data.

2 From Asymptotic Analysis to Actual Running Times

2.1 General considerations

An algorithm is typically described at a very high level, using abstract data types (ADTs) rather than specific data structures. For instance, Prim's algorithm uses a priority queue, with operations DECREASEKEY and DELETEMIN, in which to store vertices; in terms of this ADT, the algorithm is usually described by the inner loop, repeated $|V|$ times,

```
DeleteMin(v);
for each vertex w adjacent to v do
  if length(v,w) < dist(w) then
    DecreaseKey(w,length(v,w));
    dist(w) := length(v,w);
    neighbor(w) := v
  endif
endfor
```

where `dist(w)` is the distance from vertex w to the tree under construction. The initialization phase builds a trivial priority queue (all vertices have the same "infinite" key and the initial vertex is subjected to an initial DECREASEKEY to a key of 0); since such initialization takes linear time, an asymptotic analysis of Prim's algorithm leads to a running time of $O(|E| + a|V| + bD)$, where a is the cost of a DELETEMIN, b the cost of a DECREASEKEY, and D is the actual number of DECREASEKEY operations—in the worst case, of course, $D = \Theta(|E|)$. In a second step, one considers possible implementations of the abstract data type, in terms of the asymptotic running time of its operations. In the case of Prim's algorithm, using a simple binary heap for the priority queue gives $a = b = O(\log|V|)$, so that we get a total asymptotic worst-case running time of $O(|E|\log|V|)$; using a Fibonacci or relaxed heap improves b, which is now $O(1)$ in amortized terms, so that the overall running time becomes $O(|E| + |V|\log|V|)$.

Most algorithmic analysis stops at this point. But much more remains to be considered when developing an efficient implementation. First, there may be variations in the high-level algorithm itself; in the case of Prim's algorithm, for instance, one may build the priority queue dynamically rather than all at once, using a slightly different inner loop and an ADT with the additional operation INSERT:

```
DeleteMin(v);
for each vertex w adjacent to v do
  if length(v,w) < dist(w) then
```

```
    if dist(w) = infinity then
      Insert(w,length(v,w))
    else
      DecreaseKey(w,length(v,w))
    endif;
    dist(w) := length(v,w);
    neighbor(w) := v
  endif
endfor
```

The initialization phase now sets the `dist` field of each vertex to `infinity`, with the exception of the initial vertex, which is inserted into the priority queue. This different implementation has an asymptotic cost of $O(|E| + a|V| + b(D - |V|) + c|V|)$, where a, b, and D are as before and where c is the cost of an insertion. We have $c = O(\log |V|)$ with binary heaps and $c = O(1)$ with Fibonacci heaps, so that overall asymptotic running times are the same for this variant as for the original one. However, dynamic insertions suffers from a higher overhead due to the additional `if` test, yet they may also result in keeping the priority queue smaller, so that a complex tradeoff arises, which is not amenable to formal analysis.

Secondly, implementations of ADTs usually offer a wide range of possibilities, particularly among implementations that provide the same asymptotic performance. In the case of Prim's algorithm, Fibonacci heaps and relaxed heaps, to name only the best known structures, both offer constant-time amortized performance for DECREASEKEY (pairing heaps are conjectured to offer the same performance). While storage costs or coding difficulties may offer some guidance in choosing among competing designs, experimentation remains the best tool. This last statement is made all the more obvious when we consider that each data structure is in turn subject to a multitude of variations, all of which share the same asymptotic running times. For instance, among candidates for priority queue implementations, splay trees can be implemented with top-down or with bottom-up splaying and with or without parent pointers; pairing heaps can be implemented with any variation[1] of the scheduling of the pairing operations (two-pass or multi-pass) and with or without an auxiliary structure for retaining partial trees; and so on. Some variants require more storage than others (generally by some constant factor), but their respective speeds cannot be easily predicted.

Thirdly, many low-level implementation decisions have to be made, several of which can influence the running time in dramatic fashion, depending on the language chosen, the compiler used, and the architecture of the machine on which the software is to be run. A typical example is the use of linked structures *vs.* arrays with index pointers. Since we want to keep interaction with the memory management subsystem to a minimum, the use of arrays with index pointers is preferable in most cases, but it may incur increased overhead on some architectures. Pascal does not allow one to use both approaches within the same data structure, whereas C, because of its flexible pointer handling, does allow one to do so, often with substantial gain. Another example is the handling of the `dist` field in implementing a binary heap for Prim's algorithm: should it be part of the record of each heap item? stored in a separate array and accessed from the heap through indexing? stored both in the heap and outside—at the cost of additional storage, but with possibly improved efficiency? The effect of this implementation decision depends on the specifics of the architecture, because it depends mostly on the relative merits of indirection *vs.* data movement.

Finally, the actual performance of an implementation will depend most on the conditions under which it is used. The overhead of most sophisticated implementations, for instance, makes them unsuitable for very small applications. On larger instances, the overhead may remain a serious obstacle, simply because much of that overhead is present only to guard against worst-case behavior on certain types of instances; if this type of instance never arises, then less sophisticated implementations may well run faster. In the case of MST algorithms, the characteristics of graphs are fundamentally

[1] At least with respect to correctness. Several variations are conjectured to lead to amortized bounds as good as those of Fibonacci heaps, but others are known to lead to worse bounds.

important. For instance, on a family of graphs where $|E| = \Omega(|V|\log|V|)$, Prim's algorithm implemented with Fibonacci or relaxed heaps runs in linear time and so is asymptotically optimal; even on sparser graphs, the more complex algorithms of Cheriton and Tarjan, Fredman and Tarjan, and Gabow *et al.* suffer from higher overhead and offer only a slight asymptotic advantage, so that they may well prove non-competitive. But the structure of the graph is also important. For instance, when run on very sparse graphs, where $|E| = \Theta(|V|)$, Prim's algorithm with dynamic insertion never uses any DECREASEKEY operations if the graphs are trees, but may have to use a linear number of such operations if the graphs are arbitrary sparse graphs; when the graphs are trees, the priority queue remains of unit size if the graph is a line graph, but must grow to size $|V|$ if the graph is a star graph.

2.2 Implementation choices for the MST algorithm

We have already described some choices in the previous section; we content ourselves here with an annotated list of the implementation choices that we considered in our study. At the highest design level, we considered Prim's algorithm, with both static and dynamic insertion as discussed above; Kruskal's algorithm, with both presorting and demand sorting of edges; Cheriton and Tarjan's algorithm; and Fredman and Tarjan's algorithm.

In implementing Prim's algorithm, we used binary heaps, d-heaps (where we used $d = \max(2, |E|/|V|)$, following Johnson [7], so as to balance the worst-case cost of the DELETEMIN and DECREASEKEY operations), splay trees, Fibonacci heaps, rank-relaxed heaps, and pairing heaps. The asymptotic running times (amortized or worst-case) of the main operations on these data structures are summarized in Table 2. (A DECREASEKEY operation in a splay tree is implemented as a DELETE followed by an INSERT of the same item with the smaller key.) In implementing Kruskal's algorithm with presorting, we used quicksort (in the fast version discussed in [8], Section 8.4), while we used both an interruptible quicksort and a binary heap for the sorting on demand. Cheriton and Tarjan's algorithm offers little choice for the data structure, in view of all the requirements placed on it (such as constant-time MELD and nearly constant-time DELETE): we used the lazy leftist heaps proposed in the original paper. Finally, Fredman and Tarjan's algorithm can be implemented with Fibonacci or relaxed heaps, since it uses these structures much like Prim's algorithm, only ensuring that none ever grows beyond a certain threshold; we used Fibonacci heaps in our implementation.

Implementing even as simple a data structure as a binary or d-heap involves a number of important choices. We must consider, as discussed earlier, the maintenance of the current distance of each unincluded vertex to the tree under construction—inside, outside, or both inside and outside the heap. Since sifting down a binary heap requires two comparisons per step (to find the larger child and to decide whether or not to move down), but sifting up requires only one, considerable time can be saved by using Floyd's "bounce" heuristic, which sifts an item all the way down (as if it had an infinite key) and then "bounces" it back up; this approach to DELETEMIN significantly reduces the number of comparisons at the expense of a small number of additional data moves, because we expect an item taken from the bottom of the heap to sink almost to the bottom again. Experimentally (see [8], Section 8.6), this heuristic speeds up DELETEMIN in a binary heap by 20% or more. The

Table 2: Characteristics of data structures for priority queues

Data Structure		INSERT	DELETEMIN	DECREASEKEY
binary heaps		$O(\log n)$	$O(\log n)$	$O(\log n)$
d-heaps		$O(\log_d n)$	$O(d \log_d n)$	$O(\log_d n)$
splay trees		$O(\log n)$	$O(\log n)$	$O(\log n)$
Fibonacci heaps		$O(1)$	$O(\log n)$	$O(1)$
rank-relaxed heaps		$O(1)$	$O(\log n)$	$O(1)$
pairing heaps	proved	$O(\log n)$	$O(\log n)$	$O(\log n)$
	conjectured	$O(1)$	$O(\log n)$	$O(1)$

effectiveness of this heuristic decreases for d-heaps as d increases.

In implementing Kruskal's algorithm, we added a heuristic of our own devising to the heap implementations of dynamic sorting: whenever an edge is removed from the heap through a DELETEMIN, we do not automatically use the last edge in the heap as the new putative root, but first check that this edge remains useful (does not induce a cycle) through the UNION-FIND structure. This heuristic may considerably reduce the size of the heap, so that, whenever the dynamic sorting ends up sorting most of the edges, a substantial advantage may be gained—since, in effect, each additional check (and such checks cost us nearly constant time each) reduces the size of the heap by one.

For more complex data structures, the choices become too numerous for an exhaustive comparison. In the case of pairing heaps, we relied on the experimental results of Stasko and Vitter [10] and implemented four variants: two-pass and multi-pass, each with or without the auxiliary structure discussed in their article. However, we only implemented these versions with dynamic insertion, because static insertion raises the uninvestigated question of the ideal initial structure for a pairing heap within the context of a greedy algorithm. For splay trees, we chose top-down rather than bottom-up splaying, because bottom-up splaying gives us a losing tradeoff: without using significant extra storage, it must suffer clearly larger overhead than top-down splaying, while, with the extra storage, its running time is very close to that of top-down splaying. A different use of extra storage seems well justified, however: since we have direct access to each node in the tree, using parent pointers allows DELETE operations to treat the deleted node as the root of a splay subtree and proceed with the deletion locally (within the subtree). As about half of the nodes are leaves, a rough analysis indicates that this additional field allows the algorithm to run about twice as fast.

Fibonacci and rank-relaxed heaps (we did not implement run-relaxed heaps, which can be seen to suffer from much larger overhead than rank-relaxed heaps) offer the largest number of choices. Driscoll *et al.* [2], in their discussion of relaxed heaps, suggested storing in each node an array of child pointers of approximately $\log_2 |V|$ in length in order to reduce the large overhead associated with relaxed heaps; then, in order to keep the storage down to $O(|V|)$, they suggested 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. We also considered minor variations of rank-relaxed heaps suggested by Driscoll *et al.* as well as minor variations of Fibonacci heaps of our own design.

3 Experimental Design

3.1 General considerations

As our discussion of implementation makes clear, a very careful experimental design is necessary in order to assess the effect of installation- and application-dependent factors on the running time of MST algorithms. While the first few levels of implementation choices (choice of algorithm and of algorithmic variant, if any; choice of data structure and variant, if any) can simply be implemented and compared, the next levels (low-level implementation decisions and choice of environment in which to run the final codes) involve far too many parameters to admit exhaustive investigation. Inasmuch as possible, one should conduct experiments that will allow one to draw conclusions independent of the production environment (machine, language, compiler) in which the programs are to be used. On the other hand, one can only investigate what effect the density or structure of the graphs has on the running time of each algorithm and report on the findings. Since MST algorithms are efficient, it serves little purpose to compare their running times on small graphs; moreover, it is immediately clear that the sophisticated algorithms are not worth running on such inputs. Thus the experiments should cover a range of sizes that extends as far as possible; the limiting factor is simply the size of the available memory, since, for sizes that cause the program to exceed actual memory, paging performance completely dominates the running time and thus invalidates comparisons. The time consumed by memory management is an important factor for large graphs, making it necessary to preallocate storage as much as possible and to request from the operating system only a few very large blocks of memory for internal allocation. (In fact, the time penalty is present in all sizes of graphs, but the running time of a call to the memory management subsystem also typically increases with the size of the memory in use.) Even with the greatest care, the amount of memory used has a direct effect on the running time of the programs, through another mechanism: with very large memory requirements, the cache hit ratio decreases, leading to increased running times.

3.2 Our methodology

We attempted to remove any installation-dependent effects in the following manner. All of our programs were written by the same author, so as to ensure uniformity of style. Our timings (based on the Unix system call `getrusage`) do not include reading (or generation) and writing of data. We ran all of our programs on otherwise quiescent machines—no other users and no background processing. We coded most of our programs in both Pascal and C (some are only coded in C, taking advantage of pointer arithmetic), compiled them with different compilers (only in C, where we used both `cc` and `gcc`), and ran them on a number of different machine architectures, including both CISC and RISC machines—we used a VaxStation 2000, a SUN 3/60, a SUN SparcStation II, a DEC 5000/200, and a Silicon Graphics Iris 4D/340VGX, all running some version of Unix. Furthermore, we ran a linear-time baseline routine on each data set, in order to obtain a reference time; this routine simply counts the total number of edges in the graph by scanning its adjacency lists. The baseline routine serves two purposes: by comparing its rate of growth on various families of graphs across our architectures, we can ascertain the effects of the installation and of caching; and by using the ratio of the running time of our programs to the running time of the baseline, we can immediately visualize the rate of growth for each implementation in a manner that is, to a large extent, machine-independent.

Test data for the programs were entirely the product of random generation—real-world data would have been welcome, but such data are hard to come by, especially in the range of sizes of interest to us. To compensate for the lack of application data, we investigated a total of five different families of graphs: (i) randomly generated dense graphs (the density is a parameter between 0 and 1), where the number of edges is determined by the density, the choice of edges is random, and the length of each edge is a randomly generated value; (ii) a similar family in terms of density, but where the choice of each edge and its length are such as to ensure that Prim's algorithm uses as many DECREASEKEY operations as possible, each as expensive as possible; (iii) geometric graphs, created by placing points at random within the unit square, connecting them to their k (a parameter) closest neighbors with directed edges, then making the graph undirected and removing duplicate edges; (iv) tree graphs, with options for special tree shapes (lines, stars) or randomly generated trees (by generating a random degree sequence and building the corresponding tree), where the length of each edge is a randomly generated value; and (v) the same family, but with the length of each edge chosen as in family (ii). Since graphs in families (i) and (iii) need not be connected,[2] our generation algorithm simply iterates until a connected graph is generated. In order to test Kruskal's algorithm on its worst-case inputs, we allowed graphs in the first family to be made of two separate graphs joined by a very long bridge; on such graphs, Kruskal's algorithm must sort all the edges, since the bridge is examined last and yet must be part of any spanning tree.

We ran all of our programs on 50 to 100 instances of each graph size (from 128 to 8192 vertices for dense graphs and to 131,000 for sparse ones) in each family, noting running time and space usage for each. The data given below were obtained on the Iris machine using C programs compiled with the `cc` compiler; while the Iris is not the fastest of the machines that we used, it simply happened to be that with the most memory (65 Mbytes).

[2] Graphs in family (ii) are connected by virtue of our setting up a connection from the initial vertex to all other vertices, in order to force dynamic insertion to create a priority queue of maximal size immediately.

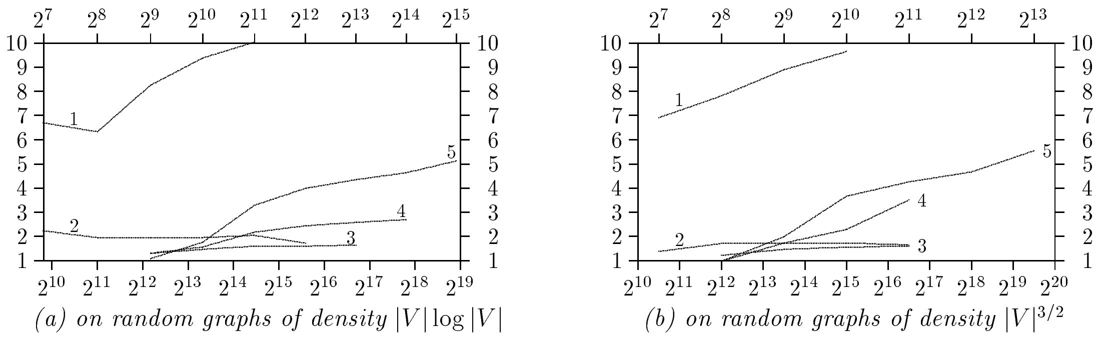
4 Some Results

We only give a sampling of observations regarding different levels of implementation and design. For a more detailed analysis of our experimental results, the reader is referred to our companion paper [9].

Our first finding was that the running time of our baseline routine is somewhat supralinear, as shown in Figure 1, where we plotted the ratio of execution time in seconds to the number of edges of the graph—the ratio being arbitrarily multiplied by 10^6 to yield results between 1 and 10. (In this figure and in all following ones, the top axis records the number of vertices while the bottom axis records the number of edges.) This figure illustrates the reason why we chose to record the performance of each algorithm as the ratio of its execution time to the execution time of the baseline routine: using such a ratio more or less eliminates the effects of increased memory requirements. However, the elimination is not complete, because the mix of instructions in the baseline routine is not the same as in the typical MST algorithm: note that the SUN 3/60, which runs generally five times more slowly than the RISC machines (and indeed ran at about that speed when executing the MST programs), ran as fast as, or even faster than, the RISC machines when executing our baseline routine—both SUN machines are very efficient at dereferencing pointers.

In general, we found that the RISC machines are fairly insensitive to low-level implementation decisions; for instance, while the VAX implementations run faster when actually moving the data and the SUN 3 implementations when using indirection to avoid data moves, the RISC implementations run equally fast in both modes. Using C rather than Pascal generally decreased any observed differences—between machines, between choices of priority queues with the same asymptotic behavior, and between low-level implementation decisions; some of this is clearly due to the compilers (on the CISC machines, the C compiler is much better than the Pascal compiler, whereas the two are much closer on the RISC machines) and some to the fact that the best C implementations typically use pointer arithmetic, which drastically reduces the overhead of, for instance, Fibonacci heaps. As illustrated by the supralinear growth in Figure 1, caching is strongly affected by memory requirements; but it is also strongly affected by the method used for storing graphs. For example, our baseline routine ran four 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—although the effect is much 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—another benefit of our approach to measurements.

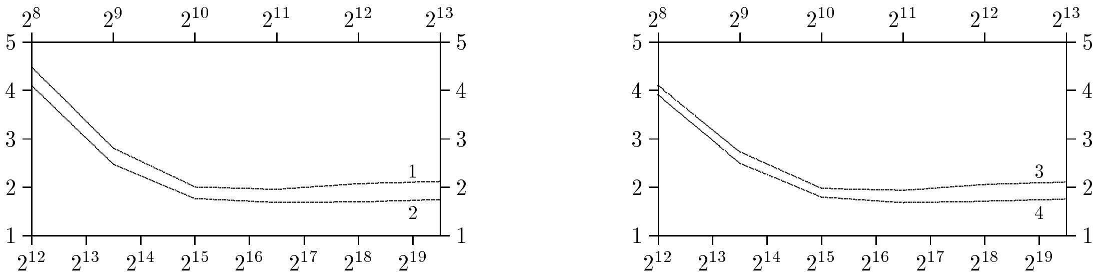
Sacks proved very important for Fibonacci and rank-relaxed heaps: for all graph families except



(a) on random graphs of density $|V| \log |V|$ *(b) on random graphs of density $|V|^{3/2}$*

(1) VaxStation 2000
(2) SUN 3/60
(3) SUN SparcStation II
(4) DEC 5000/200
(5) SGI Iris 4D/340VGX

Figure 1: Running times of the baseline routine on various machines



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

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

Figure 2: The effect of sacks on Prim's algorithm with Fibonacci heaps, using random graphs of density $|V|^{3/2}$

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 2 illustrates the behavior (as ratios to the baseline) of Prim's algorithm using Fibonacci heaps, with and without sacks, and with static or dynamic insertion. Another effect of sacks is a siginficant reduction in storage requirements. In the case of Fibonacci heaps, for instance, and within the range of sizes we used, implementations with sacks used only half as much storage as implementations without, to the point where they used less storage than conventional binary heaps.

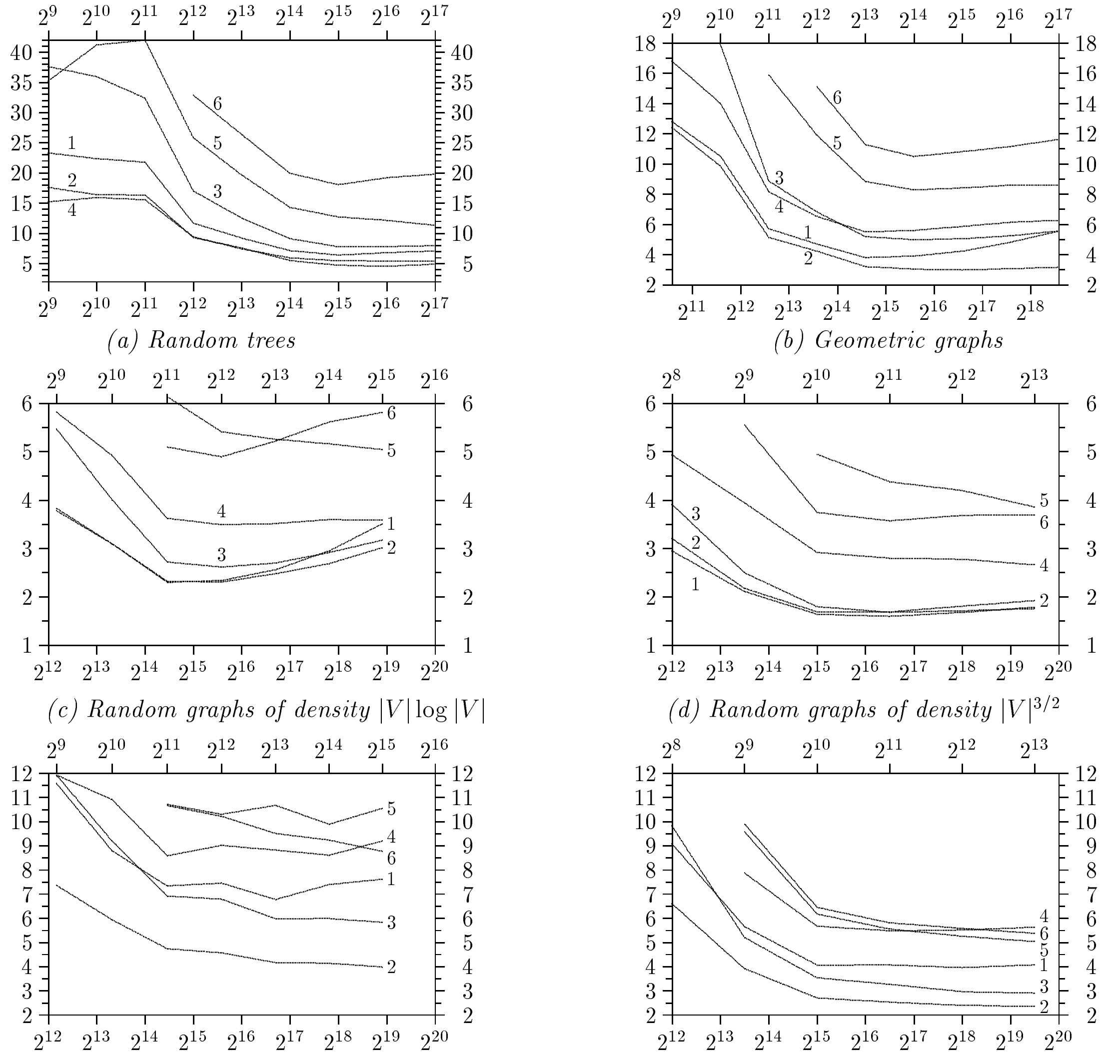
We conclude this short aperçu of our results by summarizing in one figure the behavior of some of the best implementations on our various graph families. Figure 3 shows the running time ratios of the best three implementations of Prim's algorithm (binary heaps, auxiliary two-pass pairing heaps, and Fibonacci heaps with sacks, all with dynamic insertion), the best implementation of Kruskal's algorithm (demand sorting with quicksort), and the two sophisticated algorithms of Cheriton and Tarjan and of Fredman and Tarjan, all on six graph varieties, namely random trees, geometric graphs, random graphs of density $|E| = |V| \log |V|$ and of density $|E| = |V|^{3/2}$, and worst-case graphs for Prim's algorithm of the same two densities.

5 Conclusion

We have argued that many questions in algorithmic design can only be settled through experimental comparisons; while we chose to work with minimum spanning tree algorithms, many other areas suggest themselves, in all of which a large number of efficient algorithms have been proposed, such as shortest path problems, matching problems, and network flow problems. (DIMACS sponsored last year a cooperative experimental effort on the network flow problem.) We have discussed the factors that differentiate asymptotic analysis from the actual behavior of an implementation. We have presented a careful experimental design for the MST problem.

Our results strongly suggest that the algorithm of choice for the MST problem is Prim's algorithm: it is faster than any of the others and also uses less storage than any of the others. The choice of data structure for the heap depends on the needs of the application. For a one-time (code it yourself) use, binary heaps are best (simplest and yielding very little in efficiency); for a library routine, pairing heaps are clearly preferable (the extra coding time is repaid in better running times). (In the absence of a proof of the conjectured time bounds for pairing heaps, a conservative programmer may prefer using Fibonacci heaps with sacks, a choice which sacrifices very little performance while safeguarding the worst-case bounds.)

More work remains to be done on the MST problem. Although the linear-time algorithm of Fredman and Willard does not at present seem competitive, further study may lead to implementations with lower coefficients. Our pairing heap implementations of Prim's algorithm need further study regarding static insertion and the use of sacks. And the issue of extremely large graphs (with millions



(a) Random trees *(b) Geometric graphs*

(c) Random graphs of density $|V|\log|V|$ *(d) Random graphs of density $|V|^{3/2}$*

(e) Worst-case graphs of density $|V|\log|V|$ *(f) Worst-case graphs of density $|V|^{3/2}$*

(1) Prim's, binary heaps
(2) Prim's, pairing heaps
(3) Prim's, Fibonacci heaps
(4) Kruskal's, interrupted quicksort
(5) Cheriton and Tarjan's
(6) Fredman and Tarjan's

Figure 3: Behavior of the main algorithms on a variety of graphs

of vertices and edges) remains completely uninvestigated.

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., 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.

[4] 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.

[5] 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.

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

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

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

[9] Moret, B.M.E., and H.D. Shapiro, "An empirical analysis of algorithms for constructing a minimum spanning tree," Proc. 2nd Ann. Workshop on Data Structs. and Algs. WADS-91, Ottawa (CA), to appear in Lecture Notes in Computer Science.

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