A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity*

Bernard Chazelle†

NECI Research Tech Report 99-099 (July 1999)
Journal of the ACM, 47(6), 2000, pp. 1028–1047.

Abstract

A deterministic algorithm for computing a minimum spanning tree of a connected graph is presented. Its running time is $O(m\alpha(m,n))$, where α is the classical functional inverse of Ackermann's function and n (resp. m) is the number of vertices (resp. edges). The algorithm is comparison-based: it uses pointers, not arrays, and it makes no numeric assumptions on the edge costs.

1 Introduction

The history of the minimum spanning tree (MST) problem is long and rich, going as far back as Borůvka's work in 1926 [1, 9, 13]. In fact, MST is perhaps the oldest open problem in computer science. According to Nešetřil [13], "this is a cornerstone problem of combinatorial optimization and in a sense its cradle." Textbook algorithms run in $O(m \log n)$ time, where n and m denote, respectively, the number of vertices and edges in the graph. Improvements to $O(m \log\log n)$ were given independently by Yao [18] and by Cheriton and Tarjan [4]. In the mid-eighties, Fredman and Tarjan [6] lowered the complexity to $O(m\beta(m,n))$, where $\beta(m,n)$ is the number of log-iterations necessary to map n to a number less than m/n. In the worst case, $m = O(n)$ and the running time is $O(m \log^* m)$. Soon after, the complexity was further reduced to $O(m \log \beta(m,n))$ by Gabow et al. [8]. Recently, Karger et al. [10] have discovered a randomized algorithm with linear expected complexity. If the edge costs are integers and the model allows bucketing and bit manipulation, it is possible to solve the problem in linear time deterministically, as was shown by Fredman and Willard [7]. To achieve a similar result in a comparison-based model has long been a high-priority objective in the field of algorithms. The reason why is, first, the illustrious history of the MST problem; second, the fact that it goes to the heart of matroid optimization.

This paper does not resolve the MST problem, but it takes a significant step towards a solution and charts out a new line of attack. Our main result is a deterministic algorithm for computing the MST of a connected graph in time $O(m\alpha(m,n))$, where α is the functional

*A preliminary version of this paper appeared in *"A faster deterministic algorithm for minimum spanning trees"*, by B. Chazelle, Proc. 38th Ann. IEEE Symp. Found. Comp. Sci. (1997), 22–31. This work was supported in part by NSF Grant CCR-93-01254, NSF Grant CCR-96-23768, ARO Grant DAAH04-96-1-0181, and NEC Research Institute.

†Department of Computer Science, Princeton University, and NEC Research Institute, chazelle@cs.princeton.edu and chazelle@research.nj.nec.com

inverse of Ackermann's function defined in [15]. The algorithm is comparison-based: it uses pointers, not arrays, and it makes no numeric assumptions on the edge costs.

In addition to providing a new complexity bound, the larger contribution of this paper is to introduce a nongreedy approach to matroid optimization, which we hope will prove useful beyond minimum spanning trees. The key idea is to compute sub-optimal independent sets in a *nongreedy* fashion, and then progressively improve upon them until an optimal basis is reached. Specifically, an approximate priority queue, called a soft heap [2], is used to construct a good, but not necessarily minimum, spanning tree. The quality of the tree is progressively refined until an MST is finally produced.

Theorem 1.1 *The* MST *of a connected graph with* n *vertices and* m *edges can be computed in* $O(m\alpha(m,n))$.

How good is the $m\alpha(m,n)$ bound? Is it optimal? Doubtful. Is it natural? Definitely. Given a spanning tree T, to verify that it is minimum can be done in linear time [5, 11, 12]: the problem is to check that any edge outside T is the most expensive along the cycle it forms with T. With real costs, this can be viewed as a problem of computing over the semigroup $(\mathbf{R},\max)$ along paths of a tree. Interestingly, this problem requires $\Omega(m\alpha(m,n))$ time over an arbitrary semigroup [3, 16]. This lower bound suggests that in order to improve upon our algorithm specific properties of $(\mathbf{R},\max)$ will have to be exploited. This is done statically in [5, 11, 12]. We speculate that an answer might come from a dynamic equivalent.

This paper is organized as follows. This section proceeds with a brief overview of the algorithm (§1.1), a discussion of the key concept of edge corruption (§1.2) and a review of soft heaps (§1.3). In §2 we introduce the main structural invariants of the algorithm and we discuss its components in detail. We prove its correctness in §3, and we analyze its complexity in §4 and §5.

1.1 The Algorithm at a Glance

The input is a connected, undirected graph G, with no self-loops, where each edge e is assigned a cost $c(e)$. These costs are assumed to be distinct elements from a totally ordered universe (a nonrestrictive assumption, as ties can be broken arbitrarily). As is well known, the MST of such a graph is unique. A subgraph C of G is *contractible* if its intersection with MST (G) is connected. What makes this notion useful is that MST (G) can be assembled directly from MST (C) and MST (G'), where G' is the graph derived from G by contracting C into a single vertex. Previous algorithms identify contractible subgraphs on the fly as the explored portion of the MST grows. Our first idea is to reverse this process, ie, to certify the contractibility of C *before* computing its MST. The advantage should be obvious. Computing MST (C) is bound to be easier if we already know that C is contractible, for then we need to look only at edges with *both* endpoints in C. Otherwise, we must also visit edges with one endpoint in C. This makes genuine divide-and-conquer possible. The challenge is, how can we discover a contractible subgraph without computing its MST at the same time? Current methods are not helpful, and that is why we turn to soft heaps.

To compute MST (G), first we decompose G into vertex-disjoint contractible subgraphs of suitable size. Next, we contract each subgraph into a single vertex to form a minor[1] of G, which we similarly decompose into vertex-disjoint contractible subgraphs, etc. We iterate on

[1] A minor is a graph derived from a sequence of edge contractions and their implied vertex deletions.

this process until G becomes a single vertex. This forms a hierarchy of contractible subgraphs, which we can model by a perfectly balanced tree $\mathcal{T}$: its leaves are the vertices of G; an internal node z with children $\{z_i\}$ is associated with a graph C_z whose vertices are the contractions of the graphs $\{C_{z_i}\}$. Each level of $\mathcal{T}$ represents a certain minor of G, and each C_z is a contractible subgraph of the minor associated with the level of its children. In this association, the leaf level corresponds to G while the root corresponds to the whole graph G contracted into a single vertex. Once $\mathcal{T}$ is available, we compute the MST of each C_z recursively. Because the C_z's are contractible, glueing together the trees MST (C_z) produces MST (G).

We have considerable freedom in choosing the height d of $\mathcal{T}$ and the number n_z of vertices of each C_z (which, as we should note, is also the number of children of z). The tree $\mathcal{T}$ is then computed in $O(m + d^3 n)$ time. If d is chosen large then the n_z's can be kept small; the recursive computation within each C_z is very fast but building $\mathcal{T}$ is slow. Conversely, a small height speeds up the construction of $\mathcal{T}$ but, by making the C_z's bigger, it makes the recursion more expensive. This is where Ackermann's function comes into play by providing the best possible tradeoff. Let d_z denote the height of z in $\mathcal{T}$, which is defined as the maximum number of edges from z to a leaf below. A judicious choice is $n_z = S(t,1)^3 = 8$ if $d_z = 1$, and $n_z = S(t-1, S(t, d_z - 1))^3$ if $d_z > 1$, where $t > 0$ is minimum such that $n \le S(t,d)^3$, with $d = c\lceil (m/n)^{1/3} \rceil$, for a large enough constant c, and

$$\begin{cases} S(1,j) &=& 2j, \text{ for any } j > 0; \\ S(i,1) &=& 2, \text{ for any } i > 0; \\ S(i,j) &=& S(i,j-1)S(i-1,S(i,j-1)), \text{ for any } i,j > 1. \end{cases}$$

We easily prove by induction that the *expansion* of C_z relative to G (ie, the subgraph of G whose vertices end up in C_z) has precisely $S(t,d_z)^3$ vertices. This follows from the identity

$$S(t,d_z-1)^3 n_z = S(t,d_z-1)^3 S(t-1,S(t,d_z-1))^3 = S(t,d_z)^3.$$

If we assume for simplicity that n is actually equal to $S(t,d)^3$, the previous statement is true for all z, including the root of $\mathcal{T}$. It follows immediately that d coincides with the height of $\mathcal{T}$. Let us prove by induction on t that if the number of vertices of G satisfies $n = S(t,d)^3$, then MST (G) can be computed in $bt(m + d^3 n)$ time, where b is a large enough constant. For the sake of this overview, we omit the basis case $t = 1$. To apply the induction hypothesis on the computation cost of MST (C_z), we note that the number of vertices of C_z satisfies $n_z = S(t-1, S(t,d_z-1))^3$, so by visual inspection we see that, in the formula above, t must be replaced by $t-1$ and d by $S(t,d_z-1)$. This gives a cost of $b(t-1)(m_z + S(t,d_z-1)^3 n_z)$, where m_z is the number of edges in C_z. Summing up over all internal nodes $z \in \mathcal{T}$ allows us to bound the computation costs of all the MST (C_z)'s by

$$b(t-1)\Big(m + \sum_z S(t,d_z-1)^3 S(t-1,S(t,d_z-1))^3\Big) = b(t-1)\Big(m + \sum_z S(t,d_z)^3\Big),$$

which is

$$b(t-1)\Big(\, m \; + \; \# \text{ vertices in expansion of } C_z \,\Big) = b(t-1)(m + dn).$$

Adding to this the time claimed earlier for computing $\mathcal{T}$ yields, for b large enough,

$$b(t-1)(m + dn) + O(m + d^3 n) \le bt(m + d^3 n),$$

which proves our claim. As we show later, our choice of t and d implies that $t = O(\alpha(m,n))$, and so the running time of the MST algorithm is $O(m\alpha(m,n))$.

This informal discussion leads to the heart of the matter: how to build $\mathcal{T}$ in $O(m + d^3 n)$ time. We have swept under the rug a number of peripheral difficulties, which in the end results in an algorithm significantly more sophisticated than the one we have outlined. Indeed, quite a few things can go wrong along the way; none as serious as *edge corruption*, an unavoidable byproduct of soft heaps which we discuss next.

1.2 Edge Corruption

In the course of computing $\mathcal{T}$, certain edges of G become *corrupted*, meaning that their costs are raised. To make matters even worse, the cost of a corrupted edge can be raised more than once. The reason for all this has to do with the soft heap, the approximate priority queue which we use for identifying contractible subgraphs (more on this later). Some corrupted edges cause trouble, while others do not. To understand this phenomenon—one of the most intriguing aspects of the analysis—we must discuss how the overall construction of $\mathcal{T}$ is scheduled.

It would be tempting to build $\mathcal{T}$ bottom-up level by level, but this would be a mistake. Indeed, it is imperative to maintain a connected structure. So, instead, we compute $\mathcal{T}$ in postorder: children first, parent last. Let z be the current node visited in $\mathcal{T}$, and let $z_1, \ldots, z_k = z$ be the *active path*, ie, the path from the root z_1. The subgraphs $C_{z_1}, \ldots, C_{z_k}$ are being currently assembled, and as soon as C_{z_k} is ready it is contracted into one vertex, which is then added to $C_{z_{k-1}}$. A minor technical note: if z_{i+1} is the leftmost child of z_i, ie, the first child visited chronologically, then C_{z_i} does not yet have any vertex, and so it makes sense to omit z_i from the active path altogether. The benefit of such shortcuts is that by avoiding one-child parents, we ensure that each of $C_{z_1}, \ldots, C_{z_k}$ now has at least one vertex, which itself is a vertex of G or a contraction of a connected subgraph.

As long as an edge of G has exactly one vertex in $C_{z_1} \cup \cdots \cup C_{z_k}$, it is said to be of the *border* type. Of course, the type of an edge changes over time: successively, unvisited, border, $\in C_z$ along active path, contracted. Corruption can strike edges only when they are of the border type (since it is then that they are in soft heaps). At first, corruption might seem fatal: if all edges become corrupted, aren't we solving an MST problem with entirely wrong edge costs? But in fact, rather miraculously, corruption causes harm only in one specific situation: we say that an edge becomes *bad* if it is a corrupted border edge at the time its incident C_z is contracted into one vertex. Once bad always bad, but like any corrupted edge its cost can still rise. Remarkably, it can be shown that if no edges ever turned bad, the algorithm would behave as though no corruption ever occurred, regardless of how much actually took place. Our goal, thus, is to fight badness rather than corruption. We are able to limit the number of bad edges to within $m/2 + d^3 n$. The number of edges corrupted but never bad is irrelevant.

Once $\mathcal{T}$ is built, we restore all the edge costs to their original values, and we remove all the bad edges. We recurse within what is left of the C_z's to produce a spanning forest F. Finally, we throw back in the bad edges and recurse again to produce the minimum spanning tree of G. There are subtleties in these various recursions, which we explain in the next section. We close this overview with a quick sketch of the soft heap.

1.3 The Soft Heap

A simple priority queue, called a *soft heap*, is the main vehicle for selecting good candidate edges. The data structure stores items with keys from a totally ordered universe, and supports the operations:

- `create` $(\mathcal{S})$: Create an empty soft heap $\mathcal{S}$.
- `insert` $(\mathcal{S}, x)$: Add new item x to $\mathcal{S}$.
- `meld` $(\mathcal{S}, \mathcal{S}')$: Form a new soft heap with the items stored in $\mathcal{S}$ and $\mathcal{S}'$ (assumed to be disjoint), and destroy $\mathcal{S}$ and $\mathcal{S}'$.
- `delete` $(\mathcal{S}, x)$: Remove item x from $\mathcal{S}$.
- `findmin` $(\mathcal{S})$: Return an item in $\mathcal{S}$ with the smallest key.

A soft heap may, at any time, increase the value of certain keys. Such keys, and by extension, the corresponding items, are called *corrupted*. Naturally, `findmin` returns the item of minimum current (not original) key. A parameter ε controls the amount of corruption. The soft heap is introduced in the companion paper [2]. We summarize its main features below.

Theorem 1.2 [2]. *Beginning with no prior data, consider a mixed sequence of operations that includes n inserts. For any $0 < \varepsilon \leq 1/2$, a soft heap with error rate ε supports each operation in constant amortized time, except for insert, which takes $O(\log 1/\varepsilon)$ time. The data structure never contains more than εn corrupted items at any given time.*

2 The MST Algorithm

We begin with a review of a well-known procedure known as a *Borůvka phase.* Pick a vertex and contract the cheapest edge incident to it. As is well known, this transforms G into a graph with the same set of non-MST edges. We clean up the graph, if necessary, by removing the self-loops that might have been created in the process. (Recall that the graph may have multiple edges, to begin with.) We can generalize this process by selecting the cheapest edge incident to each vertex in G, and contracting each connected component in the subgraph of selected edges. Again, we clean up afterwards. This is called a Borůvka phase; it is easily performed in $O(m)$ time in one pass through the graph. The number of vertices drops by at least a factor of two. This simple procedure is useful for reducing the number of vertices and it plays an important role in our MST algorithm.

To compute the MST of a connected graph G with n vertices and m edges, we call the function `msf` (G, t) for the parameter value

$$t = \min\{\, i > 0 \mid n \leq S(i,d)^3 \,\}, \tag{1}$$

where $d = c\lceil (m/n)^{1/3} \rceil$. Throughout this paper, c denotes a large enough integral constant. The function `msf` takes as input an integer $t \geq 1$ and a graph with distinct edge costs, and returns its minimum spanning forest (MSF). As this suggests, we no longer assume that the input graph should be connected. As will soon be clear, dropping the connectivity assumption is mandated by the recursion invariants. The choice of t is arbitrary and affects only the running time. The particular setting in (1) is the best possible, but because of the recursive nature of the algorithm it is necessary to allow t to vary as a parameter.

$\texttt{msf}(G, t)$

[1] If $t = 1$ or $n = O(1)$, return MSF (G) by direct computation.

[2] Perform c consecutive Borůvka phases.

[3] Build $\mathcal{T}$ and form the graph B of bad edges.

[4] Set $F \leftarrow \bigcup_{z \in \mathcal{T}} \texttt{msf}(C_z \setminus B, t-1)$.

[5] Return $\texttt{msf}(F \cup B, t) \cup \{$ edges contracted in [2] $\}$.

Steps [1,2]: Borůvka Phases

The case $t = 1$ is special. We solve the problem in $O(n^2)$ time by performing Borůvka phases until G is contracted to a single vertex. If we are careful to remove multiple edges (by keeping only the cheapest edge in each group) and to keep the graph free of multiple edges, we easily carry out all phases in a total of $O(n^2 + (n/2)^2 + \cdots) = O(n^2)$ time. If, on the other hand, $n = O(1)$, we compute the MST in $O(m)$ time.

We apply the remainder of the algorithm to each connected component of G separately. For the purpose of this explanation, therefore, we might as well assume that G is connected. The aim of step [2] is simply to reduce the number of vertices. The Borůvka phases transform G into a graph G_0 with $n_0 \leq n/2^c$ vertices and $m_0 \leq m$ edges. This transformation requires $O(n + m)$ time.

Step [3]: Building the Hierarchy $\mathcal{T}$

With $t > 1$ specified, so is the target size n_z of each C_z, ie, $n_z = S(t-1, S(t, d_{z_k} - 1))^3$, where d_{z_k} is the height of $z \in \mathcal{T}$. We use the word "target" because the algorithm sometimes fails to meet those intended sizes. This is no mere technicality but rather a deep structural aspect of the algorithm, whose explanation is best postponed at this point.

To get things off the ground is routine, and so we discuss the algorithm in mid-action. Let $z_1, \ldots, z_k = z$ denote the active path. As shown in Fig. 1, the subgraphs $C_{z_1}, \ldots, C_{z_k}$ currently under construction are linked together in a cost-decreasing *chain* of edges. The algorithm occasionally *discards* edges from G_0. (The word "discarded" has a technical meaning and refers to edges removed from consideration in specific circumstances explained below.) Each graph C_z includes all the non-discarded edges of G_0 whose endpoints map into it, therefore to specify a given C_z it suffices to provide its vertices. The invariants below hold at any time with respect to the current graph G_0 (ie, the original G_0 minus all the edges previously discarded). We need the important concept of a *working cost*: at any time, the working cost of an edge is its current cost if the edge is bad, and its original cost otherwise. Thus, we distinguish among three types of cost: original, current, and working.

INV1 For all $i < k$, we keep an edge (called a *chain-link*) joining C_{z_i} to $C_{z_{i+1}}$ whose current cost is (i) at most that of any border edge incident to $C_{z_1} \cup \cdots \cup C_{z_i}$ and (ii) less than the working cost of any edge joining two distinct C_{z_j}'s

($j \leq i$). To enforce the latter condition efficiently, we maintain a *min-link*, if it exists, for each pair $i < j$: this is an edge of minimum working cost joining C_{z_i} and C_{z_j}.

INV2 For all j, the border edges (u, v) with $u \in C_{z_j}$ are stored either in a soft heap, denoted $H(j)$, or in one, $H(i, j)$, where $0 \leq i < j$. No edge appears in more than one heap. Besides the condition $u \in C_{z_j}$, membership in $H(j)$ implies that v is incident to at least one edge stored in some $H(i, j)$; membership in $H(i, j)$ implies that v is also adjacent to C_{z_i} but not to any C_{z_l} in between ($i < l < j$). We extend this to $i = 0$ to mean that v is incident to no C_{z_l} ($l < j$). All the soft heaps have error rate $1/c$.

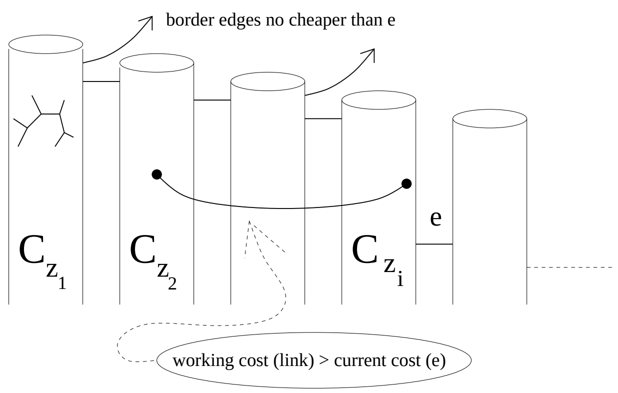

Figure 1: *The chain of subgraphs along the active path, with edge costs indicated by vertical height. Recall that working costs are current for bad edges and original for others.*

The main thrust of INV1 is to stipulate that the active path corresponds to a descending chain of edges connecting various C_z's. This descending property is essential for ensuring contractibility. The chain-link between C_{z_i} and $C_{z_{i+1}}$ is the edge that contributes $C_{z_{i+1}}$ its first vertex. Subsequently, as $C_{z_{i+1}}$ grows, lower-cost edges might connect it to C_{z_i} and so, in general, the chain-link is likely to be distinct from the min-link between C_{z_i} and $C_{z_{i+1}}$.

Why do we need so many heaps? The short answer is, to fight badness. The problem is that when bad edges are deleted from a soft heap, Theorem 1.2 allows the same amount of corruption to be produced anew. These newly corrupted edges might then turn bad and be deleted. Cycling through this process could have disastrous effects, perhaps even making every single edge bad. We use separate heaps in order to create a buffering effect to counter this process. This mechanism relies on structural properties of minimum spanning trees and a subtle interplay among heaps.

The tree $\mathcal{T}$ is built in a postorder traversal driven by a stack whose two operations, pop and push, translate into, respectively, a *retraction* and an *extension* of the active path.

- RETRACTION: This happens for $k \geq 2$ when the last subgraph C_{z_k} has attained its target size, ie, its number n_{z_k} of vertices has reached the value of $S(t-1, S(t, d_{z_k} - 1))^3$, where d_{z_k} is the height of z_k in $\mathcal{T}$. Recall that this target size is also the number of children of z_k in $\mathcal{T}$. In the particular case, $d_{z_k} = 1$, the target size is set to $S(t, 1)^3 = 8$.

The subgraph C_{z_k} is contracted and becomes a new vertex of $C_{z_{k-1}}$. That vertex is joined to $C_{z_{k-1}}$ by the chain-link (plus maybe other edges) between $C_{z_{k-1}}$ and (now contracted) C_{z_k}; these edges are *not* contracted. As a net result, $C_{z_{k-1}}$ gains one vertex and one or several new edges, and the end of the active path is now z_{k-1}. A minor technicality: because of the aforementioned shortcuts taken in forming the active path to avoid zero-vertex C_z's, we must add a new node between z_{k-1} and z_k in case their heights differ by more than one. (The numbering of the C_z's is implicit and no updating is necessary; heights are understood here with respect to the full—not partially constructed—tree $\mathcal{T}$.) Maintaining INV1 in $O(k)$ time is straightforward; INV2 is less so. Here is what we do:

The heaps $H(k)$ and $H(k-1,k)$ are destroyed. All corrupted edges are discarded. (Note that these edges, if not bad already, become so now.) The remaining items are partitioned into subsets, called *clusters*, of edges that share the same endpoint outside the chain. For each cluster in turn, select the edge (r,s) of minimum current cost and discard the others (if any). Next, insert the selected edge into the heap implied by INV2. Specifically, if (r,s) comes from $H(k)$ and shares s with an edge in $H(k-1,k)$, or if it comes from $H(k-1,k)$, then by INV2 it also shares s with an edge in some $H(i,k-1)$ already, and so it can be inserted into $H(k-1)$. Otherwise, (r,s) comes from $H(k)$ and so, by INV2, it shares s with an edge in some $H(i,k)$, with now $i<k-1$. The edge (r,s) should be inserted into $H(i,k)$. Finally, for each $i<k-1$, meld $H(i,k)$ into $H(i,k-1)$.

Two remarks: (i) by inserting (r,s) into $H(i,k)$, we force into the heap at least a second edge pointing to s; (ii) since $H(i,k)$ is melded into $H(i,k-1)$, we could have inserted (r,s) into $H(k-1)$, instead of $H(i,k)$, and still maintain INV2. We choose not to because of the risk of seeing edges hopping between $H(\star)$'s at each retraction, which would be too costly.

- EXTENSION: Do `findmin` on *all* the heaps, and retrieve the border edge (u,v) of minimum current cost $c(u,v)$. This is called the *extension edge*. Of all the min-links of working cost at most $c(u,v)$, find the one (a,b) incident to the C_{z_i} of smallest index i. If such an edge indeed exists, we perform a *fusion*: we contract the whole subchain $C_{z_{i+1}} \cup \cdots \cup C_{z_k}$ into a (Fig. 2). It is best to think of this as a two-step process: first, contract all the edges with both endpoints in $C_{z_{i+1}} \cup \cdots \cup C_{z_k}$. By abuse of notation, call b the resulting vertex, and now contract the edge(s) joining a to b.

  Next, we update all relevant min-links, which is easily done in $O(k^2)$ time. To update heaps we generalize the retraction recipe in the obvious manner: We extend the destruction of heaps to include not just $H(k)$ and $H(k-1,k)$ but $H(i+1),\dots,H(k)$, and all $H(j,j')$, $i \le j < j'$. First, we discard all corrupted edges from those heaps since they are now bad. Then, we regroup the remaining edges into clusters and, for each one in turn, we reinsert the edge (r,s) of minimum current cost and discard the others. As before, we distinguish between two cases:

  1. If (r,s) comes from some $H(j,j')$ or $H(j')$, but in the latter case shares s with an edge in $H(j,j')$, where $i \le j < j'$, then by iterative application of INV2 it also shares s with an edge (r',s) in some $H(h,l)$, with $h<i\le l$. As we explain below, the edge (r',s) is to migrate into $H(h,i)$ through melding, if it is not there already, ie, if $l>i$, therefore by INV2 we can insert (r,s) into $H(i)$.
  2. Otherwise, (r,s) comes from $H(j)$, where $i<j$, and it shares s with an edge in some $H(h,j)$, with now $h<i$. We insert the edge (r,s) into $H(h,j)$.

Finally, for each h, j with $h < i < j$, we meld $H(h,j)$ into $H(h,i)$. Observe that by Inv1(i) and the choice of the vertex a, it cannot be further down the chain than u; hint: consider the chain-link leaving the chain from the same C_z as (u,v). Therefore, whether u originally belonged to the last C_z in the chain, it now does. Regardless of whether a fusion took place, we extend the chain by making v into the single-vertex C_{z_k} and the extension edge (u,v) into the chain-link incident to it. The end of the active path is now z_k. Note that this is not the same z_k as before: without fusion, the new value of k is the old one plus one; with fusion, it is $i+1$.

Old border edges incident to v cease to be of the border type: delete them from their respective heaps, and find among them the min-link between v and each of $C_{z_1}, \ldots, C_{z_{k-1}}$. Insert the new border edges incident to v into the appropriate $H(i,k)$; in case of multiple edges, keep only the cheapest in each group and discard the others.

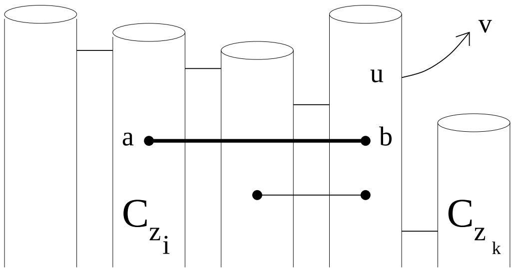

Figure 2: *Extension: all three subgraphs right of C_{z_i} collapse into b and then into a.*

Having explained how retractions and extensions work, we review the construction of $\mathcal{T}$. At any given node z_k of height at least 1, we perform extensions (and their accompanying fusions) as long as we can (stack push), stopping only when the size condition for retraction at that node has been met (stack pop). There is no retraction condition for the root z_1, and so the algorithm stops only when border edges run out and extensions are no longer possible. Assume that no fusion ever takes place. From the identity

$$S(t, d_z) = S(t, d_z - 1)S(t-1, S(t, d_z - 1)),$$

it immediately follows by induction that, for any internal z distinct from the root, the expansion of C_z has exactly $n_z S(t, d_z - 1)^3$ vertices, which is $S(t, d_z)^3$; recall that the expansion of C_z consists of the vertices of G_0 mapping into C_z, plus all the edges of G_0 joining pairs of them. Fusions muddy the waters by forcing contractions before the target size has been reached and also by creating arbitrarily large expansions. As we shall see, however, any C_z whose expansion exceeds its allowed size is naturally broken down into subgraphs of the right size, which then can be treated separately.

Remarks:

- A fusion is *not* a retraction into C_{z_i}. Because the edge (a,b) is contracted, too, and does not become an edge of C_{z_i}, a fusion, unlike a retraction, does not increase the number of vertices in C_{z_i}. A fusion reflects the difficulty we may have to "grow" to its proper size the subgraph C_z of the last node z of the active path. The solution is to contract "just enough" to be able to resume the extension from that (new) last node. For the algorithm designer, fusions are a nuisance but for the running time, they are a blessing. In fact, one could imagine the entire algorithm reduced to one giant fusion, and computing the MST in linear time.

- The formation of clusters is a way of discarding edges once and for all. Because no edge is ever discarded from $H(0, j)$, the set of non-discarded edges always spans all of G_0 and the algorithm never terminates prematurely with vertices still unexplored. Recall that the graph G and, hence, G_0 are assumed connected for the purpose of this discussion, but in case they are not, we simply repeat the construction of $\mathcal{T}$ for each connected component.

- The algorithm needs to keep track of the bad edges in order to form B in step [3]. Badness occurs to the corrupted border edges incident to C_{z_k} (in retraction) or $C_{z_{i+1}} \cup \cdots \cup C_{z_k}$ (in fusion). All such edges are either explicitly examined (and discarded) or they belong to heaps that are being melded: $H(i,k)$ into $H(i,k-1)$ in a retraction or possibly several $H(h,j)$ into the corresponding $H(h,i)$ in a fusion. A soft heap gives ready access to its corrupted items, so we can mark the relevant edges bad. To make this cost negligible, we ensure that edges are marked only once by, for example, stringing all the non-bad corrupted edges together in a linked list.

- In step [3], the graph B takes its vertices from G_0, but in the next step an edge of B has its endpoints in C_z (and so, not necessarily, in G_0). This minor ambiguity simplifies the notation and is easily resolved from the context.

Step [4]: Recursing in Subgraphs of $\mathcal{T}$

Having built the tree $\mathcal{T}$, we consider each node z: recall that C_z does not include any of the discarded edges. Let $C_z \setminus B$ denote the subgraph of C_z obtained by removing all the bad edges. (Not all bad edges may have been discarded; in fact, bad edges can be selected as extension edges and thus play a direct role in building the C_z's.) We apply the algorithm recursively to $C_z \setminus B$ after resetting all edge costs to their original values, and decrementing the parameter t by one. The only effect of this parameter change is to modify the target sizes for the new trees $\mathcal{T}$ to be built. The output F is a set of edges joining vertices in C_z, but once again we maintain this convenient ambiguity which allows us to treat them as edges of G_0 in step [5].

The correspondence between the vertices of C_z and the children of z should be obvious, were it not for the presence of fusions. Consider the first fusion into vertex a of C_z. Prior to it, vertex a corresponds to a child v of z, meaning that it is the contraction of C_v. What happens after the fusion? As far as a is concerned in step [4] the answer is: nothing. Vertex a still corresponds to the same C_v, and step [4] recurses in $C_v \setminus B$. Next, we treat the part of $\mathcal{T}$ corresponding to the subchain $C_{z_{i+1}} \cup \cdots \cup C_{z_k}$ as a hierarchy of its own. Further fusions into a are handled in the same way. In the end, vertex a is the contraction of not just C_v but also of a number of subchains of the form $C_{z_{i+1}} \cup \cdots \cup C_{z_k}$ equal to the number of fusions.

Going back to a particular fusion of b into a, of all the edges currently joining a and (contracted) b we retain in F the original min-link (a, b) if it is not bad, else none of them. Note that retaining in F only one edge of the form (a, b) per fusion is the same as solving the MSF problem, relative to original costs, for the group of non-bad, non-discarded multiple edges joining a and b. The reason is that if the min-link is not bad, then its working cost is also its original cost and it is minimum in the group: consequently, the edge is the MST of that group. If the min-link is bad, we do not need to include any edge of the group into F, since bad edges are all reprocessed in step [5]. (As we show in the proof of correctness below, we need not be concerned with non-bad discarded edges joining a and b: they are dominated in original cost by other edges of the form (a,b) that are themselves bad or not discarded, and hence, processed in this step or the next.)

Slightly anticipating the complexity discussion, we note that because each subchain individually stays within its mandated size, their potential proliferation does not increase the per-edge complexity of the algorithm. Fusions might muddy the picture a little but, from a complexity viewpoint, in fact the more of them the better. Think of the ultimate case, where the graph G_0 is made of a single path of edges with costs in increasing order, together with an arbitrary number of other edges of much higher cost. Using a soft heap with zero error rate (for illustrative purposes only), every extension gives rise to a fusion, and every edge of the path is discovered as a fusion edge of the form (a,b). The tree $\mathcal{T}$ remains of height one and step [4] is trivial.

Step [5]: The Final Recursion

In step [3], we collected all the bad edges created during the construction of $\mathcal{T}$ and we formed the graph B. We now add to it the edges of F to assemble the subgraph $F \cup B$ of G_0. Again, we emphasize the fact that the vertices of this graph are original vertices of G_0 and not the contracted endpoints from which they might have emerged during the construction of $\mathcal{T}$. Applying the same sort of transfer, the output of $\texttt{msf}(F \cup B, t)$ is now viewed as a set of edges with endpoints in G, not in G_0. Adding the edges contracted in step [2] produces the MST of G.

3 Correctness

We prove by induction on t and n that $\texttt{msf}(G,t)$ computes the MSF of G. Since the algorithm iterates through the connected components separately, we can again assume that G is connected. Because of step [1] we can obviously assume that $t > 1$ and n is larger than a suitable constant. Borůvka phases contract only MST edges, therefore, by induction on the correctness of msf, the output of step [5] is, indeed, MST(G), provided that any edge e of G_0 outside $F \cup B$ is also outside MST(G_0). In other words, the proof of correctness of our MST algorithm will be complete once we prove the following:

Lemma 3.1 *If an edge of G_0 is not bad and lies outside F, then it lies outside* MST(G_0).

In the lemma, all costs are understood as original. Of course, this innocent-looking statement is the heart of the matter, since it pertains directly to the hierarchy $\mathcal{T}$. To begin with, we must verify the two main invariants. Using heaps to choose extension edges, and hence chain-links, ensures INV1(i); similarly, we resort to fusions simply to maintain INV1(ii). Observe that if an extension edge is corrupted but not bad, then it becomes a chain-link whose current cost exceeds its working cost, so the distinction between working and current in INV1 is not meaningless. Intuitively, (i) reflects the structure provided by the heaps, and (ii) the structure needed for contractibility. Only border edges can be discarded, so the discarding process itself cannot violate these invariants. As we discussed earlier, INV2 is preserved through our updating of the heaps. We must now show why maintaining these invariants produces contractible C_z's.

Contractibility is defined in terms of MST, a global notion. Fortunately, we can certify by local means that the subgraph C of G_0 spanned by a subset of the vertices is contractible. Indeed, it suffices to check that C is *strongly contractible*, meaning that for every pair of edges e, f in G_0, each with exactly one vertex in C, there exists a path in C that connects e to f along which no edge exceeds the cost of both e and f. This implies contractibility (but not the other way around). Why? We argue by contradiction, assuming that C is not contractible. By definition, $C \cap \mathrm{MST}(G_0)$ must have more than one connected component. Consider a shortest path π in $\mathrm{MST}(G_0)$ that joins two distinct components. The path has no edge in C (else it could be shortened) and it has more than one edge (else it would be in C), so its end-edges e and f are distinct and each has exactly one endpoint in C. Any path in C joining e and f forms a cycle with π and by elementary properties of MST, the most expensive cycle edge is outside $\mathrm{MST}(G_0)$, ie, outside of π, and hence in C. This contradicts the assumption and proves the claim. The proof extends easily to accommodate nondistinct edge costs.

Lemma 3.2 *Consider the subgraph C_z at the time of its contraction. With respect to working costs, C_z is strongly contractible and the same holds of every fusion edge (a, b).*

Proof: The lemma refers to the edges present in C_z and in its neighborhood at the time C_z is contracted: it does not include the edges of G_0 that have been discarded (in fact the lemma is false otherwise). The graph C_z is formed by incrementally adding vertices via retractions. Occasionally, new neighboring edges are added by fusion into some $a \in C_z$. Because C_z does not contain border edges, edge discarding never takes place within it, and so it grows monotonically. (This does not mean, of course, that C_z includes all the edges of the original G_0 that join pairs of vertices in it.) Assume for now that no fusion occurs. Each retraction has a unique chain-link (ie, an extension edge) associated with it, and together they form a spanning tree of C_z. Thus, given any two edges e, f, each with exactly one endpoint in C_z, the tree has a unique path π joining (but not including) them. Let $g = (u, v)$ be the edge of π of highest current cost ever; break ties, if necessary, by choosing as g the last one selected for extension chronologically. As usual, we make the convention that v is the endpoint outside the chain at the time of extension. Along π, the vertex u lies between v and one of the two edges, say, e (Fig. 3). Throughout this proof the term "working" is to be understood at the time right after C_z is contracted, while "current" refers to the time when g is selected as a new chain-link (u, v) through extension. We claim that the working cost of e is at least the current cost of g. Since the working cost of no edge in π can ever exceed the current cost of g, the lemma follows.

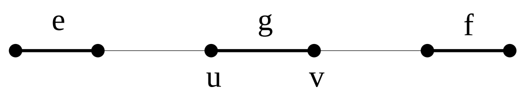

Figure 3: *Proving contractibility.*

We prove the claim. If e currently joins C_z to some other $C_{z'}$, it follows from INV1(ii). Otherwise, let e' be the first (current) border edge encountered along the path from g to e.

By $\textsc{Inv}1(\mathrm{i})$, its current cost is at least that of g, and so by our choice of g, we have $e' \notin \pi$, and hence, $e' = e$. Consequently, e currently is and still will be a border edge when C_z is contracted. If it is in a corrupted state then, it becomes bad after the contraction (were it not so already) and so, by definition, its working cost is at least its current cost (it could be higher); otherwise, both costs coincide with the original one. In both cases, the claim is true.

To deal with a fusion into C_z, we should think of it as a two-step process: (i) a sequence of retractions involving, successively, $C_{z_k}, C_{z_{k-1}}, \ldots, C_{z_{i+1}}$, where in this notation $C_z = C_{z_i}$, and (ii) the contraction of (a,b) into a. For the purpose of this discussion, let us run the algorithm right until the time C_z is contracted, while skipping step (ii) in all fusions into C_z. Then, as far as C_z is concerned its evolution is indistinguishable from the no-fusion case discussed earlier, and the same result applies. Executing all delayed applications of step (ii) now results in contracting a number of edges already within C_z, which therefore keeps C_z strongly contractible. This proves the first part of the lemma.

Now, going back to the normal sequencing of the algorithm, consider the min-link (a,b) right before step (ii) in a fusion into C_z. By construction, no other edge incident to (contracted) b is cheaper than (a,b) relative to working costs; remember that all corrupted border edges incident to b become bad, and so working and current costs agree. This shows that the edge (a,b) is strongly contractible. $\Box$

Proof of Lemma 3.1: The computation of $\mathcal{T}$ corresponds to a sequence of contractions of minors, which transforms G_0 into a single vertex. Denote these minors by $S_1, S_2, \ldots$ in chronological order of their contractions. Note that either S_i is of the form C_z or it consists of the multiple edges of some fusion edge (a,b).

Let G_0' be the graph G_0 minus all the edges discarded during step [3]. As we have observed (see *Remarks* in previous section), the discarding of edges does not disconnect G_0, so G_0' still spans all the vertices of G_0. Lemma 3.2 applies to C_z at the time of its contraction. The working costs of all edges within C_z are frozen once and for all. Current costs of edges with one endpoint in C_z might change, but working costs can never decrease, so the lemma still applies relative to final working costs, ie, with each edge assigned its last working cost chronologically. Unless specified otherwise, such costs are understood throughout the remainder of our discussion.

Fix some S_i. A vertex of S_i is either a vertex of G_0 or the contraction of some S_j $(j < i)$. In turn, the vertices of S_j are either vertices of G_0 or contractions of S_k $(k < j)$, etc. By repeated applications of Lemma 3.2 (and again identifying graphs with their edge sets) it follows that the $\textsc{mst}$ of G_0' is the union of all the $\textsc{mst}\,(S_i)$'s: we call this the *composition property*. Keep in mind that the S_i's might include bad edges and so the composition property does not necessarily hold for the graphs of the form $C_z \setminus B$. In fact, it is worth noticing that, for all their "badness," bad edges are useful for making contractibility statements.

In proving Lemma 3.1, we begin with the case where the edge e under consideration is never discarded, ie, belongs to G_0'. Consider the unique S_i that contains both endpoints of e among its vertices. By induction on the correctness of `msf`, the fact that e is not in F implies that it is not in $\textsc{msf}\,(S_i \setminus B)$. Since it is not bad, the edge e is then outside $\textsc{mst}\,(S_i)$, and by the composition property, outside $\textsc{mst}\,(G_0')$. Recall that this holds relative to final working costs. Now, switch all edge costs to their original values. If changes occur, they can only be downward. The key observation now is that, by not being bad, the edge e witnesses no change and so still remains the most expensive edge on a cycle of G_0', with respect to original costs. This shows that e is not in $\textsc{mst}\,(G_0')$, and hence $\textsc{mst}\,(G_0)$, relative to original costs.

Assume now that e is not in G_0'. Before being discarded, $e = (u, v)$ shared a common endpoint v with a cheaper edge $e' = (u', v)$. In the case of a retraction, u and u' coincide, while in a fusion, both are merged together through the contraction of a subgraph. In both cases, u and u' end up in a vertex which, by repeated applications of Lemma 3.2, is easily seen to be the contraction of a contractible subgraph of G_0' relative to working costs. By the discarding rule, e and e' are not corrupted and the former is more expensive than the latter. It follows that e is outside $\textsc{mst}\,(G_0)$. Again, observe the usefulness of bad edges: indeed, because e' might become bad, we cannot conclude that e is outside $\textsc{mst}\,(G_0 \setminus B)$. This completes the proof of Lemma 3.1. □

4 Bounding the Bad Edges

We bound the number of edges in the graph B, ie, the number of bad edges created during the construction of $\mathcal{T}$ in step [3]. To prove the lemma below, we begin with a bound on the total number of inserts. Recall that n_0 (resp. m_0) denotes the number of vertices (resp. edges) of G_0.

Lemma 4.1 *The total number of bad edges produced while building $\mathcal{T}$ is $|B| \leq m_0/2 + d^3 n_0$.*

Lemma 4.2 *The total number of inserts in all the heaps is at most $4m_0$.*

Proof: Edges are inserted for the first time into a heap during an extension. In fact all extensions witness exactly m_0 inserts. To bound the number of reinserts, we provide each edge with three credits when it is first inserted. At a currency rate of one credit per reinsert, we show that the credits injected cover the reinserts.

We maintain the following invariant: For any j, any edge in $H(j)$ has two credits; for any i, j and any vertex s outside the chain, the κ edges of $H(i, j)$ incident to s contain a total of $\kappa + 2$ credits (or, of course, 0 if $\kappa = 0$). With its three brand-new credits the first insertion of edge (r, s), which takes place in some $H(i, k)$, easily conforms with the invariants.

Consider the case of a reinsert of (r, s) through retraction. If (r, s) comes from $H(k-1, k)$ or comes from $H(k)$ but shares s with an edge in $H(k-1, k)$, then its cluster of edges pointing to s releases at least 3 credits, ie, $\kappa + 2 +$ (zero or more credits from $H(k)$), for $\kappa > 0$: one pays for the insert into $H(k-1)$, while the other two are enough to maintain the credit invariant for $H(k-1)$. Otherwise, (r, s) comes from $H(k)$ and has two credits at its disposal, as it is inserted into some $H(i, k)$, where $i < k - 1$. After the insertion, the heap $H(i, k)$ will contain more than one edge pointing to s, and so only one credit is needed for the heap as (r, s) moves into it. The remaining credit pays for the insert.

What we just did is to revisit the retraction procedure step-by-step and follow the movement of credits alongside. We can do exactly the same for a fusion. Being so similar to a sequence of retractions, the fusion operation leads to an almost identical discussion, which we may omit. □

Let $B(i, j)$ be the bad edges in the heap $H(i, j)$ at the time of its disappearance (either via melding or actual destruction). To avoid double-counting, we focus only on the edges of $B(i, j)$ that were not already bad while in $B(i', j')$, for any (i', j') lexicographically greater than (i, j). Actually we can assume that $i = i'$, since for $i' > i$ all such edges are and thus denied a chance to appear in $B(i, j)$. We also have the bad edges from the heaps $H(\star)$.

These are easy to handle because unlike $H(\star,\star)$ those heaps are never melded together: By Theorem 1.2 and Lemma 4.2, the total number of corrupted edges in all the $H(\star)$'s at the time of their destruction is at most $4m_0/c$. Thus, the total number $|B|$ of bad edges satisfies: (by abuse of notation, i,j is a shorthand for all pairs (node, descendant) in $\mathcal{T}$)

$$|B| \leq 4m_0/c + \sum_{i,j} \Big| B(i,j) \setminus \bigcup_{j'>j} B(i,j') \Big|. \tag{2}$$

Define the *multiplicity* of $H(i,j)$ to be the maximum number of edges in it that share the same endpoint (outside the chain). Melding $H(i,j')$ into $H(i,j)$ does not increase its multiplicity. (That is precisely the reason why we keep separate heaps for each pair i,j.) An insert into some $H(i,j)$ during an extension sets the multiplicity to one. During a retraction, an insert can increment the multiplicity by at most one, but then the heap is immediately melded into $H(i,j-1)$. It follows that the multiplicity of any $H(\star,\star)$ is at most the height of $\mathcal{T}$, which is itself at most d.

Any edge in $H(i,l)$ that ends up in $H(i,j)$ passes through all the intermediate heaps $H(i,l')$ created ($i<l'<l$). So, with the summation sign ranging over the children j' of node j in $\mathcal{T}$, we find that the summand in (2) is equal to

$$|B(i,j)| - \sum_{j'} |B(i,j')| + \sum_{j'} \#\,\text{bad edges deleted from } H(i,j') \text{ during extensions.} \tag{3}$$

The last additive term comes from the fact that the only deletes from $H(i,j')$ are caused by extensions. Indeed, deletes occur after findmins: all the edges sharing the same endpoint with the edge selected by findmin are deleted. As we just observed there are at most d of them in each $B(i,j)$. There are at most $\binom{d+1}{2} \leq d^2$ heaps $H(\star,\star)$ at any time, so the total number of edges deleted from $B(i,j')$, for all i,j', is at most $d^3 n_0$. In view of (3), expanding (2) gives us a telescoping sum resulting in

$$|B| \leq 4m_0/c + d^3 n_0 + \sum_{i,i'} |B(i,i')|,$$

where i' denotes any child of node i. The inserts that caused corruption within the $H(i,i')$'s are all distinct, and so again by Theorem 1.2 and Lemma 4.2, the $|B(i,i')|$'s sum up to at most $4m_0/c$. We conclude that $|B| \leq 8m_0/c + d^3 n_0$. (In fact, we are overcounting.) With c large enough, Lemma 4.1 is now proven. □

5 Bounding the Running Time

We prove by induction on t and n that $\mathtt{msf}(G,t)$ takes time at most $bt(m + d^3(n-1))$, where b is a constant large enough (but arbitrarily smaller than c), and d is any integer large enough so that $n \leq S(t,d)^3$. The basis case, $t=1$, is easy. We have $n \leq S(1,d)^3 = 8d^3$ and the computation takes time $O(n^2) = O(d^3 n) \leq b(m + d^3(n-1))$. So we assume that $t>1$ and, because of step [1], that n is large enough.

We claim that d is an upper bound on the height of $\mathcal{T}$, and so we can apply the results of the previous section with the same value of d. Indeed, suppose that no fusion ever occurs. If $n = S(t,d)^3$, then d is precisely the height of $\mathcal{T}$ for reasons already explained (see paragraph preceding *Remarks* in §2). If $n < S(t,d)^3$, then obviously the construction terminates before the root of $\mathcal{T}$ attains its target degree, and our claim holds. In the presence of fusions, the key fact is that a fusion prunes a part of the existing $\mathcal{T}$ to create another one. Repeated fusions create as many new trees, but each of them can be treated as a tree to which the fusion-free case applies. Any one of them involves fewer than n vertices, so our claim holds.

The Borůvka phases in step [2] transform G into a graph G_0 with $n_0 \leq n/2^c$ vertices and $m_0 \leq m$ edges. This transformation requires $O(n+m)$ time. The complexity of building $\mathcal{T}$ in step [3] is dominated by the heap operations. By Lemma 4.2, there are $O(m_0)$ inserts, and hence, $O(m_0)$ deletes and melds. There are $n_0 - 1$ edge extensions, each of them calling up to $O(d^2)$ findmins. Each heap operation takes constant time so, conservatively, computing $\mathcal{T}$ takes $O(m_0 + d^2 n_0)$ time plus the bookkeeping costs of accessing the right heaps at the right time.

Bookkeeping is fairly straightforward, but two important points need to be understood: one concerns the exclusive use of pointers, the other the identification of the heaps $H(i,j)$ for insertion. We do not use tables (arrays being disallowed from the model), and as alluded to earlier, the notation $H(i,j)$ is merely shorthand for $H(z_i, z_j)$. For each node z_j, we maintain a linked list giving access to $H(j)$ and the $H(i,j)$'s, with the nodes z_i appearing in order of increasing height. The correspondence between the heap $H(i,j)$ and the node z_i is enforced by pointers linking them. (Recall that these z_i's need not be consecutive along the active path.)

For each v adjacent to the current chain, we maintain the border edges incident to v in a linked list sorted in order of nondecreasing height along the active path; this forms what we call the *height list* of v. In addition, for each z, we maintain the *border structure* of z. This allows us to find, in $O(1)$ time, the node z whose C_z is incident upon a given border edge.

1. If the number of border edges incident to C_z is less than d, then we keep a pointer from each of them to z.

2. Otherwise, we partition the border edges incident to C_z into groups of size d (plus a remainder group of size $< d$). Each edge points to a group representative, which itself points to z. (In other words, we create a tree of height 2 with z at the root.)

The combination of height lists and border structures allows us to answer the following question in constant amortized time: Given a border-edge (u,v) incident to C_z, what is the next z' after z, up the active path, such that v is adjacent to $C_{z'}$? A minor technicality: Since several edges in the height list might join the same C_z, we might have to walk ahead in the list to find z'. Close inspection reveals that the difficulty arises only during retraction or fusion, when inserting into $H(\star,\star)$ an edge formerly in $H(\star)$. In that case, all but one of the edges of $H(\star)$ incident to C_z are discarded, so the extra walk can be amortized over the discarded edges. To avoid traversing the other edges incident to C_z, which all come from $H(\star,\star)$, we regroup them into a sublist within the height list. As it turns out, the question above about finding z' is never asked with (u,v) being one of those edges so the sublist never needs to be traversed, and the question can always be answered in constant amortized time.

Where all that is useful is in locating the heaps $H(i,j)$ in which to insert edges (in extension and retraction). In all cases, z_j is fixed and we must insert edges into the appropriate $H(i,j)$'s. For each edge, we must find the next z_i up the active path such that C_{z_i} is adjacent to an endpoint of the edge. As we just observed, this can be done in constant amortized time. So, for fixed j, in one pass through the list of heaps $H(i,j)$ we can identify which ones to select for insertion in constant time per edge-to-be-inserted plus $O(d)$ time in overhead.

What are the maintenance costs of border structures and height lists? We give a brief sketch only, attempting mostly to explain why we use shallow trees for border structures. An extension typically causes new edges to be added to the height lists of the vertices incident to the extension edge. No border structure needs updating, however, except the one corresponding to the newly created C_{z_k}. This requires time proportional to the number of new edges considered. Note that deleting or discarding edges is trivial to handle in the case of height lists, and can simply be ignored in the case of border structures. During a retraction (or fusion), two or more C_z's collapse together. Height lists are easy to maintain. In the case of border structures, the updating cost per C_z is $O(d)$ for reconfiguring plus $1/d$ per edge for updating the root of the shallow tree (if any). The latter cost can be incurred at most d times by a given border edge, since the height in $\mathcal{T}$ corresponding to its incident C_z increases by at least one at every retraction/fusion. This gives a per-edge cost of $O(1)$; note that without shallow trees, this cost would be a prohibitive $O(d)$. There are at most d C_z's involved during a given retraction or extension, so conservatively the reconfiguration costs of $O(d)$ add up to $O(d^2 n_0)$. All together, this gives bookkeeping costs of $O(m_0 + d^2 n_0)$. In sum, by choosing b large enough, we ensure that

$$\texttt{time for step } [2,\,3] \;<\; \frac{b}{2}(n + m + d^2 n_0).$$

Turning now to step [4], consider an internal node z of $\mathcal{T}$. If $d_z = 1$, we say that z is *full* if its number n_z of children (ie, # vertices in C_z) is equal to $S(t,1)^3 = 8$. If $d_z > 1$, the node z is full if $n_z = S(t-1, S(t, d_z - 1))^3$ and its children are also full. Given a full z, the expansion of C_z relative to G_0 has a number N_z of vertices equal to $S(t, d_z)^3$; we do not include the fusion subgraphs in the count. For z not to be full, the construction of C_z must have terminated prematurely, either because a fusion pruned a part of $\mathcal{T}$ including z or more simply because the algorithm finished. Therefore, either z is full or else all its children but (the last) one are. This shows that $N_z \geq (n_z - 1) S(t, d_z - 1)^3$, for all $d_z > 1$. By construction, the number of vertices in $C_z \setminus B$ is at most $S(t-1, S(t, d_z - 1))^3$, and so we can apply the induction hypothesis and bound the time for $\texttt{msf}(C_z \setminus B, t-1)$ by

$$b(t-1)\Big(m_z + S(t, d_z - 1)^3 (n_z - 1)\Big) \leq b(t-1)(m_z + N_z), \tag{4}$$

where m_z is the number of edges in $C_z \setminus B$. Accounting for fusions, recall that a vertex of C_z may not be the contraction of just one $C_{z'}$, for some child z' of z in $\mathcal{T}$, but also subgraphs of the form C_v, where v is a node pruned from the active path of $\mathcal{T}$ together with its "fusion tree" below. Fusion trees are treated separately, and so the inequality in (4) applies to any such v as well. Over all nodes of $\mathcal{T}$ (and all fusion trees), we have $\sum m_z \leq m_0 - |B|$ and $\sum N_z \leq d n_0$ (at most n_0 vertices per level), so the overall recursion time is bounded by $b(t-1)(m_0 - |B| + d n_0)$.

$$\texttt{time for step } [4] \;<\; b(t-1)(m_0 - |B| + d n_0).$$

Finally, step [5] recurses with respect to the graph $F \cup B$. Its number of vertices is $n_0 < n \leq S(t,d)^3$ and F is cycle-free, so by induction,

$$\texttt{time for step } [5] \;<\; bt(n_0 - 1 + |B| + d^3(n_0 - 1)).$$

Adding up all these costs gives a running time at most

$$btm_0 + b\Big(\frac{m}{2} - m_0 + |B|\Big) + 2btd^3 n_0 + \frac{bn}{2}\,.$$

By Lemma 4.1, this is no more than

$$btm - b(m - m_0)\Big(t - \frac{1}{2}\Big) + 3btd^3 n_0 + \frac{bn}{2}\,.$$

Finally, using the fact that $n_0 \le n/2^c$, we find that the complexity of $\mathtt{msf}(G,t)$ is bounded by $bt(m + d^3(n-1))$, which completes the proof by induction.

When using **msf** to compute the MST of a connected graph with n vertices and m edges, our particular choice of d ensures that $d^3 n = O(m)$ and, as shown below, $t = O(\alpha(m,n))$. It follows that the MST of G is computed in time $O(m\alpha(m,n))$ and Theorem 1.1 is thus proven. □

Lemma 5.1 *If* $d = c\lceil (m/n)^{1/3} \rceil$ *and* $t = \min\{\, i > 0 \,|\, n \le S(i,d)^3 \,\}$, *then*

$$t = O(\alpha(m,n)).$$

Proof: Ackermann's function $A(i,j)$ is defined for any integers $i,j \ge 0$ [15]:

$$\begin{cases} A(0,j) &=& 2j, \text{ for any } j \ge 0; \\ A(i,0) &=& 0 \quad \text{and} \quad A(i,1) = 2, \text{ for any } i \ge 1; \\ A(i,j) &=& A(i-1, A(i,j-1)), \text{ for any } i \ge 1,\ j \ge 2, \end{cases}$$

and for any $n, m > 0$,

$$\alpha(m,n) = \min\Big\{\, i \ge 1 \,:\, A(i, 4\lceil m/n \rceil) > \log n \,\Big\}.$$

For $i \ge 1$ and $j \ge 4$,

$$A(3i,j) = A(3i-1, A(3i,j-1)) > 2^{A(3i,j-1)} = 2^{A(3i-1,A(3i,j-2))}.$$

Using the monotonicity of A, since $A(3i,j-2) \ge j$, we have

$$A(3i,j) > 2^{A(i,j)}. \tag{5}$$

It is easily shown by induction that, for any $u \ge 2, v \ge 3$, $A(u,v) \ge 2^{v+1}$, and so

$$A(3i,j) = A(3i-1, A(3i,j-1)) \ge A(3i-1, 2^j) \ge A(i, 2^j). \tag{6}$$

Trivially, $A(u-1,v) \le S(u,v)$, for any $u,v \ge 1$, which implies that

$$S(9\alpha(m,n)+1, d) \ge A(9\alpha(m,n), d).$$

Therefore, by (5, 6) and with $d \ge 4$,

$$\begin{aligned} S(9\alpha(m,n)+1,d) &> 2^{A(3\alpha(m,n),d)} \ge 2^{A(\alpha(m,n),2^d)} \\ &\ge 2^{A(\alpha(m,n),4\lceil m/n \rceil)} > n, \end{aligned}$$

and therefore the smallest t such that $n \le S(t,d)^3$ satisfies $t \le 9\alpha(m,n)+1$. □

References

[1] Borůvka, O. *O jistém problému minimálním*, Práce Moravské Přirodovědecké Společnosti, 3 (1926), 37–58. (In Czech.)

[2] Chazelle, B. *The soft heap: an approximate priority queue with optimal error rate*, to appear in J. ACM.

[3] Chazelle, B., Rosenberg, B. *The complexity of computing partial sums off-line*, Internat. J. Comput. Geom. and Appl., 1 (1991), 33–45.

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

[5] Dixon, B., Rauch, M., Tarjan, R.E. *Verification and sensitivity analysis of minimum spanning trees in linear time*, SIAM J. Comput., 21 (1992), 1184–1192.

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

[7] Fredman, M.L., Willard, D.E. *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, J. Computer and System Sciences, 48 (1993), 424–436.

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

[9] Graham, R.L., Hell, P. *On the history of the minimum spanning tree problem*, Ann. Hist. Comput., 7 (1985), 43–57.

[10] Karger, D.R., Klein, P.N., Tarjan, R.E. *A randomized linear-time algorithm to find minimum spanning trees*, J. ACM, 42 (1995), 321–328.

[11] King, V. *A simpler minimum spanning tree verification algorithm*, Algorithmica, 18 (1997), 263–270.

[12] Komlós, J. *Linear verification for spanning trees*, Combinatorica, 5 (1985), 57–65.

[13] Nešetřil, J. *A few remarks on the history of MST-problem*, Archivum Mathematicum, Brno, 33 (1997), 15–22. Prelim. version in KAM Series, Charles University, Prague, No. 97–338 (1997).

[14] Sharir, M., Agarwal, P.K. *Davenport-Schinzel sequences and their geometric applications*, Cambridge Univ. Press, 1995.

[15] Tarjan, R.E. *Efficiency of a good but not linear set-union algorithm*, J. ACM, 22 (1975), 215–225.

[16] Tarjan, R.E. *Complexity of monotone networks for computing conjunctions*, Annals Discrete Math., 2 (1978), 121–133.

[17] Tarjan, R.E. *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA, 1983.

[18] Yao, A. *An* $O(|E| \log\log |V|)$ *algorithm for finding minimum spanning trees*, Inf. Process. Lett., 4 (1975), 21–23.