DIMACS Series in Discrete Mathematics
and Theoretical Computer Science

An Empirical Assessment of Algorithms for Constructing a Minimum Spanning Tree

Bernard M.E. Moret and Henry D. Shapiro

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 the results from a study in which we used: multiple languages, compilers, and machines; all the major variants of the comparison-based algorithms; and eight varieties of graphs in five families, with sizes of up to 0.5 million vertices (in sparse graphs) or 1.3 million 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 $O\big(|E| \log \beta(|E|, |V|)\big)$, where $\beta(m, n) = \min\{\, i \mid \log^{(i)} n \leq m/n \,\}$ (in particular, $\beta(|E|, |V|)$ is $O(1)$ 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. The classical algorithms of Prim and Kruskal have slower asymptotic running times, but also tend to be simpler; for a review of these, consult [**12**], 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 algorithms, for two reasons. The first reason is that $O(\,)$ bounds are worst-case bounds; even if they are exact, these bounds may only be reachable with very unusual graph structures, so that the average performance of the algorithms is much better. The second reason is that the $O(\,)$ bounds are asymptotic and generally given without leading coefficients

1991 *Mathematics Subject Classification.* Primary 68A20, 68Q25; Secondary 68E10.

The second author did much of his work on this project while on leave at the Technische Universität Graz, Austria.

This paper is in final form and no version of it will be submitted for publication elsewhere.

©0000 (copyright holder)

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	$O\big((\|E\| + d\|V\|) \log_d \|V\|\big)$	$O(\|V\|)$
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.*		$O(\|E\| \cdot \log \beta(\|E\|, \|V\|))$	$O(\|E\|)$

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

Although the algorithm of Gabow *et al.* is asymptotically fastest, there are good reasons to expect that simpler algorithms may run faster, even on very large graphs. Both Kruskal's and Prim's algorithms have pessimistic worst cases: in practice, we do not expect either one 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 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 the results we observed in a large experimental study of MST algorithms. Some of these results and remarks appeared in earlier conference presentations [**14, 15**].

Since we experimented with both low- and intermediate-level implementations of the various priority queues used with Prim's algorithm, our study also 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 [**9**], who experimented with various implementations of priority queues under random distributions of operations (not including DECREASEKEY), and of Stasko and Vitter [**17**], who experimented with a variety of pairing heap designs under a variety of distributions of operations (including DECREASEKEY).

In the following, we carefully separate our prior analysis of the problem (Section 2), our experimental design (Section 3), and our results and their analysis (Section 4). In any experimental endeavor, it is critical that the data-gathering procedure be as well thought-out as possible beforehand and that it not be modified as observations proceed. The organization of Sections 2 through 4 reflects our methodology: once we settled on an experimental design (after much beforehand analysis and some preliminary testing), we ran all of our experiments with no modification whatsoever, gathered all of the data without doing any piecemeal analysis, and only set about analyzing and interpreting the results once the experiments were complete.

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);
dist(v) := -dist(v)  (* flag v as visited *)
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 is $\Theta(|E|)$. 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 operation Insert:

```
DeleteMin(v);
dist(v) := -dist(v)  (* flag v as visited *)
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 $O\big(|E|+a|V|+b(D-|V|)+c|V|\big)$, 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. Dynamic insertions suffer from a higher overhead due to the additional `if` test, but, since they may also keep the priority queue smaller, a complex tradeoff arises, which is not amenable to formal analysis.

Secondly, implementations of ADTs usually offer a wide range of possibilities, particularly among data structures that provide the same asymptotic performance. While storage costs or coding difficulties 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 interaction with the memory management subsystem: because calls to this subsystem have an unpredictable, but generally high cost, minimizing calls by using very large arrays (allocated in large chunks through a few calls) is desirable. Such practice leads to the use of arrays with indices over linked structures, although both can be used within the same array when programming in C. A more specific example is the handling of the `dist` field in implementing a binary heap for Prim's algorithm: since its value is identical to the key value stored in the priority queue, 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 important. For instance, on a family of graphs where $|E|$

[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.

TABLE 2. Characteristics of data structures for priority queues

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

is $\Omega(|V| \log |V|)$, 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 noncompetitive. Even for a given density, the fine-grained structure of the graph is important. For instance, when run on very sparse graphs, where $|E|$ is $\Theta(|V|)$, 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. At the highest design level, we considered Prim's algorithm, with both preinsertion 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 normally used $d = \max(2, |E|/|V|)$, following Johnson [**7**], so as to balance the worst-case cost of the DELETEMIN and DECREASEKEY operations, but also experimented with a wide range of choices for d in order to determine which value better reflects average-case behavior), 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 [**12**], 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 [**12**], Section 8.6), this heuristic speeds up DELETEMIN in a binary heap by 20% or more. The 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 only cost us roughly constant amortized 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 [**17**] 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 preinsertion 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 clearly must suffer larger overhead than top-down splaying, while, with the extra storage, its running time must remain 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 should allow the algorithm to run about twice as fast.

Fibonacci and rank-relaxed heaps (we did not implement run-relaxed heaps, which 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 implementation of priority queues and so experimented with sacks in binary,

pairing, and Fibonacci 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

Experimental work in the area of combinatorial algorithms has been largely confined to heuristics for NP-hard problems such as the Travelling Salesperson Problem (see [**10**]) or the Minimum Test Set problem [**13**]; or to exact algorithms for complex tractable problems such as the the Planar Graph Coloring problem (see [**16**]). In the case of NP-hard optimization problems, moreover, much of the experimental work revolves around the quality of the approximation, not the running time of the algorithm. Work designed to compare the relative running time of competing algorithms or data structures for a well-solved problem is drawing increasing attention, however, with the priority queue study of Jones [**9**], the study of pairing heaps of Stasko and Vitter [**17**], a small-scale study of three algorithms that use priority queues [**11**], and the DIMACS-sponsored Internet challenge on network flow and matching algorithms [**8**]. We begin with a discussion, within the MST framework, of problems common to any experimental study of algorithms.

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 algorithms. While the first few levels of implementation choices (choice of algorithm and of algorithmic variant; choice of data structure and variant) 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. Nor would such an investigation be really profitable: many of the variants are significant only in certain environments or under certain circumstances. Inasmuch as possible, therefore, one should design 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 cannot design experiments that would allow one to draw conclusions independent of the density or structure of the graphs; here one must 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. Memory considerations are indeed paramount in the experimental design, due to three distinct problems. First, if experiments are run that cause the program to exceed actual memory, paging costs will completely dominate the running time; thus entirely different algorithms (algorithms with the best possible locality of reference) would be favored in such an environment. Secondly, 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 (both for the data cache and, more importantly, for the translation look-aside buffer), leading to increased running times. In planning the experiments, this effect must

be taken into account. Finally, as noted earlier, the time consumed by the memory management subsystem is an important factor, making it necessary to preallocate storage as much as possible through a few very large requests.

3.2. Our methodology. We tried 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 the C versions with different compilers (`cc` and `gcc`), and ran all of our versions 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.

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. We then took the ratio of the running time of our programs to the running time of the baseline. These ratios are largely machine-independent, since the baseline routine closely resembles our MST routines (in its pattern of addressing and in its mix of instructions), so that the effects of caching, CPU speed, and preference for data movement over pointer dereferencing more or less cancel out. Moreover, these ratios should remain constant for an optimal (linear-time) algorithm; any sustained growth in the ratio denotes a supra-linear behavior. Hence the use of ratios to a baseline also enables us to judge very quickly the behavior of each algorithm.

3.3. Graph generation. 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 causing its target (chosen as a leaf in the binary heap) to become (temporarily) the smallest key in the priority queue; (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

[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.

allowed graphs in family (i) 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 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.

3.4. The experiments. We ran all of our programs on 50 to 100 instances of each graph size (from 128 to 12,288 vertices for dense graphs and to 512,000 for sparse ones) in each family, noting running time and space usage for each, along with other useful data, such as the number of DECREASEKEY operations performed, etc. (All bookkeeping for non-timing data is done in separate, non-timing, runs, so that the measured performance is not affected by the gathering of other data.) The data given below in Figures 2 *et seq.* were obtained on the Sparc II machine using C programs compiled with the `gcc` compiler, but other machines gave very similar results, obscured somewhat for small sizes by architectural differences.

4. Results

4.1. Low-level considerations. Our first finding was that the running time of our linear-time (as per analysis) baseline routine indeed runs in supralinear time, 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 scaled by 10^6 so that the vertical scale can be read as microseconds per edge. (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 confirms observations we had made in an earlier experimental study focussed on internal sorting algorithms (reported in [**12**], Chapter 8): whereas analysis predicts that radix sort should run in linear time for a fixed key size, we found that it exhibited supralinear behavior. 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 architecture and increased memory requirements. Of course, the elimination cannot be complete, because the mix of instructions in the baseline routine is not exactly the same as in the typical MST algorithm.

As illustrated by the supralinear growth in Figure 1, caching performance, especially the performance of the address translation cache (the translation lookaside buffer, or TLB), is strongly affected by memory requirements. Data caching, in turn, is strongly affected by the method used for storing graphs (not shown in the figure). 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 in our final experiments.

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

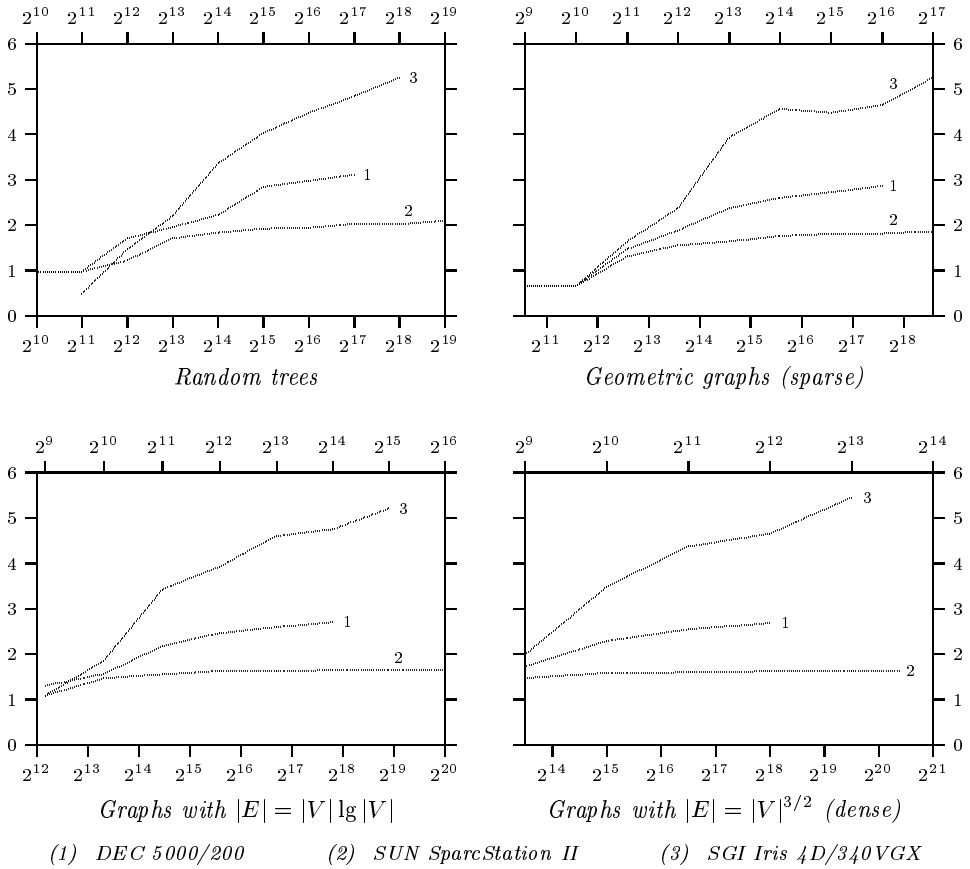

FIGURE 1. Running times of the baseline routine on various machines

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 significantly reduces the overhead of, for instance, Fibonacci heaps.

4.2. Data structure choices. 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 2 illustrates the behavior (as ratios to the baseline) of Prim's algorithm using Fibonacci heaps, with and without sacks, and with preinsertion and dynamic insertion. Another effect of sacks is a significant 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.

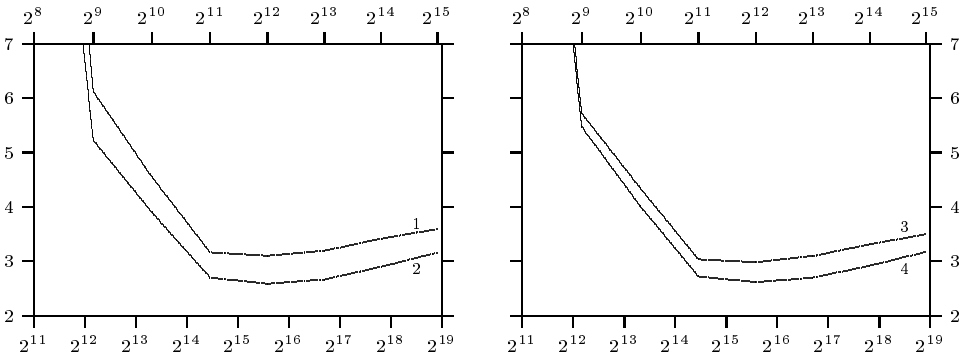

FIGURE 2. The effect of sacks on Prim's algorithm with Fibonacci heaps

4.3. Results with priority queues. As mentioned earlier, we implemented Prim's algorithm with a wide variety of priority queues: binary heaps, pairing heaps, Fibonacci heaps, rank-relaxed heaps, and splay trees, each with a number of variations. In all, we ran 42 different versions, after eliminating a larger number of variations based on preliminary testing. All of our results point to the conclusion that pairing heaps (without sacks) are the best choice for implementing Prim's algorithm. In our implementation, they use only 25% more storage than binary heaps and are consistently the fastest or second fastest implementation across all graph families. Figure 3 illustrates the behavior of the best implementations (all using dynamic insertion) of each of the five data structures.

Comparing our results with those of Stasko and Vitter or of Jones is difficult, because they both used random sequences of operations in testing their implementations of priority queues, whereas our sequences of operations are determined by the MST algorithms. This distinction may account for the fact that, in contrast to the results of Stasko and Vitter, we did not observe any significant difference between the four variants of pairing heaps that we implemented (with and without auxiliary data structure; two-pass *vs.* multipass). Jones' results are yet more difficult to compare to ours, because his random sequences of operations did not include any DECREASEKEY operations; only in the case of random trees, where (due to dynamic insertion) DECREASEKEY operations do not occur, can we compare his results and ours—and they are then in close agreement, both pointing to splay trees as the best structure. (Note, however, that pairing heaps remain very close to the best structure even in this case.)

Prim's algorithm, when run with pairing heaps, Fibonacci heaps, or rank-relaxed heaps should exhibit linear running time on graphs with $\Omega(|V| \log |V|)$ edges, whereas, when run with binary heaps or splay trees, it should retain a supralinear growth by a factor of $\log |V|$, a growth that should be apparent for all five methods on sparse graphs. Actually, a more precise, if still fairly rough, analysis shows that the behavior of binary heaps and splay trees on the densest random graphs should appear linear. Briefly stated, the reason for this behavior is that the number of DECREASEKEY operations does not increase linearly with the number

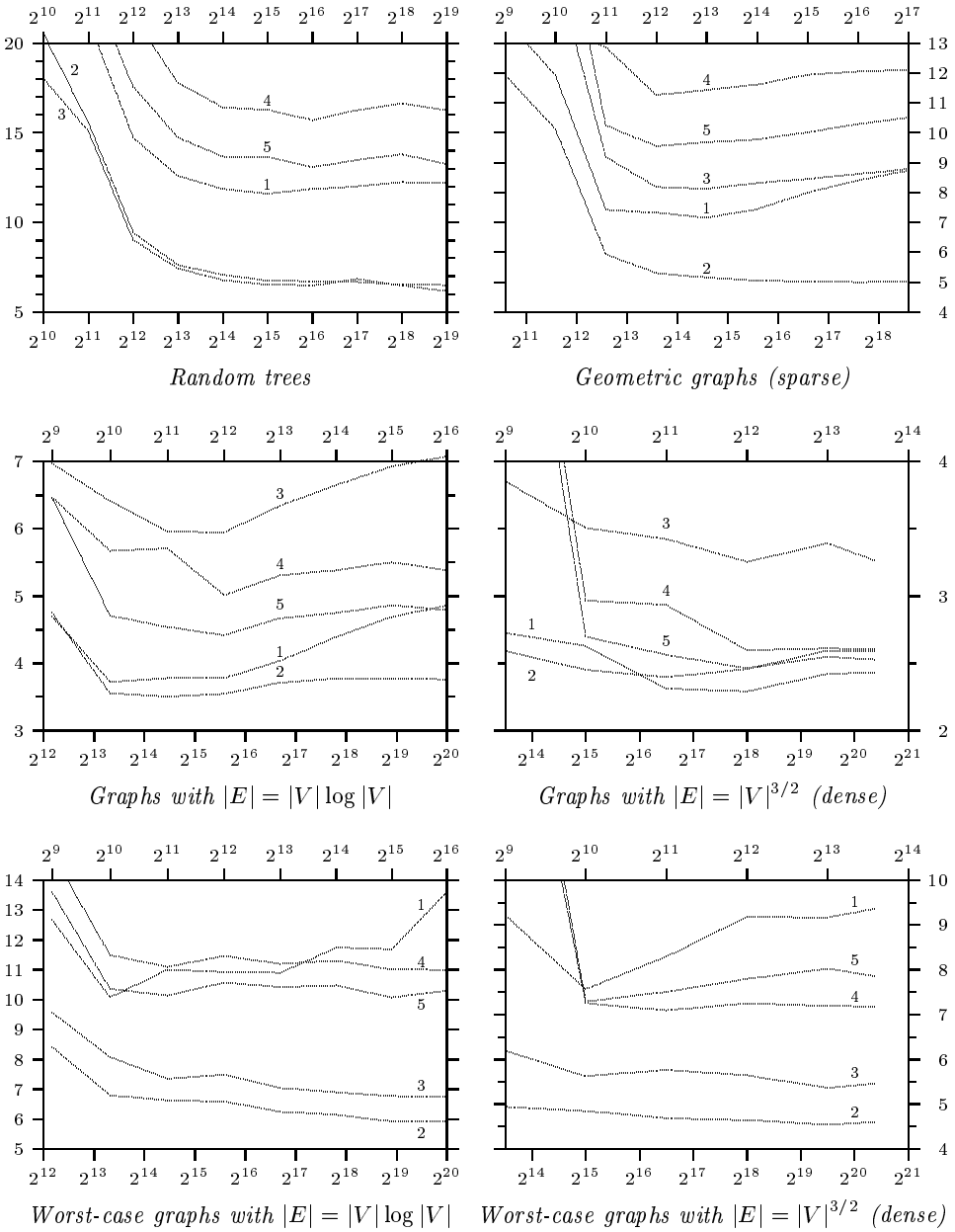

FIGURE 3. Results with priority queues for Prim's algorithm

of edges in a random graph. As the algorithm examines the ith edge to an as yet "unexplored" vertex, this edge has probability $1/i$ of causing a DecreaseKey for that vertex; therefore, as it gets included in the tree, a vertex will cause roughly H_j DecreaseKey operations, where H_j is the jth harmonic number and j itself depends on when the vertex is included in the tree. Thus, in a dense graph (where $|E| = |V|^{1+\epsilon}$), we would expect $O(\log |V|)$ DecreaseKey operations per vertex, each taking $O(\log |V|)$ time, for a total contribution of $O(|V| \log^2 |V|)$ time to the running time of the algorithm; but this term is dominated by the $O(|E|)$ term present in any version of Prim's algorithm and thus the algorithm appears to run in linear time. In a graph of critical density (where $|E| = |V| \log |V|$), we would expect $O(\log \log |V|)$ DecreaseKey operations per vertex, each taking $O(\log |V|)$ time, for a total contribution of $O(|V| \log |V| \log \log |V|)$ time to the running time of the algorithm; since the other terms contributing to the running time are all linear in $|E|$, the cost of DecreaseKey operations eventually dominates the total cost.

The results on geometric graphs and on graphs with $|E| = |V| \log |V|$ or $|E| = |V|^{3/2}$ follow these expectations exactly. On dense graphs with $|E| = |V|^{3/2}$, splay trees and binary heaps indeed appear to run in linear time; the fraction of edges that cause a DecreaseKey operation is small and decreases rapidly, from 17% for the smallest graphs down to 5% for the largest; on graphs with $|E| = |V| \log |V|$, the fraction of edges that cause a DecreaseKey operation decreases more slowly and remains significantly larger—from 35% for the smallest graphs down to 23% for the largest.

DecreaseKey operations do not occur when Prim's algorithm with dynamic insertion is run on random trees; nevertheless, we would expect supralinear behavior ($|V| \log |V|$) on these trees, due to the cost of Insert and DeleteMin operations. Yet all methods appear to run in linear time, presumably because running times are very small, priority queues remain of very small size, and the logarithmic growth does not show with "only" half a million edges. Confirming this last statement is the fact that binary heaps, which have the least overhead, do appear to grow slightly from about 100,000 vertices.

The behavior of the data structures on the worst-case graphs may appear puzzling, until one recalls that the adversary used in constructing these graphs is designed to foil binary heaps, by making each DecreaseKey as expensive as possible within these heaps. But this very design creates a situation that is abnormally favorable for splay trees: each DecreaseKey operation is composed of a Delete operation, at least 50% of which are local, and of an Insert operation, which, since the item is now the smallest in the priority queue, uses very few rotations. (Indeed, further reflection shows that the splay tree will reshape itself into a degenerate right-leaning tree over time; in this tree, all three operations take constant time.) What the worst-case graphs actually show is, first, that splay trees indeed adapt well to unusual patterns of use (as intended in their design); secondly, that binary heaps remain competitive even when in their absolute worst case; and finally, that the behavior of pairing heaps is quite independent of the specific graphs used.

4.4. Results with the four main 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 comparatively inefficient binary heaps yield better running times than the highly optimized quicksort presorting pass. Overall, Kruskal's algorithm always runs significantly more slowly than Prim's (contrary to received opinion and to many textbook pronouncements) 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^{18} vertices, it built a total of over 2^{16} 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 4 shows the running time ratios of the best implementation of Prim's algorithm (pairing heaps, for which we arbitrarily chose auxiliary two-pass pairing heaps with dynamic insertion), the best implementation of Kruskal's algorithm (demand sorting with interruptible quicksort), and the algorithms of Cheriton and Tarjan and of Fredman and Tarjan, all four run on six graph varieties.

Note that Kruskal's algorithm, if given an (unsorted, modifiable) array of edges, actually runs very quickly—often faster than the best version of Prim's algorithm: on 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: on the worst-case graphs constructed for this algorithm (graphs with a single

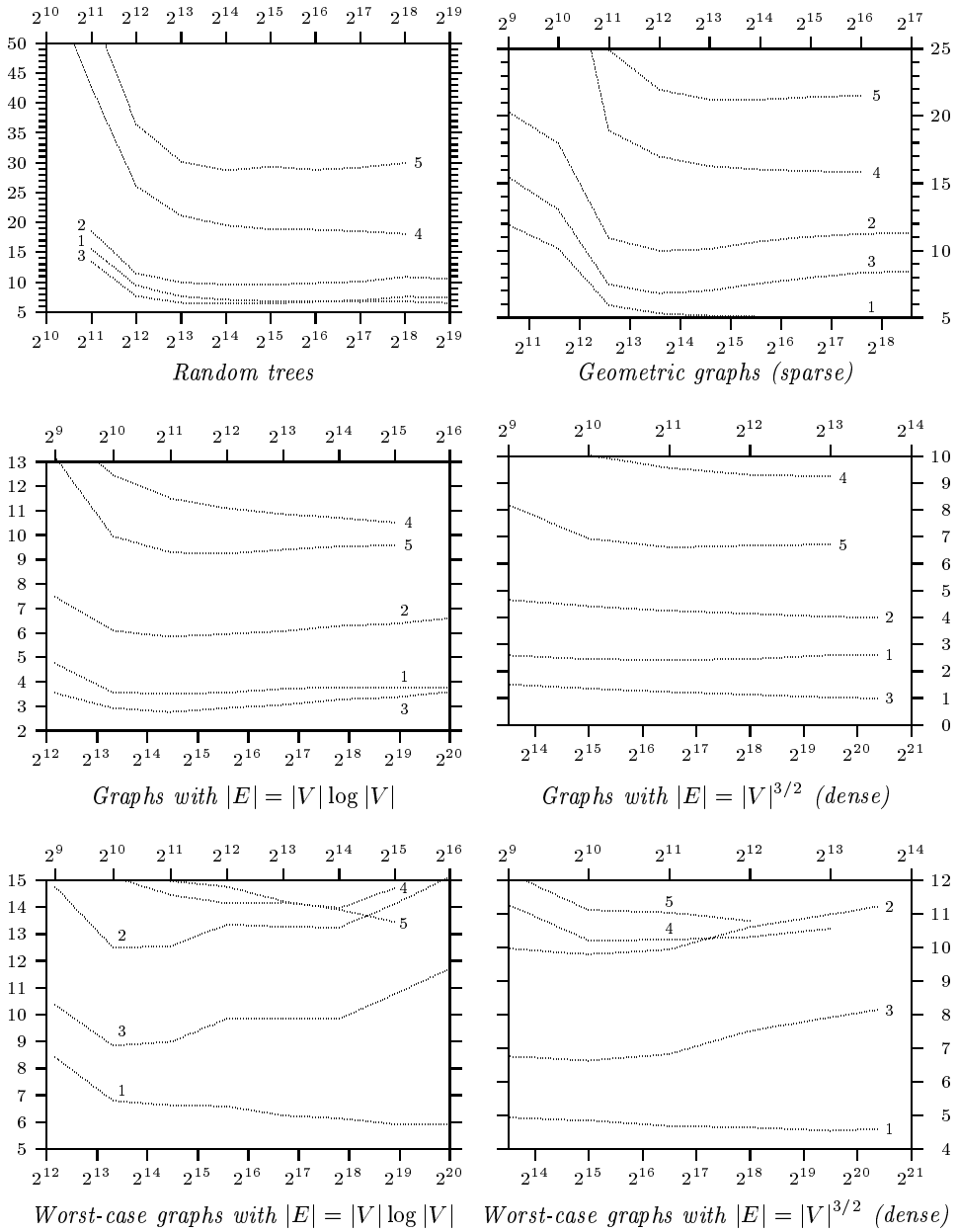

FIGURE 4. Behavior of the main algorithms on a variety of graphs

long bridge), Kruskal's algorithm runs slowly with or without the preparation time. (The worst-case graphs shown in Figure 4 are still those designed to fool Prim's algorithm with binary heaps, but also prove to be nearly worst cases for Kruskal's algorithm.)

Prim's algorithm with pairing heaps (as previously observed) and the two sophisticated algorithms (as expected) exhibit linear-time behavior on all dense or moderately dense graphs, with very small growth on sparse graphs, while Kruskal's algorithm (also as expected) shows supralinear growth on all but the densest graphs (where the number of edges of the graph quickly outgrows the number of edges that the algorithm actually examines).

We have not shown any actual times, but let us simply note that, on a SUN SparcStation II with 65 Mbytes of memory, the source for the figures shown here, the baseline for a dense graph with 8,192 vertices and about 750,000 edges takes about 1.2s and Prim's algorithm takes about 3.0s, while the baseline for a random tree with about 524,000 vertices and as many edges takes 1.1s 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.

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, network flow problems, and a variety of geometric problems (paths and neighborhoods). 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).

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 might be improved by the use of preinsertion. Finally, the issue of extremely large graphs (with millions 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] Johnson, D.S., and C.C. McGeoch, eds., *DIMACS implementation challenge workshop: algorithms for network flows and matching*, DIMACS Tech. Rep. 92-4.

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

[10] Lawler, E.L., J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys (eds.) *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, New York, 1985.

[11] Liao, A.M., *Three priority queue applications revisited*, Algorithmica **7** (1992), pp. 415–427.

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

[13] ———, *On minimizing a set of tests*, SIAM J. Scientific & Statistical Comp. **6**, 4 (1985), pp. 983–1003.

[14] ———, *An empirical analysis of algorithms for constructing a minimum spanning tree*, Proc. 2nd Ann. Workshop on Data Structs. and Algs. WADS-91, Ottawa (CA), in Lecture Notes in Computer Science, Vol. 519, pp. 400–411.

[15] ———, *How to construct a minimum spanning tree in practice*, Proc. Conf. on New Results and New Trends in Computer Science, Graz, Austria (1991); in Lecture Notes in Computer Science, Vol. 555, pp. 192–203.

[16] Morgenstern, C.A., and H.D. Shapiro, *Heuristics for rapidly four-coloring large planar graphs*, Algorithmica **6** (1991), pp. 869–891.

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

Department of Computer Science, University of New Mexico, Albuquerque, NM 87131-1836

E-mail address: moret@cs.unm.edu, shapiro@cs.unm.edu