PERGAMON

Computers & Operations Research 28 (2001) 767–785

computers & operations research

www.elsevier.com/locate/dsw

Minimum-weight spanning tree algorithms A survey and empirical study

Cüneyt F. Bazlamaçcı[a], Khalil S. Hindi[b,*]

[a]*Electrical and Electronics Engineering Department, Middle East Technical University, Ankara 06531, Turkey*
[b]*Department of Systems Engineering, Brunel University, Uxbridge, Middlesex UB8 3PH, UK*

Received 1 August 1999; received in revised form 1 December 1999

Abstract

The minimum-weight spanning tree problem is one of the most typical and well-known problems of combinatorial optimisation. Efficient solution techniques had been known for many years. However, in the last two decades asymptotically faster algorithms have been invented. Each new algorithm brought the time bound one step closer to linearity and finally Karger, Klein and Tarjan proposed the only known expected linear-time method. Modern algorithms make use of more advanced data structures and appear to be more complicated to implement. Most authors and practitioners refer to these but still use the classical ones, which are considerably simpler but asymptotically slower. The paper first presents a survey of the classical methods and the more recent algorithmic developments. Modern algorithms are then compared with the classical ones and their relative performance is evaluated through extensive empirical tests, using reasonably large-size problem instances. Randomly generated problem instances used in the tests range from small networks having 512 nodes and 1024 edges to quite large ones with 16 384 nodes and 524 288 edges. The purpose of the comparative study is to investigate the conjecture that modern algorithms are also easy to apply and have constants of proportionality small enough to make them competitive in practice with the older ones.

Scope and purpose

The minimum-weight spanning tree (MST) problem is a well-known combinatorial optimisation problem concerned with finding a spanning tree of an undirected, connected graph, such that the sum of the weights of the selected edges is minimum. The importance of this problem derives from its direct applications in the design of computer, communication, transportation, power and piping networks; from its appearance as part of solution methods to other problems to which it applies less directly such as network reliability, clustering and classification problems and from its occurrence as a subproblem in the solution of other problems like the travelling salesman problem, the multi-terminal flow problem, the matching problem and the capacitated

* Corresponding author. Tel.: + 1895-203299; fax: + 171-691-9454.
E-mail address: khalil.hindi@brunel.ac.uk (K.S. Hindi).

PII: S0305-0548(00)00007-1

MST problem. Although efficient solution techniques capable of solving large instances had existed, there has been sustained effort over the last two decades to invent asymptotically faster algorithms. With each new algorithm found the time bound approached linearity. Finally, an expected linear-time method has been proposed. The purpose of this work is to survey the classical and modern solution techniques and empirically compare the performance of the existing methods.

Keywords: Network optimisation; Graph algorithms; Minimum spanning tree; Linear-time algorithms; Performance evaluation

1. Introduction

The minimum-weight spanning tree problem (MSTP), one of the most typical and well-known problems of combinatorial optimisation, is that of finding a spanning tree of an undirected, connected graph, such that the sum of the weights of the selected edges is minimum. MSTP is important and very popular for the following reasons:

1. It has direct applications in the design of computer and communication networks, power and leased-line telephone networks, wiring connections, links in a transportation network, piping in a flow network, etc.
2. It offers a method of solution to other problems to which it applies less directly, such as network reliability, clustering and classification problems.
3. It often occurs as a subproblem in the solution of other problems. For example, minimum spanning tree (MST) algorithms are used in several exact and approximation algorithms for the travelling salesman problem, the multi-terminal flow problem, the matching problem and the capacitated MST problem.
4. Efficient solution techniques capable of solving large instances exist.

The long and rich history of the problem is detailed in the surveys by Pierce [1], Maffioli [2] and Graham and Hell [3]. Maffioli's survey takes a broader view and classifies various forms of optimum undirected tree problems, with emphasis on their computational complexity. The survey of Graham and Hell concentrates solely on the MST; not only surveying the algorithms but also tracing their independent sources. However, this informative survey stops at 1985 emphasising classical algorithms mainly.

In the last two decades, efforts were concentrated on finding ever faster algorithms, making use of more modern data structures, with each new algorithm bringing the time bound a step closer to linearity. Finally, Karger et al. [4] proposed the only known linear expected-time algorithm for the restricted random access computation model. This is a randomised, recursive algorithm and requires the solution of a related problem, that of verifying whether a given spanning tree is minimum.

Numerous authors have studied different forms of the classical algorithms and proposed efficient implementation techniques (see for example [5]). Reports on computational experiments of various types have also appeared [6,7].

The continuing interest and developments in the field and the discovery of recent algorithms, including the randomised linear-time algorithm of Karger et al. [4], argue for a comprehensive review of the existing methods, classical and modern, and for a comparative empirical study. Most of the modern algorithms are not difficult to implement. Yet, it is almost standard practice among authors and practitioners to refer to these, but still use one of the classical algorithms which are considerably simpler, though asymptotically slower. The present work aims to verify empirically the conjecture that modern algorithms are also easy to use and have constants of proportionality small enough to make them competitive in practice with older algorithms.

In the next two sections, the formal definition of the problem and a general solution technique are given. Section 4 is an overview of the classical methods and Section 5 includes a survey of the more recent algorithmic developments. Section 6 empirically evaluates the comparative performance of modern and classical MST algorithms and investigates the feasibility of using the recent methods in solving large problem instances. Section 7 presents a summary and the conclusions.

2. Problem definition

Given an undirected graph $G = (V, E)$, where V denotes the set of vertices with $n = |V|$ and E the set of edges with $m = |E|$ and a real number $w(e)$ for each edge $e \in E$ called the weight of edge e, the MSTP is formally defined as finding a spanning tree T^* on G, such that $w(T^*) = \min_T \sum_{e \in T} w(e)$ is the minimum taken over all possible spanning trees of G.

Given a graph G, a spanning tree T and a cotree edge $e = \{i, j\} \in E - T$, the unique cycle in G consisting of the edge e and the edges of the unique chain in T between i and j is called a *fundamental cycle* of G relative to T with respect to e.

For a graph G and two distinct vertices say i and j, let X be any subset of vertices that contains i but not j and let $\bar{X}$ be its complement (i.e., $\bar{X} = V \backslash X$), then set X is a *cut* and the set of edges $\{(i, j) \mid i \in X, j \in \bar{X}, (i, j) \in E\}$ is called a *cutset*. The removal of the arcs in a cutset leaves a disconnected subgraph of G. Corresponding to each edge e of a spanning tree T of a connected graph G, there is a unique cutset called the *fundamental cutset* of T with respect to edge e.

The two theorems on the optimality conditions for a MST are stated below without proofs. Two elegant proofs can be found in [8,9].

Theorem 1 (Fundamental cutset optimality). *A spanning tree T in a weighted graph is a MST if and only if every edge in the tree is a minimum-weight edge in the fundamental cutset defined by that edge.*

Theorem 2 (Fundamental cycle optimality). *A spanning tree T in a graph G is a MST if and only if every edge $e \in E - T$ is a maximum weight edge in the unique fundamental cycle defined by that edge.*

3. General solution

The MSTP is solved by a simple incremental, *greedy* method: the MST is built edge by edge, including appropriate small edges and excluding appropriate large ones. The technique is greedy in the sense that at every stage, the best possible edge is chosen for inclusion in the MST without

producing a cycle in the subgraph constructed so far or for exclusion from the MST without disconnecting the graph.

In his monograph, Tarjan [10] represents the construction process as one of edge colouring. Starting from an initial uncoloured edge set, we colour one edge at a time either blue (included) or red (excluded) according to the following two rules:

Blue rule: Select a cutset that does not contain a blue edge. Among the uncoloured edges in the cutset, select one of minimum weight and colour it blue.

Red rule: Select a simple cycle containing no red edges. Among the uncoloured edges on the cycle, select one of maximum weight and colour it red.

The above blue and red rules are direct applications of the fundamental cutset and fundamental cycle optimality theorems, respectively.

Either rule can be applied at any time, making the method non-deterministic. The most important property of the greedy method is that it colours all the edges of any connected graph and maintains a MST containing all of the blue edges and none of the red ones.

In the next two sections an overview of the classical methods and a survey of the more recent algorithmic developments are given. Table 1 summarises and presents the existing MST algorithms

Table 1
Minimal spanning tree algorithms

Algorithm	Special data structures and subproblems used	Time complexity
Boruvka [11]	Disjoint set union algorithm	$O(m \log n)$
Kruskal (sorted) [12]	Disjoint set union (with path compression and merging with rank)	$O(m\alpha(m,n))$[a]
Kruskal (not sorted) [12]	Disjoint set union and heapsort	$O(m \log n)$
Prim [13]	–	$O(n^2)$
Prim	Binary heap	$O(m \log n)$
Prim	d-Heap	$O(nd \log_d n + m \log_d n)$
Prim	F-Heap	$O(n \log n + m)$
Yao [15]	Heaps of size k, a selection algorithm	$O(m \log \log n)$
Cheriton and Tarjan [16]	Doubly linked queue, leftist heap with lazy meld and lazy deletion, disjoint set union	$O(m \log \log n)$
Fredman and Tarjan [17]	F-heap, doubly linked queue	$O(m\beta(m,n))$[b]
Gabow et al. [18]	F-heaps with packets, disjoint set union	$O(m \log \beta(m,n))$
Karger [22]	Randomisation	$O(n \log n + m)$
Karger et al. [4]	Randomisation, recursion, a linear-time verification algorithm	$O(m)$[c]

[a] $\alpha(m,n)$: Inverse Ackermann's function.
[b] $\beta(m,n) = \min\{i \mid \log^{(i)} n \leqslant m/n\}$ and $\log^{(0)} n = n$.
[c] Expected running time.

in five categories: classical algorithms (Boruvka, Kruskal, Prim); Prim implementations using different forms of heap structure; algorithms which introduced for the first time the special use of heaps and obtained faster running times than the classical algorithms (Yao, Cheriton and Tarjan); algorithms which used F-heaps for the first time (Fredman and Tarjan, Gabow et al.); and the more recent algorithms which use randomisation (Karger, Karger et al.). For each algorithm, all the special structures used and the corresponding time complexities are also included.

4. Classical algorithms

There are three classical algorithms; namely, Boruvka [10,11], Kruskal [12] and Prim [13]; the last two of which are presented in almost every well-known text in the field.

All three algorithms use the blue rule. Although algorithms that use the red rule can be designed, until recently, they were thought to be less efficient. The algorithm of Karger et al. [4] uses the blue rule in conjunction with the red.

The blue edges form a forest, consisting of a set of trees that are called blue trees. The most general blue rule algorithm begins with an initial set of blue trees and applies the following colouring step $n-1$ times: *Colouring step*: (General) Select a blue tree; find a minimum-weight edge incident to this tree and colour it blue.

The three classical algorithms differ in their starting states and colouring steps; in other words, they differ in the criterion used to select the next edge or edges to be added in each iteration. A brief summary of the three classical algorithms is given below with the conventions of Tarjan [10].

The Boruvka algorithm: Start with the initial set of n blue trees, each consisting of one vertex and no edges, and repeat the following step until there is only one blue tree. *Colouring step*: For every blue tree T, select a minimum weight edge incident to T. Colour all selected edges blue.

The algorithm builds the trees uniformly throughout the graph. It is, therefore, suitable for use in parallel computations.

The Kruskal algorithm: Sort the edges in the order of increasing weights and apply the following step to the edges in the sorted list until the number of blue edges is $n-1$. *Colouring step*: If the edge considered has both endpoints in the same blue tree, colour it red; otherwise colour it blue.

The algorithm builds up blue trees in an irregular fashion dictated only by edge weights. It is worth noting that the algorithm is best in situations where the edges are given in sorted order or can be sorted fast (like when the weights are small integers, making it possible to employ radix sort) or where the graph is sparse.

The Prim algorithm: Use an arbitrary starting vertex s and apply the following step $n-1$ times. *Colouring step*: Let T be the blue tree containing s. Select a minimum weight edge incident to T and colour it blue.

The algorithm generates only one nontrivial blue tree from a single root.

The Boruvka algorithm runs in $O(m \log n)$ time (see Table 1). If edges are in disorder with respect to their weight, the Kruskal algorithm requires $O(m \log n)$ time. Given a sorted edge list, finding a MST requires $O(m\alpha(m,n))$ time, assuming the use of the disjoint set union algorithm of Tarjan

[10], where $\alpha(m,n)$ is the inverse Ackermann's function. $\alpha(m,n)$ is defined as $\alpha(m,n) = \min\{i \geqslant 1 \mid A(i, \lfloor m/n \rfloor) > \log n\}$ for $m \geqslant n \geqslant 1$ where Ackermann's function $A(i,j)$ for $i,j \geqslant 1$ is given by

$$A(1,j) = 2^j \quad \text{for } j \geqslant 1,$$

$$A(i,1) = A(i-1,2) \quad \text{for } i \geqslant 2,$$

$$A(i,j) = A(i-1, A(i,j-1)) \quad \text{for } i,j \geqslant 2.$$

The function $A(i,j)$ grows exponentially fast and hence $\alpha(m,n)$ is a very slowly growing function. $\alpha(m,n) \leqslant 3$ for $n < 2^{16}$. Hence, for all practical purposes $\alpha(m,n)$ is assumed to be a constant not larger than four.

The Prim algorithm with an implementation using d-heaps, as proposed by Johnson [14], runs in $O(nd \log_d n + m \log_d n)$. In the case of binary heaps (i.e., $d = 2$), the time complexity is $O(m \log n)$. Choosing $d = \lceil 2 + m/n \rceil$, the time bound becomes $O(m \log_{(2+m/n)} n)$. If $m = \Omega(n^{1+\varepsilon})$, for some positive ε, the running time is $O(m/\varepsilon)$. Thus, the algorithm with Johnson's implementation is well suited for dense graphs as well as sparse ones and the method is asymptotically worse than that of Kruskal only if the edges are pre-sorted.

5. Modern algorithms

Renewed interest in the MSTP in the last two decades led to faster algorithms. Yao [15] was the first to discover an implementation of Boruvka which had a faster asymptotic running time. Cheriton and Tarjan [16] followed by proposing a similar but slightly more practical algorithm. Both algorithms are based on the general blue rule algorithm given above. They both use heaps to implement efficient Boruvka-like colouring steps and run in $O(m \log\log n)$ time. With the invention of an efficient form of heap, called the Fibonacci heap (F-heap), Fredman and Tarjan [17] developed an implementation of the MST algorithm which runs in $O(m\beta(m,n))$, where $\beta(m,n) = \min\{i \mid \log^{(i)} n \leqslant m/n\}$ with $\log^{(0)} n = n$. In addition, they pointed out that the use of an F-heap in its simplest form in the implementation of the Prim algorithm, i.e., replacing the binary or d-heap with an F-heap, solves the MSTP in $O(n \log n + m)$ time. Gabow et al. [18] introduced the use of packets in conjunction with F-heaps and modified the Fredman and Tarjan algorithm to achieve the slightly better time complexity of $O(m \log \beta(m,n))$. The method of Gabow et al. was the fastest theoretically until recently.

Currently, the linear expected-time algorithm of Karger et al. [4] stands as theoretically the fastest. It is a randomised and recursive algorithm which also requires the solution of a related problem, that of verifying whether a given spanning tree is minimum. This subproblem itself has been studied earlier, but it was the linear verification algorithm invented by Dixon et al. [19] which made the proof of the existence of a linear expected time MST algorithm possible. The $O(m)$-time verification algorithm of King [20] seems to be much simpler.

In the following, we briefly describe one example for each of the last three groups in Table 1. The modern algorithms chosen and used in this work are either the fastest or the most practical in each group and are described along with implementation details in [21].

The Cheriton and Tarjan Algorithm

A heap is kept for each blue tree. Each heap holds the edges with at least one endpoint in the tree and which are candidates for becoming blue; the cost of an edge being its key in the heap. The major difference from the Boruvka algorithm is that in each iteration we pick one blue tree in the forest for the merge operation. In conjunction with the heap, the selection rule for picking a blue tree for merging plays an important role in beating the $O(m \log n)$ time bound. Two alternatives are [16]:

- pick uniformly the first in a queue of candidate blue trees (implementation with a doubly linked list is sufficient to have an O(1) time operation including deletion from the queue),
- pick the smallest candidate tree (less efficient with an $O(n)$ implementation).

This algorithm is asymptotically faster than any of the three classical algorithms for sparse graphs, but is slower by a factor of $O(\log \log n)$ than the Prim algorithm for dense graphs.

The Fredman and Tarjan Algorithm

The invention of Fibonacci heaps (F-heaps) by Fredman and Tarjan [17] opened the way to improving some network optimisation algorithms, including those for the shortest path and the MST problems. The most important property of F-heaps is that arbitrary deletion from an n-item heap is possible in $O(\log n)$ amortised time, while all other standard heap operations require O(1) amortised time. In many network optimisation algorithms, the number of deletions is relatively small compared to the total number of heap operations. Thus F-heaps provide a very efficient tool for obtaining asymptotically faster algorithms.

The simplest and most straightforward use of an F-heap in the context of the MSTP is in the implementation of the Prim algorithm to find the minimum weight edge incident to the blue tree containing the starting vertex.

The algorithm requires n deletemin operations and $O(m)$ other heap operations, none of them deletions. Therefore, using F-heaps leads to $O(n \log n + m)$ running time (Table 1).

The $O(m\beta(m, n))$ time bound of Fredman and Tarjan is due to a different strategy, which grows a single tree only until its heap of neighbouring vertices exceeds a certain critical size. Then, another tree starting from a new vertex is grown until the heap gets large again. Continuing in this way, the stopping criterion is that every vertex should be in a tree. The algorithm consists of one or more passes of the above form. Each pass implicitly considers the trees of the forest obtained in the previous pass as super vertices. These super vertices are then grown into larger trees in the next pass. After a sufficient number of passes, only the MST remains.

Fredman and Tarjan's implementation uses a clean-up operation to achieve condensing the graph before starting each pass. This requires the numbering of the trees consecutively from one and assigning to each vertex the number of the tree containing it. The clean-up requires, a two-pass radix sort for the lexicographic ordering of the edges on the numbers of the trees containing their endpoints, the scanning of the edge list to save the appropriate ones, and the construction for each old tree t of a list of the edges with one endpoint in t after the clean-up.

The method of Gabow et al. [18] is an improved version of the Fredman and Tarjan algorithm. The main idea is to group the edges with a common vertex into 'packets' and to work only with packet minima. The structure is similar to that of Fredman and Tarjan in general, but additional

F-heaps are used to handle the packets and to find the packet minima. The packets are initialised according to parameter *packet size*. In order to achieve the claimed time bound $O(m \log \beta(m,n))$, *packet size* should be chosen as $\beta(m,n)$. Moreover, the heap size selected for the ith pass is chosen as $k_1 = 2^{2m/n}$ and $k_i = 2^{k_{i-1}}$ for $i \geqslant 2$.

For practical problems, even if the network is very sparse, *packet size* is usually small. Hence, the advantage of using packets and using F-heaps for such a small number of elements in each packet is not very big. Therefore, we feel that for most realistic problem sizes, the Fredman and Tarjan algorithm will be comparable to, or even better than, that of Gabow et al. However, this conjecture remains to be tested.

The Algorithm of Karger et al.

The algorithm of Karger et al. solves the more general problem of finding a minimum forest in a possibly disconnected graph, but we assume the graph to be connected for simplicity. We also assume that all edge weights are distinct, since an ordering is possible even if they are not. Unlike most algorithms for finding a MST, the Karger et al. algorithm makes use of both the blue and red rules.

The algorithm relies on a random sampling step to discard edges that cannot be in the MST. We need the following terminology to state the result that indicates the effectiveness of this edge elimination step. Given an edge $(x,y) \in E$ and a forest F in G, $F(x,y)$ denotes the path connecting x and y in F with $w_F(x,y)$ denoting the maximum weight of an edge on $F(x,y)$. We use the convention $w_F(x,y) = \infty$ if x and y are not connected in F. An edge is F_{heavy} if $w(x,y) > w_F(x,y)$ and F_{light} otherwise. It is worth noting that all edges of F are F_{light} and no F_{heavy} edges can be in the MST of G as a result of the red rule. There exist $O(m)$-time algorithms to compute the F_{heavy} edges (for example, [19,20]).

Lemma 1 (Karger et al. [4]). *Let H be a subgraph obtained from G by including each edge independently with probability p, and let F be the minimum spanning forest of H. The expected number of F_{light} edges in G is at most n/p where n is the number of vertices.*

We omit the proof and only state that the lemma generalises to matroids.

There are three major operations, namely the Boruvka step, the random sampling step and the verification step:

Boruvka step: Given a graph G, apply the Boruvka algorithm to carry out one colouring step only. Contract the graph G, i.e., replace each blue tree by a single vertex, delete edges whose endpoints are the same after the replacement and delete all but the lowest-weight edge among each set of multiple edges, again after the replacement.

Random sampling step: Given a contracted graph G, choose a subgraph H by selecting each edge in G independently with a probability 1/2.

Verification step: Given any minimum spanning forest F of $H \subseteq G$, find all the F_{heavy} edges (both those in H and not in H) and delete them from G to reduce the graph further

Each Boruvka step reduces the number of vertices by at least one half. The following algorithm is recursive and generates two subproblems after each execution of step 1.

Algorithm

1. Given $G = (V, E)$, apply two successive Boruvka steps to the graph to contract G.
2. Apply the random sampling step to the contracted graph to select H.
3. Apply the algorithm recursively producing a minimum spanning forest F of the H formed in step 2.
4. Given F of H, apply the verification step to the subgraph H, which was chosen, and obtain a graph G' which is reduced further.
5. Apply the algorithm recursively to G' to compute the minimum spanning forest F' of G'.
6. Return those edges contracted in step 1 together with the edges of F'.

The worst case and the expected running time of the minimum spanning forest algorithm is $O(\min\{n^2, m \log n\})$ and $O(m)$, respectively. It runs in $O(m)$ time with probability $1 - \exp(-\Omega(m))$ [4].

6. Computational results

Computational testing of the surveyed algorithms was carried out on problem instances of various sizes. All implementations were in Pascal and computations were carried out on an HP735 workstation running under UNIX. To the best knowledge of the authors, this is the first report in the literature on a comparative, empirical study of the performance of the algorithms of Boruvka and Karger et al. on problem instances of considerable size.

The main objectives of the computational study are to:

- Compare the performance of the classical and modern algorithms and investigate empirically the practicability of adopting the recently developed ones among the latter to substitute the former.
- Observe the effect of varying network sizes and compare the real time performance of the algorithms with the theoretical bounds.
- Pinpoint the fastest MST algorithm in practice; at least for the range of problem sizes tested.

Random problem instances were generated. The size of the largest problem instances was chosen such that they are solvable on a relatively small computer. The numbers of nodes, ranged, in powers of two, from 512 (0.5K) to 16K. The large resulting networks were chosen to be sparse; with m/n ratios of two, four, eight, sixteen and thirty two. Each $n \times m$ combination constitutes a group of five different problem instances; all the timings are averages for each group. For each problem instance, nodes were selected randomly from a 1000×1000 grid with a uniform probability density. To ensure connectivity, edges $(i, j) | j = i + 1$ were chosen first, i.e., each node i was connected to node $i + 1$, then the end nodes of the remaining edges were selected from the complete graph at random. The cost of an edge was calculated as the integer part of the Euclidean distance between its two endpoints.

Each problem instance was solved to find the corresponding MST using the different algorithms. Dynamic memory management techniques are employed to make handling large networks possible. Since memory management operations are machine and compiler dependent, we consider

them as general utility operations and do not include their execution times in the following discussion.

Results regarding the Kruskal algorithm in relation to sorting methods can be analysed in two ways. One is by keeping n constant and increasing m or vice versa; the other is by increasing the input size while keeping the m/n ratio constant. For the latter, we double both input parameters n and m. It is worth noting that the density of the graph decreases with such an approach.

It is well known that none of the existing sorting algorithms is the best all round. Some are good for small m, others for large m; where m is the number of items in the list to be sorted. Insertion sort works well when the list is already partially ordered. Merge sort has the best worst-case behaviour, but it requires more storage than a heap sort. Quick sort has the best average behaviour, but its worst-case behaviour is $O(m^2)$. In this work, we employed both the quick sort and the heap sort in computing the MST with the Kruskal algorithm.

We analyse the results by keeping n constant first. Even though it requires some extra memory, quick sort is usually recommended for large m. The results obtained in our tests in sorting rather large lists demonstrate that it is fast and advantageous up to $m = 16$K, but at $m = 32$K, quick sort and heap sort become comparable. Beyond this point, we observe the non-linear character of quick sort and heap sort dominates. Fig. 1 illustrates these thresholds for $n = 2$K. Fig. 2 on the other hand demonstrates the non-linearity of quick sort and the superiority of heap sort for $n = 8$K. Although heap sort is also non-linear, its non-linearity was not observed within the problem range under consideration. The conjecture that heap sort is usually not quite as fast as quick sort does not

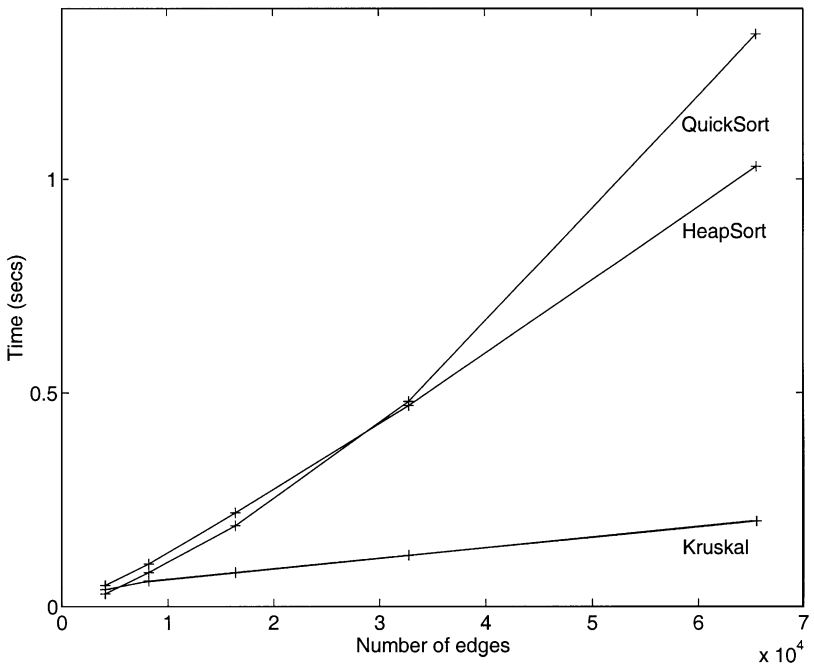

Fig. 1. Performance of Kruskal and the sorting algorithms used for $n = 2$K.

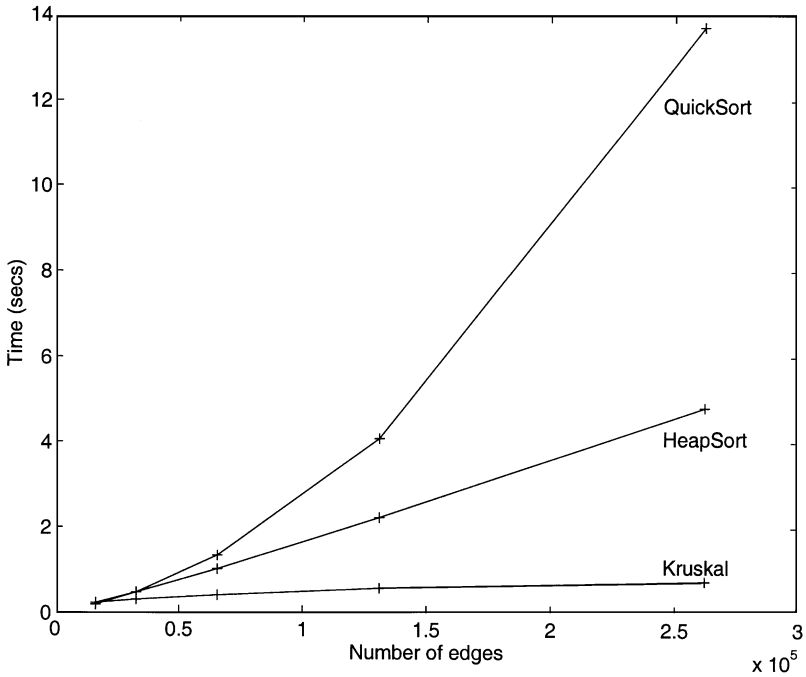

Fig. 2. Performance of Kruskal and the sorting algorithms used for $n = 8$K.

hold for very large m. Once the edges are available in sorted order, the average computing time for Kruskal is extremely small (see Figs. 1 and 2).

The second type of analysis, in which the m/n ratio is kept constant while increasing the input size, produced similar results. The gap between quick sort and heap sort increases as the m/n ratio increases for large n.

The Prim algorithm performs better with a conventional binary heap than with any other heap. In general, the implementation with d-heaps comes second, but the implementation with F-heaps gains an advantage over it as the number of edges increases (see Table 2). Increasing m/n decreases the difference between the F-heap and d-heap implementations, and the implementation with F-heap dominates the one with d-heap for $m/n = 32$, except for small n ($n \leqslant 2$K). It is worth noting that implementing the Prim algorithm without the use of heaps is extremely inefficient comparatively (see column no-hp in Table 2). The results also reveal the quadratic nature of the algorithm.

The Boruvka algorithm can be implemented in various ways. In the current study, we used the disjoint set-union algorithm with and without clean-up, i.e., permanently deleting the edges that have both end-points in the same connected component. Figs. 3 and 4 are performance plots.

An interesting observation is that the use of clean-up turns out to be counter productive. In our implementation, we first form a new edge list in the form of a doubly linked list, so that each time an edge is found to have two end points in the same subtree, it can be deleted permanently from the edge list in O(1) time. However, the overhead of the initial copying process and accessing the edges

Table 2
Performance of Prim's algorithm using different forms of heaps (s)[a]

m	$n = 0.5$K				$n = 1$K				$n = 2$K			
	no-hp	d-hp	b-hp	F-hp	no-hp	d-hp	b-hp	F-hp	no-hp	d-hp	b-hp	F-hp
1K	0.08	0.01	0.01	0.01	—	—	—	—	—	—	—	—
2K	0.09	0.01	0.01	0.02	0.37	0.03	0.03	0.03	—	—	—	—
4K	0.09	0.02	0.02	0.03	0.38	0.03	0.03	0.04	1.88	0.06	0.06	0.06
8K	0.11	0.03	0.04	0.04	0.40	0.05	0.05	0.06	1.90	0.08	0.07	0.09
16K	0.14	0.07	0.06	0.08	0.43	0.08	0.08	0.10	1.93	0.11	0.11	0.13
32K	—	—	—	—	0.48	0.15	0.13	0.16	2.00	0.19	0.18	0.21
64K	—	—	—	—	—	—	—	—	2.16	0.34	0.31	0.37
m	$n = 4$K				$n = 8$K				$n = 16$K			
	no-hp	d-hp	b-hp	F-hp	no-hp	d-hp	b-hp	F-hp	no-hp	d-hp	b-hp	F-hp
8K	12.69	0.14	0.14	0.15	—	—	—	—	—	—	—	—
16K	12.73	0.19	0.18	0.21	54.38	0.33	0.30	0.33	—	—	—	—
32K	12.80	0.27	0.26	0.30	54.46	0.42	0.40	0.46	220.41	0.71	0.66	0.70
64K	12.94	0.44	0.41	0.47	54.68	0.61	0.57	0.66	220.51	0.92	0.88	0.97
128K	13.25	0.81	0.73	0.81	55.09	0.99	0.91	1.03	221.86	1.32	1.26	1.40
256K	—	—	—	—	55.75	1.80	1.60	1.75	222.03	2.15	1.98	2.15

[a]no-hp: without any heap; d-hp: with d-heap; b-hp: with binary heap; F-hp: with Fibonacci heap.

in the list is larger than the gain from reducing the number of edges. Although the problems considered are quite large, they require few Boruvka phases and the reduction in the total number of edges does not compensate for the list processing overhead. Nevertheless, in much larger problems, Boruvka's algorithm with clean-up may dominate.

Next, we compare one representative of each group of modern algorithms in Table 1. Cheriton and Tarjan, Fredman and Tarjan and Karger et al. are chosen for their practicality or recency as representatives of each group. Figs. 5–8 present their performances, respectively.

The Cheriton and Tarjan algorithm is relatively the simplest and the most efficient in terms of real CPU time for the problem instances considered. This result is verified up to $n = 16$K and $m = 256$K. Due to physical memory limitations, the observed non-linearity beyond $n = 16$K and $m = 128$K was not verified using even larger problem instances.

The Fredman and Tarjan algorithm seems to be most interesting from the viewpoint of practical performance. Although it is fast in theory, the break even point seems far away. Moreover, unless n is sufficiently large, it is highly sensitive to the m/n ratio. The sparser the graph, the more inefficient is the method, which, at least for the range of problem sizes we have used, conflicts with the asymptotic bound. Figs. 6 and 7 are performance plots which illustrate that given n, increasing m/n decreases the running time until the point where the heap size k is larger than n and hence where only one pass is performed. The choice of the parameter k dictates the number of passes. Smaller values of k reduce the time per pass; larger values reduce the number of passes. In other

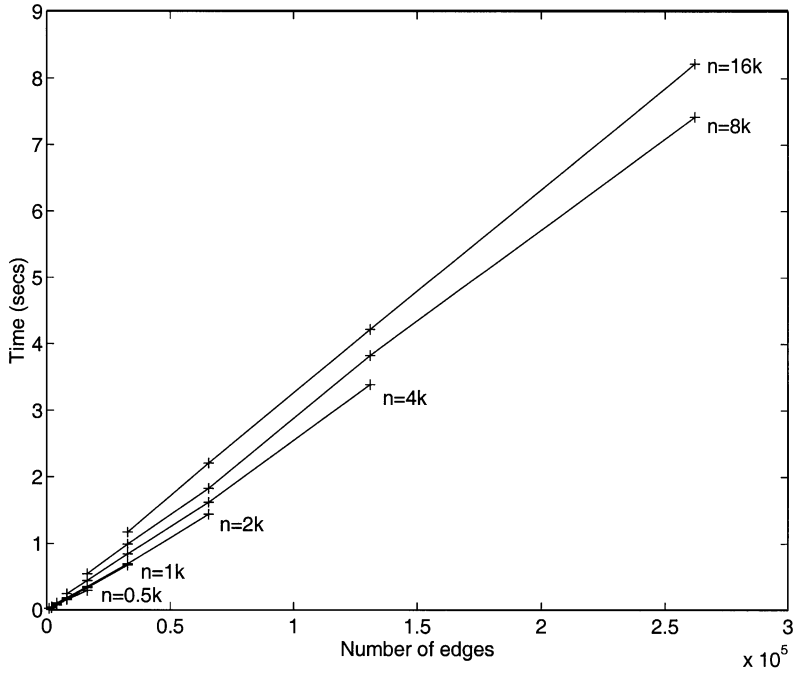

Fig. 3. Performance of Boruvka's algorithm.

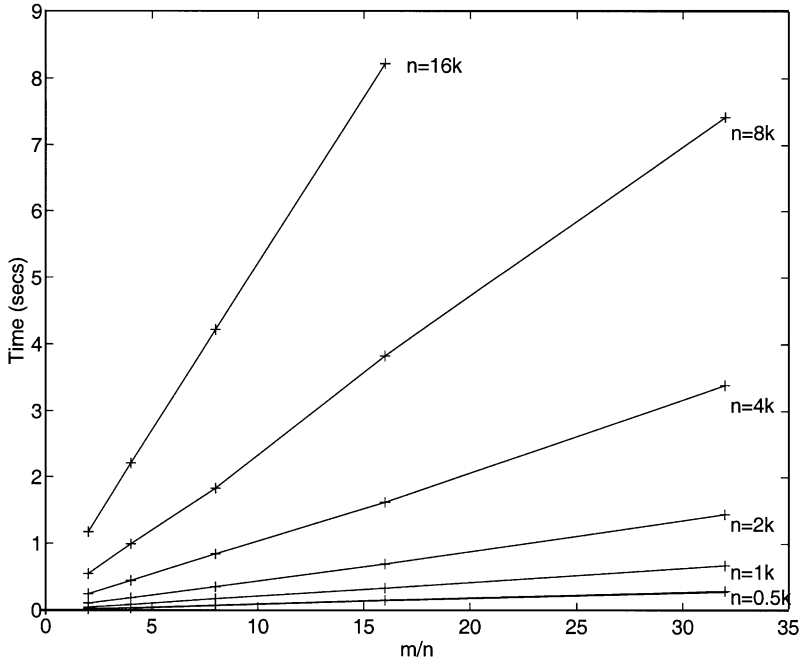

Fig. 4. Performance of Boruvka's algorithm w.r.t. to m/n ratio.

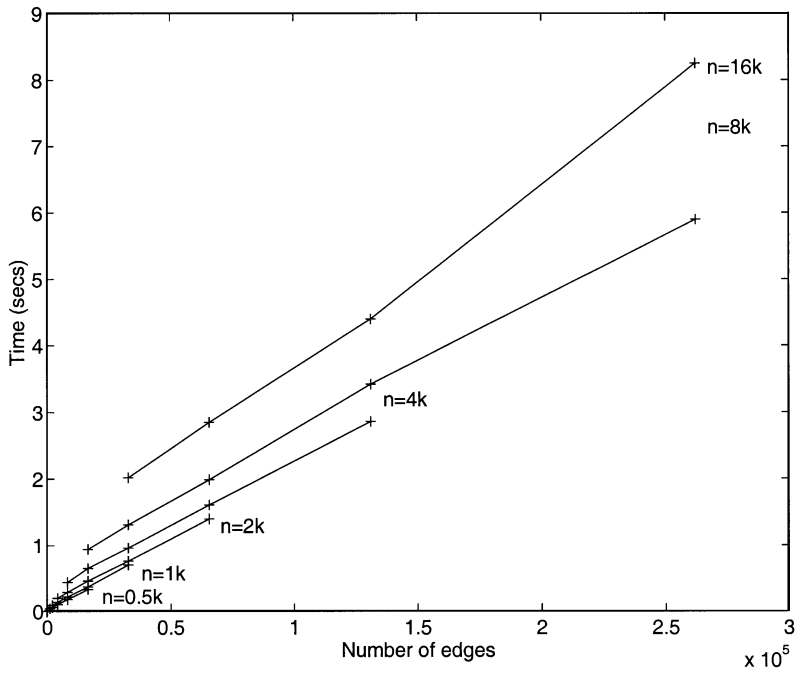

Fig. 5. Performance of Cheriton and Tarjan's algorithm.

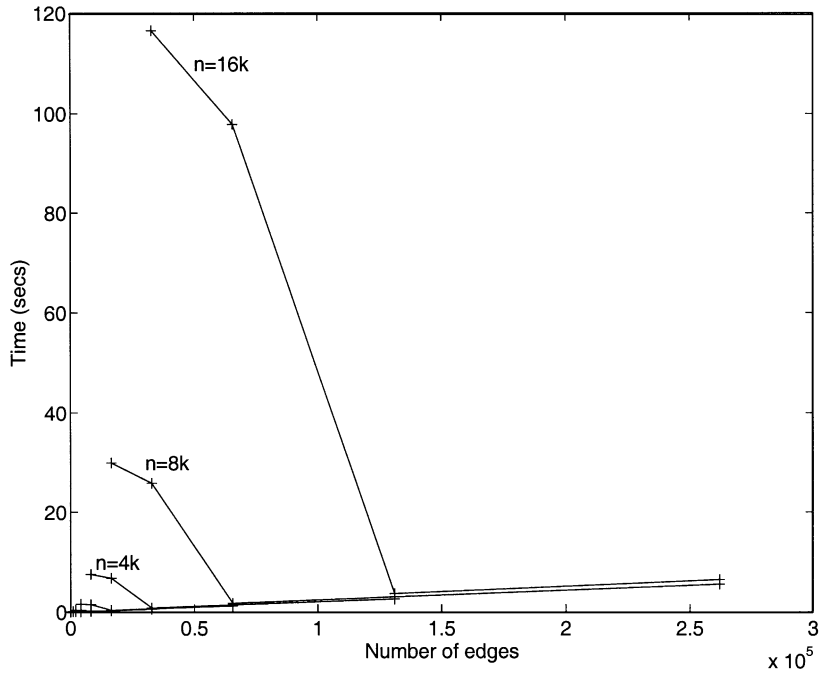

Fig. 6. Performance of Fredman and Tarjan's algorithm.

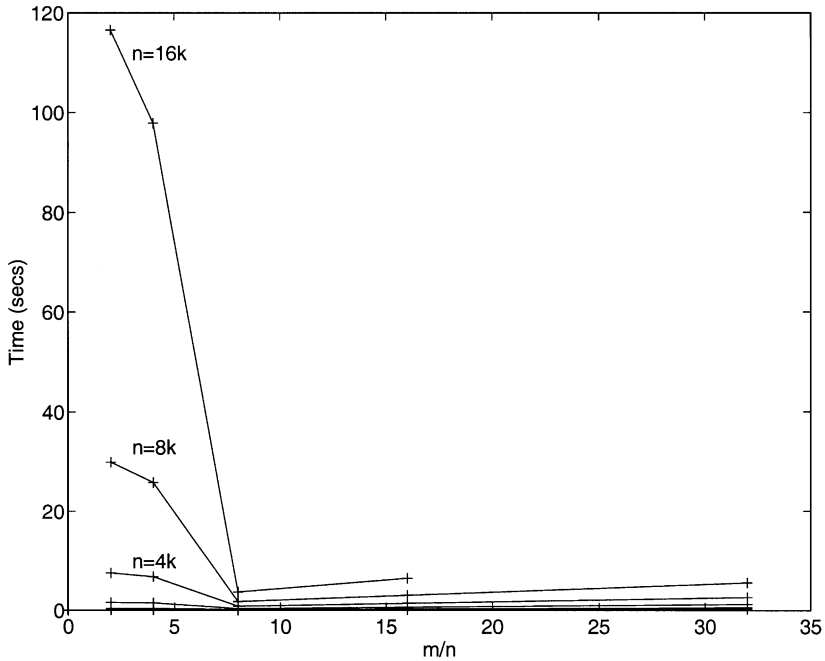

Fig. 7. Performance of Fredman and Tarjan's algorithm w.r.t. m/n.

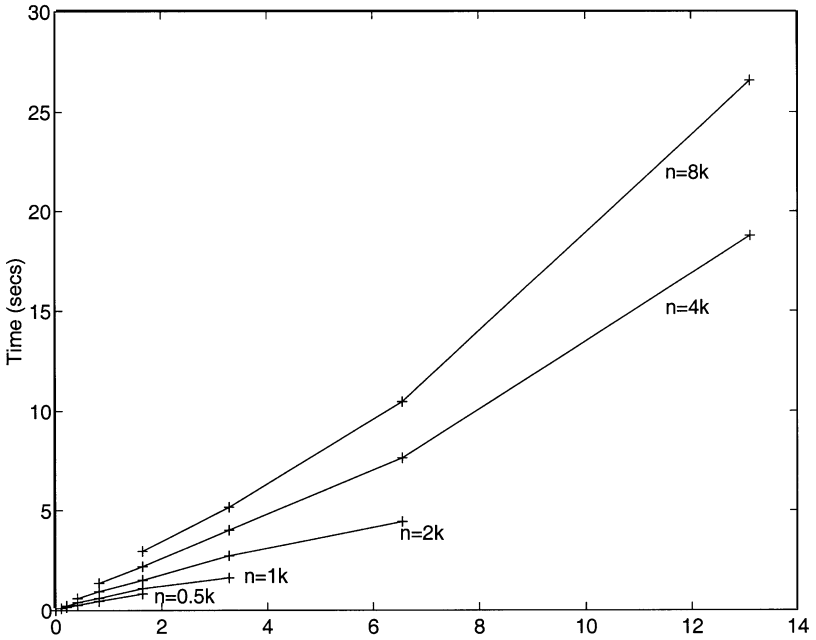

Fig. 8. Performance of Karger et al.'s algorithm.

words, the trees that are grown without interruption tend to be small in passes with a small k. Practical results demonstrate that stopping the tree growing process and restarting growing a new tree from scratch very often, following the necessary heap initialisation and other assignments, is inefficient. Choosing $k = 2^{2m/t}$, where t is the number of trees before the pass, optimises the running time per pass to be O(m), but this is not observed for inputs which are not sufficiently large.

The inefficiency for small m/n is also partly due to the use of the clean-up process after each pass. Unless n and m are sufficiently large, each additional pass adds a substantial amount of clean-up time.

The algorithm of Karger et al. is also not practical for the problem sizes we considered. Due to its recursive nature, we store a reduced network in each subproblem and, hence, spend more time in copying and processing these local networks. Such overheads are either not small or even inordinate. Moreover, it is important to note that the expected time linearity of the algorithm is not observed with the data set used.

Table 3 offers a comparison of the modern algorithms with the classical for $n = 8$K and $n = 16$K, with m/n ranging between 2 and 32. The results indicate that the conventional binary heap implementation of the Prim algorithm is the fastest for the problem sizes considered.

Table 4 indicates roughly the amount of work one has to do to implement each algorithm. The algorithm of Karger et al. is the most difficult to implement as it also requires the solution of the minimum spanning tree verification and the least-common ancestor subproblems in linear time.

A problem related to that addressed in this work; namely, the problem of re-optimisation of solutions, is noteworthy. This problem, also known in the literature as the problem of MST maintenance on dynamic graphs, deals with maintaining a minimum spanning tree on a graph subject to modifications (change of edge weights, insertion and deletion of edges and vertices). Neither classical nor modern algorithms provide adequate data structures and mechanisms for fast maintenance. However, there are for this purpose fast methods that use special structures and

Table 3
Performance of MST algorithms (s)

n	m	Kruskal	Prim[a]	Boruvka	Cheriton	Fredman	Karger
4K	8K	0.21	0.14	0.24	0.44	7.53	1.36
	16K	0.36	0.18	0.44	0.65	6.78	2.19
	32K	0.66	0.26	0.84	0.96	0.84	4.02
	64K	1.31	0.41	1.62	1.61	1.44	7.64
	128K	2.69	0.73	3.39	2.86	2.64	18.77
8K	16K	0.46	0.30	0.54	0.94	29.91	2.96
	32K	0.78	0.40	0.99	1.31	25.82	5.17
	64K	1.44	0.57	1.83	1.99	1.78	10.47
	128K	2.83	0.91	3.83	3.42	3.06	26.58
	256K	5.58	1.60	7.42	5.90	5.58	[b]

[a] With conventional binary heap.
[b] Not performed due to memory limitations.

Table 4
Code sizes of MST algorithms (bytes)

Code	Kruskal	Prim[a]	Boruvka	Cheriton	Fredman	Karger
Source	10664	8788	12456	25761	35742	82121
Executable	7168	6656	7680	13312	15872	38656

[a]With conventional binary heap.

techniques (see for example [23]). A brief survey of the MST maintenance problem can be found in [21].

7. Conclusions

The paper briefly surveys the existing algorithms for solving the MSTP and classifies them according to their similarities or the tools and subproblems they use. The feasibility of using the modern methods, which are more complicated compared to the classical ones, has been investigated.

The empirical results lead to the conclusion that while the algorithms developed in recent years are theoretically very important, they require excessive problem sizes to justify their use as an alternative to the classical methods. The problem dimensions we have experimented with are rather large, but still not sufficiently large to make the asymptotically fast algorithms competitive in practice with the older ones. Among modern methods, the Cheriton and Tarjan round robin algorithm is relatively the simplest. It has constants of proportionality small enough to make it the most efficient and most competitive of the modern methods.

This work has helped in observing the effect of varying network sizes and in comparing the empirical time performance of the algorithms with the theoretical bounds. In the case of the Fredman and Tarjan algorithm, it is found that extremely large problem sizes are required to reach the break even point and hence the method is not viable to apply in realistic cases.

The expected linear-time algorithm of Karger et al. was also found to be not feasible to use, at least for the range of problem sizes we considered. It uses the complicated subproblem of verifying a MST in linear time, which in turn requires the solution of the nearest common ancestor problem in linear time.

For the range of problem sizes considered here, the Prim algorithm with a conventional binary heap implementation is found to be the fastest. In view of its simplicity and practicality, it is safe to conclude that the Prim algorithm will continue to serve efficiently with a conventional binary heap implementation in most cases and possibly with a F-heap implementation in problem instances much larger than those considered in this study.

References

[1] Pierce AR. Bibliography on algorithms for shortest path, shortest spanning tree and related circuit routing problems (1956–1974). Networks 1975;5:129–49.

[2] Maffioli F. Complexity of optimum undirected tree problems: a survey of recent results. In: Ausiello G, Lucertini M, editors. Analysis and design of algorithms in combinatorial optimization. International Center for Mechanical Sciences. CISM Courses and Lectures, 266. New York: Springer, 1981. p. 107–28.
[3] Graham RL, Hell P. On the history of the minimum spanning tree problem. Annals of the History of Computing 1985;7:43–57.
[4] Karger DR, Klein PN, Tarjan RE. A randomized linear-time algorithm to find minimum spanning trees. Journal of the Association for Computing Machinery 1995;42/2:321–8.
[5] Haymond RE, Jarvis JP, Shier DR. Computational methods for minimum spanning tree algorithms. SIAM Journal on Scientific and Statistical Computing 1984;5/1:157–74.
[6] Glover F, Klingman D, Krishnan R, Padman A. An in-depth empirical investigation of non-greedy approaches for the minimum spanning tree problem. European Journal of Operational Research 1992;56:343–56.
[7] Moret BME, Shapiro D. How to find a minimum spanning tree in practice. In: Maurer H, editor. New results and new trends in computer science: proceedings, Graz, Austria, June 1991. Lecture Notes in Computer Science, Vol. 555. Berlin: Springer, 1991. p. 192–203.
[8] Balakrishnan VK. Network optimization. London: Chapman and Hall, 1995.
[9] Gondran M, Minoux M. Graphs and Algorithms. New York: Wiley, 1984.
[10] Tarjan RE. Data structures and network algorithms. In CBMS-NFS Regional Conference Series in Applied Mathematics. Philadelphia: Society for Industrial and Applied Mathematics, 1983. p. 44.
[11] Boruvka O. O jistém problému minimálním. Práca Moravské Přírodovědecké Společnosti 1926;3:37–58 [in Czech], cited in [3,10].
[12] Kruskal JB. On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society 1956;7:48–50.
[13] Prim RC. Shortest connection networks and some generalizations. Bell System Technical Journal 1957;36:1389–401.
[14] Johnson DB. Priority queues with update and finding minimum spanning trees. Information Processing Letters 1975;4:53–7.
[15] Yao A. An $O(|E|\log\log|V|)$ algorithm for finding minimum spanning trees. Information Processing Letters 1975;4:21–3.
[16] Cheriton D, Tarjan RE. Finding minimum spanning trees. SIAM Journal on Computing 1976;5:724–42.
[17] Fredman ML, Tarjan RE. Fibonacci heaps and their uses in improved network optimisation algorithms. Journal of the Association for Computing Machinery 1987;34/3:596–615.
[18] Gabow HN, Galil Z, Spencer TH, Tarjan RE. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. Combinatorica 1986;6:109–22.
[19] Dixon B, Rauch M, Tarjan RE. Verification and sensitivity analysis of minimum spanning trees in linear time. SIAM Journal on Computing 1992;21:1184–92.
[20] King V. A simpler minimum spanning tree verification algorithm. In: Akl SG, Dehne F. editors. Algorithms and data structures: Proceedings of the 4th International Workshop, Kingston, Canada. Lecture Notes in Computer Science, 955. Berlin: Springer, 1995. p. 440–8.
[21] Bazlamaçcı CF. Optimised network design: minimum spanning trees and minimum concave-cost problems. Ph.D. thesis, University of Manchester, Institute of Science and Technology, Manchester, England, 1996.
[22] Karger DR. Random sampling in matroids, with applications to graph connectivity and minimum spanning trees. In Proceedings of the 34th Annual IEEE Symposium on Foundations of Computer Science. Los Alamitos, California: IEEE Computer Society Press, 1993. pp. 84–93.
[23] Eppstein D, Italiano GF, Tamassia R, Tarjan RE, Westbrook J, Yung M. Maintenance of a minimum spanning forest in a dynamic plane graph. Journal of Algorithms 1992;13:33–54.

Cüneyt F. Bazlamaçcı is an Assistant Professor in Electrical and Electronics Engineering Department, Middle East Technical University (METU), Ankara, Turkey. He received his B.S. and M.S. degrees, both in electrical and electronics engineering, from the Middle East Technical University and his Ph.D. degree in Computation from the University of Manchester, Institute of Science and Technology (UMIST), Manchester. His current research interests include network

and graph algorithms, synthesis and design of computer, communication and transportation networks, applications of graph theory in operations research, metaheuristics and advanced data structures and algorithms.

Khalil S. Hindi is a Professor of Systems Engineering at Brunel University, Greater London. His current research interests are in computer-aided management, planning, operation, scheduling and control of engineering systems; particularly manufacturing systems, electric power systems and gas and water distribution and transmission systems, employing computing and mathematical modelling techniques.