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

August 4, 1999

UTCS Technical Report TR99-17

Abstract

We present a deterministic algorithm to find a minimum spanning forest of an edge-weighted undirected graph. On a graph with n vertices and m edges, the algorithm runs in time $O(\mathcal{T}^*(m,n))$ where $\mathcal{T}^*$ is the decision-tree complexity of the problem. This time bound is provably optimal as a function of n and m. The algorithm is quite simple, and can be implemented on a pointer machine.

The exact function describing the running time of our algorithm is not known at present. The current best bounds known for $\mathcal{T}^*$ (and hence the running time of our algorithm) are $\mathcal{T}^*(m,n) = \Omega(m)$ and $\mathcal{T}^*(m,n) = O(m \cdot \alpha(m,n) \cdot \log\alpha(m,n))$, where α, a certain natural inverse of Ackermann's function, is an extremely slow-growing function.

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. The first algorithms for finding MSTs were published in 1926 and 1930 by Borůvka [Bor26] and Jarník [Jar30] and for many years the only progress made on the MST problem was in rediscovering these algorithms. Borůvka's algorithm was rediscovered by Choquet [Cho38], Florek et al. [FLPSZ51], and Sollin [BG65], and both Prim [Prim57] and Dijkstra [Dij59] give descriptions of Jarník's algorithm, together with some important implementation details. (See [GH85] for a history of the MST problem.) Kruskal [Kr56] presented an algorithm that matched previous algorithms in terms of simplicity but did not improve on the $O(m \log n)$ time bound first established by Borůvka. Here m and n are the number of edges and vertices in the graph.

The $m \log n$ barrier was broken in the mid-1970s by $O(m \log\log n)$ time algorithms by [Yao75] and Cheriton and Tarjan [CT76]. The MST problem saw no new developments until the mid-1980s when Fredman and Tarjan [FT87] used Fibonacci heaps (presented in the same paper) to give an algorithm running in $O(m\,\beta(m,n))$ time[1]; in the worst case this algorithm runs in $O(m \log^* n)$ time[2]. Soon thereafter Gabow et al. [GGST86] refined this algorithm to obtain a running time of $O(m \log \beta(m,n))$. Then recently Chazelle [Chaz97] presented an MST algorithm running in

[1] By definition, $\beta(m,n) = \min\{i : \log^{(i)} n \le \frac{m}{n}\}$; here $\log^{(1)} n = \log n$, $\log^{(i+1)} n = \log\log^{(i)} n$.

[2] $\log^* n = \min\{i \ge 1 : \log^{(i)} n \le 1\}$.

time $O(m\alpha(m,n)\log\alpha(m,n))$, where α is a certain inverse of Ackermann's function which grows extremely slowly. This algorithm uses a data structure called the Soft Heap [Chaz98]. This is the fastest algorithm to date, although it is not known to be optimal.

All algorithms mentioned thus far require a relatively weak model of computation. Each can be implemented on a deterministic pointer machine model [Tar79] which does not allow pointer arithmetic, hence certain techniques such as table lookup cannot be used, and in which the only operations allowed on edge weights are binary comparisons. If more powerful models of computation are used then finding minimum spanning trees can be done even faster. Under the assumption that edge weights are integers, Fredman and Willard [FW90] showed that on a unit-cost RAM in which the bit-representation of edge weights may be manipulated, the MST can be computed in linear time. Karger et al. [KKT95] considered a unit-cost RAM with access to a stream of random bits and showed that with high probability, the MST can be computed in linear time, even if edge weights are only subject to comparisons. Buchsbaum et al. [BKRW98] recently showed that the randomized algorithm of [KKT95] can be made to run on a pointer machine.

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 MST algorithm that runs on a pointer machine and is provably optimal; specifically, we prove that our algorithm runs 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. This implies that any future algorithm for computing MST that performs $f(m,n)$ edge-weight comparisons (for some function f) demonstrates that our algorithm performs $O(f(m,n))$ operations in total; note that the cost incurred by using data structures which require super-linear time to maintain (such as union-find), may be exempted. Further, if one is able to show that the MST problem can be solved with no more than $f(m,n)$ comparisons through any type of reasoning, including a nonconstructive proof, this would imply that our algorithm runs in $O(f(m,n))$ time.

Although our algorithm is optimal, its precise running time is not known at this time. In view of Chazelle's algorithm [Chaz97] we can state that the running time of our algorithm is $O(m \cdot \alpha(m,n) \cdot \log\alpha(m,n))$. Clearly, its running time is also $\Omega(m)$.

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 on a graph G. Let x and y be vertices in T, let w and z be vertices not in T, and let (x, w) and (y, z) be edges in $G - T$. If f is the edge of maximum weight on the path in T connecting x and y, then the weight of f cannot be larger than the weights of both (x, w) and (y, z).*

Proof: Let $f = (a, b)$ and let the path $\mathcal{P}$ in T connecting x and y consist of a path from x to a, followed by edge f, followed by a path from b to y.

Consider the step in which f was chosen to be added to the DJP tree and assume w.l.o.g. that at this time a is in the tree and b is not. Let $\mathcal{P}'$ be the subpath of $\mathcal{P}$ that is present in the tree at this step, and let its endpoints be a and c. If $c \neq x$ then the next edge in $\mathcal{P}$ incident on c has smaller cost than f and is eligible to be picked in this step, hence f would not be chosen in this step. Hence $c = x$. But then f must have smaller weight than edge (x, w), since (x, w) is eligible to be picked at this step. □

2.3 The Dense Case Algorithm

Our algorithm will switch to another MSF algorithm when the graph becomes sufficiently dense, allowing its MSF to be computed in linear time by one of several existing algorithms. Here density refers to the edge-to-vertex ratio. The procedure DenseCase$(G; F)$ takes as input a graph G and returns the MSF F of G. Our algorithm guarantees DenseCase will be called on graphs of density $\Omega(\log^{(3)} n)$, thus the algorithms presented in [FT87, GGST86, Chaz97] could be used as DenseCase since each runs in linear time for that density.

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. However, the values of some keys may be increased, *corrupting* the associated items and potentially causing later Findmins to report the wrong answer. The guarantee is that after n Insert operations, no more than ϵn corrupted items are in the heap. Note that because of deletes, the proportion of corrupted items could be much greater than ϵ. The following result in 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.

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 $weight_G(e)$ be the weight of edge e in graph G (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 some amount (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. To be very explicit, for each connected component C' of C, we add a new vertex c' to G and reassign the endpoints of edges with one or more endpoint in C'. If x is an endpoint for some edge and $x \in C'$, that endpoint is reassigned to c'. Finally we remove from G all vertices and edges in C.

We define a subgraph C of G to be *DJP-contractible* if the tree that results by executing the DJP algorithm on G for some number of steps, starting with a vertex in C, 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 $H = G \backslash C - M_C$. Our goal is to show that no edge in $H -$ MSF(H) is in MSF(G).

Let e be an edge in $H -$ MSF(H). Then e must be the heaviest edge on the cycle $\mathcal{X}$ formed in H when e is added to MSF(H). In G the cycle $\mathcal{X}$ forms a path $\mathcal{P}$ whose end points, say x and y, are both in C; let these end edges in $\mathcal{P}$ be (x, w) and (y, z). In H we removed all corrupted edges with one end point in C. Hence both (x, w) and (y, z) are not in M.

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 f be the heaviest edge in $\mathcal{Q}$. Notice that $\mathcal{P} \cup \mathcal{Q}$ forms a cycle. The weight of e is larger than the weights of the other edges in the path $\mathcal{P}$. Hence if e is in MSF(G), then by the cycle property $weight_{G \Uparrow M}(f) > weight_G(e)$. But this requires $weight_{G \Uparrow M}(f)$ to be greater than the weights of edges (x, w) and (y, z) since both of these edges have smaller weight than e. By Lemma 2.1 this is not possible. Hence e cannot be in MSF(G). □

3.2 The Partition Procedure

Our algorithm uses the Partition procedure 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

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

Below, arguments appearing before the semicolon are inputs; the outputs will be returned in the other arguments. 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 $M, C_1, \ldots, C_k$.

Partition$(G, maxsize, \epsilon\,;\ M, \mathcal{C})$
```
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.
```

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. In place of a correct heap, 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 the Partition procedure 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 $|M| \leq 2\epsilon m$. Summarizing the specification of Partition, we have the following lemma.

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 connected component (i.e., 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 The Optimal Algorithm

4.1 Overview

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 above.

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.

4.2 Decision Trees

An MSF decision tree is a rooted tree having an edge-weight comparison associated with each internal node (i.e. $weight(x, y) < weight(w, z)$). 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 4.1, 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. This allows us to 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 $m \cdot \alpha(m, n) \cdot \log \alpha(m, n)$). 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 to establish our main result that OptimalMSF runs in $O(\mathcal{T}^*(m, n))$ time.

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

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

Claim 5.3 *For fixed n and $m' > m$, $\mathcal{T}^*(m', n) \geq \mathcal{T}^*(m, n)$.*

Claim 5.1 is obviously true since every edge should participate in a comparison to determine inclusion in or exclusion from the MSF. Claim 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. To see that Claim 5.3 holds, we observe that we can add edges of very large cost to a graph without altering its MSF.

We now state a Condition that is used by Lemmas 5.5 and 5.6.

Condition 5.4 *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 Condition 5.4 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 $\bigcup_{j<i} C_j$.

Lemma 5.5 *If Condition 5.4 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 Condition 5.4 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. □

Lemma 5.6 *If Condition 5.4 holds for a graph 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 sums 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 5.5). 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 will establish 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 the 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 5.7 *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, $\sum_i \mathcal{T}^*(C_i) \leq \mathcal{T}^*(G) \leq \mathcal{T}^*(m,n)$.*

Corollary 5.8 *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.

$$\begin{aligned}
T(m,n) &\leq \sum_i \mathcal{T}^*(C_i) + T(m/2, n/4) + c_1 \cdot m \\
&\leq \mathcal{T}^*(m,n) + T(m/2,n/4) + c_1 \cdot m \quad \textit{(Corollary 5.7)} \\
&\leq \mathcal{T}^*(m,n) + c \cdot \mathcal{T}^*(m/2,n/4) + c_1 \cdot m \quad \textit{(assume inductively)} \\
&\leq \mathcal{T}^*(m,n)(1 + c/2 + 2c_1) \quad \textit{(Corollary 5.8 and Claims 5.1, 5.2)} \\
&\leq c \cdot \mathcal{T}^*(m,n) \textit{ (for some } c \textit{ sufficiently large; this completes the induction)}
\end{aligned}$$

This gives us the desired theorem.

Theorem 5.9 *Let $\mathcal{T}^*(m,n)$ be the decision tree complexity of the MSF problem on graphs with m edges and n nodes. The 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 [BKRW98], but we ensure that the time taken 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 $O(r^{r^2})$ entries in order to match up the graphs. This takes time $O(m + r^{r^2}) = O(m)$.

7 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.

An intriguing aspect of our algorithm is that we do not know its precise running time. The presence of Chazelle's algorithm [Chaz97] shows that its running time is $O(m\alpha(m, n) \log \alpha(m, n))$. This could conceivably be the correct bound for our algorithm — all that we can say at this time is that the bound lies between this rather unwieldy upper bound and the obvious linear-time lower bound.

Since our time bound depends only on the decision tree complexity of the MSF problem, the running time of our algorithm (and hence of the fastest MSF algorithm) depends only on the number of edge-weight comparisons needed to resolve this problem, and not on data structural issues. Hence if the complexity of the problem turns out to be, say, $\Theta(m\alpha(m, n))$, the α will not be due to the use of a data structure with that complexity, but rather due to the nature of edge-weight comparisons required. This also means that in order to determine the complexity of the MSF problem one can now look solely at its decision tree complexity without considering the data structures needed to implement the other features of the algorithm. This could potentially simplify proofs on the algorithmic complexity of the MST problem.

Pinning down the function that describes the worst-case complexity of our algorithm is the main open question that remains for the sequential complexity of the MSF problem. One can also ask for the parallel complexity of this problem. Here, the randomized complexity of the MSF problem on the EREW PRAM was recently resolved in [PR99]. For deterministic parallel MSF algorithms, the time complexity on the EREW PRAM was resolved recently in [CHL99]. An open question that remains here is to obtain a deterministic parallel MSF algorithm with optimal work and time bounds.

References

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

[BG65] C. Berge, A. Ghouila-Houri. *Programming, Games, and Transportation Networks.* John Wiley, New York, 1965.

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

[BKRW98] 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. of the 30th ACM Symposium on Theory of Computing*, 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, Lecture Notes in Comp. Sci., 1461, Springer, Berlin, 1998.

[Cho38] G. Choquet. Etude de certains réseaux de routes. *Comptes Rendus Acad. Sci.*, 206 (1938), pp. 310-313.

[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* 1999, pp. 225-234.

[CT76] D. Cheriton, R. E. Tarjan. Finding minimum spanning trees. In *SIAM J. Comput.* 5 (1976), pp. 724–742.

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

[FLPSZ51] K. Florek, L. Lukasziewicz, J. Perkal, H. Steinhaus, S. Zubrzycki. Sur la liaison et la division des points d'un ensemble fini. In *Colloq. Math.*, 2 (1951), pp. 282–285

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

[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 (1986), pp. 109–122.

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

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

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

[Kr56] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. In *Proc. Amer. Math. Soc.* 7 (1956), pp. 48–50.

[PR99] S. Pettie, V. Ramachandran. A Randomized Time-Work Optimal Parallel Algorithm for Finding a Minimum Spanning Forest To appear in *Proc. RANDOM '99*, also Tech. Report TR99-13, Univ. of Texas at Austin, April 1999.

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

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

[Tar83] R. E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[Yao75] A. Yao. An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees. *Information Processing Letters* 4 (1975), pp. 21–23.