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.