An Optimal Minimum Spanning Tree Algorithm

SETH PETTIE AND VIJAYA RAMACHANDRAN

The University of Texas at Austin, Austin, Texas

Abstract. We establish that the algorithmic complexity of the minimum spanning tree problem is equal to its decision-tree complexity. Specifically, we present a deterministic algorithm to find a minimum spanning tree of a graph with n vertices and m edges that runs in time $O(\mathcal{T}^*(m, n))$ where $\mathcal{T}^*$ is the minimum number of edge-weight comparisons needed to determine the solution. The algorithm is quite simple and can be implemented on a pointer machine.

Although our time bound is optimal, the exact function describing it is not known at present. The current best bounds known for $\mathcal{T}^*$ are $\mathcal{T}^*(m, n) = \Omega(m)$ and $\mathcal{T}^*(m, n) = O(m \cdot \alpha(m, n))$, where α is a certain natural inverse of Ackermann's function.

Even under the assumption that $\mathcal{T}^*$ is superlinear, we show that if the input graph is selected from $G_{n,m}$, our algorithm runs in linear time with high probability, regardless of n, m, or the permutation of edge weights. The analysis uses a new martingale for $G_{n,m}$ similar to the edge-exposure martingale for $G_{n,p}$.

Categories and Subject Descriptors: F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*; G.3 [**Probability and Statistics**]

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Graph algorithms, minimum spanning tree, optimal complexity

1. *Introduction*

The minimum spanning tree (MST) problem has been studied for much of this century and yet despite its apparent simplicity, the problem is still not fully understood. Graham and Hell [1985] give an excellent survey of results from the earliest known algorithm of Borůvka [1926] to the invention of Fibonacci heaps, which were central to the algorithms in Fredman and Tarjan [1987] and Gabow et al. [1986]. Chazelle [1997] presented an MST algorithm based on the Soft Heap [Chazelle 2000a] having complexity $O(m\alpha(m, n) \log \alpha(m, n))$, where α is a certain inverse of Ackermann's function. Recently Chazelle [2000b] modified the

A preliminary version of this article appeared in *Proceedings of the 27th International Colloquium on Automata, Languages and Programming* (*ICALP*) (Geneva, Switzerland). Springer-Verlag, New York, 2000.

Part of this work was supported by Texas Advanced Research Program Grant 003658-0029-1999.

S. Pettie was also supported by an MCD Graduate Fellowship.

Authors' address: The University of Texas at Austin, Department of Computer Science, Taylor Hall 2.124 (Mailcode 0500), Austin, TX 78712, e-mail: {seth;vlr}@cs.utexas.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this worked owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2002 ACM 0004-5411/02/0100–0016 $5.00

algorithm in Chazelle [1997] to bring down the running time to $O(m \cdot \alpha(m, n))$. Later a similar algorithm of the same running time was presented by Pettie [1999], which gives an alternate exposition of the $O(m \cdot \alpha(m, n))$ result. This is the tightest time bound for the MST problem to date, though not known to be optimal.

All algorithms mentioned above work on a pointer machine [Tarjan 1979a] under the restriction that edge weights may only be subjected to binary comparisons. If, in addition, we have access to a stream of perfectly random bits, Karger et al. [1995] showed that the MST can be computed in linear time with high probability. Fredman and Willard [1994] gave a deterministic linear time MST algorithm under the unit-cost RAM model, assuming edge weights are integers represented in binary.

It is still unknown whether these more powerful models are necessary to compute the MST in linear time. However, in this article, we give a deterministic, comparison-based MST algorithm that runs on a pointer machine in $O(\mathcal{T}^*(m, n))$ time, where $\mathcal{T}^*(m, n)$ is the number of edge-weight comparisons needed to determine the MST on any graph with m edges and n vertices. Additionally, we show that our algorithm runs in linear time for the vast majority of graphs, regardless of the number of edges in the graph or the permutation of edge weights.

Because of the nature of our algorithm, its exact running time is not known. This might seem paradoxical at first. The source of our algorithm's optimality, and its mysterious running time, is the use of precomputed "MST decision trees" whose exact depth is unknown but nonetheless provably optimal. The technique of obtaining optimal algorithms via precomputation was used in a simpler setting in Larmore [1990] for searching convex matrices and in Dixon et al. [1992] for MST sensitivity analysis. We should point out that precomputing optimal decision trees does *not* increase the constant factor hidden by big-Oh notation, nor does it result in a nonuniform algorithm. A trivial lower bound on the running time of our algorithm is $\Omega(m)$; the best upper bound, $O(m\alpha(m, n))$, is due to Chazelle [2000b].

Our optimal MST algorithm should be contrasted with the complexity-theoretic result that any optimal verification algorithm for some problem can be used to construct an optimal algorithm for the same problem [Jones 1997]. Though asymptotically optimal, this construction hides astronomical constant factors and proves nothing about the relationship between algorithmic complexity and decision-tree complexity. See Section 8 for a discussion of these and other related issues.

In the next sections, we review some well-known MST results that are used by our algorithm. In Section 3, we prove a key lemma and give a procedure for partitioning the graph in an MST-respecting manner. Section 4 gives an overview of our optimal algorithm and discusses the structure and use of precomputed decision-trees for the MST problem. Section 5 gives the algorithm and a proof of optimality. Section 6 shows how the algorithm may be modified to run on a pointer machine. In Section 7, we show our algorithm runs in linear-time with high probability if the input graph is selected at random. Sections 8 and 9 discuss related problems and algorithms, open questions, and the actual complexity of MST.

2. *Preliminaries*

The input is an undirected graph $G = (V, E)$ where each edge is assigned a distinct real-valued *weight*. By convention, $|V| = n$ and $|E| = m$. The *minimum spanning*

forest (*MSF*) problem asks for a spanning acyclic subgraph of G having the least total weight. In this article, we assume for convenience that the input graph is connected, since otherwise we can find its connected components in linear time and then solve the problem on each connected component. Thus, the MSF problem is identical to the minimum spanning *tree* problem.

It is well known that one can identify edges provably in the MSF using the *cut* property, and edges provably not in the MSF using the *cycle* property. The cut property states that the lightest edge crossing any partition of the vertex set into two parts must belong to the MSF. The cycle property states that the heaviest edge in any cycle in the graph cannot be in the MSF.

2.1. BORŮVKA STEPS. The earliest known MSF algorithm is due to Borůvka [1926]. The algorithm is quite simple: It proceeds in a sequence of stages, and in each stage, or *Borůvka step*, it identifies a forest F consisting of the minimum-weight edge incident to each vertex in the graph G, then forms the graph $G_1 = G\backslash F$ as the input to the next stage. Here $G\backslash F$ denotes the graph derived from G by contracting edges in F (by the cut property these edges belong to the MSF.) Each Borůvka step takes linear time, and since the number of vertices is reduced by at least half in each step, Borůvka's algorithm takes $O(m \log n)$ time.

Our optimal algorithm uses a procedure called Boruvka2(G; F, G'). This procedure executes two Borůvka steps on the input graph G and returns the contracted graph G' as well as the set of edges F identified as part of the MSF during these two steps.

2.2. DIJSKTRA-JARNÍK-PRIM ALGORITHM. Another early MSF algorithm that runs in $O(m \log n)$ time is the one by Jarník [1930], rediscovered by Dijkstra [1959] and Prim [1957]. We refer to this algorithm as the *DJP* algorithm. Briefly, the DJP algorithm grows the MSF T one edge at a time. Initially, T is an arbitrary vertex. In each step of the DJP algorithm, T is augmented with the least-weight edge (x, y) such that $x \in T$ and $y \notin T$. By the cut property, all edges added to T are in the MSF.

LEMMA 2.1. *Let T be the tree formed after the execution of some number of steps of the DJP algorithm. Let e and f be two arbitrary edges, each with exactly one endpoint in T, and let g be the maximum weight edge on the path from e to f in T. Then g cannot be heavier than both e and f.*

PROOF. Let $\mathcal{P}$ be the path in T connecting e and f, and assume the contrary, that g is the heaviest edge in $\mathcal{P} \cup \{e, f\}$. Now consider the moment when g is selected by DJP and let $\mathcal{P}'$ be the portion of $\mathcal{P}$ present in the tree. There are exactly two edges in $(\mathcal{P} - \mathcal{P}') \cup \{e, f\}$ that are eligible to be chosen by the DJP algorithm at this moment, one of which is the edge g. If the other edge is in $\mathcal{P}$, then by our choice of g it must be lighter than g. If the other edge is either e or f, then by our assumption it must be lighter than g. In both cases, g could not be chosen next by the DJP algorithm, a contradiction. □

2.3. THE DENSE CASE ALGORITHM. The algorithms presented in Fredman and Tarjan [1987], Gabow et al. [1986], Chazelle [1997, 2000b], and Pettie [1999] will find the MSF of a graph in linear time if the graph is sufficiently dense, that is, has a sufficiently large edge-to-vertex ratio. For our purposes, *sufficiently dense* will mean $m/n \geq \log^{(3)} n$. All of the above algorithms run in linear time for that

density, the simplest of which is easily that of Fredman and Tarjan [1987]. This algorithm executes a number of phases, where the purpose of each phase is to amplify the "nominal density" of the graph by contracting a large number of MSF edges; here the *nominal density* is the ratio m/n', where m, as usual, is the number of edges in the original graph, and n' is the number of vertices in the current graph. Each phase of the algorithm runs in $O(m + n)$ time, and works by executing the DJP algorithm many times, each for a limited number of steps. If n' is the number of vertices before a phase, the number of vertices after the phase is no more than $n'/2^{m/n'}$, hence no more than $\log^* n - \log^*(m/n)$ phases are needed.

The procedure DenseCase(G; F) takes as input an n-node graph G and returns the MSF F of G in linear time for graphs with density at least $\log^{(3)} n$.

Our optimal algorithm will actually call DenseCase on a graph derived from an n-node, m-edge graph by contracting vertices so that the number of vertices is reduced by a factor of at least $\log^{(3)} n$. The number of edges in the contracted graph is no more than m. Hence, DenseCase will run in $O(m + n)$ time on such a graph.

2.4. SOFT HEAP. The main data structure used by our algorithm is the *Soft Heap* [Chazelle 2000a]. The Soft Heap is a kind of priority queue that gives us an optimal trade-off between accuracy and speed. It is parameterized by an error tolerance ϵ, and supports the following operations:

—`MakeHeap`(): returns an empty soft heap.
—`Insert`(S, x): insert item x into heap S.
—`Findmin`(S): returns item with smallest key in heap S.
—`Delete`(S, x): delete x from heap S.
—`Meld`(S_1, S_2): create new heap containing the union of items stored in S_1 and S_2, destroying S_1 and S_2 in the process.

All operations take constant amortized time, except for Insert, which takes $O(\log(1/\epsilon))$ time. To save time the Soft Heap allows items to be grouped together and treated as though they have a single key. An item adopts the largest key of any item in its group, *corrupting* the item if its new key differs from its original key. Thus, the original key of an item returned by Findmin (i.e., any item in the group with minimum key) is no more than the keys of all uncorrupted items in the heap. The guarantee is that after n Insert operations, no more than ϵn corrupted items are in the heap. The following result is shown in Chazelle [2000a].

LEMMA 2.2. *Fix any parameter $0 < \epsilon < 1/2$, and beginning with no prior data, consider a mixed sequence of operations that includes n inserts. On a Soft Heap, the amortized complexity of each operation is constant, except for insert, which takes $O(\log(1/\epsilon))$ time. At most ϵn items are corrupted at any given time.*

3. *A Key Lemma and Procedure*

3.1. A ROBUST CONTRACTION LEMMA. It is well known that if T is a tree of MSF edges, we can *contract* T into a single vertex while maintaining the invariant that the MSF of the contracted graph plus T gives the MSF for the graph before contraction.

In our algorithm, we find a tree of MSF edges T in a *corrupted* graph, where some of the edge weights have been increased due to the use of a Soft Heap. In the lemma given below, we show that useful information can be obtained by contracting certain corrupted trees, in particular those constructed using some number of steps from the Dijkstra–Jarnik–Prim (DJP) algorithm. Ideas similar to these are used in Chazelle's [1997] algorithm and more explicitly in the recent algorithm of Chazelle [2000b] (see also Pettie [1999]).

Before stating the lemma, we need some notation and preliminary concepts. Let $V(G)$ and $E(G)$ be the vertex and edge sets of G, and n and m be their cardinality, respectively. Let the G-weight of an edge be its weight in graph G (the G may be omitted if implied from context).

For the following definitions, M and C are subgraphs of G. Denote by $G \Uparrow M$ some graph derived from G by raising the weight of each edge in M by arbitrary amounts (these edges are said to be corrupted). Let M_C be the set of edges in M with exactly one endpoint in C. Let $G\backslash C$ denote the graph obtained by contracting all connected components induced by C, that is, by replacing each connected component with a single vertex and reassigning edge endpoints appropriately.

Definition 3.1. A subgraph C is said to be *DJP*-contractible with respect to G if after executing the *DJP* algorithm on G for some number of steps, with a suitable start vertex in C, the tree that results is a spanning tree for C.

LEMMA 3.2. *Let M be a set of edges in a graph G. If C is a subgraph of G that is DJP-contractible with respect to $G \Uparrow M$, then $MSF(G)$ is a subset of $MSF(C) \cup MSF(G\backslash C - M_C) \cup M_C$.*

PROOF. Each edge in C that is not in MSF(C) is the heaviest edge on some cycle in C. Since that cycle exists in G as well, that edge is not in MSF(G). So we need only show that edges in $G\backslash C$ that are not in MSF($G\backslash C - M_C$) $\cup M_C$ are also not in MSF(G).

Let $H = G\backslash C - M_C$; hence, we need to show that no edge in $H - MSF(H)$ is in $MSF(G)$. Let e be in $H - MSF(H)$, that is, e is the heaviest edge on some cycle χ in H. If χ does not involve the vertex derived by contracting C, then it exists in G as well and $e \notin MSF(G)$. Otherwise, χ forms a *path* $\mathcal{P}$ in G whose endpoints, say x and y, are both in C. Let the end edges of $\mathcal{P}$ be (x, w) and (y, z). Since H includes no corrupted edges with one endpoint in C, the G-weight of these edges is the same as their $(G \Uparrow M)$-weight.

Let T be the spanning tree of $C \Uparrow M$ derived by the DJP algorithm, $\mathcal{Q}$ be the path in T connecting x and y, and g be the heaviest edge in $\mathcal{Q}$. Notice that $\mathcal{P} \cup \mathcal{Q}$ forms a cycle. By our choice of e, it must be heavier than both (x, w) and (y, z), and by Lemma 2.1, the heavier of (x, w) and (y, z) is heavier than the $(G \Uparrow M)$-weight of g, which is an upper bound on the G-weights of all edges in $\mathcal{Q}$. So with respect to G-weights, e is the heaviest edge on the cycle $\mathcal{P} \cup \mathcal{Q}$ and cannot be in $MSF(G)$. □

3.2. THE PARTITION PROCEDURE. Our algorithm uses the Partition procedure that is given below. This procedure finds DJP-contractible subgraphs $C_1, \ldots, C_k$ in which edges are progressively being corrupted by the Soft Heap. Let M_{C_i} contain only those corrupted edges with one endpoint in C_i at the time it is completed.

```
Partition(G, maxsize, ϵ ; M_C, C)
    All vertices are initially ``live''
    M_C := ∅
    i := 0
    While there is a live vertex
        Increment i
        Let V_i := {v}, where v is any live vertex
        Create a Soft Heap consisting of v's edges (uses ϵ)
        While all vertices in V_i are live and |V_i| < maxsize
            Repeat
                Find and delete min-weight edge (x, y) from Soft Heap
                W.l.o.g, assume x ∈ V_i
            Until y ∉ V_i
            V_i := V_i ∪ {y}
            If y is live then insert each of y's edges into the Soft Heap
        Set all vertices in V_i to be dead
        Let M_{V_i} be the corrupted edges with one endpoint in V_i
        M_C := M_C ∪ M_{V_i}
        G := G − M_{V_i}
        Dismantle the Soft Heap
    Let C := {C_1, ..., C_i} where C_z is the subgraph of G induced by V_z
    Exit.
```

FIG. 1. The Partition procedure.

Each subgraph C_i will be DJP-contractible with respect to a graph derived from G by several rounds of contractions and edge deletions. When C_i is finished, it is contracted and all incident corrupted edges are discarded. By applying Lemma 3.2 repeatedly, we see that after C_i is built, the MSF of G is a subset of

$$\bigcup_{j=1}^{i} MSF(C_j) \cup MSF\left(G \setminus \bigcup_{j=1}^{i} C_j - \bigcup_{j=1}^{i} M_{C_j}\right) \cup \bigcup_{j=1}^{i} M_{C_j}.$$

The Partition procedure is shown in Figure 1. The arguments appearing before the semicolon are inputs; the others are outputs. $\mathcal{C} = \{C_1, \ldots, C_k\}$ is a set of subgraphs of G, and M_C is a set of corrupted edges with endpoints in different C_i's. No edge will appear in more than one of $M_C, C_1, \ldots, C_k$.

Initially, Partition sets every vertex to be *live*. The objective is to convert each vertex to *dead*, signifying that it is part of a component C_i with $\leq$ *maxsize* vertices and part of a *conglomerate* of $\geq$ *maxsize* vertices, where a conglomerate is a connected component of the graph $\bigcup E(C_i)$. Intuitively a conglomerate is a collection of C_i's linked by common vertices. This scheme for growing components is similar to the one given in Fredman and Tarjan [1987].

We grow the C_i's one at a time according to the DJP algorithm, except that we use a Soft Heap. A component is done growing if it reaches *maxsize* vertices or if it attaches itself to an existing component. Clearly, if a component does not reach *maxsize* vertices, it has linked to a conglomerate of at least *maxsize* vertices. Hence, all its vertices can be designated dead. Upon completion of a component C_i, we discard the set of corrupted edges with one endpoint in C_i.

The running time of **Partition** is dominated by the heap operations, which depend on ϵ. Each edge is inserted into a Soft Heap no more than twice (once for each endpoint), and extracted no more than once. We can charge the cost of dismantling the heap to the insert operations which created it; hence, the total running time is $O(m \log(1/\epsilon))$. By Lemma 2.2, the number of discarded edges is bounded by the number of insertions scaled by ϵ; thus, $|M_C| \leq 2\epsilon m$. Thus, we have

LEMMA 3.3. *Given a graph G, any $0 < \epsilon < 1/2$, and a parameter maxsize, Partition finds edge-disjoint subgraphs $M_C, C_1, \ldots, C_k$ in time $O(|E(G)| \cdot \log(1/\epsilon))$ while satisfying several conditions:*

(*a*) *For all $v \in V(G)$, there is some i such that $v \in V(C_i)$.*
(*b*) *For all i, $|V(C_i)| \leq$ maxsize.*
(*c*) *For each conglomerate $P \in \bigcup_i C_i$, $|V(P)| \geq$ maxsize.*
(*d*) $|E(M_C)| \leq 2\epsilon \cdot |E(G)|$.
(*e*) $MSF(G) \subseteq \bigcup_i MSF(C_i) \cup MSF(G \backslash (\bigcup_i C_i) - M_C) \cup M_C$.

We observe that the full suite of soft heap operations is not needed as we never employ the *meld* operation. We can therefore use a more space-efficient version of the soft heap where its nodes are placed in an array and the links between them represented implicitly, as in a binary heap.

4. *Overview of the Optimal Algorithm*

Here is an overview of our optimal MSF algorithm.

—In the first stage, we find DJP-contractible subgraphs $C_1, C_2, \ldots, C_k$ with their associated set of edges $M_C = \bigcup_i M_{C_i}$, where M_{C_i} consists of corrupted edges with one endpoint in C_i.
—In the second stage we find the MSF F_i of each C_i, and the MSF F_0 of the contracted graph $G \backslash (\bigcup_i C_i) - M_C$. By Lemma 3.2, the MSF of the whole graph is contained within $F_0 \cup (\bigcup_i F_i) \cup M_C$. Note that, at this point, we have not identified any edges as being in the MSF of the original graph G.
—In the third stage, we find some MSF edges, via Borůvka steps, and recurse on the graph derived by contracting these edges.

We execute the first stage using the Partition procedure described in the previous section.

We execute the second stage with *optimal decision trees*. Essentially, these are hardwired algorithms designed to compute the MSF of a graph using an optimal number of edge-weight comparisons. In general, decision trees are much larger than the size of the problem that they solve and finding optimal ones is very time consuming. We can afford the cost of building decision trees by guaranteeing that each one is extremely small. At the same time, we make each conglomerate formed by the C_i to be sufficiently large so that the MSF F_0 of the contracted graph can be found in linear time using the DenseCase algorithm.

Finally, in the third stage, we have a reduction in vertices due to the Borůvka steps, and a reduction in edges due to the application of Lemma 3.2. In our optimal algorithm, both vertices and edges reduce by a constant factor, thus resulting in the recursive applications of the algorithm on graphs with geometrically decreasing sizes.

4.1. DECISION TREES. Consider a computation that takes as input a fixed graph G and computes the minimum spanning tree for G for any given permutation of edge weights. If we are only interested in the edge-weight comparisons performed, this computation can be described in terms of an *MSF decision tree* for G. An MSF decision tree is a rooted tree having an edge-weight comparison associated with each internal node (e.g., $weight(x, y) < weight(w, z)$). Each internal node has exactly two children, one representing that the comparison is true, the other that it is false. The leaves of the tree list off the edges in some spanning tree of the graph. An MSF decision tree for G is said to be *correct* if the edge-weight comparisons encountered on any path from the root to a leaf uniquely identify the spanning tree at that leaf as the MSF. A decision tree for G is said to be *optimal* if it is correct and there exists no correct decision tree for G with lesser depth.

Let us bound the time needed to find optimal decision trees for all graphs on r vertices by brute force search. There are $2^{\binom{r}{2}}$ such graphs and for each graph we must check all possible decision trees bounded by a sufficient depth. Since the DJP algorithm uses no more than $r(r-1)$ comparisons on any graph on r vertices, a depth of r^2 is sufficient. Hence, the tree has fewer than 2^{r^2} internal nodes. There are $<r^4$ possibilities for each internal node (its comparison must identify two edges); hence, there are $<r^{2^{r^2+2}}$ distinct decision trees to check. To determine if a decision tree is correct, we generate all possible permutations of the edge weights and for each, solve the MSF problem on the given graph. Now we simultaneously check all permutations against a decision tree as follows: First, we place all permutations at the root; then move them to the left or right child depending on the truth or falsity of the edge-weight comparison with respect to each permutation. We repeat this step until all permutations reach a leaf. If for each leaf, all permutations sharing that leaf agree on the MSF, then the decision tree is correct. This process takes no longer than $(r^2+1)!$ for each decision tree, hence the total time required to find an optimal decision tree for all graphs on fewer than r vertices is bounded by $2^{r^2} \cdot r^{2^{r^2+2}} \cdot (r^2+1)!$, which is less than $2^{2^{r^2+o(r)}}$. Setting $r = \log^{(3)} n$ allows us to precompute all optimal decision trees in $o(n)$ time.[1]

Observe that, in the high-level algorithm we gave in Section 4, if the maximum size of each component C_i is sufficiently small, the components can be organized into a relatively small number of groups of isomorphic components (ignoring edge weights). For each group, we use a single precomputed optimal decision tree to determine the MSF of components in that group.

In our optimal algorithm, we use a procedure DecisionTree($\mathcal{G}; \mathcal{F}$), which takes as input a collection of graphs $\mathcal{G}$, each with at most r vertices, and returns their minimum spanning forests in $\mathcal{F}$ using the precomputed decision trees.

5. *The Algorithm*

As discussed above, the optimal MSF algorithm is as follows: First, precompute the optimal decision trees for all graphs with $\leq \log^{(3)} n$ vertices. Next, divide the input graph into subgraphs $C_1, C_2, \ldots, C_k$, discarding the set of corrupted edges M_{C_i} as

[1] We can set r as high as $\sqrt{\log \log n} - 1$; however, this provides no benefit to our algorithm.

each C_i is completed. Use the decision trees found earlier to compute the MSF F_i of each C_i, then contract each connected component spanned by $F_1 \cup \cdots \cup F_k$ (i.e., each conglomerate) into a single vertex. The resulting graph has $\leq n/\log^{(3)} n$ vertices since each conglomerate has at least $\log^{(3)} n$ vertices by Lemma 3.3. Hence, we can use the DenseCase algorithm to compute its MSF F_0 in time linear in m. At this point, by Lemma 3.2 the MSF is now contained in the edge set $F_0 \cup \cdots \cup F_k \cup M_{C_1} \cup \cdots \cup M_{C_k}$. On this graph, we apply two Borůvka steps, reducing the number of vertices by a factor of four, and then compute recursively. The algorithm is given below.

Let $\epsilon = 1/8$ (this is used by the Soft Heap in the Partition procedure).

Precompute optimal decision trees for all graphs with $\leq \log^{(3)} n_0$ vertices, where n_0 is the number of vertices in the original input graph.

OptimalMSF(G)
  `If` $E(G) = \emptyset$ `then Return`$(\emptyset)$
  $r := \lceil \log^{(3)} |V(G)| \rceil$
  `Partition`$(G, r, \epsilon;\ M, \mathcal{C})$
  `DecisionTree`$(\mathcal{C};\ \mathcal{F})$
  `Let` $k := |\mathcal{C}|$ `and let` $\mathcal{C} = \{C_1, \ldots, C_k\}$, $\mathcal{F} = \{F_1, \ldots, F_k\}$
  $G_a := G \backslash (F_1 \cup \cdots \cup F_k) - M$
  `DenseCase`$(G_a;\ F_0)$
  $G_b := F_0 \cup F_1 \cup \cdots \cup F_k \cup M$
  `Boruvka2`$(G_b;\ F', G_c)$
  $F :=$ `OptimalMSF`(G_c)
  `Return`$(F \cup F')$

Apart from recursive calls and using the decision trees, the computation performed by OptimalMSF is clearly linear since Partition takes $O(m \log(1/\epsilon))$ time, and owing to the reduction in vertices, the call to DenseCase also takes linear time. For $\epsilon = 1/8$, the number of edges passed to the final recursive call is $\leq m/4 + n/4 \leq m/2$, giving a geometric reduction in the number of edges. Since no MSF algorithm can do better than linear time, the bottleneck, if any, must lie in using the decision trees, which are optimal by construction.

More concretely, let $T(m, n)$ be the running time of OptimalMSF. Let $\mathcal{T}^*(m, n)$ be the optimal number of comparisons needed to determine the MSF on any graph with n vertices and m edges and let $\mathcal{T}^*(H)$ be the optimal number of comparisons needed on a *specific* graph H. That is, $\mathcal{T}^*(m, n) = \max\{\mathcal{T}^*(H) : |V(H)| = n,\ |E(H)| = m\}$. We also refer to $\mathcal{T}^*$ as the *decision-tree complexity* of MSF, as it corresponds to the height of an optimal decision-tree. The recurrence relation for T is given below. For the base case, note that the graphs in the recursive calls will be connected if the input graph is connected. Hence, the base case graph has no edges and one vertex, and we have $T(0, 1)$ equal to a constant

$$T(m, n) \leq \sum_i c_1 \mathcal{T}^*(C_i) + T\left(\frac{m}{2}, \frac{n}{4}\right) + c_2 \cdot m.$$

It is straightforward to see that, if $\mathcal{T}^*(m, n) = O(m)$, then the above recurrence gives $T(m, n) = O(m)$. One can also show that $T(m, n) = O(\mathcal{T}^*(m, n))$ for many natural functions for $\mathcal{T}^*$ (including $m \cdot \alpha(m, n)$). However, to show that this result

holds no matter what the function describing $\mathcal{T}^*(m, n)$ is, we need to establish some results on the decision tree complexity of the MSF problem, which we do in the next section.

5.1. SOME RESULTS FOR MSF DECISION TREES. In this section, we establish some results on MSF decision trees that allow us to establish our main result that OptimalMSF runs in $O(\mathcal{T}^*(m, n))$ time.

PROPOSITION 5.1. *For* $m, n > 2$, $\mathcal{T}^*(m, n) \geq m/2$.

PROPOSITION 5.2. *For* $n' > n$, $\mathcal{T}^*(m, n') \geq \mathcal{T}^*(m, n)$, *and for* $m' > m$, $\mathcal{T}^*(m', n) \geq \mathcal{T}^*(m, n)$.

Proposition 5.1 is true since for $m, n > 2$ every edge can be placed on some cycle, and must therefore participate in at least one comparison. Proposition 5.2 holds since we can always add isolated vertices or edges of very high weight, neither of which affects the MSF.

We now state a property that is used by Lemmas 5.4 and 5.5.

PROPERTY 5.3. *Let H be a graph which is the union of edge-disjoint subgraphs $C_1, \ldots, C_k$. The structure of H dictates that $MSF(H) = MSF(C_1) \cup \cdots \cup MSF(C_k)$.*

If $C_1, \ldots, C_k$ are the components returned by Partition, it can be seen that the graph $H = \bigcup_i C_i$ satisfies Definition 5.3 since every simple cycle in this graph must be contained in exactly one of the C_i. To see this, consider any simple cycle and let i be the largest index such that C_i contains an edge in the cycle. Since each C_i shares no more than one vertex with $\bigcup_{j<i} C_j$, this cycle cannot contain an edge from $\bigcup_{j<i} C_j$.

LEMMA 5.4. *If Property 5.3 holds for H, then there exists an optimal MSF decision tree for H that makes no comparisons of the form $e < f$ where $e \in C_i$, $f \in C_j$ and $i \neq j$.*

PROOF. Consider a subset $\mathcal{P}$ of the permutations of all edge weights where for $e \in C_i$, $f \in C_j$ and $i < j$, it holds that $weight(e) < weight(f)$. Permutations in $\mathcal{P}$ have two useful attributes that can be readily verified. First, any number of intercomponent comparisons shed no light on the relative weights of edges in the same component. Second, any spanning forest of a component is the MSF of that component for some permutation in $\mathcal{P}$.

Now consider any optimal decision tree T for H. Let T' be the subtree of T that contains only leaves that can be reached by some permutation in $\mathcal{P}$. Each intercomponent comparison node in T' must have only one child, and by the first attribute, the MSF at each leaf was deduced using only intracomponent comparisons. By the second attribute, T' must determine the MSF of each component correctly, and thus by Property 5.3 it must determine the MSF of the graph H correctly. Hence, we can contract T' into a correct decision tree T'' by replacing each one-child node with its only child. □

LEMMA 5.5. *If Property 5.3 holds for H, then $\mathcal{T}^*(H) = \sum_i \mathcal{T}^*(C_i)$.*

PROOF. Given optimal decision trees T_i for the C_i we can construct a decision tree for G by replacing each leaf of T_1 by T_2, and in general replacing each leaf of T_i by T_{i+1} and by labeling each leaf of the last tree by the union of the labels of the original trees along this path. Clearly, the height of this tree is the sum of the heights of the T_i, and hence $\mathcal{T}^*(G) \leq \sum_i \mathcal{T}^*(C_i)$. So we need only prove that no optimal decision tree for G has height less than the sum of the heights of the T_i.

Let T be an optimal decision tree for G that has no intercomponent comparisons (as guaranteed by Lemma 5.4). We show that T can be transformed into a "canonical" decision tree T' for G of the same height as T, such that in T', all comparisons for C_i precede all comparisons for C_{i+1}, for each i, and further, the subtrees of T' containing intra-C_i comparisons are all identical. That is, they have the same shape and the same comparisons are associated with corresponding nodes. This establishes the desired result since T' must contain a path that is the concatenation of the longest path in an optimal decision tree for each of the C_i.

We first prove this result for the case when there are only two components, C_1 and C_2. Assume inductively that the subtrees rooted at all vertices at a certain depth d in T have been transformed to the desired structure of having the C_1 comparisons occur before he C_2 comparisons, and with all subtrees for C_2 within each of the subtrees rooted at depth d being identical. (This is trivially the case when d is equal to the height of T.)

Consider any node v at depth $d - 1$. If the comparison at that node is a C_1 comparison, then all C_2 subtrees at descendent nodes must compute the same set of leaves for C_2. Hence, the subtree rooted at v can be converted to the desired format simply by replacing all C_2 subtrees by one having minimum depth (note that there are at most two different C_2 subtrees: all those descending from v's left child (right child) are identical). If the comparison at v is a C_2 comparison, we know that the C_1 subtrees rooted at its left child x and its right child y must both compute the same set of leaves for C_1. Hence, we pick the C_1 subtree of smaller height (without loss of generality, let its root be x) and replace v by x, together with the C_1 subtree rooted at x. We then copy the comparison at node v to each leaf position of this C_1 subtree. For each such copy, we place one of the isomorphic copies of the C_2 subtree that is a descendant of x as its left subtree, and the C_2 subtree that is a descendant of y as its right subtree. The subtree rooted at x, which is now at depth $d - 1$, is now in the desired form; it computes the same result as in T, and there was no increase in the height of the tree. Hence, by induction, T can be converted into canonical decision tree of no greater height.

Assume inductively that the result hold for up to $k - 1 \geq 2$ components. The result easily extends to k components by noting that we can group the first $k - 1$ components as C'_1 and let C_k be C'_2. By the above method, we can transform T to a canonical tree in which the C_k comparisons appear as leaf subtrees. We now strip the C_k subtrees from this canonical tree and then, by the inductive assumption, we can perform the transformation for remaining $k - 1$ components. □

COROLLARY 5.6. *Let the C_i be the components formed by the Partition routine applied to graph G, let $H = \bigcup_i C_i$ and let G have m edges and n vertices. Then, $\sum_i \mathcal{T}^*(C_i) = \mathcal{T}^*(H) \leq \mathcal{T}^*(m, n)$.*

COROLLARY 5.7. *For any m and n, $2 \cdot \mathcal{T}^*(m, n) \leq \mathcal{T}^*(2m, 2n)$.*

We can now solve the recurrence relation for the running time of OptimalMSF given in the previous section.

$$\begin{aligned} T(m,n) &\le \sum_i c_1 \mathcal{T}^*(C_i) + T\left(\frac{m}{2}, \frac{n}{4}\right) + c_2 \cdot m \\ &\le c_1 \mathcal{T}^*(m,n) + T\left(\frac{m}{2}, \frac{n}{4}\right) + c_2 \cdot m \quad (\textit{Corollary } 5.6) \\ &\le c_1 \mathcal{T}^*(m,n) + c \cdot \mathcal{T}^*\left(\frac{m}{2}, \frac{n}{4}\right) + c_2 \cdot m \quad (\textit{assume inductively}) \\ &\le \mathcal{T}^*(m,n)\left(c_1 + \frac{c}{2} + 2c_2\right) \quad (\textit{Corollary } 5.7 \textit{ and Propositions } 5.1, 5.2) \\ &\le c \cdot \mathcal{T}^*(m,n) \ (\textit{for } c = 2c_1 + 4c_2; \textit{ this completes the induction}). \end{aligned}$$

This gives us the desired theorem.

THEOREM 5.8. *Let $\mathcal{T}^*(m,n)$ be the decision-tree complexity of the MSF problem on graphs with m edges and n nodes. Algorithm* `OptimalMSF` *computes the MSF of a graph with m edges and n vertices deterministically in $O(\mathcal{T}^*(m,n))$ time.*

6. *Avoiding Pointer Arithmetic*

We have not precisely specified what is required of the underlying machine model. Upon examination, the algorithm does not seem to require the full power of a random access machine (RAM). No bit manipulation is used and arithmetic can be limited to just the increment operation. However, if procedure DecisionTree is implemented in the obvious manner, it will require using a table lookup, and thus random access to memory. In this section, we outline the pointer machine [Tarjan 1979a] a model that does not allow random access to memory, and describe some techniques we use for implementing the DecisionTree procedure on a pointer machine. Our method is similar to that described in Buchsbaum et al. [1998], but we ensure that the time overhead in performing the table look-ups during a call to DecisionTree is linear in the size of the *current* input to DecisionTree.

A pointer machine distinguishes pointers from all other data types. The only operations allowed on pointers are assignment, comparison for equality and dereferencing. Memory is organized into records, each of which holds some constant number of pointers and normal data words (integers, floats, etc.). Given a pointer to a particular record, we can refer to any pointer or data word in that record in constant time. On nonpointer data, the usual array of logical, arithmetic, and binary comparison operations are allowed.

Every MSF decision tree solves the MSF problem on a particular graph topology: we call this the *generic graph*. In order to solve the MSF problem on an *actual* graph, we *bind* corresponding edges of the actual and generic graphs, such that given one edge the other can be found in constant time. Comparisons in the MSF decision tree refer to edges in the generic graph; hence, they too can be translated into comparisons in the actual graph in constant time.

We match up identical actual graphs and generic graphs using the method given in Buchsbaum et al. [1998]. If all graphs have fewer than r vertices we represent the graphs as a string of numbers between 1 and r, then perform a lexicographic sort [Aho et al. 1974] on all the graphs (both actual and generic).

A generic graph will appear adjacent to all identical actual graphs. All that remains to be done is bind the actual graphs to the appropriate generic graph and run the associated MSF decision tree. If the total size of all actual subgraphs is s ($s \leq m$), the sorting step takes $O(s + r^2 2^{r^2})$ time, which is $O(m + n)$ for $r = \log^{(3)} n$. The lexicographic sort guarantees that in the recursive calls it suffices to scan an initial prefix of the sorted list whose size is linear in the size of the current graph.

7. *Performance on Random Graphs*

Even if we assume that MST has some super-linear complexity, we show below that our algorithm runs in linear time for nearly all graphs, for arbitrarily chosen edge weights. This improves upon the expected linear-time result of Karp and Tarjan [1980], which depended on the edge weights being chosen randomly. Our result may also be compared with the randomized algorithm of Karger et al. [1995], which is shown to run in $O(m)$ time with high probability. However, for any given graph the Karger et al. [1995] algorithm can be made to run in $\Omega(m \log n)$ time by an adversary that controls the edge weights. In contrast, we show below that our algorithm runs in linear time for the vast majority of graphs, for every possible assignment of edge weights.

None of the earlier published MST algorithms appear to have this property of running in linear time with high probability on random graphs for all edge-weights. Using the analysis of this section and suitably souped-up versions of earlier algorithms [Fredman and Tarjan 1987; Gabow et al. 1986; Chazelle 2000b], we may obtain similar high probability results.

Our analysis hinges on the observation that for sparse random graphs, with high probability any subgraph constructed by the Partition routine has only a miniscule number in edges in excess of the number of spanning forest edges in that subgraph. The MST of such graphs can be computed in linear time, and hence the computation on optimal decision trees takes linear time on these graphs.

Throughout this section, α will denote $\alpha(m, n)$.

The random graph model $G_{n,m}$ [Erdös and Rényi 1961] assigns all $\binom{\binom{n}{2}}{m}$ graphs with m edges equal probability. In $G_{n,p}$ any graph on m edges is assigned probability $p^m(1-p)^{\binom{n}{2}-m}$. In other words, each possible edge is included independently with probability p.

THEOREM 7.1. *The MST of a graph can be found in linear time with probability*
(1) $1 - exp(-\Omega(m/\alpha^2))$, *for a graph drawn from* $G_{n,m}$
(2) $1 - exp(-\Omega(pn^2/\alpha^2))$, *for a graph drawn from* $G_{n,p}$.
Both (1) *and* (2) *hold regardless of the permutation of edge weights.*

In the next section, we describe the *edge-addition martingale* for the $G_{n,m}$ model. In Section 7.2, we use this martingale and Azuma's inequality to prove part (1) of Theorem 7.1. Part (2) is shown to follow from part (1).

7.1. THE EDGE-ADDITION MARTINGALE.

It was observed Erdös and Rényi [1961] that a random graph from the $G_{n,m}$ model can be generated in an incremental fashion as follows: We begin with n labeled vertices, adding one random edge at a time that was not previously selected. Let X_i be a random edge such that $X_i \neq X_j$ for $j < i$, and $G_i = \{X_1, \ldots, X_i\}$ be the graph made up of the first i edges, with G_0 being the graph on n vertices having no edges.

A *martingale* is a sequence of random variables $Y_0, \ldots, Y_m$ such that $\mathrm{E}[Y_i \mid Y_{i-1}] = Y_{i-1}$ for $0 < i \leq m$. We now prove that if g is any graph-theoretic function and $g_E(G_i) = \mathrm{E}[g(G_m)|G_i]$, then $g_E(G_i)$, for $0 \leq i \leq m$ is a martingale.

LEMMA 7.2. *The sequence* $g_E(G_i) = E[g(G_m) \mid G_i]$, *for* $0 \leq i \leq m$, *is a martingale, where g is any graph theoretic function,* G_0 *is the edge-free graph on n vertices, and* G_i *is derived from* G_{i-1} *by adding a random edge not in* G_{i-1} *to* G_{i-1}.

PROOF. Let $X_i^j = \{X_i, \ldots, X_j\}$. Given that G_{i-1} has been fixed,

$$\begin{aligned}
\mathrm{E}[g_E(G_i)] &= \sum_{X_i = x_i} \Pr[X_i = x_i \mid G_{i-1}] \\
&\quad \cdot \sum_{X_{i+1}^m = x_{i+1}^m} \Pr\left[X_{i+1}^m = x_{i+1}^m \middle| G_{i-1}, X_i = x_i\right] \cdot g\left(G_{i-1} \cup x_i^m\right) \\
&= \sum_{X_i^m = x_i^m} \Pr\left[X_i^m = x_i^m \middle| G_{i-1}\right] \cdot g\left(G_{i-1} \cup x_i^m\right) \\
&= \mathrm{E}[g(G_m) \mid G_{i-1}] = g_E(G_{i-1}). \qquad \square
\end{aligned}$$

We call the sequence proved to be a martingale in Lemma 7.2 the *edge-addition* martingale in contrast to the *edge-exposure* martingale for $G_{n,p}$.

We now recall the well-known Azuma's inequality (see, e.g., Alon and Spencer [1992]).

THEOREM 7.3 (AZUMA'S INEQUALITY). *Let* $Y_0, \ldots, Y_m$ *be a martingale with* $|Y_i - Y_{i-1}| \leq 1$ *for* $0 < i \leq m$. *Let* $\lambda > 0$ *be arbitrary. Then* $\Pr[|Y_m - Y_0| > \lambda\sqrt{m}] < exp(-\lambda^2/2)$.

To facilitate the application of Azuma's inequality to our edge-addition martingale, we establish the following lemma:

LEMMA 7.4. *Consider the sequence proved to be a martingale in Lemma* 7.2. *Let g be any graph-theoretic function such that* $|g(G) - g(G')| \leq 1$ *for any pair of graphs G and G′ of the form* $G = H \cup \{e\}$ *and* $G' = H \cup \{e'\}$, *for some graph H. Then* $|g_E(G_i) - g_E(G_{i-1})| \leq 1$, *for* $0 < i \leq m$.

PROOF. $g_E(G_i)$ and $g_E(G_{i-1})$ are the average of $g(G_i \cup X_{i+1}^m)$ and $g(G_{i-1} \cup X_i^m)$ where X_{i+1}^m and X_i^m range over their possible outcomes, given G_i and G_{i-1}, respectively. We identify each outcome of X_{i+1}^m with equal-size disjoint sets of outcomes of X_i^m that cover all outcomes of X_i^m. Then $g_E(G_{i-1})$ may be regarded as an average of set averages. If, for each set corresponding to an outcome P of X_{i+1}^m, we establish that the set average differs from $g(G_i \cup P)$ by no more than 1, the Lemma follows.

The correspondence is as follows: Let $G_i = G_{i-1} \cup \{a\}$ (i.e., $X_i = a$). For each outcome x_{i+1}^m, the corresponding set consists of outcomes of the form $x_{i+1}^j \, a \, x_{j+1}^m$ for $i < j \leq m$ (i.e., the same graph but a appears at different times), and of the form $x_i \, x_{i+1}^m$ where x_i ranges over all edges not appearing in G_{i-1} and x_{i+1}^m. For each outcome $P = x_{i+1}^m$ of X_{i+1}^m and all Q in P's associated set, $|g(G_i \cup P) - g(G_{i-1} \cup Q)| \leq 1$ since the graphs differ in at most one edge. Clearly, $|g(G_i \cup P) - \mathrm{AVG}_Q\{g(G_{i-1} \cup Q)\}| \leq 1$ holds as well, where the average is over outcomes Q in P's associated set. $\square$

7.2. ANALYSIS. We define the *excess* of a subgraph H to be $|E(H)| - |F(H)|$, where $F(H)$ is any spanning forest of H. *Let $f(G)$ be the maximum excess of the graph made up of intracomponent edges, where the sets of components range over all possible sets returned by the Partition procedure.* (Recall that the size of any component is no more than $k = maxsize = \log^{(3)} n$.)

Define $f_E(G_i) = \mathrm{E}[f(G_m)|G_i]$.

The key observation leading to our linear-time result is that each pass of our optimal algorithm definitely runs in linear time if $f(G) \leq m/\alpha(m,n)$. To see this, note that if this bound on $f(G)$ holds, we can reduce the *total* number of intracomponent edges to $\leq 2m/\alpha$ in linear time using $\log \alpha$ Borůvka steps, and then, clearly, the MST of the resulting graph can be determined in $O(m)$ time. We show below that if a graph is randomly chosen from $G_{n,m}$, $f(G) \leq m/\alpha(m,n)$ with high probability.

We now show that Lemma 7.4 applies to the graph-theoretic function f, and then apply Azuma's inequality to obtain our desired result.

LEMMA 7.5. *Let $G = H \cup \{e\}$ and $G' = H \cup \{e'\}$ be two graphs on a set of labeled vertices that differ by no more than one edge. Then $|f(G) - f(G')| \leq 1$.*

PROOF. Suppose without loss of generality that $f(G) - f(G') > 1$, then we could apply the optimal set of components of G to G'. Every intracomponent edge of G remains an intracomponent edge, except possibly e. This can reduce the excess by no more than one, a contradiction. The possibility that e' may become an intracomponent edge can only help the argument. □

LEMMA 7.6. $f_E(G_0) = o(m/\alpha)$.

PROOF. Notice that if $m/n \geq \alpha k$, it is simply impossible to have m/α intracomponent edges, so we assume $m/n < \alpha k$.

An upper bound on $f_E(G_0)$ is the expected number of indices i such that edge X_i completed a cycle of length $\leq k$ in G_{i-1}, since all edges which caused f to increase must have satisfied this criterion. Let p_i be the probability that X_i completed a cycle of length $\leq k$. By bounding the number of such cycles, and the probability they exist in the graph, we have

$$\begin{aligned} p_i &< \sum_{j=3}^{k} n^{j-2} \left(\prod_{\ell=1}^{j-1} \frac{i-\ell}{\binom{n}{2} - (\ell-1)} \right) \\ &< \frac{1}{n} \sum_{j=3}^{k} \left(\frac{nm}{\binom{n}{2}} \right)^{j-1} \quad \text{(recall that } i \leq m\text{)} \\ &= O\left(k \frac{m^{k-1}}{n^k} \right) \qquad \text{if } m = \Omega(n) \\ or &= O\left(\frac{m^2}{n^3} \right) \qquad \text{if } m = o(n) \end{aligned}$$

In either case, $f_E(G_0) \leq \sum_i p_i = o(m/\alpha)$. □

LEMMA 7.7. *Let G be chosen from $G_{n,m}$. Then* $\Pr[f(G) > m/\alpha] < exp(-\Omega(m/\alpha^2))$.

PROOF. By applying Azuma's inequality, we have that $\Pr[|f_E(G_m) - f_E(G_0)| > \lambda\sqrt{m}] < \exp(-\lambda^2/2)$. Setting $\lambda = \sqrt{m}/\alpha - f_E(G_0)/\sqrt{m}$ gives the lemma. Note that, by Lemma 7.6, $f_E(G_0)$ is quite insignificant. □

We are now ready to prove Theorem 7.1.

PROOF. We examine only the first $\log k$ passes of our optimal algorithm, since all remaining passes certainly take $o(m)$ time. Lemma 7.7 assures us that the first pass runs in linear time with high probability. However, the topology of the graph examined in later passes *does* depend on the edge weights. Assuming the Borůvka steps contract all parts of the graph at a constant rate, which can easily be enforced, a partition of the graph in one pass of the algorithm corresponds to a partition of the original graph into components of size less than k^c, for some fixed c. Using k^c in place of k does not affect Lemma 7.6, which gives the Theorem for $G_{n,m}$, that is, part (1). For $G_{n,p}$, note that the probability that there are not $\Theta(pn^2)$ edges is exponential in $-\Omega(pn^2)$; hence, the probability that the algorithm fails to run in linear time is dominated by the bound in part (1). □

For the sparse case where $m < n/\alpha$, Theorem 7.1 part (1) holds with probability 1, and for $p < 1/n\alpha$, by a Chernoff bound, part (2) holds with probability $1 - \exp(-\Omega(n/\alpha))$.

8. *Discussion*

An intriguing aspect of our algorithm is that we do not know its precise deterministic running time although we can prove that it is within a constant factor of optimal. Results of this nature have been obtained in the past for sensitivity analysis of minimum spanning trees [Dixon et al. 1992] and convex matrix searching [Larmore 1990]. Also, for the problem of triangulating a convex polygon, it was observed in Dixon et al. [1992] that an alternate linear-time algorithm could be obtained using optimal decision trees on small subproblems. However, these earlier algorithms make use of decision trees in more straightforward ways than the algorithm presented here.

As noted in Section 4.1, the construction of optimal decision trees takes sublinear time. Thus, it is important to observe that our use of decision trees does not result in a large constant factor in the running time. Further, this construction of optimal decision trees is performed by a straightforward brute-force search; hence, the resulting algorithm is *uniform* (i.e., it is a fixed algorithm that works for all problem sizes).

It was mentioned in the introduction that an optimal algorithm can be constructed for any problem, given an optimal verification algorithm for that problem. We briefly sketch this construction [Jones 1997]. Consider a problem that has an optimal verification algorithm that runs in time $Ver(n)$. The above-mentioned construction produces an algorithm that enumerates programs $P_1, P_2, \ldots$ for some machine model and executes them incrementally as follows: for each $i \geq 1$, for every two operations executed by program P_i, program P_{i+1} executes one operation. Whenever any of the programs halts the verifier checks its output for correctness. The algorithm terminates once the verifier determines that a correct output has been

produced by one of the programs. If P_C is the optimal program for the problem with running time $Opt(n)$, then this construction gives an algorithm that takes no more than $2^{C+1}(Opt(n) + Ver(n))$ steps. Since C is a constant and $Ver(n) = O(Opt(n))$, this gives an algorithm that is within a constant factor of $Opt(n)$.

Using a linear-time MST verification algorithm such as Dixon et al. [1992], King [1997], and Buchsbaum et al. [1998], the above construction yields an optimal MST algorithm; however, it is unsatisfactory for several reasons. It is truly impractical since it asks for an enumeration of all possible algorithms and its constant factor is exponential in the position of the actual optimal algorithm in this enumeration. Further, it sheds no light on the relation between the algorithmic and decision-tree complexity of the problem. Our result, in contrast, has a very reasonable constant factor in the running time, and it is robust in that it ties the algorithmic complexity of MST to its decision-tree complexity, a limiting factor in any machine model. It is not always the case that algorithmic complexity and decision-tree complexity are asymptotically equivalent: for instance, two sorting-type problems whose decision-tree complexity and algorithmic complexity provably diverge are described in Goddard et al. [1993] and [Pettie and Ramachandran [2002a, Sect. 8]. In fact, one can easily concoct simple problems that are NP-hard but nevertheless have polynomial-depth decision-trees.

9. *Conclusion*

We have presented a deterministic MSF algorithm that is provably optimal. The algorithm runs on a pointer machine, and on graphs with n vertices and m edges, its running time is $O(\mathcal{T}^*(m, n))$, where $\mathcal{T}^*(m, n)$ is the decision-tree complexity of the MSF problem on n-node, m-edge graphs. Also, on random graphs our algorithm runs in linear time with high probability for all possible edge-weights. In fact, a hybrid of our algorithm and the randomized algorithm of Karger et al. [1995] runs in expected linear time using only $\log^* n$ random bits [Pettie and Ramachandran 2000b]. Although the exact running time of our algorithm is not known, we have shown that the time bound depends only on the number of edge-weight comparisons needed to determine the MSF, and not on any data structural issues.

Determining the worst-case complexity of our algorithm is the main open question remaining in the MSF problem; however, there is a subtler open question. We have given an optimal uniform algorithm for the MSF problem that uses precomputed decision trees. Is there an optimal uniform algorithm that does *not* use precomputed decision trees (or some similar technique)? More generally, are there problems where precomputation is necessary? One may wish to study this issue in a simpler setting, say the MSF verification problem on a pointer machine. Here, there is still an $\alpha(m, n)$ factor separating the best pointer machine algorithm that uses precomputed decision trees [Buchsbaum et al. 1998] and the one which does not [Tarjan 1979b].

One may also ask for the parallel complexity of the MSF problem. Here, resolved recently were the deterministic time complexity [Chong et al. 2001] and the randomized work-time complexity [Pettie and Ramachandran 1999] of the MSF problem on the EREW PRAM. An open question that remains here is to obtain a deterministic work-time optimal parallel MSF algorithm. Parallelizing our optimal

algorithm is not at all straightforward. Although handling decision trees does not present any problems in the parallel context, we still need a method for identifying contractible components in parallel and a base case algorithm that performs linear work for graph-densities of $\log^{(3)} n$. Existing sequential algorithms that are suitable for the base case, such as the one in Fredman and Tarjan [1987] are also not easily parallelizable.

REFERENCES

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. 1974. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass.

ALON, N., AND SPENCER, J. 1992. *The Probabilistic Method*. Wiley, New York.

BORŮVKA, O. 1926. O jistém problému minimaálním. *Moravské Přírodovědecké Společnosti 3*, 37–58. (In Czech.)

BUCHSBAUM, A. L., KAPLAN, H., ROGERS, A., AND WESTBROOK, J. R. 1998. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In *Proceedings of the ACM Symposium on the Theory of Computing*. ACM, New York, 279–288.

CHAZELLE, B. 1997. A faster deterministic algorithm for minimum spanning trees. In *Proceedings of the IEEE Symposium on Foundations of Computing Science*. IEEE Computer Society Press, Los Alamitos, Calif., 22–31.

CHAZELLE, B. 2000a. The soft heap: An approximate priority queue with optimal error rate. *J. ACM 47*, 6, 1012–1027.

CHAZELLE, B. 2000b. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *J. ACM 47*, 6, 1028–1047.

CHONG, K. W., HAN, Y., AND LAM, T. W. 2001. Concurrent threads and optimal parallel minimum spanning trees algorithm. *J. ACM 48*, 2, 297–323.

DIJKSTRA, E. W. 1959. A note on two problems in connexion with graphs. *Numer. Math. 1*, 269–271.

DIXON, B., RAUCH, M., AND TARJAN, R. E. 1992. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput. 21*, 1184–1192.

ERDŐS, P., AND RÉNYI, A. 1961. On the evolution of random graphs. *Bull. Inst. Internat. Statist. 38*, 343–347.

FREDMAN, M. L., AND TARJAN, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM 34*, 596–615.

FREDMAN, M., AND WILLARD, D. E. 1994. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci. 48*, 3, 533–551.

GABOW, H. N., GALIL, Z., SPENCER, T., AND TARJAN, R. E. 1986. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica 6*, 109–122.

GRAHAM, R. L., AND HELL, P. 1985. On the history of the minimum spanning tree problem. *Ann. Hist. Comput. 7*, 43–57.

GODDARD, W., KENYON, C., KING, V., AND SCHULMAN, L. 1993. Optimal randomized algorithms for local sorting and set-maxima. *SIAM J. Comput. 22*, 2, 272–283.

JARNÍK, V. 1930. O jistém problému minimaálním. *Moravské Přírodovědecké Společnosti 6*, 57–63. (In Czech.)

JONES, N. 1997. *Computability and Complexity: From a Programming Perspective*. MIT Press, Cambridge, Mass.

KARGER, D. R., KLEIN, P. N., AND TARJAN, R. E. 1995. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM 42*, 321–328.

KARP, R. M., AND TARJAN, R. E. 1980. Linear expected-time algorithms for connectivity problems. *J. Algorithms 1*, 4, 374–393.

KING, V. 1997. A simpler minimum spanning tree verification algorithm. *Algorithmica 18*, 2, 263–270.

LARMORE, L. L. 1990. An optimal algorithm with unknown time complexity for convex matrix searching. *Infor. Process. Lett. 36*, 147–151.

PETTIE, S. 1999. Finding minimum spanning trees in $O(m\alpha(m, n))$ time. Tech. Rep. TR99-23. Univ. of Texas at Austin, Austin, Tex.

PETTIE, S., AND RAMACHANDRAN, V. 1999. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. In *Proceedings of RANDOM '99*. Lecture Notes in Computer Science, vol. 1671. Springer-Verlag, New York, 233–244.

PETTIE, S., AND RAMACHANDRAN, V. 2002a. Computing shortest paths with comparisons and additions. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 267–276.

PETTIE, S., AND RAMACHANDRAN, V. 2002b. Minimizing randomness in minimum spanning tree, parallel connectivity, and set maxima algorithms. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*. ACM, New York, 713–722.

PRIM, R. C. 1957. Shortest connection networks and some generalizations. *Bell Syst. Tech. J. 36*, 1389–1401.

TARJAN, R. E. 1979a. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci. 18*, 2, 110–127.

TARJAN, R. E. 1979b. Applications of path compression on balanced trees. *J. ACM 26*, 4, 690–715.

RECEIVED AUGUST 2000; REVISED OCTOBER 2001; ACCEPTED NOVEMBER 2001