Finding Minimum Spanning Trees in $O(m\,\alpha(m,n))$ Time

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

October 21, 1999

UTCS Technical Report TR99-23

Abstract

We describe a deterministic minimum spanning tree algorithm running in time $O(m\,\alpha(m,n))$, where α is a natural inverse of Ackermann's function and m and n are the number of edges and vertices, respectively. This improves upon the $O(m\,\alpha(m,n)\log\alpha(m,n))$ bound established by Chazelle in 1997.

A similar $O(m\,\alpha(m,n))$-time algorithm was discovered independently by Chazelle, predating the algorithm presented here by many months. This paper may still be of interest for its alternative exposition.

1 Introduction

We consider the problem of finding a minimum spanning tree on a weighted, undirected graph. This problem has been studied in its present form for many decades and yet to date, no proof of its complexity has been found. The first MST algorithms were discovered by Borůvka [Bor26] and Jarník [Jar30] and for many years the only progress made on the MST problem was in rediscovering these algorithms. (See [GH85] for an historical survey of MST.) Kruskal [Kr56] presented an algorithm that rivaled previous algorithms in terms of simplicity but did not improve on the $O(m\log n)$ time bound first established by Borůvka. Here m (resp. n) is the number of edges (resp. vertices) in the graph.

The $m\log n$ barrier was broken by Yao's $O(m\log\log n)$ time algorithm[1] [Yao75], which was followed quickly by Cheriton and Tarjan's $O(m\log\log_d n)$ time algorithm [CT76], where $d=\max\{2,\frac{m}{n}\}$. 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[2]. In the worst case, $\beta(m,n)=\log^* n$). Before the ink had dried on this result Gabow et al. [GGST86] upped the ante to $O(m\log\beta(m,n))$, a result which stood for a decade.

Recently Chazelle described a non-greedy approach to solving the MST problem which makes use of the soft heap [Chaz98a], a priority queue which is allowed to corrupt its own data in a controlled fashion. This led to an algorithm [Chaz97, Chaz98b] running in time $O(m\alpha\log\alpha)$, where $\alpha=\alpha(m,n)$ is a certain inverse of Ackermann's function.

Pettie and Ramachandran [PR99] have just developed an optimal MST algorithm by breaking the larger MST problem into manageable subproblems and finding the MSTs on these subproblems using optimal decision trees. In the decision tree model, edge cost comparisons take unit time and all other operations are free. The overhead for this algorithm is linear, thus its running time is asymptotically the same as the decision tree complexity of the MST problem. Considering this result, in the analysis of our algorithm we will not address the time spent on operations which do not involve edge cost comparisons. Henceforth, *running time* refers to time under the decision tree model.

[1] Actually, Yao cites an unpublished algorithm of Tarjan running in $O(m\sqrt{\log n})$ time.
[2] $\beta(m,n)=\min\{i:\log^{(i)} n\le\frac{m}{n}\}$.

All algorithms mentioned thus far require a relatively weak model of computation. Each can be implemented on a pointer machine[3] in which the only operations allowed on edge costs are comparisons. If more powerful models of computation are used then finding minimum spanning trees can be done even faster. Under the assumption that edge costs are integers, Fredman and Willard [FW90] showed that on a unit-cost RAM in which the bit-representation of edge costs may be manipulated, the MST can be computed in linear time. Karger et al. [KKT95] considered a model 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 costs are only subject to comparisons. It is still unknown whether these more powerful models are necessary to compute the MST in linear time.

In this paper we present a deterministic minimum spanning tree algorithm running in time $O(m\alpha(m,n))$. The increase in speed over [Chaz97, Chaz98b] is the result of dealing with "bad" edges[4] more intelligently, which also calls for changes to the recursive structure of the 1997 algorithm. In addition, we believe our exposition highlights the underlying elegance of the algorithm.

We would like to give due credit to Chazelle on two matters. First, the bulk of our algorithm was in place in his $O(m\alpha\log\alpha)$-time algorithm [Chaz97, Chaz98b]. Second, he has independently lowered the complexity of his 1997 algorithm to $O(m\alpha)$ [Chaz99]. There is no question that this result predates our algorithm.

2 The Soft Heap

The soft heap [Chaz98a] 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 $O(\log(\frac{1}{\epsilon}))$ amortized time. Here's the catch: to make its job easier, the soft heap may increase the values of any keys, *corrupting* the associated items and potentially causing later findmins to report the wrong answer. Once corrupted, an item's key may still increase, though never decrease. 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.

3 Preliminaries

The input is an undirected graph $G = (V, E)$ with a distinct cost associated with each edge. We make no other assumptions about the costs, but require that any two may be compared in constant time. The minimum spanning tree problem can be stated in just a handful of words: find the tree spanning the vertices of G which is of minimum total cost.

Although we must minimize the total cost, edges may be certified as being inside or outside the MST by observing just a subset of G. By the *cycle property*, the costliest edge on any cycle in G is not in the MST. Assume for the purposes of contradiction that such an edge, call it e, was in the MST. Edge e separates the vertices of the MST into two groups, meaning there must be at least one edge from the cycle, call it f, which has one endpoint in each group. We can thus produce a tree of lesser total cost by substituting f for e. Dual to the cycle property is the *cut property* which states that for any cut $X \subset V(G)$, the cheapest edge

[3]The pointer machine model prohibits pointer arithmetic, so certain techniques such as table lookup cannot be used. See [Tar79].

[4]Bad edges will be discussed in later sections. Briefly, the algorithm finds a spanning tree where the only edges that could possibly decrease its weight are the bad ones. They are reconsidered in recursive calls in order to find the *minimum* spanning tree.

with exactly one endpoint in X is in the MST. This follows directly from the cycle property since such an edge cannot be the costliest in any cycle.

Traditional MST algorithms identify the minimum cost edge crossing a cut X by keeping all eligible edges incident to vertices in X in a heap. We will use this same strategy, using a soft heap in place of a correct heap. Edges identified in this manner will be in the MST of $G \Uparrow R$, a graph derived from G by raising the costs of all edges in $R \subseteq E(G)$. How do the cut and cycle properties fare in this corrupted graph? Unless all edges crossing a cut are uncorrupted (not in R), the minimum such edge is not guaranteed to be in MST(G). Similarly, the costliest edge in some cycle is definitely not in MST(G) only if it is uncorrupted (all corrupted edges in the cycle having higher costs than w.r.t the graph G).

Using these two properties for the purpose of classifying edges will not prove useful. However, we may derive useful information about the MST by certifying that *regions* of the graph are *contractible*. We say that a subgraph C is contractible if for any edges e and f, each having one endpoint in C, there exists a path connecting e to f in C consisting of edges with costs less than either e or f. The notation $G \backslash C$ is used to mean the graph G with the subgraph C contracted into a single vertex c. Edges incident to one vertex in C become incident to c and edges internal to C are removed. The following Lemma is very well known.

Lemma 3.1 *If C is contractible w.r.t G, then $MST(G) = MST(G \backslash C) \cup MST(C)$.*

Proof: Edges in C which are not in MST(C), being the costliest on some cycle, are also not in MST(G) since that cycle exists in G as well. We need only examine edges which are the most expensive on a cycle in $G \backslash C$ involving vertex c. Let e and f be the two edges incident to c in such a cycle. By the contractibility of C, there is a path connecting e to f in C, the edges of which are cheaper than $\max\{cost(e), cost(f)\}$. Therefore, the costliest edge on any cycle in $G \backslash C$ is the costliest edge on a corresponding cycle in G.
□

This idea of contractibility is surprisingly robust when applied to corrupted graphs. Clearly Lemma 3.1 does not work as is. With a little adjustment however, we obtain a Lemma which is crucial to the correctness of the algorithm.

Lemma 3.2 *If C is contractible w.r.t $G \Uparrow R$, and R_C are those edges in R with one endpoint in C, then $MST(G) \subseteq MST(C) \cup MST(G \backslash C \Uparrow R_C) \cup R_C$*

Proof: First note that C is also contractible w.r.t $G \Uparrow R_C$ since returning the edges of C to their uncorrupted state only lowers their cost. Edges in C which are not in MST(C) are the most expensive along some cycle and thus are not in MST(G) since the cycle exists there as well. Consider the edge e, the costliest on some cycle in $G \backslash C \Uparrow R_C$ involving vertex c (derived by contracting C). If e is not corrupted, i.e. not in R_C, then by the contractibility of C, it is also the costliest edge in some cycle in $G \Uparrow R_C$, and thus in G as well. However, if e is corrupted it is not necessarily the costliest edge in some cycle in G (though it is for some cycle in $G \Uparrow R_C$.) This forces us to reconsider all edges in R_C.
□

The Lemma given above is enough to show the correctness of the following generic algorithm.

1. Consider a graph $G_0 = G \Uparrow R$ derived from the input graph by corrupting all edges in $R \subseteq E(G)$. Partition G_0 into contractible subgraphs, then contract each subgraph into a single vertex, forming the graph G_1. Repeat the partition-contraction step (creating graphs $G_2, G_3, \ldots$) until the whole graph contracts into a single vertex. Whenever a subgraph is contracted, corrupted edges with one endpoint in that subgraph are marked as *bad*. They, as well as any other corrupted edges, remain corrupted.

2. Next, recurse on the non-bad edges of each contracted subgraph, returning its MST. Non-bad edges should be restored to their original cost before the algorithm is applied recursively.

3. Finally, recurse on the graph consisting of the edges returned in step 2 and the bad edges found in the step 1, returning them to their original cost. By repeated application of Lemma 3.2, this set of edges contains the MST of the original graph G.

In the actual algorithm edges will be corrupted progressively, not in one swift stroke. However, let us momentarily abstract away this aspect of the algorithm.

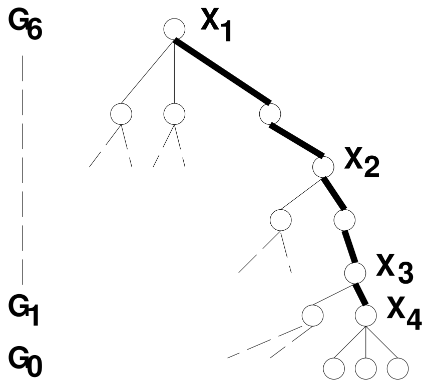

Figure 1: The contraction tree, partially built.

We can represent the contractions made in the first step by a *contraction tree* T. A node x in T with height h represents both a vertex v_x of G_h *and* a subgraph of G_{h-1}. The children of x in T represent those vertices of G_{h-1} contracted to form v_x.

Building T efficiently is central to this algorithm but doing it bottom-up (or equivalently, top-down) does not give us the desired running time (though it is possible to match the performance of the algorithm of Gabow et al. [GGST86] using this method). We, however, will build T in post-order: the children of a node $x \in T$ will be assembled from left to right in post-order followed by node x. If only some of x's children are complete we say that x is "under construction". At any given time there will be no more than one subgraph from each of $G_d, G_{d-1}, \ldots, G_0$ under construction, where d is the height of T. As soon as the subgraph of G_i has reached a critical size (which depends on i), it is contracted into a single vertex and added to the subgraph of G_{i+1} under construction (which might cause *it* to contract, and so on).

We maintain a sequence of subgraphs $\mathcal{X} = \langle X_1, \ldots, X_k \rangle$ which are under construction and currently have at least one vertex. At all times $h_i \geq h_{i+1}$ where h_i is the height of X_i in T. (This is in contrast to the G_i, which are indexed by increasing height.) Note that indices in this sequence do not directly relate to the height of the subgraph in T since a subgraph is not represented until its left-most child is contracted. See Figure 1.

Building T in post-order is no simple matter. In the next section we discuss the structure of $\mathcal{X}$, how new vertices outside of $\mathcal{X}$ are incorporated into it, and how to decide when subgraphs should be contracted This is where Ackermann's function comes into play.

The recursive structure of the algorithm is described in Section 5. In Section 6 we analyze the running time, and in Section 7 we prove bounds on the number of bad edges generated whilst building T. The substantive changes to the algorithm of [Chaz97, Chaz98b] will be found in Section 5, along with a short summary. Several other changes were made to simplify the description and analysis of the algorithm.

4 Building the Contraction Tree

Before getting into the details of building T we must define some terminology. Edges with one endpoint in $\mathcal{X}$ are called *external*; those with both endpoints in $\mathcal{X}$ are *internal.* Every external edge is either in some soft heap, has been certified to be outside of MSF(G) and discarded, or has been marked bad and set aside. Note that not all bad edges are set aside; some remain in soft heaps. When an edge makes the transition from external to internal (after an outside vertex is added to $\mathcal{X}$), it is deleted from the soft heap which contains it, hence it cannot be corrupted further.

At any time during the construction of T the *heap* cost of an edge is the cost given to it by the soft heap (which is its original cost if uncorrupted). The *current* cost of an edge is its heap cost if it is external or bad, and its original cost otherwise. When there are no more external edges (i.e. T has been built), the *final* cost is the current cost. To avoid possible confusion, let us stress that after T is built, the only pertinent information about an edge is its original cost and whether it is bad or not. The heap costs of corrupted edges are never used again and may be forgotton.

One may wish to verify the following truths, which will be taken as self-evident in the proof of correctness.

1. Heap costs never decrease.
2. For each internal edge, current cost = final cost = original cost if not bad.
3. For each bad edge, its final cost will be no less than its current cost.

Obviously we do not know the final cost of every edge until T is built. However, we will contract subgraphs consistent with final costs, whatever they may be. To that end, we maintain the following important invariant.

Invariant 1 *Let g be the edge of minimum current cost between X_i and X_{i+1}. The current cost of g is cheaper than the current cost of all external edges incident on X_j for $j \le i$, and internal edges between distinct X_j, $X_{j'}$, for $j, j' \le i$.*

It will not be difficult to maintain the invariant w.r.t. external edges. For internal edges we keep an edge min-link(i, j) (for each pair i, j) which is the edge of minimum current cost (= final cost) connecting X_i to X_j. Internal edges which are not min-links will not be examined again until T is built, and thus can be ignored for the moment.

The two basic operations on $\mathcal{X}$ are the extension and the retraction (see Figure 1). We use them in concert to construct contractible subgraphs whose size and density are well balanced, all the while maintaining Invariant 1. In a retraction all corrupted external edges incident to X_k (the last subgraph under construction in $\mathcal{X}$) are marked bad; X_k, which has height h in T, is contracted into a single vertex x_k which is then added to the subgraph of G_h under construction. Generally this will be X_{k-1} but if the height of X_{k-1} and X_k differ by more than one, x_k will simply be christened X_k having height $h+1$. An extension consists of selecting the external edge (u, v) (where $u \in \mathcal{X}$) of minimum current cost and adding v to $\mathcal{X}$, followed by a round of retractions if some subgraphs have reached their critical size. Extending $\mathcal{X}$ to include v can be a simple operation. However, in order to preserve Invariant 1 we may need to prepare for v's arrival by performing a sequence of premature retractions followed by a *fusion*. The difference between a fusion and a retraction is this: in a retraction the last subgraph X_i is contracted and the resulting vertex is added to X_{i-1}. In a fusion, after X_i is contracted, the resulting vertex and an existing vertex of X_{i-1} are then contracted (or *fused*).

The procedure Build-T (described later), decides how to order the extensions and retractions. Note that a retraction can happen either because Build-T issued one, or prematurely as part of a complex extension.

Retraction

Let X_k be the last subgraph under construction in $\mathcal{X}$.
Mark bad all corrupted edges incident on X_k.
Contract X_k, let the resulting vertex be x_k.
Let X_k and X_{k-1} have height h_k and h_{k-1} resp.
If $h_{k-1} = h_k + 1$ then
    $X_{k-1} = X_{k-1} \cup \{x_k\}$
    $k = k - 1$
Otherwise
    $X_k = \{x_k\}$
    $h_k = h_k + 1$
Update the min-links

Simple Extension

Let (u,v) have minimum current cost among external edges.
If $u \in X_k$ and current-cost$(u,v) <$ min-link(i,j) for all i,j
Then $X_{k+1} = \{v\}$.
  Let X_{k+1} have height zero in T.
  Find min-link(i,k+1) for all $i \le k$.
  $k = k+1$
Otherwise perform a Complex Extension.

Complex Extension

Find min i s.t. min-link$(i,j) <$ current-cost(u,v) (for some j).
Perform retractions until X_{i+1} is last subgraph in $\mathcal{X}$.
Contract X_{i+1} into a single vertex z.
Perform a fusion:
  Let (w,z) be min-link$(i,i+1)$, $w \in X_i$.
  Contract the edge (w,z). This is the fusion edge.
Edge (u,v) is now cheaper than all min-links, hence a simple extension can be performed.

It is not too difficult to see that a simple extension preserves Invariant 1. By our choice of edge (u,v), it must have current cost less than any other external edge, and by being eligible for a simple extension its cost must be less than all min-links. When (u,v) is added to $\mathcal{X}$, its current cost may only drop (if it is corrupted, but not bad), hence Invariant 1 is preserved.

If the edge (u,v) was ineligible for a simple extension, then some subset of the min-links had costs less than (u,v). The solution, in essence, is to perform retractions until these min-links do not exist. This operation should seem very suspicious since the algorithm depends upon the contraction tree being built in such a regimented manner. However, it turns out that complex extensions are quite desirable and in the end save us a lot of work.

The key to the complex extension is the fusion (see Figure 2). Let i be minimal s.t. for some j, min-link(i,j) is less than the cost of (u,v). Consider the state of affairs just after we have contracted the subgraphs $X_{i+1}, X_{i+2}, \ldots, X_k$ into a single vertex z, and let (w,z) be min-link$(i,i+1)$. Since (w,z) is contractible w.r.t. final costs (see Lemma 4.2), we can safely contract it; call the resulting vertex $fuse(w,z)$. Note that this is *not* a retraction, hence no new vertices get added to X_i. The effect of a fusion on the contraction tree is to create a dummy node $fuse(w,z)$ whose children correspond to w and z. We do not recalculate the height of nodes in T after a fusion; the dummy node in T is just meant to document that w and z were contracted. See Figure 3.

Lemma 4.1 *At all times and for all i, X_i is contractible w.r.t. current costs.*

Proof: Consider some X_i just before it acquires another vertex by way of a retraction, and assume inductively that X_i is contractible. Let (u,v) be min-link$(i,i+1)$, where $u \in X_i$ and v is the vertex derived by contracting X_{i+1}. Consider a pair of edges e and $f = (v,w)$ incident on $X_i \cup \{v\}$, and a path $\mathcal{P} \cup \{(u,v)\}$ connecting e to f where $\mathcal{P}$ is the cheapest path connecting e to (u,v) in X_i. By Invariant 1 the current cost of (u,v) is less than that of e, and by the contractibility of X_i, the costliest edge in $\mathcal{P}$ is also less than e. Hence, $X_i \cup \{v\}$ is contractible w.r.t. current costs. Any future corruption of external edges will not affect the contractibility of $X_i \cup \{v\}$. $\square$

Lemma 4.2 *Let C be a subgraph or fusion edge contracted while building T, then C is contractible w.r.t. final costs.*

Proof: Suppose $C = X_i$ at the time of its contraction. For edges internal to X_i, their current cost equals their final cost. By Lemma 4.1 X_i is contractible w.r.t. current costs. Since all external corrupted edges incident to X_i are marked bad upon its contraction, their final cost can never be less than their current cost, thus C is contractible w.r.t. final costs.

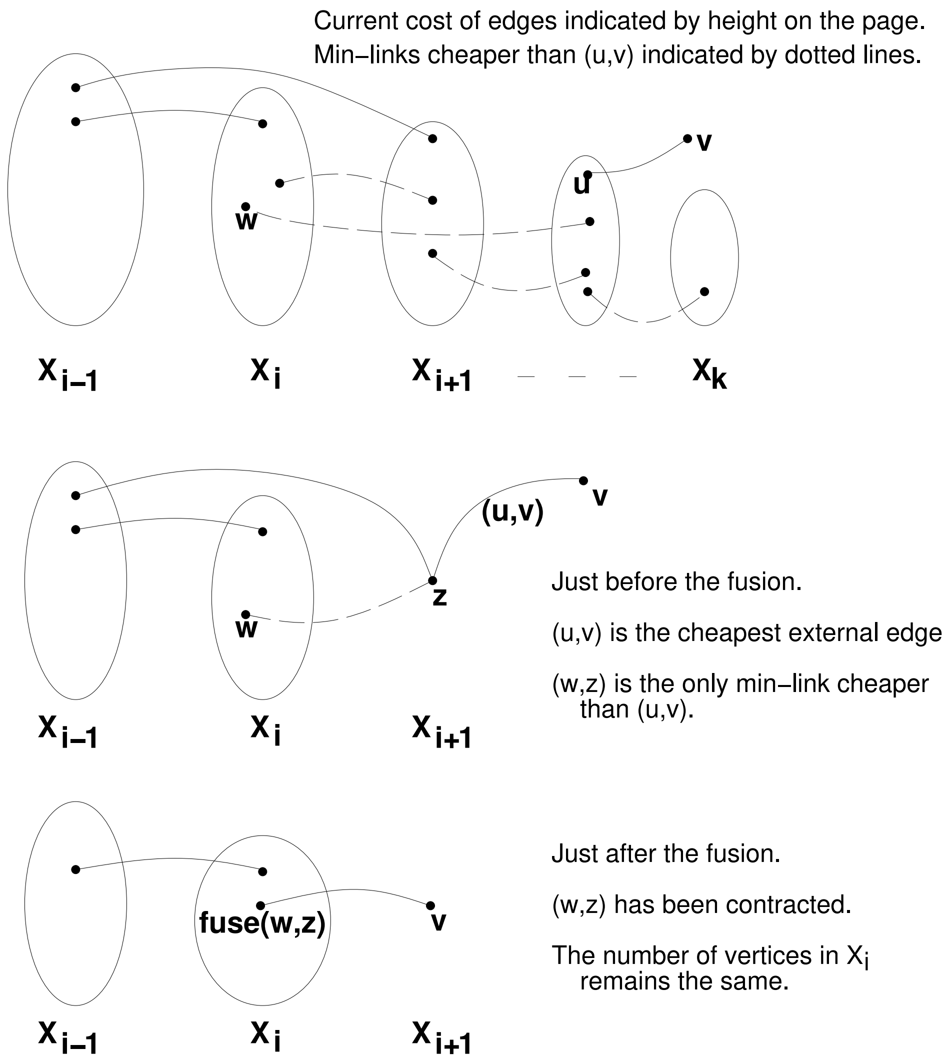

Figure 2: A complex extension, step by step.

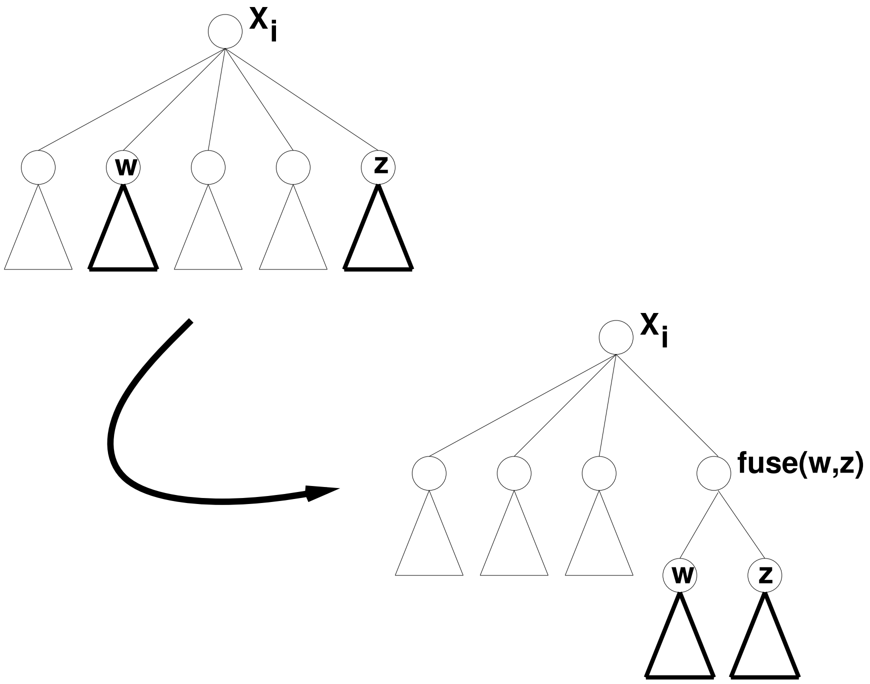

Figure 3: A fusion's effect on the contraction tree.

Consider the case when $C = (w, z)$ is a fusion edge, where $w \in X_i$. By construction C is the minimum cost edge incident on z in terms of current cost, which corresponds to its final cost since C is internal. When the subgraphs $X_{i+1}, \ldots, X_k$ were contracted to form z all external corrupted edges incident on these subgraphs were marked bad, hence their final cost is no less than their current cost. Therefore, C is contractible w.r.t. final costs. $\square$

Lemma 4.3 *Let $G \Uparrow B$ be the graph under final costs, where B is the set of bad edges. Let x be some node in T and C_x be the corresponding subgraph. Then $MST(G) \subseteq \bigcup_{x \in T} MST(C_x - B) \cup B$.*

Proof: The edges not appearing in this expression are the non-bad edges of some C_x which are not in $\mathrm{MST}(C_x - B)$. All such edges are the most expensive in some cycle in $C_x - B$ and by the contractibility of C_x w.r.t. final costs, are also the most expensive in some corresponding cycle in $G \Uparrow B$. Since the costs of other edges in the cycle can only drop when going from $G \Uparrow B$ to G, all edges not in $\mathrm{MST}(C_x - B)$ are also not in G.
$\square$

We use Ackermann's function to regulate the construction of T. Although this is a well known function, several variants of it have been used in the analysis of algorithms. The variant we use is given below.

$$\begin{array}{ll} A(1,j) = 2^j & (j \geq 1) \\ A(i,1) = A(i-1,2) & (i > 1) \\ A(i,j) = A(i-1, A(i,j-1)) & (i,j > 1) \end{array}$$

$$\text{Let } \alpha(m,n) \text{ be the minimum } i \text{ such that } A(i, \lceil \tfrac{m}{n} \rceil) > \log n.$$

Let $d = \lceil \left(\frac{m}{n}\right)^{\frac{1}{4}} \rceil$ and t be minimal s.t. $A(t,d) \geq n$. It will be shown later that t is no more than $\alpha(m,n) + 2$. The necessity of defining t and d in this way will soon become evident.

Let $|V(X_i)|$ denote the number of vertices in X_i. Recall that each retraction into X_i adds one vertex to it, but a fusion does not affect the number of vertices. If X_i has height h_i in T then it has reached its critical size (and should be retracted) when $|V(X_i)| \geq A(t, h_i)$. If $h_i = 0$ then the critical size is 1, hence X_i would be retracted the moment it comes into existence. Assuming retractions are made on schedule, it follows that the height of T is bounded by d.

Assume there is a fictitious vertex v_0 with an infinite-cost edge connecting every vertex to v_0. This vertex provides a nice starting place for the following procedure and forces the graph to be connected.

Build-T
- Let $X_1 = \{v_0\}$ and $\mathcal{X} = \{X_1\}$
- While there are still external edges:
  - Perform an extension.
  - Repeat as many times as necessary:
    - Let X_k be the last subgraph in $\mathcal{X}$
    - If X_k has reached its critical size
      - Perform a retraction (this decrements k).

Lemma 4.4 *Call a subgraph trivial if it has only one vertex. The number of vertices in non-trivial subgraphs of G_i contracted while building T is no more than $2n/A(t, i)$.*

Proof: We claim that each vertex in a non-trivial subgraph C of G_i represents at least $A(t, i)$ vertices in G, except perhaps the last, which did not achieve its intended size. Each vertex v in C was born out of a retraction, so let us examine the two types of retractions. If v was the result of a scheduled retraction (one issued by the Build-T routine), then it must be the result of contracting $A(t, i)$ vertices from G_{i-1}. On the other hand, if v was the result of a premature retraction (as part of a complex extension), then it will either be fused with an existing vertex in C, which satisfies the claim, or C itself will be retracted. If the latter is the case, no new vertices may be added to C, also confirming the the claim.

In the worst case, each subgraph of G_i contracted has two vertices, one that reached its mature size and another of trivial size. This gives the desired bound. □

5 The Algorithm

Once we know how to build T, describing the high-level algorithm is quite simple. We use the term *density* to refer to the the edge-to-vertex ratio $\frac{m}{n}$. We assume that G has density $\geq c\alpha(m, n)$ for some suitable constant c. The algorithm is such that all graphs in recursive calls have density at least that of G, so increasing the density once (before the algorithm begins), is sufficient. There are any number of ways to do this within $O(m\alpha(m, n))$ time (see Lemma 5.1 for one).

MST(G)
(1) Compute d and t. If $t = 1$, return MST of G.
(2) Build-T, let B be the set of bad edges.
(3) For all $x \in T$ let C_x be corresponding subgraph, sans bad edges.
Improve density of C_x, yielding C'_x; put contracted edges in F_x
(4) Let $G' = \bigcup_{x \in T}(\text{MST}(C'_x) \cup F_x) \cup B$.
(5) Improve density of G', yielding G''; put contracted edges in F'.
(6) Return MST$(G'') \cup F'$

The recursion bottoms out when $t = 1$ meaning the density is at least $\log n$. Using the Dijkstra-Jarník-Prim algorithm[5], the MST of G can be found using $O(m + n \log n) = O(m)$ comparisons.

By Lemma 4.3, G' contains the MST of G, and we will show that the contracted graph G'' has only a fraction of the edges in G. The edge contraction performed in steps (3) and (5) insures that the d and t parameters behave correctly when they are recalculated in recursive calls. Specifically, we show that for all recursive calls made in step (4) the t parameter is decremented, and that it does not increase for the recursive call in step (6). The following Lemma will help bound the time to do steps (3) and (5).

[5]This simple algorithm was discovered independently by Dijkstra [Dij59], Jarník [Jar30], and Prim [Prim57]. The stated running time assumes Fibonacci heaps are used [FT87].

Lemma 5.1 *Let G have m edges and n vertices. The density of G can be increased to D (by contracting MST edges), in $O(m + n \log D)$ time.*

Proof Sketch: Performing one pass of the Fredman-Tarjan [FT87] algorithm suffices to prove the lemma. $\square$

In step (3), if x is a node introduced by a fusion, then the corresponding subgraph just has two vertices and its MST is simply the fusion edge. Otherwise, let x have height $h_x > 1$, and let d_x and t_x be the d and t parameters as calculated in the recursive call MST(C_x). We will increase the density of C_x to $\geq A(t, h_x - 1)^4$, hence d_x will be recalculated as $\geq A(t, h_x - 1)$ and t_x will be minimal s.t. $A(t_x, d_x) \geq |V(C_x)|$. By the definition of Ackermann's function (and the fact that $A(\cdot, \cdot)$ is increasing in both arguments), $A(t_x, d_x) \geq A(t_x, A(t, h_x - 1)) = A(t, h_x) \geq |V(C_x)|$ for $t_x = t - 1$. For $h_x = 1$ we just increase the density to that of G, which will also have the effect of decrementing the t parameter.

For the final recursive call we increase the density of G' to D^2 where $D = \frac{m}{n}$ is the density of G. Clearly the t parameter does not increase as G'' has fewer vertices and higher density than G.

The time spent improving density is linear in the size of G, which can be seen from Lemmas 4.4 and 5.1. Let m_i and n_i be the number of edges and vertices in G_i. By Lemma 4.4, $n_i \leq 2n/A(t,i)$, and by Lemma 5.1, the time to improve the density for all G_i is $\sum_{i=1}^{d} O(m_i + \frac{2n \log A(t,i)^4}{A(t,i)}) = O(m)$. Similarly, the time spent improving the density of G_0 and G' is also linear in m.

This algorithm differs from the one presented in [Chaz97, Chaz98b] in its recursive structure and in how edges, specifically the bad edges, move between recursive calls. If an edge is passed to a recursive call with a smaller t parameter, we say that edge is propagated *downward.* If the recursive call has the same t parameter then that edge is propagated *laterally.* If the edge is next examined in a recursive call with a greater t parameter it is propagated *upward.* In our algorithm only MST edges are propagated upward and all bad edges are propagated laterally. This is in contrast to [Chaz97, Chaz98b] in which bad edges at each level of recursion are grouped together and passed to the *same* recursive call. To counter the effect of bad edges propagating upward, one must ensure that the number of bad edges is much smaller. Chazelle sets ϵ to be roughly $1/\alpha$, giving the $\log \alpha$ term in the $O(m\alpha \log \alpha)$ running time.

6 Running Time

For the sake of simplicity we will postpone proving the following Lemma until Section 7.

Lemma 6.1 *Build-T runs in time $O(m \log(\frac{1}{\epsilon}) + d^2 n)$ and generates $O(\epsilon m + d^3 n)$ bad edges, for any $\epsilon < \frac{1}{2}$.*

The ϵ mentioned above is the same one used by the soft heap. We will set it to some small constant s.t. the number of bad edges is $\leq m/2$ (since we set $d = (\frac{m}{n})^{\frac{1}{4}}$, the $d^3 n$ term contributes very little).

The running time of the algorithm is given by the following recurrence. Let $c_2 m$ be the time spent building T and improving the density of various subgraphs. Let m_x be the number of edges in subgraph C_x where $x \in T$, and m_B be the number of bad edges. We have that $\sum_x m_x + m_B \leq m$ and $m_B \leq m/2$. The number of edges passed to the final recursive call is bounded by $m_B(1 + \frac{1}{D})$, where D is the density of G. This follows from the fact that G' has $< m_B + n$ edges and the density of G'' is at least a factor of D larger than G'. This was described in the previous section.

$$\begin{aligned}
T(m,t) &\leq \sum_x T(m_x, t-1) + T(m_B(1 + \tfrac{1}{D}), t) + c_2 m \\
&\leq \sum_x c_1(t-1)m_x + c_1 t m_B(1 + \tfrac{1}{D}) + c_2 m \quad \text{(Assume inductively)} \\
&\leq c_1(t-1)\tfrac{m}{2} + c_1 t \tfrac{m}{2}(1 + \tfrac{1}{D}) + c_2 m \\
&= c_1 t m(1 + \tfrac{1}{2D} - \tfrac{1}{2t} + \tfrac{c_2}{c_1 t}) \\
&\leq c_1 t m \quad \text{(for } D \geq 2t, c_1 \geq 4c_2\text{, this completes the induction)}
\end{aligned}$$

Theorem 6.1 *The minimum spanning forest of a graph with m edges and n vertices can be computed deterministically in $O(m\,\alpha(m,n))$ time.*

To prove Theorem 6.1 we need only show that $t = O(\alpha(m,n))$. By definition $\alpha(m,n) = \min\{j : A(j, \lceil \frac{m}{n} \rceil) > \log n\}$. We defined t to be minimal s.t. $A(t, \lceil (\frac{m}{n})^{\frac{1}{4}} \rceil) \geq n$. where we maintain $\frac{m}{n}$ to be at least some constant. It is straightforward to prove that

$$\forall i,j \qquad A(i,j) \geq 2^j \tag{1}$$

$$\text{and} \qquad A(i,j) \leq \log A(i+1,j) \tag{2}$$

Let $\alpha = \alpha(m,n)$ and $D = \frac{m}{n}$ be the density.

$$\begin{aligned} A(\alpha+2, D^{\frac{1}{4}}) &= A(\alpha+1, A(\alpha+2, D^{\frac{1}{4}}-1)) \\ &\geq A(\alpha+1, D) \ \ \text{(from 1,2)} \\ &\geq 2^{A(\alpha,D)} \ \ \text{(from 2)} \\ &\geq 2^{\log n} = n \ \ \text{(definition of } \alpha(m,n)\text{)} \end{aligned}$$

Hence t does not exceed $\alpha + 2$, which proves Theorem 6.1.

7 Heap Management and the Bad Edges

The need for a good heap management scheme is best exemplified by illustrating the defects in a naive scheme. Suppose that we maintained k heaps, one associated with each X_i. All external edges incident on X_i would be kept in the corresponding heap, and the effect of retracting X_k would be to mark bad all corrupted edges in X_k's heap, then meld it with X_{k-1}'s heap. Extensions would be handled in the obvious manner: edges that made the transition from external to internal would be deleted from their respective heaps, and new external edges would be inserted into the proper heap. This scheme is correct, but in the worst case nearly all edges can become bad. The problem is that as soon as bad edges are deleted from a soft heap, it is free (according to its specification) to corrupt an equal number of new edges. Thus whenever some X_i is retracted, the number of edges marked bad can be up to ϵI where I is the *total* number of insertions made to X_i's heap or any heaps melded into it.

The solution is to divide the heaps into two types; one type will contain no bad edges but will suffer many deletes, the other may contain bad edges but the number of deletes will be bounded. Consider an edge (u,v) where $u \in X_i$. This edge will be kept in either the heap $H(i)$ or some $H_j(i)$ for $j < i$. We now describe the qualifications for membership in these heaps and how they behave during extensions and retractions.

$H_j(i)$:	If (u,v) is kept in this heap then v is also incident to X_j but to no other X_ℓ, for $j < \ell < i$. If $j = 0$ this implies that v is incident to no X_ℓ for $\ell < i$.
$H(i)$:	Edge (u,v) will be kept here only if v is incident to some edge kept in $H_j(i)$ for some j. No bad edges will be kept in this heap nor will it ever meld with another.

After an extension, all newly internal edges will be deleted from their respective heaps, and newly external edges will be inserted into some heap. If an edge is incident on X_i and qualifies for heap $H(i)$ then that is where it should be inserted. Otherwise, it should be inserted into the only eligible $H_j(i)$.

After the retraction of X_k the heaps $H(k)$ and all $H_j(k)$ (for $j < k$) must be dealt with. First, mark all corrupted edges in these heaps bad. Destroy the heaps $H_{k-1}(k)$ and $H(k)$, discarding all bad edges (this is tantamount to increasing their costs to ∞). Group the remaining (uncorrupted) edges according to their endpoint outside $\mathcal{X}$. For each group, reinsert the cheapest edge into another heap (described next) and discard the rest of the edges, all of which are not in the MST of G and need not be reprocessed.

Consider an edge which was the cheapest in its group. If it, or any edge in its group was kept in $H_{k-1}(k)$, then it is eligible to be reinserted into $H(k-1)$ and should go there. The remaining edges kept in $H(k)$ should be reinserted into the only $H_j(k)$ for which they are eligible. Finally, we meld $H_j(k)$ into $H_j(k-1)$

for all j. This scheme is essentially performing bulk decrease key operations on the items held in $H(k)$ and $H_{k-1}(k)$. Recall that the soft heap has no decrease key operation of its own.

Lemma 7.1 *The number of insertions to all the heaps is $O(m)$.*

Proof: The number of first-time insertions is clearly m. We perform reinsertions only after grouping edges with a common endpoint and discarding all but the cheapest. Thus if the group size is greater than 1 we can charge the cost of reinsertion to one of the discarded edges. If the group is a singleton, notice that all edges coming from $H(k)$ are reinserted into some $H_j(k)$. These edges were kept in $H(k)$ precisely because they shared an endpoint with some edge in $H_j(k)$, so the next time these edges are up for reinsertion, they will be in a group of at least 2. Since singleton edges from some $H(k)$ are never reinserted into $H(k-1)$, we can ignore all reinsertions into the $H(\cdot)$ heaps and be off by no more than a factor of 2. $\Box$

Lemma 7.2 *Define the multiplicity of a heap to be the maximum number of edges with the same endpoint outside $\mathcal{X}$. Then the multiplicity of $H_j(i)$ is bounded by d.*

Proof: We will actually show that the multiplicity of $H_j(i)$ is bounded by h_i, the height of X_i in T, which is of course $\leq d$. This is certainly true when X_i has height 1 since all first-time insertions which would increase the multiplicity to 2 are instead directed towards $H(i)$. If X_i has height greater than 1 then its growth comes only from reinsertions and melds performed just after a retraction.

Assume inductively that just before a retraction, $H_j(k-1)$ and $H_j(k)$ have multiplicity $\leq h_{k-1}$ and $\leq h_k \leq h_{k-1} - 1$ respectively. No two edges reinserted into $H_j(k)$ have the same endpoint (one would have been discarded already), so the multiplicity of $H_j(k)$ is increased by no more than 1, bringing it to no more than h_{k-1}. The multiplicity of $\mathrm{Meld}(H_j(k), H_j(k-1))$ is just the larger of the multiplicities of $H_j(k)$ and $H_j(k-1)$, since the set of vertices adjacent to edges in $H_j(k)$ is disjoint from those adjacent to edges in $H_j(k-1)$. Thus the multiplicity of $H_j(k-1)$ post-retraction remains no more than h_{k-1}.

In the case where h_{k-1} and h_k differ by more than 1, the proof is simpler since $H_j(k-1)$ and $H_j(k)$ are not melded. $\Box$

We now restate and prove Lemma 6.1.

Build-T runs in time $O(m \log(\frac{1}{\epsilon}) + d^2 n)$ and generates $O(\epsilon m + d^3 n)$ bad edges, for any $\epsilon < \frac{1}{2}$.

Proof: By Lemma 7.1 the time for all insertions is $O(m \log(\frac{1}{\epsilon}))$, which is clearly a bound on the time for deletes and melds. For each extension we perform one findmin operation per heap and one comparison with each min-link, hence the $d^2 n$ term. Updating the min-links requires $< m$ comparisons since each time two potential min-links are compared, one is never eligible to be a min-link again. Since we are using the decision tree model, the time spent deciding where to insert edges can be ignored.

Observe that all bad edges are discarded in one of two ways. They are either still in a heap at the time of its destruction (just after a retraction), or they are deleted from a heap after an extension. The number of bad edges discarded in the first way is no more than ϵI where I is the total number of insertions. By Lemma 7.1 this is $O(\epsilon m)$. Consider all the edges deleted from heaps just after an extension. We can ignore those deleted from some $H(i)$ since all edges in such heaps are not bad (they become bad only upon retraction of X_i). The number of edges deleted from $H_j(i)$ is bounded by its multiplicity, which by Lemma 7.2 is no more than d. Since there are $< d^2$ such heaps, the total number of bad edges is $O(\epsilon m + d^3 n)$. $\Box$

8 Conclusion

The complexity of the minimum spanning tree problem could conceivably be $\Theta(m\alpha(m,n))$, though this, as would any super-linear bound, seems very unlikely. Indeed, there is still some slack in this algorithm that could be exploited.

9 Acknowledgment

I would like to thank Vijaya Ramachandran for her comments.

References

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

[Chaz97] B. Chazelle. A Faster Deterministic Algorithm for Minimum Spanning Trees. In *FOCS '97*, pp. 22–31, 1997.

[Chaz98a] 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.

[Chaz98b] B. Chazelle. A Deterministic Algorithm for Minimum Spanning Trees. Undated manuscript, received February 1998.

[Chaz99] B. Chazelle. A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity. NECI Tech Report 99–099 (2-062-0347-97005), July 1999.

[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.

[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. An Optimal Minimum Spanning Tree Algorithm. Tech. Report TR99-17, Univ. of Texas at Austin, 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.