An Optimal Minimum Spanning Tree Algorithm*

Seth Pettie and Vijaya Ramachandran
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
seth@cs.utexas.edu, vlr@cs.utexas.edu

May 15, 2001

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 forest 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 super-linear, we show that if the input graph is selected from $G_{n,m}$, our algorithm runs in linear time w.h.p., 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 $G_{n,p}$.

Keywords: 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 [GH85] give an excellent survey of results from the earliest known algorithm of Borůvka [Bor26] to the invention of Fibonacci heaps, which were central to the algorithms in [FT87, GGST86]. Chazelle [Chaz97] presented an MST algorithm based on the Soft Heap [Chaz98] having complexity $O(m\alpha(m,n)\log\alpha(m,n))$, where α is a certain inverse of Ackermann's function. Recently Chazelle [Chaz00] modified the algorithm in [Chaz97] to bring down the running time to $O(m \cdot \alpha(m,n))$. Later, and in independent work, a similar algorithm of the same running time was presented in Pettie [Pet99], 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.

*This is an updated version of UTCS Technical Report TR99-17 which includes performance analysis on random graphs and new references. Part of this work was supported by Texas Advanced Research Program Grant 003658-0029-1999. Seth Pettie was also supported by an MCD Fellowship.

All algorithms mentioned above work on a pointer machine [Tar79] under the restriction that edge weights may only be subjected to binary comparisons. If a more powerful model is assumed, the MST can be computed optimally. Fredman and Willard [FW90] showed that on a unit-cost RAM where the bit-representation of edge weights may be manipulated, the MST can be computed in linear time. Karger et al. [KKT95] presented a *randomized* MST algorithm that runs in linear time with high probability, even if edge weights are only subject to comparisons.

It is still unknown whether these more powerful models are necessary to compute the MST in linear time. However, in this paper 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 density 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. A trivial lower bound on our algorithm is $\Omega(m)$; the best upper bound, $O(m\alpha(m,n))$, is due to Chazelle [Chaz00]. 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 non-uniform algorithm.

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 [Jo97]. 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 section 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 the optimal algorithm and discusses the structure and use of pre-computed 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 w.h.p. if the input graph is selected at random. Sections 8 & 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*. The *minimum spanning forest (MSF)* problem asks for a spanning acyclic subgraph of G having the least total weight. In this paper 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 Boruvka steps

The earliest known MSF algorithm is due to Borůvka [Bor26]. The algorithm is quite simple: It proceeds in a sequence of stages, and in each stage it executes a *Borůvka step* on the graph G, which identifies the set F consisting of the minimum-weight edge incident on each vertex in G, adds these edges to the MSF (since they must be in the MSF by the cut property), and then forms the graph $G_1 = G \backslash F$ as the input to the next stage, where $G \backslash F$ is the graph obtained by contracting each connected component formed by F. This computation can be performed in linear time. Since the number of vertices reduces by at least a factor of two, the running time of this algorithm is $O(m \log n)$, where m and n are the number of vertices and edges in the input graph.

Our optimal algorithm uses a procedure called Boruvka2($G; F, G'$). This procedure executes two Boruvka steps on the input graph G and returns the contracted graph G' as well as the set of edges F identified for 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 [Jar30], re-discovered by Dijkstra [Dij59] and Prim [Prim57]. We will refer to this algorithm as the *DJP* algorithm. Briefly, the DJP algorithm grows a tree T, which initially consists of an arbitrary vertex, one edge at a time, choosing the next edge by the following simple criterion: Augment T with the minimum weight edge (x, y) such that $x \in T$ and $y \notin T$. By the cut property, all edges in 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 $(\mathcal{P} - \mathcal{P}') \cup \{e, f\}$ which 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. $\square$

2.3 The Dense Case Algorithm

The algorithms presented in [FT87, GGST86, Chaz97, Chaz00, Pet99] will find the MSF of a graph in linear time if the graph is sufficiently dense, i.e., has a sufficiently large edge-to-vertex ratio. For our purposes, 'sufficiently dense' will mean $\Omega(\log^{(3)} n)$, where n is the number of vertices in the graph. All of the above algorithms run in linear time for that density.

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 $\Omega(\log^{(3)} n)$.

Our optimal algorithm will 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 $\Omega(\log^{(3)} n)$. The number of edges in the contracted graph is no more than m. It is straightforward to see that 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* [Chaz98]. The Soft Heap is a kind of priority queue that gives us an optimal tradeoff between accuracy and speed. It 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 $O(\log(\frac{1}{\epsilon}))$ 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 [Chaz98].

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 will 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 [Chaz97], and more explicitly in the recent algorithms of Pettie [Pet99] and Chazelle [Chaz00].

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 $G \Uparrow M$ a 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, i.e. by replacing each connected component with a single vertex and reassigning edge endpoints appropriately.

We define a subgraph C of G to be *DJP-contractible* 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 3.1 *Let M be a set of edges in a graph G. If C is a subgraph of G that is DJP-contractible w.r.t. $G \Uparrow M$, 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 the heaviest edge on some cycle χ in H (i.e. $e \in H - MSF(H)$). 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 end points, say x and y, are both in C. Let the end edges of $\mathcal{P}$ be (x, w) and (y, z). Since H included no corrupted edges with one end point 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, y) and (w, z), and by Lemma 2.1, the heavier of (x, y) and (w, 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 w.r.t. G-weights, e is the heaviest edge on the cycle $\mathcal{P} \cup \mathcal{Q}$ and cannot be in $MSF(G)$. $\Box$

3.2 The Partition Procedure

Our algorithm uses the Partition procedure which 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.

Each subgraph C_i will be DJP-contractible w.r.t 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 3.1 repeatedly we see that after C_i is built, the MSF of G is a subset of

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

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

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 [FT87].

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

```
Partition(G, maxsize, ε; M, C)
      All vertices are initially ''live''
      M := ∅
      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
                  Until y ∉ V_i   (Assume w.l.o.g. x ∈ 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 := M ∪ 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.
```

Figure 1: The Partition Procedure.

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 $O(m\log(\frac{1}{\epsilon}))$. The number of discarded edges is bounded by the number of insertions scaled by ϵ, thus $|M| \leq 2\epsilon m$. Thus we have

Lemma 3.2 *Given a graph G, any $0 < \epsilon < \frac{1}{2}$, and a parameter maxsize, Partition finds edge-disjoint subgraphs $M, C_1, \ldots, C_k$ in time $O(|E(G)| \cdot \log(\frac{1}{\epsilon}))$ while satisfying several conditions:*
a) For all $v \in V(G)$ there is some i s.t. $v \in V(C_i)$.
b) For all i, $|V(C_i)| \leq maxsize$.
c) For each conglomerate $P \in \bigcup_i C_i$, $|V(P)| \geq maxsize$.
d) $|E(M)| \leq 2\epsilon \cdot |E(G)|$
e) $MSF(G) \subseteq \bigcup_i MSF(C_i) \cup MSF(G\backslash(\bigcup_i C_i) - M) \cup M$

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 $M = \bigcup_i M_{C_i}$, 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 $G \backslash (\bigcup_i C_i) - \bigcup_i M_{C_i}$. By Lemma 3.1, the MSF of the whole graph is contained within $F_0 \cup \bigcup_i (F_i \cup M_{C_i})$. 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 3.1. 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

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. An MSF decision tree 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 is said to be *optimal* if it is correct and there exists no correct decision tree with lesser depth.

Let us bound the time needed to find all optimal decision trees for graphs of $\leq r$ vertices by brute force search. There are fewer than 2^{r^2} such graphs and for each graph we must check all possible decision trees bounded by a depth of r^2. There are $< r^4$ possibilities for each internal node and $< r^{2^{r^2+O(1)}}$ 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. First put 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 w.r.t to each permutation. 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. Setting $r = \log^{(3)} n$ allows us to precompute all optimal decision trees in $o(n)$ time.

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 will 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 $C_1, C_2, ..., C_k$, discarding the set of corrupted edges M_{C_i} as 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 $F_1 \cup \ldots \cup F_k$ (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 3.2. Hence we can use the DenseCase algorithm to compute its MSF F_0 in time linear in m. At this point, by Lemma 3.1 the MSF is now contained in the edge set $F_0 \cup \ldots \cup F_k \cup M_{C_1} \cup \ldots M_{C_k}$. 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(`$\emptyset$`)`
    $r := \log^{(3)} |V(G)|$
    `Partition(`$G, r, \epsilon;\ M, \mathcal{C}$`)`
    `DecisionTree(`$\mathcal{C}; \mathcal{F}$`)`
    `Let` $k := |\mathcal{C}|$ `and let` $\mathcal{C} = \{C_1, \ldots, C_k\}$, $\mathcal{F} = \{F_1, \ldots, F_k\}$
    $G_a := G \backslash (F_1 \cup \ldots \cup F_k) - M$
    `DenseCase(`$G_a; F_0$`)`
    $G_b := F_0 \cup F_1 \cup \ldots \cup F_k \cup M$
    `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 $O(m \log(\frac{1}{\epsilon}))$ time, and owing to the reduction in vertices, the call to DenseCase also takes linear time. For $\epsilon = \frac{1}{8}$, 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 on any graph with n vertices and m edges and let $\mathcal{T}^*(G)$ be the optimal number of comparisons needed on a *specific* graph G. 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.

$$T(m, n) \quad \leq \quad \sum_i \mathcal{T}^*(C_i) + T(m/2, n/4) + c_1 \cdot m$$

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

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

Proposition 5.1 is obviously true since every edge should participate in a comparison to determine inclusion in or exclusion from the MSF. Proposition 5.2 holds since we can add isolated vertices to a graph, which obviously does not affect the MSF or the number of necessary comparisons.

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

Property 5.3 *The structure of G dictates that MSF(G) = MSF(C_1) $\cup \ldots \cup$ MSF(C_k), where $C_1, \ldots, C_k$ are edge-disjoint subgraphs of G.*

If $C_1, \ldots, C_k$ are the components returned by Partition, it can be seen that the graph $\bigcup_i C_i$ 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 an edge from $\bigcup_{j<i} C_j$. The proof of the following lemma can be found in [PR99b].

Lemma 5.4 *If Property 5.3 holds for G, then there exists an optimal MSF decision tree for G which 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 properties which can be readily verified. First, any number of inter-component 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 G. Let T' be the subtree of T which contains only leaves that can be reached by some permutation in $\mathcal{P}$. Each inter-component comparison node in T' must have only one child, and by the first property, the MSF at each leaf was deduced using only intra-component comparisons. By the second property, T' must determine the MSF of each component correctly, and thus by Property 5.3 it must determine the MSF of the graph G correctly. Hence we can contract T' into a correct decision tree T'' by replacing each one-child node with its only child. $\Box$

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

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 inter-component 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, for each i, the subgraph of T' containing the comparisons within C_i consists of a collection of isomorphic trees. 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 isomorphic. (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 only two different C_2 subtrees – all C_2 subtrees descendent to the left (right) child of v must be isomorphic). 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 (w.l.o.g. 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, and let G have m edges and n vertices. Then, $\sum_i \mathcal{T}^*(C_i) = \mathcal{T}^*(G) \leq \mathcal{T}^*(m,n)$.*

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.

$$\begin{aligned} T(m,n) &\leq \sum_i \mathcal{T}^*(C_i) + T(m/2, n/4) + c_1 \cdot m \\ &\leq \mathcal{T}^*(m,n) + T(m/2,n/4) + c_1 \cdot m \quad \textit{(Corollary 5.6)} \\ &\leq \mathcal{T}^*(m,n) + c \cdot \mathcal{T}^*(m/2,n/4) + c_1 \cdot m \quad \textit{(assume inductively)} \end{aligned}$$

$$
\begin{aligned}
&\leq \mathcal{T}^*(m,n)(1+c/2+2c_1) \quad \textit{(Corollary 5.7 and Propositions 5.1, 5.2)} \\
&\leq c \cdot \mathcal{T}^*(m,n) \ \textit{(for sufficiently large } c\textit{; this completes the induction)}
\end{aligned}
$$

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 describe an alternate method of handling the decision trees which can run on a pointer machine [Tar79], a model which does not allow random access to memory. Our method is similar to that described in [B+98], but we ensure that the time overhead in performing the table lookups 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 non-pointer data, the usual array of logical, arithmetic, and binary comparison operations are allowed.

We first describe the representation of a decision tree. Each decision tree has associated with it a *generic* graph with no edge weights. This decision tree will determine the MST of each permutation of edge weights for this generic graph. At each internal node of the decision tree are four pointers, the first two point to edges in the generic graph being compared and the second two point to the left and right child of the node. Each leaf lists the edges in some spanning tree of the generic graph. Since a decision tree is a pointer-based structure, we can construct each precomputed decision tree (by enumerating and checking all possibilities) without using table lookups.

We now describe our representation of the generic graphs. The vertices of a generic graph are numbered in order by integers starting with 1, and the representation consists of a listing of the vertices in order, starting from 1, followed by the adjacency list for each vertex, starting with vertex 1. Each generic graph will have a pointer to the root of its decision tree.

Recall that we precomputed decision trees for all generic graphs with at most $\log^{(3)} n_0$ vertices (where n_0 is the number of vertices in the input graph whose MSF we need to find). The generic graphs will be generated and stored in lexicographically sorted order. Note that with our representation, in the sorted order the generic graphs will appear in nondecreasing order of the number of vertices in the graph.

Before using a decision tree on an *actual* graph (which must be isomorphic to the generic graph for that decision tree), we must associate each edge in the actual graph with its counterpart in the generic graph. Thus a comparison between edge weights in the generic graph can be substituted by the corresponding weights in the actual graph in constant time.

On a random access machine, we can encode each possible graph in a single machine word (say, as an adjacency matrix), then index the generic graph in an array according to this representation. Thus given a graph we can find the associated decision tree in constant time. On a pointer machine however, converting a bit vector or an integer to a pointer is specifically disallowed.

We now describe our method to identify the generic graph for each C_i efficiently. We assume that each C_i is specified by the adjacency lists representation, and that each edge (x, y) has a pointer to the occurrence of (y, x) in y's adjacency list. Each edge also has a pointer to a record containing its weight. Let m and n be the number of edges and vertices in $\bigcup_i C_i$, and let $r = \log^{(3)} n$.

We rewrite each C_i in the same form as the generic graphs, which we will call the *numerical representation*. Let C_i have p vertices (note that $p \leq r$). We assign the vertices numbers from 1 to p in the order in which they are listed in the adjacency lists representation, and we rewrite each edge as a pair of such numbers indicating its endpoints. Each edge will retain the pointer to its weight, but that is separate from its numerical representation.

We then change the format for each graph as follows: Instead of a list of numbers, each in the range $[1..r]$, we will represent the graph as a list of pointers. For this we initialize a linked list with r buckets, labeled 1 through r. If, in the numerical representation the number j appears, it will be replaced by a pointer to the j^{th} bucket.

We transform a graph into this pointer representation by traversing first the list of vertices and then the list of edges in order, and traversing the list of buckets simultaneously, replacing each vertex entry, and the first vertex entry for each edge by a pointer to the corresponding bucket. Thus edge (x, y), also appearing as (y, x), will now appear as $(ptr(x), y)$ and $(ptr(y), x)$. We then employ the twin pointers to replace the remaining y and x with their equivalent pointers. Clearly this transformation can be performed in $O(m)$ time, where m is the sum of the sizes of all of the C_i.

We will now perform a lexicographic sort [AHU74] on the sequence of C_i's in order to group together isomorphic components. With our representation we can replace each bucket indexing performed by traditional lexicographic sort by an access to the bucket pointer that we have placed for each element. Hence the running time for the pointer-based lexicographic sort is $O(\sum_i \ell_i + Lr)$ where ℓ_i is the length of the i^{th} vector and $L = \max_i\{\ell_i\}$ [AHU74]. Since DecisionTree is called with graphs of size $r = O(\log^{(3)} n)$, we have $L = O(r^2)$ and the sum of the sizes of the graphs is $O(m)$. Hence the radix sort can be performed in $O(m + r^3) = O(m + n)$ time.

Finally, we march through the sorted list of the C_i's and the sorted list of generic graphs, matching them up as appropriate. We will only need to traverse an initial sequence of the sorted generic graphs containing $O(2^{r^2})$ entries in order to match up the graphs. This takes time $O(m + 2^{r^2}) = O(m)$.

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, regardless of edge weights. This improves upon the expected linear-time result of Karp and Tarjan [KT80], which depended on the edge weights being chosen randomly. Our result may also be contrasted with the randomized algorithm of Karger et al. [KKT95], which is shown to run in $O(m)$ time w.h.p. by a proof that depends on the permutation of edge weights and random bits chosen, not the graph topology. In fact, none of the earlier published MST algorithms appear to have this property of running in linear time w.h.p. on random graphs for all edge-weights. Using the analysis of this section and suitably souped-up versions of earlier algorithms [FT87, GGST86, Chaz00], we may obtain the same high probability result.

Our analysis hinges on the observation that for sparse random graphs, w.h.p. any subgraph constructed by the Partition routine has only a miniscule number in edges in excess of the number of spanning forst 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 $\alpha(m, n)$.

Theorem 7.1 *The MST of a graph can be found in linear time with probability*
1) $1 - e^{-\Omega(m/\alpha^2)}$, *for a graph drawn from* $G_{n,m}$
2) $1 - e^{-\Omega(pn^2/\alpha^2)}$, *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

Consider the $G_{n,m}$ random graph model in which each graph with n labeled vertices and m edges is equally likely. For analytical purposes, we select a random graph by beginning with n vertices and adding one edge at a time [ER61]. Let X_i be a random edge s.t. $X_i \neq X_j$ 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 $Y_0, \ldots, Y_m$ s.t. $\mathrm{E}[Y_i \,|\, Y_{i-1}] = Y_{i-1}$ 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* $g_E(G_i) = \mathrm{E}[g(G_m) \,|\, G_i]$, *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,

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

□

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., [AS92]).

Theorem 7.3 *(Azuma'a Inequality.) Let* $Y_0, \cdots, Y_m$ *be a martingale with* $|Y_i - Y_{i-1}| \leq 1$ *for* $0 < i \leq m$. *Let* $\lambda > 0$ *be arbitrary. Then* $\Pr[|Y_m - Y_0| > \lambda\sqrt{m}] < e^{-\lambda^2/2}$.

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 which 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 $X_i = a$. For each outcome x_{i+1}^m, the corresponding set consists of outcomes $x_{i+1}^j \, a \, x_{j+1}^m$ for $i < j \leq m$, and $x_i \, x_{i+1}^m$ 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. □

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 intra-component 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 intra-component 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 which differ by no more than one edge. Then $|f(G) - f(G')| \leq 1$.*

Proof: Suppose w.l.o.g. that $f(G) - f(G') > 1$, then we could apply the optimal set of components of G to G'. Every intra-component edge of G remains an intra-component edge, except possibly e. This can reduce the excess by no more than one, a contradiction. The possibility that e' may become an intra-component edge can only help the argument. □

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

Proof: Notice that if $\frac{m}{n} \geq \alpha k$, it is simply impossible to have m/α intra-component edges, so we assume $\frac{m}{n} < \alpha k$.

An upper bound on $f_E(G_0)$ is the expected number of indices i s.t. 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

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

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* $\Pr[f(G) > m/\alpha] < e^{-\Omega(m/\alpha^2)}$.

Proof: By applying Azuma's inequality, we have that $\Pr[|f_E(G_m) - f_E(G_0)| > \lambda\sqrt{m}] < e^{-\lambda^2/2}$. 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 w.h.p. 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 $1 - e^{-\Omega(n/\alpha)}$.

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 [DRT92] and convex matrix searching [Lar90]. Also, for the problem of triangulating a convex polygon, it was observed in [DRT92] 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 sub-linear 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*.

It was mentioned in the introduction that an optimal algorithm can be constructed for any problem, given an optimal verification algorithm for that problem [Jo97]. This construction produces an algorithm which enumerates programs (for some machine model) and executes them incrementally. Whenever one of the programs halts the verifier checks its output for correctness. Using a linear-time MST verification algorithm such as [DRT92, K97, B+98], this construction yields an optimal MST algorithm, however it is unsatisfactory for several reasons. Aside from truly astronomical constant factors (roughly exponential in the size of the optimal program), the algorithm is optimal only with respect to a particular machine model (say a TM, a RAM, or a pointer machine). Our result, in contrast, 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. In fact, one can easily concoct simple problems which are NP-hard but nevertheless have polynomial-depth decision-trees (e.g. find the lightest edge on any Hamiltonian path). See [GKS93], [PR01, Section 8] for two sorting-type problems whose decision-tree complexity and algorithmic complexity provably diverge.

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. 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. Is there an optimal uniform algorithm which 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 which uses precomputed decision trees [B+98] and the one which does not [Tar79b].

One may also ask for the parallel complexity of the MSF problem. Here, resolved recently were the randomized work-time complexity [PR99] and the deterministic time complexity [CHL99] 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 which are suitable for the base case, such as the one in [FT87], are also not easily parallelizable.

References

[AHU74] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[AS92] N. Alon, J. Spencer. *The Probabilistic Method.* Wiley, New York, 1992.

[Bor26] O. Borůvka . O jistém problému minimaálním. *Moravské Přírodovědecké Společnosti* 3, pp. 37-58, 1926. (In Czech).

[B+98] A. L. Buchsbaum, H. Kaplan, A. Rogers, J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In *Proc. STOC'98*, pp. 279–288, 1998.

[Chaz97] B. Chazelle. A faster deterministic algorithm for minimum spanning trees. In *FOCS '97*, pp. 22–31, 1997.

[Chaz98] B. Chazelle. Car-pooling as a data structuring device: The soft heap. In *ESA '98* (Venice), pp. 35–42, LNCS 1461, Springer, 1998.

[Chaz00] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. NECI Technical Report 99-099, 1999.

[CHL99] K. W. Chong, Y. Han and T. W. Lam. On the parallel time complexity of undirected connectivity and minimum spanning trees. In *Proc. SODA*, pp. 225-234, 1999.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. In *Numer. Math.*, 1, pp. 269-271, 1959.

[DRT92] B. Dixon, M. Rauch, R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Jour. Comput.*, vol 21, pp. 1184-1192, 1992.

[ER61] P. Erdös, A. Rényi On the evolution of random graphs. *Bull. Inst. Internat. Statist.* 38, pp. 343–347, 1961.

[FT87] M. L. Fredman, R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *JACM* 34, pp. 596–615, 1987.

[FW90] M. Fredman, D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. In *Proc. FOCS '90*, pp. 719–725, 1990.

[GGST86] H. N. Gabow, Z. Galil, T. Spencer, R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. In *Combinatorica* 6, pp. 109–122, 1986.

[GH85] R. L. Graham, P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing* 7, pp. 43–57, 1985.

[GKS93] W. Goddard, V. King, L. Schulman. Optimal randomized algorithms for local sorting and set-maxima. *SIAM J. on Comput.* 22 (1993), no. 2, 272–283.

[Jar30] V. Jarník. O jistém problému minimaálním. *Moravské Přírodovědecké Společnosti* 6, pp. 57-63, 1930. (In Czech).

[Jo97] N. Jones. *Computability and Complexity: From a Programming Perspective.* MIT Press, 1997.

[KKT95] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *JACM*, 42:321–328, 1995.

[KT80] R. M. Karp, R. E. Tarjan. Linear expected-time algorithms for connectivity problems. *J. Algorithms* 1 (1980), no. 4, pp. 374–393.

[K97] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica* 18 (1997), no. 2, 263–270.

[Lar90] L. L. Larmore. An optimal algorithm with unknown time complexity for convex matrix searching. *IPL*, vol. 36, pp. 147-151, 1990.

[PR99] S. Pettie, V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest *Proc. RANDOM '99*, LNCS 1671, Springer, pp. 233-244, 1999.

[PR99b] S. Pettie, V. Ramachandran. An optimal minimum spanning tree algorithm. Tech Report TR99-17, Univ. of Texas at Austin, 1999.

[PR01] S. Pettie, V. Ramachandran. Computing undirected shortest paths with comparisons and additions. Tech Report TR01-12, Univ. of Texas at Austin, 2001.

[Pet99] S. Pettie. Finding minimum spanning trees in $O(m\alpha(m,n))$ time. Tech Report TR99-23, Univ. of Texas at Austin, 1999.

[Prim57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389-1401, 1957.

[Tar79] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. In *JCSS*, 18(2), pp 110–127, 1979.

[Tar79b] R. E. Tarjan. Applications of path compression on balanced trees. *J. Assoc. Comput. Mach.* 26 (1979), no. 4, 690–715.