State-of-the-Art Algorithms for Minimum Spanning Trees*

A Tutorial Discussion

Jason Eisner
University of Pennsylvania

April 1997

*This report was originally submitted in fulfillment of the Written Preliminary Exam II, Department of Computer and Information Science, University of Pennsylvania, and was supported in part by a National Science Foundation Graduate Research Fellowship. Many thanks to my prelim committee—Tandy Warnow, Sampath Kannan, and Fan Chung—not only for pointing me to the papers that I discuss here, but also for their friendship and willingness over the past years to hang out with me and talk about algorithms, puzzles, or even the Aristotelian world!

A future version ought to illustrate the ideas with pictorial examples.

Abstract

The classic "easy" optimization problem is to find the minimum spanning tree (MST) of a connected, undirected graph. Good polynomial-time algorithms have been known since 1930. Over the last 10 years, however, the standard $O(m \log n)$ results of Kruskal and Prim have been improved to linear or near-linear time. The new methods use several tricks of general interest in order to reduce the number of edge weight comparisons and the amount of other work. This tutorial reviews those methods, building up strategies step by step so as to expose the insights behind the algorithms. Implementation details are clarified, and some generalizations are given.

Specifically, the paper attempts to shed light on the classical algorithms of Kruskal, of Prim, and of Borůvka; the improved approach of Gabow, Galil, and Spencer, which takes time only $O(m \log(\log^* n - \log^* \frac{m}{n}))$; and the randomized $O(m)$ algorithm of Karger, Klein, and Tarjan, which relies on an $O(m)$ MST verification algorithm by King. It also considers Frederickson's method for maintaining an MST in time $O(m^{1/2})$ per change to the graph. An appendix explains Fibonacci heaps.

Contents

Chapter §1

Introduction

The *minimum spanning tree* or *MST* problem is one of the simplest and best-studied optimization problems in computer science. Given an undirected, connected graph with m weighted edges, it takes an $O(m)$-time depth-first search to find an arbitrary spanning tree, i.e., a tree that connects all the vertices of G using only edges of G. The MST problem is to find a spanning tree *of minimum total weight.* Remarkably, this optimization version of the problem can be solved in little worse than $O(m)$ time.

The problem has obvious applications to network organization and touring problems. It also arises frequently in other guises. For example, the single-link clustering technique, used by statisticians to partition data points into coherent groups, or to choose a tree topology for dendroid distributions, is essentially MST under another name.

We will only consider algorithms for the case where edge weights may be compared but not otherwise manipulated. The paper begins by reviewing the classical 1950's MST-construction algorithms of Kruskal [11] (previously invented by Varník in 1930) and Prim [13], as well as the 1926 algorithm of Borůvka [2]. With this foundation, the paper proceeds with its main task: to explicate four algorithms from the more recent literature.

(a) The asymptotically fastest deterministic algorithm known is [7], an improvement on [5]. These algorithms are all-but-linear in m, i.e., linear times a log*-type factor.

(b) A randomized Las Vegas algorithm [8] achieves expected time that is truly linear in m, and in fact guarantees linear-time performance with all but exponentially small probability.

(c) Verification of a putative MST [4, 9] can be done in time linear in m. (The randomized algorithm above uses this result.)

(d) The MST (more generally, MSF) of a fully dynamic graph can be maintained in time $O(\sqrt{m})$ per change to the graph [6].

While these algorithms are quite different in many respects, it is notable that they all rely on a recursive partitioning of the MST into connected subtrees. We will be able to use some common notation and data structures to discuss this recursive partitioning.

Throughout the presentation, I will try to elucidate where the algorithms came from—that is, how might someone have come up with them?—and how each gains a purchase on the problem. My principal goal is to convey clear intuitions. Although I will rarely waste time repeating material from the original papers unless I can fill in significant missing details or offer a fresh perspective, the present paper is essentially a tutorial, and the level of detail does resemble that of the original texts.

Chapter §2

A Whimsical Overview

This paper could also have been called: "How to Organize a Grass-Roots Revolutionary Movement As Quickly As Possible." The present section explains why.

§2.1 Problem. Russia, 1895. Idealists are scattered throughout the country. Their revolutionary ambitions hinge on their ability to make and maintain contact with each other. But channels of communication can be risky to establish, use, and maintain.

The revolutionaries need to establish enough channels between pairs of individuals that a message can be sent—perhaps indirectly—from anyone to anyone else. Naturally they prefer safe channels, and no more channels than necessary: that is, the total risk is to be minimized. In addition, they want to speed the revolution by establishing this network as quickly as possible!

§2.2 Formalization. This is, of course, an instance of the minimum spanning tree problem. Imagine a graph whose vertices correspond to the revolutionaries. If two revolutionaries can open up a channel of communication, there is a graph edge between them, weighted by the risk of that channel.[1] We wish to rapidly choose a subset of these edges so as to connect all the vertices together with minimum total edge weight (i.e., risk).

§2.3 Implementation. Various organizing strategies are presently being proposed by theoreticians. Remarkably, all strategies arrive at the same communications network—a historical inevitability. But the inevitable may arrive quickly or slowly, depending on whether the intelligentsia join the workers to hasten the revolutionary tide.

Kruskal's (serial distributed) "Safe channels should be opened first and risky ones left for a last resort. The first two individuals in Russia to establish contact should be those who can most safely do so. The next safest pair, even if in far-off Siberia, should establish contact with each other next."

Prim's (serial centralized) "Starting with Lenin, the central revolutionary organization in St. Petersburg must grow by recruiting new members who are close friends of some existing member—as close as can be found."

[1]One could define the risk of a channel as $-\log \Pr(\text{channel remains secure})$. The total risk for a set of channels is then $-\sum_i \log \Pr(\text{channel } i \text{ remains secure}) = -\log \prod_i \Pr(\text{channel } i \text{ remains secure})$, which equals $-\log \Pr(\text{all channels remain secure})$ if the channels are assumed to face independent dangers. Minimizing the latter quantity means minimizing the chance that the network will somewhere be compromised.

The optimal network (the minimal spanning tree) can be found rapidly only because of the above assumption that the risks of the various channels are independent. Formally, it is crucial that edge weights can be determined independently and need only be summed to determine the total risk of the network. In other words, edges do not interact combinatorially. They can be chosen fairly independently of each other.

Borůvka's (parallel distributed) "Every one of us should at once reach out to his or her closest acquaintance. This will group us into cells. Now each cell should reach out to another nearby cell, in the same way. That will form supercells, and so forth."

Fredman & Tarjan's "The St. Petersburg organization should grow just as Comrade Prim suggests, but only until its Rolodex is so full that it can no longer easily choose its next recruit.[2] At that time we can start a new revolutionary organization in Moscow, and then another in Siberia. Once we have covered Russia with regional organizations, the St. Petersburg organization can start recruiting other entire organizations—so that cells will be joined into supercells, as Comrade Borůvka has already proposed."

Gabow, Galil & Spencer's "We approve of the radical ideas of Comrades Fredman and Tarjan. However, may we suggest that each of us take a moment to glue together some cards in his personal Rolodex, so that our close friends are a bit easier to find than our distant acquaintances? The cells and supercells and supersupercells will *all* be flipping through those cards in search of new recruits, once our Rolodexes are combined. It is our duty to make it easy for them."

Karger, Klein & Tarjan's "Suppose we make this easier on ourselves by ignoring half the possible channels, at random. We will still [by recursion] be able to establish some fairly good paths of communication—so good that the channels we ignored are mainly riskier than these paths. A few of the channels we ignored, but only a few, will be safe enough that we should try to use them to improve the original paths."

Frederickson's "Comrades! Surely you see that we must adapt to events. If any channel whatsoever should be compromised, splitting us into two movements, then the movements must be able to find another safe channel by which to communicate. I propose that we build a resilient hierarchy of cells, supercells, supersupercells, and so on. Should we be split temporarily into two movements, each movement must be able to reorganize quickly as a hierarchy of its own. Then individuals and cells who know of safe channels to members of the other movement can rapidly pass their ideas up the hierarchy to the leadership."

[2] As in Prim's algorithm, the organization's Rolodex grows rapidly. Every new member's Rolodex is merged into the organization's, so that the organization can keep track of all friends of existing members.

Chapter §3

Classical Methods

This section presents the two classic MST algorithms, Kruskal's [11] and Prim's [13]. The remarks focus on how the high-level strategies of these algorithms differ, and why, intuitively, this leads to differences in asymptotic performance.

We begin with some general notation and a brief discussion of why the Cut/Cycle properties are both true and useful.

§3.1 NOTATION AND CONVENTIONS. $G = (V, E)$ denotes the (undirected) input graph, M denotes the correct minimum spanning tree, and F denotes a forest that will eventually be the MST. G has n vertices and m edges. We standardly assume that G is connected, so $m \geq n - 1$. An edge $e \in E$ has weight $\mathbf{w}(e)$; if e's endpoints in a graph are u and v we may sometimes denote it by uv.

Edge weights need not be real numbers. We assume they are distinct elements of an ordered set W, which is closed under an associative, commutative operation + that preserves the ordering, in the weak sense that $(w < w') \Rightarrow (w + x < w' + x)$. Our assumption that the edges have *distinct* weights is a harmless simplification, as we may break ties arbitrarily in any consistent manner (e.g., with reference to an enumeration of the edges).

For any vertex v, $\Gamma(v)$ or $\Gamma_G(v)$ denotes the set of edges incident on v in graph G. All graphs are stored using a standard adjacency-list representation. This allows a vertex of degree d to enumerate its incident edges $\Gamma(v)$ in time $O(d + 1)$.

$\log n$ denotes logarithm to the base 2. $\lg n$ denotes $\log(n + 1)$, the number of bits in the base-2 representation of n; so $\lg 0 = 0$ and $\lg 1 = 1$.

§3.2 PROPERTIES OF SPANNING TREES. A spanning tree, T, is defined as a connected acyclic spanning subgraph of G. "Connected" means that T includes at least one edge across each cut of G. "Acyclic" means that T excludes at least one edge from each cycle of G. A minimum spanning tree is a spanning tree of G whose edges have minimal total weight.

Spanning trees are connected and acyclic

§3.3 PROPERTIES OF THE MST. Under our convention that all edges have distinct weights (§3.1), the *minimum* spanning tree M has the following well-known complementary properties:

MST has cut and cycle properties

- **Strong cut property:**
  $e \in M \Leftrightarrow e$ is the lightest edge across some cut of G.
- **Strong cycle property:**
  $e \notin M \Leftrightarrow e$ is the heaviest edge on some cycle of G.

Either property implies at once that M is unique.

§3.4 PROOF. The following proof is especially quick and symmetric.

$\Rightarrow$: Every $e \in M$, when removed from M, *determines* a cut across which it is lightest; every $e \notin M$, when added to M, *determines* a cycle on which it is heaviest.

Specifically, $e \in M$ is the lightest edge between the two components of $M - e$ (for any lighter alternative e' would yield a lighter spanning tree $M - e + e'$). Similarly, $e \notin M$ is the heaviest edge on the cycle it completes in $M + e$ (for any heavier alternative e' would yield a lighter spanning tree $M + e - e'$.)

$\Leftarrow$: We derive the contrapositives from the cases just proved. $e \notin M$ cannot be lightest across any cut, because we have just seen that it is heaviest on some cycle, which will cross the cut again more lightly. Likewise $e \in M$ cannot be heaviest on any cycle, because we have just seen that it is lightest across some cut, which the cycle will cross again more heavily.

§3.5 REMARK. As we will see, all MST algorithms are rapid methods for ruling edges in or out of M. While either the Strong Cut Property or the Strong Cycle Property would suffice to define which edges appear in M, the most practical techniques for ruling edges in or out use only the $\Leftarrow$ implications of §3.3.[1] The reason: the $\Leftarrow$ implications have existential antecedents ("if $\exists$ cut") while their converses have universal ones ("if $\nexists$ cut"). Why does this matter? Because the $\Leftarrow$ statements let us classify an edge as soon as we find a *single* cut or cycle to witness the classification. Moreover, such witnesses can be produced in any order, because their qualification as witnesses depends only on G, not on what has already been determined about M.

Using cut and cycle properties to find the MST

§3.6 KRUSKAL'S ALGORITHM. This classical algorithm [11] is derived directly from §3.3. We consider each edge e, and use the Cut Property and Cycle Property to decide correctly whether $e \in M$. If so, we add it to a growing forest $F \subseteq M$; if not, we discard it. F is simply the graph $(V, \{\text{edges added so far}\})$.

Try adding edges in increasing weight order

Suppose Kruskal's algorithm is considering whether to add edge uv to F. If added, uv would be either F's first-added edge across some cut or F's last-added edge in some cycle, depending on whether there was previously a uv path in F. Kruskal's insight is to consider the edges in order of increasing weight. Then if uv is the *first* edge across a cut, it is the *lightest* such, and so must be added to F. If it is the *last* edge in a cycle, it is the *heaviest* such, and so must be excluded from F. One of these two possibilities always holds, so we can classify each edge as soon we see it.[2]

§3.7 VISUALIZATION. Whenever Kruskal's algorithm adds an edge, it connects two component trees in F, reducing the number of components by 1. At the start, F has n components, the isolated vertices. By the end, when $n - 1$ edges have been added, it has only a single component, namely the true MST.

§3.8 IMPLEMENTATION AND ANALYSIS. Kruskal's algorithm has two main chores. First, it must enumerate the edges in weight order; since only comparisons are allowed, this requires an

Kruskal's algorithm is $\Theta(m \log m)$

[1] Indeed, the $\Rightarrow$ statements will play no role in this paper except to enable the pretty proof we just saw in §3.4!

[2] While I am doing my best to frame this as a symmetric proof, it is slightly less symmetric than it sounds. Negative decisions $uv \notin F$ are justified by "local" evidence, namely an exhibitable path of lighter edges from u to v. But to justify a positive decision $uv \in F$ requires the inductive assumption that the algorithm has behaved correctly so far. To wit, the algorithm adds uv if uv would be the first edge *added* that crosses the cut, whereas what we really need to know is that uv is the first edge *considered* that crosses the cut. The latter proposition follows, but only because we know inductively that the algorithm made the right choice at previous steps: the first edge considered that crosses the cut definitely belongs in M (by §3.3), so if that edge had been considered earlier it would have been added then.

$\Omega(m \log m)$ sort, which dominates the runtime. The other chore is to determine for arbitrary edges uv whether there is already a $u \ldots v$ path in F. This can be done in a total of $m\alpha(m,n)$ time using the Union-Find algorithm [16].

Grow the tree using the lightest edge possible

§3.9 Prim's algorithm (Overview). Prim's algorithm is faster than Kruskal's. Like Kruskal's, it starts with no edges in F and grows the forest one edge at a time. However, rather than growing many trees simultaneously, it devotes all its energy to growing a single tree T of the forest F.

1. $F := (V, \emptyset)$ *(* Kruskal's also starts this way *)*
2. $u_1 :=$ an arbitrary "start vertex" in V
3. **repeat** $n-1$ times *(* once for each remaining vertex *)*
4. *(* connect an isolated vertex to u_1's component of F, which we call T *)*
5. $e :=$ the lightest edge of G leaving T (i.e., having just one endpoint in T)
6. $F := F \cup \{e\}$

It is easy to see that F is now the MST. Every edge added belonged in F, because it was lightest across the cut from T to $V-T$. Conversely, every edge belonging in F must have been added, since we added $n-1$ such edges.

Use "buckets" to reduce the set of candidate edges

§3.10 Prim's algorithm (Development). Each pass of Prim's method needs to find the lightest edge leaving T. How to do this rapidly? It is inefficient to search through all m edges on each step. We could somehow sort the edges leaving T, or maintain them in a heap, but this would be as expensive as Kruskal's algorithm.[3]

Prim's solution is essentially to "bucket" the edges leaving T, according to their terminal vertices. Whenever a new vertex u is added to T (via line 2 or 6 of §3.9), we consider each edge $uv \in \Gamma(u)$. If $v \notin T$, we throw uv into a bucket $\ell[v]$ maintained at v, which keeps a running minimum of all edges thrown into it.[4] Thus, $\ell[v]$ remembers just the lightest edge that runs from T to v. We may think of this as recording the "distance" from T to each vertex v.

Since every edge e leaving the new version of T (as augmented with u) has now been thrown into some bucket $\ell[v]$, it is now easier to find the lightest such edge. Each bucket retains only the lightest edge thrown into it, so all we have to do is to find the lightest bucket.

Obviously we do not want to search the buckets one by one on each iteration through §3.9. Nor can we imitate Kruskal's and sort the buckets by weight once and for all, since as T grows and acquires new edges that leave it, some of the buckets may become lighter. The right solution is to maintain the vertices $V-T$ in a heap (see Appendix A), keyed on their "distance" $\mathbf{w}(\ell[v])$ from T. This heap has size $O(n)$. On every iteration we Extract-Min the vertex u that is closest to T, and add the edge $\ell[u]$ to F. We then bucket the new edges $\Gamma(u)$ as discussed above; when a light edge uv displaces the old contents of $\ell[v]$, so that the bucket becomes lighter, we perform a Decrease-Key operation so that v's key continues to reflect the decreased bucket weight $\mathbf{w}(\ell[v])$.

Prim's algorithm is $O(n \log n + m)$ with Fibonacci heaps

§3.11 Analysis. Prim's algorithm removes $n-1$ vertices from the heap and adds them to the tree. Each removal takes an Extract-Min opration. Each time it removes a vertex, it examines all edges incident on that vertex, so that it eventually examines each of the m edges.[5] Each of the m edges must be bucketed, requiring a weight comparison and perhaps a

[3] The heap of edges leaving T might contain $\Theta(m)$ edges. For example, T might be a path containing half the vertices, with every vertex in T connected to every vertex in $V-T$ and no other connections.

[4] One may regard a bucket as a simple type of collection that supports only the operations Insert and Minimum, both in $O(1)$ time. (See Appendix A for the obvious definitions of these operations.) However, I will often speak of a bucket of edges $\ell[v]$ as containing just its lightest edge, or even identify it with that edge, since that is how it is trivially implemented.

[5] To be precise, each edge is examined exactly twice, when each of its endpoints is added to T, but bucketed exactly once, when its first endpoint is added to T.

DECREASE-KEY operation.

If the heap of vertices is implemented as a standard binary heap of size $O(n)$, EXTRACT-MIN and DECREASE-KEY each take time $O(\log n)$. Thus, the total time for the algorithm is $O(n \log n + m \log n) = O(m \log n)$. Notice that Kruskal's $O(m \log m)$ is asymptotically just as good, since $n - 1 \leq m \leq n^2$ implies $\log m = \Theta(\log n)$.

The bottleneck in this analysis is the DECREASE-KEY operation, which is performed $O(m)$ times, once per comparison. If we use a Fibonacci heap (explained in Appendix A), the cost of the bottleneck DECREASE-KEY operation drops to $O(1)$. Total time is then $O(n \log n + m)$. This is considerably better than Kruskal's $O(m \log m)$.

§3.12 REMARK. Indeed, if we know that the input will consist of dense graphs where $m = \Omega(n \log n)$, then Prim's algorithm using Fibonacci heaps uses time $O(m)$. Since any correct algorithm must examine the weights of all m edges in the worst case,[6] this is in general the best we can do (asymptotically speaking). Much of the work discussed below is therefore about how to improve performance on *sparse* graphs.

Prim's is optimal for dense graphs

§3.13 DISCUSSION. Why does Prim's algorithm work better than Kruskal's? Superficially, the reason is that we are bucketing edges: rather than sort m edges (time $O(m \log m)$), we are distributing them among n buckets (time $O(m)$) and then sorting the buckets (time $O(n \log n)$). But what feature of the problem lets us use these simple, constant-space buckets?

Prim's sorts only the MST edges

Each edge that is displaced from a bucket $\ell[v]$ is forgotten forever; it will never be added to F. To see that such an edge in fact cannot be in the MST M, suppose that uv and $u'v$ are both bucketed into $\ell[v]$, and uv (as the heavier edge) is discarded. The heaviest edge on the cycle $u \ldots u'v$ cannot be in M (where $u \ldots u'$ denotes the path in T from u to u'), by §3.3. This heaviest edge must be uv: for it cannot be $u'v$, which is lighter than uv, nor can it be any edge on $u \ldots u' \subseteq T$, since these edges are already known to be in M.

The crucial point is that we discard bad edges without sorting them with respect to each other: once we know they're not in M, we don't care how heavy they are. Thus we spend only $O(1)$ time to discard a bad edge (via a bucket comparison and a fast implementation of DECREASE-KEY), as compared to $O(\log n)$ time to identify a good edge (via EXTRACT-MIN). By contrast, Kruskal's algorithm spends $O(\log n)$ time on either a good or a bad edge. It puts even the bad edges into the right order, only to discard them as soon as it sees them!

§3.14 GREEDY DISCIPLINE. Let us close this section with an unoriginal observation that will come in handy later. The algorithms of Kruskal and Prim are instances of what [7] calls the "generalized greedy algorithm" for constructing MSTs.

The key idea shared by Kruskal's and Prim's

The generalized greedy algorithm initializes a forest F to $(V, \emptyset)$, and adds edges one at a time until F is connected. Every edge that it adds is the lightest edge leaving some component T of F. An algorithm that behaves in this way is said to observe "greedy discipline."

It is easy to see that at the end of such an algorithm, F is the MST. Every edge e that we add belongs in the MST, since it is the lightest edge across the cut from some T to $V - T$. So at every stage, F is a spanning subgraph of the MST (hence a forest). Now when the loop ends, F is connected, so it cannot be any proper spanning subgraph of the MST; rather it must be the whole MST, as claimed.

In Kruskal's algorithm, each edge uv added is the lightest one leaving u's component, or indeed *any* component. (All lighter edges of G have both endpoints within a single component.)

[6] Suppose an algorithm fails to examine some edge e of a 2-connected graph. The algorithm cannot correctly tell whether e belongs in the MST or not without looking at its weight. If e happens to be the lightest edge, then Kruskal's algorithm (hence any correct algorithm) would put it in the MST. If e happens to be the heaviest, than Kruskal's would exclude it, since the other edges are sufficient to connect the graph and will be considered first.

In Prim's algorithm, an edge is added only if it is the lightest one leaving the the start vertex's component. So both algorithms observe "greedy discipline."

Chapter §4

Bottom-up clustering and Borůvka's algorithm

The remaining algorithms discussed in this paper depend on a recursive grouping of the graph vertices V into successively larger "vertex clusters," each of which induces a subtree of the minimum spanning tree M. In this section I will describe this strategy, and at the risk of some initial tedium, introduce a set of notations and implementational data structures that will serve for the rest of the paper.

As a first example of these data structures, I will also discuss a simple but less familiar MST algorithm due to Borůvka [2]. In Borůvka's algorithm, each vertex of the graph colors its lightest incident edge blue; then the blue edges are contracted simultaneously, and the process is repeated until the graph has shrunk down to a single vertex. The full set of blue edges then constitutes the MST. Both King [9] and Karger *et al.* [8] make use of this algorithm.

The generalized bottom-up method

Allow graph contraction during greedy algorithms

§4.1 Strategy. To understand the generalized bottom-up strategy for constructing an MST, consider that the generalized greedy algorithm (§3.14) can be modified to incorporate contraction. Recall that the algorithm continually adds edges to a growing forest $F \subseteq G$. We observe that it is free to pause at any time and contract G (and simultaneously $F \subseteq G$) across any of the edges already in F. So the modified algorithm intersperses edge contractions with the edge additions. Can the contractions affect the algorithm's behavior? No; they merely shrink components without changing the edges incident on them, so they do not affect whether an edge is the lightest edge leaving a compoment (i.e., whether it can be selected by the algorithm).

We will be working with a generalized bottom-up strategy, which incorporates contraction in a particular way:

```
G := (V, E)
F := (V, ∅)
until F is connected
    add some edges to F, observing greedy discipline (§3.14)      (* a "pass" of the algorithm *)
    contract F and G across all edges in F
```

F and G are eventually reduced to a single vertex. We must keep a separate list of all edges added to F; when the algorithm ends, the MST is specified by this list (not shown in the pseudocode above).

The graph shrinks from pass to pass

§4.2 Discussion. Why is such a strategy useful? Following each pass through the main loop, the contraction step shrinks every tree in F down to a single vertex, in both F and G.

The next pass therefore starts anew with F as an empty graph on G's vertices. The advantage is that now G is smaller, so that the this next pass can run faster.

In what sense is G smaller? It certainly has fewer vertices than before. It may also have substantially fewer edges, if the contraction operation of line 5 is implemented (§4.12) so as to immediately or lazily eliminate multiple edges (2-cycles) and/or self-loops (1-cycles), which would otherwise arise from contracting edges on longer cycles in G. (An implementation that eliminates multiple edges should keep only the lightest edge between a given pair of vertices, since the Cycle Property (§3.3) excludes the others from the MST; if not, G becomes a multigraph.)

§4.3 Notation. For some of the methods discussed in this paper, we must keep track of the F and G graphs constructed on each pass through the algorithm of §4.1. Formally, we will construct a sequence of graphs G_0, F_0; G_1, F_1; $\ldots G_N, F_N$. G_0 is the input graph G. Given G_i, we choose F_i (on pass i of the algorithm, counting from 0) to be some acyclic subgraph of G_i, containing only edges of the MST. We then construct G_{i+1} as the contraction of G_i over all the edges that appear in F_i.

Let n_i and m_i be the number of vertices and edges, respectively, in G_i. Because $F_i \subseteq G_i$, these two graphs share a vertex set of size n_i, which we call V_i. Suppose $i \geq 1$; then these vertices correspond 1-1 to the trees in the forest F_{i-1}, of which they are contracted versions. We describe this correspondence using functions C and P (for "children" and "parent"). For $\hat{v} \in V_i$, let $C(\hat{v}) \subseteq V_{i-1}$ denote the vertex set of the corresponding tree in F_{i-1}. Conversely, for each v in that tree, let $P(v)$ denote $\hat{v}$.

A little more notation will be helpful. We define general ancestors via $P^0(x) = x$, $P^{i+1}(x) = P(P^i(x))$. We also define $Meet(x, y)$ to be the smallest i such that $P^i(x) = P^i(y)$. Finally, if $\hat{v}$ is a vertex of F_i, we write $C^\infty(\hat{v})$ for $\{v \in V_0 : P^i(v) = \hat{v}\}$; this is the full set of vertices of the original graph that have been successively merged to become $\hat{v}$.

§4.4 Implementation. We make use of the following simple data structures.

Data structures and pseudocode

edges All the edges of $G = (V, E)$ are stored in a set E; each edge records its weight and (pointers to) its endpoints in V.

vertices The vertex sets $V_0 = V, V_1, V_2, \ldots$ are stored in disjoint sets. Each vertex $v \in V_i$ stores pointers to its adjacency lists $\Gamma_{G_i}(v)$ and $\Gamma_{F_i}(v)$. For some algorithms, we will also want v to store $P(v)$ and $C(v)$; these are respectively a pointer into V_{i+1} and a circular list of pointers into V_{i-1}.

adjacency lists An adjacency list $\Gamma(v)$ is a linked list of pointers to weighted edges in E. Some algorithms require this list to be doubly linked to support $O(1)$ deletion of edges. Some also require it to be circular, so that when we merge two vertices, we can (destructively) concatenate their adjacency lists in $O(1)$ time.

Notice that the contracted graphs G_i all reuse the same edges E (on their adjacency lists). Thus, when we select an edge from the contracted graph G_i to add to the MST, we know which original vertices of G it connects and hence where it belongs in the MST. The subgraphs F_i likewise reuse the edges E.

We now flesh out the generalized bottom-up algorithm of §4.1:

$V_0 := V$ (* *with empty adjacency lists* *)
for each edge $uv \in E$ (* *create* G_0 *)
    add uv to $\Gamma_{G_0}(u)$ and $\Gamma_{G_0}(v)$
$i := 0$
while $|V_i| > 1$ (* *each iteration is called a "pass"* *)
    (* *create* F_i *from* G_i *)

```
7.      for various edges uv chosen from G_i, each choice observing greedy discipline w.r.t. F_i's components
8.          add uv to the list of MST edges that we will return
9.          add uv to Γ_{F_i}(u) and Γ_{F_i}(v)      (* see §4.6 for discussion *)
10.     (* create G_{i+1} from G_i, F_i *)
11.     V_{i+1} := ∅
12.     for each component tree T of F_i      (* see §4.5 for discussion *)
13.         add a new vertex v̂ to V_{i+1}, with empty adjacency lists
14.         C(v̂) := ∅
15.         for each v ∈ T
16.             P(v) := v̂;  C(v̂) := C(v̂) ∪ {v};  Γ_{G_{i+1}}(v̂) := Γ_{G_{i+1}}(v̂) ∪ Γ_{G_i}(v)
17.     i := i + 1
```

Notice that this basic version of the algorithm does not eliminate multiple edges or self-loops (cf. §4.12), so the G_i are in general multigraphs. The most important difference among instances of this generalized algorithm is their realization of line 7.

Finding components

§4.5 Implementation detail. The algorithm of §4.4 needs to enumerate the vertices (in F_i) of the components T of F_i, one component at a time (lines 12, 15). This can always be done in $O(1)$ time per vertex, using the usual depth-first search algorithm for finding components (simplified by the acyclicity of F_i). For certain algorithms such as [5] (see §5), where components rarely merge, it is simpler and just as fast to keep track of the vertex list of each component as the components are constructed; that is, the work of line 16 can be moved into the loop at lines 7–9.[1]

Finding contracted endpoints

§4.6 Implementation detail. Because edges are reused, the step of §4.4 where we add uv to $\Gamma_{F_i}(u)$ and $\Gamma_{F_i}(\mathrm{v})$ (line 9) is trickier than it might appear. The edge uv is taken from an adjacency list, so it is an element of E. Since such an edge records only its endpoints from the original graph, we cannot tell that it is the edge uv in G_i; all we know is that it is the edge u_0v_0 (say) in G_0. We must rapidly determine u and v so that we can add the edge to their adjacency lists and subsequently contract it. Three solutions are available:

$O(m_i)$ **overhead per pass,** $O(1)$ **per edge lookup** Assume that we find edges only on adjacency lists. Since uv is taken from u's adjacency list $\Gamma_{G_i}(u)$, we simply arrange to have stored v on the list as well. Thus, u's adjacency list has entries of the form $\langle u_0v_0, v\rangle$, where $u_0v_0 \in E$ and $v = P^i(v_0)$. This tells us in $O(1)$ time that the endpoints of the edge are u and v.

Under this representation, once we have created the adjacency lists of G_{i+1} by concatenating adjacency lists of G_i, we must walk them and update each copied entry of the form $\langle u_0v_0, v\rangle$ to $\langle u_0v_0, P(v)\rangle$. This takes time $O(m_i)$.

$O(n)$ **overhead per pass,** $O(1)$ **per edge lookup** We will find v directly from v_0 each time we consult an edge u_0v_0. Note that v is uniquely determined as $P^i(v_0)$. During pass i, we have each $v_0 \in V_0$ store $P^i(v_0) \in V_i$ (which represents v_0's contracted component), so that we can look up $P^i(v_0)$ in $O(1)$ time. At the end of pass i, we spend $O(n)$ time to replace each stored $P^i(v_0)$ with its new parent $P(P^i(v_0)) = P^{i+1}(v_0)$.

no overhead per pass, $O(\alpha(m,n))$ **per edge lookup (assuming** $\geq m$ **lookups)** Again, we will find v directly from v_0 each time we consult an edge u_0v_0. We maintain a fast

[1] In [5], components are grown one vertex at a time, so we can easily maintain the two-way correspondence (via functions P, C) between components and their vertices. We do have to handle merger of components, since the algorithm does allow a component to grow until it "bumps into" another component, and in this case, the vertices of the former must be relabeled as being in the latter. However, on a given pass of [5], each vertex is labeled at most once and relabeled at most once, so the time is still $O(1)$ per vertex.

disjoint-set structure [16] such that during pass i, we can do so in amortized $O(\alpha(m,n))$ time.

The disjoint sets during pass i are the equivalence classes of V under P^i; that is, two original vertices are in the same set just if they have been contracted together before pass i. The operation FIND(v_0) returns a canonical element $\overline{v_0}$ of v_0's equivalence class. At this element we store the ancestor $P^i(\overline{v_0})$, which equals $P^i(v_0)$ as desired.

To maintain these properties, we must update the equivalence classes and the stored ancestors at the end of pass i. We add some instructions to §4.4's loop (lines 12–16) over components T:

for each component tree T of F_i
    $\vdots$
    for each edge of T *(* we can discover T's edges in parallel with its vertices in line 15 of §4.4 *)*
        UNION(u_0, v_0) where $u_0, v_0 \in V$ are the endpoints of the edge
    $\vdots$
    $\overline{v_0}$:=FIND(v_0) where v_0 is an arbitrary vertex of T *(* the new canonical vertex for $C^\infty(\hat{v})$ *)*
    let $\overline{v_0}$ store $\hat{v}$ as the value of $P^{i+1}(\overline{v_0})$ *(* $\hat{v}$ ($\cong$ contracted T) was added to F_{i+1} in §4.4 line 13 *)*
    $\vdots$

Notice that with this approach, once we have finished pass i, we are able to compute P^{i+1}, but we can no longer compute P^i, because of the set unions. The other approaches can be implemented non-destructively.

As for the complexity, notice that we will perform a total of n MAKE-SET operations (that is, the data structure maintains a partition of the n vertices), $n-1$ UNION operations corresponding to contractions across the MST edges as we discover them, and some number $M \geq m$ of FIND operations. Thus we have $2n + M - 1$ operations *in toto*, for which the total time required under [16] is $O((2n + M - 1)\alpha(2n + M - 1, n))$. Weakening this bound to get a more convenient expression, observe that $(2n+M-1)\alpha(2n+M-1,n) < 3M\alpha(2n+M-1,n) < 3M\alpha(m,n)$. Hence we take amortized $O(\alpha(m,n))$ time per FIND operation.

Borůvka's algorithm

§4.7 PRELIMINARY VERSION. Although I have not read the original Czech description of Borůvka's algorithm [2], it is fairly clear from some remarks in [8] how it must work. It makes a good, simple example of how bottom-up clustering can help the performance of an MST algorithm. To see this point more clearly, let us consider two versions of the algorithm: a preliminary version without contraction, and the real version, which uses contraction.

Grow trees from every vertex at once

Without contraction, the algorithm runs as follows. We initialize our growing forest F to $(V, \emptyset)$ as usual. On each pass, every tree T in F locates the lightest edge of G leaving T and colors it blue. (Some edges may be colored by two trees.) Then all these blue edges are added to F, following which we uncolor them in G and proceed to the next pass. We repeat this procedure until F is connected.

§4.8 BORUVKA'S ALGORITHM. Like Kruskal's and Prim's algorithms, §4.7 is essentially an example of the generalized greedy algorithm (§3.14), and therefore correctly finds F. To see this, imagine that we add each batch of blue edges to F in *decreasing* order of weight, rather than simultaneously. (This cannot affect correctness.) Then each addition observes greedy discipline: if edge e was originally colored blue by component T, then at the time e is added it is the lightest edge leaving T's component. Why? T's component results from the merger of T

Contract trees after each step

with other trees whose blue edges were added before e. As e is the lightest blue edge leaving any of these trees, it is in fact the lightest edge leaving any of them.

We may freely add contraction to any generalized greedy algorithm (see §4.1), so it does not damage the correctness of the algorithm to contract the blue edges at the end of each pass. This yields Borůvka's algorithm.

§4.9 Implementation. Since Borůvka's algorithm is an instance of the generalized bottom-up method of §4.1, we can implement it exactly as in §4.4. We must specify which edges uv to add on pass i (line 7): the blue edges, found by iterating over all vertices $u \in V_i$ of the contracted graph and, for each, iterating over its adjacent edges $uv \in \Gamma_{G_i}(u)$ to find the lightest one. This takes time $O(m_i)$. We are careful to mark edges so that we do not add any edge twice (i.e., as uv and vu both), and not to add any self-loops (i.e., uu) that might appear in G_i.

We must also specify which bookkeeping method to use in §4.6: the $O(m_i)$-per-pass method. The cost is that after pass i has iterated over all the edges of G_i, we must iterate over them once more, which obviously does not affect the $O(m_i)$ asymptotic analysis.

§4.10 Analysis. It is easy to see that we need $O(\log n)$ passes: each pass i at least halves the number of vertices, by merging every vertex of F_i with at least one other vertex. (Or equivalently, without contraction, each pass attaches every component of F to at least one other component, halving the number of components.) *Borůvka's is $O(\min(m \log n, n^2))$*

Each pass takes $O(m)$ time to iterate through all the edges of G_i, so the algorithm runs in $O(m \log n)$ steps *in toto*. However, we can say more strongly that pass i takes $O(m_i)$ time. This is a stronger bound than $O(m)$ if, when we form G_i from G_{i-1}, we also take the trouble to eliminate multiple edges and self-loops in G_i. (For a fast implementation see §4.12 below.) Then m_i may be less than m; in particular it is bounded by n_i^2. Since $n_i \leq n/(2^i)$, the total time spent iterating through edges, summed over all i, is only $O(n^2 + (n/2)^2 + (n/4)^2 + \cdots) = O(n^2)$. So Borůvka's algorithm runs in $O(\min(m \log n, n^2))$, as [8] mentions.

§4.11 Improved analysis. The stated $O(\min(m \log n, n^2))$ bound (§4.10) beats the $O(m \log n)$ bound achieved either by Kruskal's or by a binary-heap implementation of Prim's. However, it is an asymptotic improvement only for very dense graphs, where $m = \omega(n^2/\log n)$. We can tighten the analysis to see that Borůvka's algorithm gives an improvement even for slightly sparser graphs. Each pass i takes time $O(m_i)$ edges, where m_i is bounded both by m ("intrinsic bound") and by n_i^2 ("extrinsic bound"). Rather than consider one type of bound throughout, as in §4.10, we consider on each pass i whichever bound is smaller: i.e., the intrinsic bound until we are sure that $n_i^2 \leq m$, and the extrinsic bound thereafter. Since $n_i \leq n/2^i$, the crossover point is pass $\hat{\imath} = \frac{1}{2}\log(n^2/m)$. Borůvka's algorithm requires time $O(m)$ for each of the first $\hat{\imath}$ passes, and total time $O(\sum_{i \geq \hat{\imath}} n_i^2) = O(m + m/4 + m/16 + \cdots) = O(m)$ for all subsequent passes, for a total time cost of $O(m(1 + \hat{\imath})) = O(m \cdot (1 + \log(n^2/m)))$. *Borůvka's is $O(m \cdot (1 + \log(n^2/m)))$*

This bound on Borůvka's algorithm is an improvement over naive Prim's (that is, it's $o(m \log n)$) just if $m = \omega(n^{2-\epsilon})$ for all $\epsilon > 0$. For example, it is an improvement if $m = cn^2/(\log n)^k$ for some c, k.

§4.12 Implementation detail. For the analyses of §4.10–§4.11 to go through, pass i must consider only $m_i \leq n_i^2$ edges in G_i. So we need to have eliminated redundant edges at the end of pass $i-1$ (see §4.2). Specifically, at the end of pass $i-1$ we have in hand a multigraph version of G_i, with n_i vertices and m_{i-1} edges including multiple edges and self-loops. After redundancy elimination, only $m_i \leq n_i^2$ edges will remain in G_i, and pass i may begin. *Eliminating redundant edges after contraction*

One solution is to scan through the m_{i-1} edges, discarding self-loops and placing the other edges in bins[2] according to their endpoints in G_i. That is, multiple edges connecting $u, v \in G_i$ will be placed in the same bin, which is keyed on the pair $\langle u, v \rangle$, and only the lightest such edge kept. Unfortunately, this may take more than just the $O(m_{i-1})$ steps for binning the edges. We also need $O(n_i^2)$ time to initialize the n_i^2 bins beforehand, not to mention $O(n_i^2)$ time afterwards to scan the bins and create from them new, pruned versions of the adjacency lists for G_i.[3]

A nice fix (perhaps the one used by [2], but I don't know, as I can think of others) is to use a *two*-pass stable sort. Basically, we represent each edge as a two-digit number in base n, and radix-sort the edges. The passes of a radix sort are bucket sorts. Unlike the one-pass bucket sort above, this solution is dominated by only the $O(m_{i-1})$ time to iterate through the m_{i-1} edges of the multigraph. This is possible because the number of bins is now n_i rather than n_i^2, and the overhead is proportional to the number of bins.

Each bin stores a linked list of *all* edges placed in it, with the most recently added edge at the head of the list. We use two arrays of bins, B and B'. Note that we can find an edge's endpoints in G_i in $O(1)$ time, because of the bookkeeping choice made in §4.9.

```
for each edge e of G_i
    let u, v be the endpoints in G_i of e, where v has the higher vertex number
    add e to the bin B[v]
clear the adjacency lists of G_i
for v ∈ V_i, from high-numbered to low-numbered
    for e ∈ B[v]
        let u be the other endpoint of e
        add e to the bin B'[u]
for u ∈ V_i, from low-numbered to high-numbered
    for e ∈ B'[u]
        (* this enumerates edges e of G_i in lexicographic order by their endpoints! *)
        (* each group of multiple edges is enumerated together—now easy to identify the lightest edge in the group *)
        if e is the lightest edge between its endpoints, and not a self-loop
            add e to the adjacency lists of its endpoints in G_i
```

§4.13 Discussion. *Borůvka's sorts only the MST edges* Why is Borůvka's algorithm asymptotically faster than Kruskal's or naive Prim's? It is faster only if we eliminate multiple edges, so the speedup evidently results from the fact that the size of the graph shrinks from pass to pass (see §4.2). More deeply, the point is that elimination of multiple edges is a fast way to rule edges out of the graph. As we just saw (§4.12), Borůvka's algorithm takes only $O(1)$ time per edge to eliminate all but the lightest edge between each pair of vertices of G_i. When deleting the other edges, we do not waste time sorting them with respect to each other, which would take $O(\log m_i)$ per edge. This argument should sound familiar: we saw in §3.13 that it is for just the same reason that a good implementation of Prim's algorithm (with Fibonacci heaps) will outperform Kruskal's!

§4.14 Discussion. *Borůvka's scans the same edges on every pass* Given the above remark, why is Borůvka's algorithm not quite as good as Prim's with Fibonacci heaps? For very dense or very sparse graphs it *is* as good: if $m = \Omega(n^2)$, both algorithms take only $O(m)$ time, and if $m = O(n)$, both algorithms take $O(n \log n)$. A moderately sparse graph does eventually *become* dense during Borůvka's

[2] Usually called buckets, since this is a bucket sort. I use the term "bins" to avoid confusion with the buckets of §3.10, which could hold only one element.

[3] Actually, it happens that §4.11's bounds for Borůvka's algorithm are not irrevocably damaged by the $O(n_i^2)$ overhead. The only reason we care about eliminating edges is to achieve an $O(n_i^2)$ bound on the later passes. We can simply omit the elimination step for the first $\hat{\imath}-1$ passes. Just as claimed in §4.11, passes 0 through $\hat{\imath}-1$ take time $O(m)$ apiece, while passes $i \geq \hat{\imath}$—since multiple edges have been eliminated prior to those passes—take time $O(n_i^2)$ apiece. The very first elimination step, at the end of pass $i = \hat{\imath} - 1$, takes time $O(n_i^2 + m) = O(m)$, and the elimination step at the end of each subsequent pass $i \geq \hat{\imath}$ takes time $O(n_{i+1}^2 + m_i) = O(n_i^2)$. These additions only multiply the work for each pass by a constant, so they do not affect the asymptotic analysis. Nonetheless, for thinking about MST algorithms, it is helpful to know that redundant edges can if necessary be eliminated without this $O(n_i^2)$ overhead, as shown below.

algorithm—by pass $\hat{\imath}$ or so, it has lost enough vertices that $m = \Omega(n^2)$, and the rest of the algorithm is therefore fast. But the early passes of Borůvka's on such a graph do not do as much work: at this stage there are not many multiple edges to eliminate, and indeed our analysis §4.11 assumes pessimistically that each of these passes takes fully $\Theta(m)$ steps. The extra cost of Borůvka's algorithm is due to reconsidering nearly all the same edges on each of these early passes; nearly m edges are considered on each pass, in order to include a paltry $n_i/2$ to n_i of them in the tree and perhaps exclude a few multiple edges.

Explicit contraction is unnecessary for Borůvka's

§4.15 Discussion. Prim's algorithm does well without using contraction at all. What is the benefit of contraction in Borůvka's algorithm? Without contraction (§4.7), we would simply add the lightest edge incident on each *uncontracted tree* in the forest F. We would still have to keep track of the correspondence between trees and the vertices in them: given a tree, we need its vertices so that we can enumerate the edges incident on the tree, and given an edge, we need to know which trees its endpoints are in, so that we can detect and delete within-tree edges (corresponding to self-loops in the contracted graph) and multiple edges between a pair of trees.

The main consequence of not contracting is that we never merge the adjacency lists of the vertices in a tree. So to enumerate the edges incident on a tree, we must explicitly loop through all the tree's vertices and follow the adjacency list of each. To enumerate all edges incident on all trees therefore takes time $O(m_i + n)$, rather than only time $O(m_i + n_i) = O(m_i)$ for the version with contraction. The extra $O(n)$ work on each pass leads to total extra work of $O(n \log n)$. It so happens that this does not damage the asymptotic complexity of §4.10, nor (less obviously) that of §4.11.[4] So in the case of Borůvka's algorithm, it turns out that contraction has only pedagogy to recommend it! This is also the case for the algorithms of Fredman & Tarjan (§5), King (§6), and Frederickson (§8), although the latter two (like Borůvka's algorithm) must eliminate multiple edges as if they had contracted. By contrast, the explicit merger of adjacency lists under contraction is crucial in the algorithms of Gabow *et al.* (see §5.18) and Karger *et al.* (see §7.16).

[4] §4.10 gives $O(\min(m \log n, n^2))$, and $n \log n$ is less than both terms. §4.11 gives $O(m \cdot (1 + \log(n^2/m)))$. Rewrite $m \cdot (1 + \log(n^2/m))$ as $cn \cdot (1 + \log n - \log c)$, where we have chosen $1 \le c \le n$ such that $m = cn$. This exceeds $n \log n$ for all these values of c: it is greater for $c = 1$, and increases with c up till $c = n$, since its derivative w.r.t. c is $n(\log n - \log c)$.

Chapter §5

Faster deterministic algorithms

In this section, we will consider the MST construction algorithms of Fredman & Tarjan [5] and of Gabow, Galil, Spencer, & Tarjan [7]. These methods achieve performance that is remarkably close to linear: $O(m\beta(m,n))$ for [5], and $O(m\log\beta(m,n))$ using an additional idea of [7]. I will concentrate on showing how such good performance is achieved.

The new algorithms are nearly linear, and always better than Prim's

§5.1 The β function. Let us take a moment to understand the asymptotic complexity. $\beta(m,n)$ is defined to be $\min\{i : \log^{(i)} n \leq m/n\}$. This may be usefully regarded as $\log^* n - \log^*(m/n)$, which it approximates to within plus or minus 1. (For, once $\beta(m,n)$ successive logarithms of n have gotten it below m/n, an additional $\log^*(m/n)$ logarithms will get it below 1, for a total of $\log^* n$ logarithms.) Thus, $\beta(m,n)$ is better than $\log^* n$, particularly for dense graphs. However, it is not as good as the inverse Ackerman function $\alpha(m,n)$; nor is $\log\beta(m,n)$.

One might think that on *dense* graphs, $O(m\beta(m,n))$ would be inferior to the Prim's performance of $O(n\log n + m) = O(m)$, and indeed [5] and [3] leave one with this impression. However, notice that the denser the graph, the smaller the value of $\beta(m,n)$; for instance, for

$m = \Theta(n^2)$ we have $\beta(m,n) = O(1)$. It is possible to prove that $m\beta(m,n) = O(n \log n + m)$.[1] Hence [5] actually matches or improves on the asymptotic performance of Prim's algorithm, for all graph densities. And it is substantially faster for sparse graphs—those where $n \log n$ would dominate m in the running time of Prim's algorithm. [7] is better yet.

§5.2 REMARK. It is significant that [5] breaks the $O(n \log n + m)$ barrier on some families of graphs (although it was not the first to do so). It means that the MST problem is not inherently bound by the time of edge-sorting. We saw in §3.13 that Prim's algorithm manages to avoid sorting the bad edges with respect to each other. However, it does sort up to $\Theta(n)$ of the good edges, on the heap, as well as considering all $O(m)$ edges, resulting in its $O(n \log n + m)$-time performance. Faster algorithms such as [5] and [7] cannot afford to sort so many as $\Theta(n)$ of the good edges. *MST is not bound by edge-sorting*

While a decision-tree argument shows that comparison-sorting n numbers requires at least $\log n! = \Omega(n \log n)$ comparisons on some input, because $n!$ answers are possible before we have done any comparisons, there is no such argument showing that MST construction requires that many comparisons. An input graph of m edges has 2^m subgraphs, some $k \le 2^m$ of which are spanning trees. So all we can conclude is that some input requires at least $\lceil \log k \rceil$ comparisons, where $\log k < m$.

---

[1]That is, there exist $c > 0$ and n_0 such that for all connected graphs with more than n_0 vertices, we have $m\beta(m,n) \le c(n \log n + m)$. Thus all but finitely many graphs satisfy this property; i.e., for any infinite sequence of distinct graphs, [5] eventually runs in time proportional to Fibonacci Prim or better.

I will show specifically that for all real $n \ge 16$, $m\beta(m,n) \le n \log n + m$. Let $d = m/n \log n$. The claim is now that regardless of d,

$$m \min\{i : \log^{(i)} n \le d \log n\} \le m/d + m.$$

Equivalently, dividing by m,

$$\min\{i : \log^{(i)} n \le d \log n\} \le 1/d + 1.$$

Equivalently,

$$\log^{(\lfloor 1/d \rfloor + 1)} n \le d \log n.$$

Equivalently, exponentiating each side,

$$\log^{(\lfloor 1/d \rfloor)} n \le n^d.$$

Putting $k = \lfloor 1/d \rfloor$, it is enough to show that

$$\log^{(k)} n \le n^{1/(k+1)} \quad \text{(again, just for } n \ge 16)$$

since $1/(k+1) < d$. We may see $k = 0, 1, 2$ directly and $k > 2$ by induction.

The $k = 0$ case just states that $n \le n$. The $k = 1$ case states that $\log n \le \sqrt{n}$, which holds for $n \ge 16$ by checking derivatives. The $k = 2$ case states that $\log \log n \le \sqrt[3]{n}$: claim that this holds not only for $n \ge 16$ but (crucially) wherever these functions are defined. We take derivatives to see that the RHS grows faster just when $\sqrt[3]{n} \log n > 3/(\ln 2)^2$. Now $\sqrt[3]{n} \log n$ is an increasing function, which passes the constant $3/(\ln 2)^2$ at some $n_1 < 9$, since it is already greater when $n = 9$. Now observe that

$$(\log \log n_1)(\log n_1) < (\log \log 9)(\log 9) < 3/(\ln 2)^2 \text{ (by numerical check) } = \sqrt[3]{n_1} \log n_1$$

So $\log \log n < \sqrt[3]{n}$ for $n = n_1$. By the definition of n_1, the RHS grows more quickly than the LHS as n increases above n_1, and falls more slowly than the LHS as n decreases below n_1, so the RHS is greater everywhere as claimed.

Finally, to see that $\log^{(k)} n \le n^{1/(k+1)}$ for $k > 2$, we write

$$\begin{aligned}
\log^{(k)} n &= \log \log \log^{(k-2)} n \\
&< \sqrt[3]{\log^{(k-2)} n} \quad \text{(by the } k = 2 \text{ case, which holds everywhere)} \\
&= \sqrt[3]{n^{1/(k-1)}} \text{ provided that } n \ge 16 \quad \text{(by induction, and monotonicity of } \sqrt[3]{\ }) \\
&= n^{1/(3k-3)} \\
&< n^{1/(k+1)} \quad \text{(since } k > 2).
\end{aligned}$$

Fredman & Tarjan's algorithm

§5.3 Motivation. Recall from §3.11 that Prim's algorithm takes $n + 2m$ steps simply to enumerate neighbors, and there are also n Extract-Min and m Decrease-Key operations. Thus, a binary-heap implementation requires time $O(n \log n + m \log n)$, and a Fibonacci-heap implementation requires time $O(n \log n + m)$. Having achieved the latter performance by introducing Fibonacci heaps in [5], Fredman & Tarjan proceed by attacking the *other* $\log n$ factor, in order to achieve close to $O(m)$ performance.

Keep the heap small for speed

The $\log n$ factor arises as the penalty for a large heap: as the Prim tree T grows, it may accumulate up to $O(n)$ neighboring vertices that the heap must keep track of. If we were able to keep the heap small—no more than k of the n vertices on the heap at any time—then each Extract-Min would be faster, and we would have reduced the time requirements to $O(n \log k + m)$.

We might try to arrange for a constant size heap, say a heap of size $k = 100$. If we could do this we would certainly have $O(m)$ performance. In point of fact, for dense graphs—which have fewer vertices for the number of edges—we could let k be quite a bit larger: we only have to extract n vertices from the heap, and if n is small we can afford to take more time per extraction. In particular, we may put $k = 2^{2m/n}$ and still have $O(n \log k + m)$ be $O(m)$. Note that k is defined here as 2 to the average vertex degree of G; this is large for dense graphs.

§5.4 Development. How do we keep the Prim's heap small? The heap maintains just those vertices v that are adjacent to the growing tree T, keyed by their distance from T (i.e,. the weight of the lightest T–v edge, $\ell[v]$).[2] Thus, the trick is to ensure that T does not acquire too many neighbors. We will simply stop growing T as soon as it has more than k neighbors, so that we never Extract-Min a vertex from a heap that has grown to size $> k$.

Quit and start an additional tree if the heap gets too big

Our first tree T might actually grow to span the whole graph; then we are done. But it might happen that owing to heap overflow, we have to stop growing T prematurely. In that case, we settle for building a forest. We choose a new start vertex arbitrarily from $V - T$, and grow a companion tree from it by the same procedure, starting afresh with an empty heap and again stopping when the heap gets too large. We repeat this procedure until every vertex of the graph has been sucked, Prim-like, into some large but not unmanageable tree—that is, until we have covered the graph with a spanning forest F_0 in which (speaking roughly) every tree has just over k neighbors.

§5.5 Clarification. In the above section, I say "speaking roughly" because the picture is complicated slightly by the possibility of tree merging. As we grow a given tree T', it might acquire as neighbors not only isolated vertices of F_0, but also vertices in previously-built trees of F_0. (Such vertices may be identified by their non-empty Γ_{F_0} lists.) Eventually one of these neighbors—let us say a vertex in T''—may be pulled off the heap and connected to T', thereby linking all of T'' to T'. If so, we stop growing T' immediately, just as if the heap had overflowed (indeed, T' now has $> k$ neighbors), and start growing a new tree.

Trees may grow into each other

How big is the resulting forest? Every vertex of G ends up in a tree of the form $\bar{T} = T' \cup T'' \cup T''' \cup \cdots$, where T' has more than k neighbors in G. (Some of these neighbors may be in T'' and hence in the combined tree!) It follows that every tree $\bar{T}$ in F "touches" more than k edges in the sense that it contains at least one endpoint of each. But there are only m graph

[2] So v is first added to the heap only when it becomes adjacent to T. That is, if while growing T we discover an edge from T to a vertex v that is not yet on the heap, we set $\ell[v]$ to that edge and insert v on the heap with key $\mathbf{w}(\ell[v])$. The heap starts out empty.

This discipline differs slightly from the traditional implementation of Prim's algorithm in that it omits vertices of infinite key from the heap. In the traditional implementation, *all* vertices in $V - T$ are maintained on the heap, but the key of v is ∞ until the discovery of an edge from T to v causes it to decrease. The discussion of Prim's algorithm in §3.10 was deliberately vague on this distinction.

edges to go around. Each graph edge can be touched by at most two trees. It follows that F contains $2m/k$ or fewer trees.

The length of this pass is still $O(m)$ as intended. Although we have modified Prim's approach to grow multiple trees instead of just one, each vertex of G is still extracted from a heap exactly once,[3] at $O(\log k)$ cost, and each edge of G is still considered exactly twice and bucketed exactly once, at $O(1)$ cost. So the analysis of §5.3 is unaffected: the pass takes time $O(n \log k + m) = O(m)$.

§5.6 Development. This completes one pass of the generalized bottom-up algorithm of §4.1. As §4.1 prescribes, we now contract G over all the trees of F_0, giving G_1, and perform a new pass to link together these contracted trees as far as possible given the limitation on heap size. We repeat until the graph is finally connected.

Contract all the trees and repeat

Since every added edge was chosen to be the lightest edge incident on the currently growing tree, §3.14 and §4.1 show that the algorithm correctly finds the MST.

On each pass there are fewer vertices in the graph. Since we have fewer vertices to extract, we can take longer to extract them. Indeed, we can dramatically increase the heap bound k from pass to pass, and take only $O(m)$ steps per pass. Since the goal is to keep the Extract-Min time of $O(n \log k)$ within $O(m)$, even a small reduction in n allows a correspondingly exponential increase in k, which will lead to an equally large reduction in n on the next pass. The effect snowballs so that the algorithm finishes rapidly.

Specifically, on pass i, we take $k = k_i = 2^{2m/n_i}$, so that the pass takes time $O(n_i \log k_i + m) = O(m)$. (Note that we do not bother to eliminate multiple edges; §5.9 explains why.)

§5.7 Analysis. On pass i, we have $k_i = 2^{2m/n_i}$. We saw in §5.5 that this heap bound suffices to cover the graph with, if not a single tree, then at most $2m/k_i$ trees. So $n_{i+1} \le 2m/k_i$, and then $k_{i+1} = 2^{2m/n_{i+1}} \ge 2^{k_i}$. Thus k increases tetrationally while n decreases tetrationally:[4] it does not take long before $k_i \ge n_i$, at which point Prim's algorithm can bring all n_i vertices into one tree without overflowing the k_i-sized heap. How long does it take? Observe that $k_i \ge n$ $(\ge n_i)$ iff $\log^{(i)} k_i \ge \log^{(i)} n$ iff $2m/n \ge \log^{(i)} n$. Now $i = \beta(m, n)$ is minimal such that $(2m/n >)$ $m/n \ge \log^{(i)} n$, so we require at most $\beta(m, n)$ passes of time $O(m)$ each.

The graph shrinks extremely fast

As noted in §5.3, k is defined as 2 to the average vertex degree, which keeps shrinking as the multigraph contracts. Thus even if the graph starts out as the kind of sparse graph that Prim's algorithm finds difficult, it rapidly becomes dense.

§5.8 Implementation. We can implement the method as an instance of the general bottom-up algorithm of §4.4. For contraction, we use the $O(n)$-per-pass or $O(m_i)$-per-pass method of §4.6, since neither endangers our time bound of $O(m)$ per pass. Notice that the Union-Find method, by contrast, would add an extra $\alpha(m, n)$ factor to each pass and to the overall analysis.

§5.9 Remark. Why don't we bother to eliminate redundant edges that result from contraction? Fredman & Tarjan actually specify that we *should* eliminate edges at the end of each pass, using an $O(m)$ radix sort with two passes as in §4.12. But nothing at all in their paper hinges on reducing the number of edges. In particular, their analysis allows each pass to take fully $O(m)$ steps, even if m_i is much less than m.

No need to eliminate redundant edges

[3] This is not quite precise: in truth some vertices are never extracted and some are extracted twice. A new tree's start vertex is chosen without an extraction, while the last vertex we add to it might be have been previously extracted as well (if it happens to fall in another tree). These two effects cancel each other out in each tree.

[4] Tetration is the fourth in the sequence of operations *addition, multiplication, exponentiation, tetration*, where $a \text{ op}_{i+1} b = \underbrace{a \text{ op}_i (a \text{ op}_i (a \text{ op}_i \cdots))}_{b \text{ times}}$. Thus ^4a ("a tetrated to the fourth power") is $a^{a^{a^a}}$. The inverse operation of $^b a$ is $\log^*$ to the base a (usually 2).

Would a policy of edge elimination let us improve on Fredman & Tarjan's $O(m\beta(m,n))$ analysis? Edge elimination means that we only require $O(m_i)$ time per pass (provided that we reduce k_i from $2^{2m/n_i}$ to $2^{2m_i/n_i}$). Unfortunately m_i does not decrease quickly enough for this to help. One might expect that the rapidly decreasing n_i^2 would be a useful upper bound on m_i, as it is for Borůvka's algorithm (§4.10); but once n_i^2 becomes a better bound than m is, i.e., $m < n_i^2$, a single $O(m)$ Fredman-Tarjan-style pass is enough to finish the job anyway.

Not only does n_i^2 fail to be a useful upper bound on m_i, but m_i genuinely need not decrease very fast. We can see this by constructing an infinite family of sparse graphs G_s that require $\Theta(\log^* n)$ passes of fully $\Theta(m)$ steps each. Let s be a positive integer. G_s consists of s light paths, of $s-1$ vertices each, connected to each other by single heavy edges in a K^s topology.[5] Now we have $s(s-1)$ vertices, $s(s-2)$ light edges within paths and $s(s-1)/2$ heavy edges between paths. The light within-path edges are contracted first, taking several passes;[6] this leaves us with K^s, which we contract on the final pass. Since about 1/3 of the edges—the between-path edges—survive until the final pass, every pass takes $\Theta(m)$ time. Notice that no pass creates any multiple edges or self-loops, so edge elimination happens to make no difference here.

The packet algorithm of Gabow *et al.*

§5.10 MOTIVATION. In all versions of Prim's algorithm, we incur $O(m)$ DECREASE-KEY operations as we bucket all the neighbors of each newly added vertex. For Fredman & Tarjan, who have carefully controlled the difficulty of the harder but less numerous EXTRACT-MIN operations, this DECREASE-KEY cost, repeated on every pass, has become the performance bottleneck.

Fredman & Tarjan scan the same edges on every pass

To attack this cost, which is already down to $O(1)$ per call thanks to Fibonacci heaps, we must reduce the number of calls. Observe that Fredman & Tarjan repeatedly sift through the same graph edges from pass to pass. Every surviving edge is reconsidered on every pass, so that all edges that survive till the last pass are seen up to $\beta(m,n)$ times. There may be $\Theta(m)$ of these in the worst case (§5.9), so the cost is not trivial. Yet we are always doing the same thing with those edges: looking at their weights so that we can find a light one. Perhaps we can retain some order information from pass to pass, so that we do not have to keep looking at the same edges.

Let us consider more carefully why an edge would be reconsidered—that is, why $O(1)$ time per edge (for a total of $O(m)$) is not enough for this algorithm to decide which edges are in the tree. Suppose we are growing a tree T during pass i. Throwing s edges into the bucket of a particular vertex v requires $O(1)$ time per edge. This lets us permanently eliminate $s-1$ edges (all but the lightest), having spent $O(1)$ time on each. But we have also invested $O(1)$ time on the surviving edge in the bucket. If v happens not to be added to T on this pass, because T stops growing first, the bucket is callously forgotten and that $O(1)$ effort is wasted: we still do not know whether the surviving edge will be in the tree or not. We will have to look at it again next pass.

From this point of view, it is better to wait until later to look at edges. Wasted effort is at most $O(1)$ per bucket. On later passes, there are fewer buckets, thanks to vertex mergers, so there is less wasted effort. Put another way, it is better to throw $2s$ edges into a large, merged bucket and forget about the one survivor than to throw s edges into each of two smaller buckets and forget about one survivor for each.

[5] Call the paths $P_0, P_1, \ldots P_{s-1}$, and let $P_{i,1}$ through $P_{i,s-1}$ be the vertices of P_i. The $s-1$ vertices of a given path are linked respectively to the $s-1$ other paths: simply link $P_{i,j}$ to $P_{i+j \bmod s, s-j}$ and vice-versa.

[6] An adversary can set the edge weights so that the within-path contractions do not involve any tree merging (§5.5) and therefore take about $\log^* s$ steps.

The edges that keep on going unused are unused because they are heavy, of course. We will simply refuse to pay any attention to them until after we've dealt with lighter edges! The solution, due to Gabow *et al.* [7], is to "hide" heavier edges behind lighter ones; once a lighter edge has been processed, the heavier ones become visible. This trick will improve complexity to $O(m \log \beta(m, n))$.

§5.11 DEVELOPMENT. Let p be a small positive integer. Before the first pass of the Fredman & Tarjan algorithm, we will partition the $2m$ edges mentioned on G's adjacency lists into $2m/p$ *packets* of p edges each. (Note that these $2m$ edges are not all distinct: each of the m edges of G is mentioned twice and appears in two packets.) For now let us assume that this division is arbitrary and also exact, so that each packet holds exactly p directed edges. The initial assignment of directed edges to packets is permanent, so the number of packets stays fixed from pass to pass, although edges may be deleted from packets.

Sort a few edges at the start to postpone heavy edges

A key point is that each packet is *sorted* when it is created. This sorting costs a little time up front (a total of $O(m \log p)$, which is an incentive to keep p small), but it will save us from having to consider the heavy edges in a packet until competing light edges have already been dispensed with.

Let us call the minimum-weight edge of a packet its "top edge." We treat every packet just as if it *were* its light top edge. The other edges are essentially invisible, which is what saves us time. However, when a top edge gets "used up," in the sense that we conclude it is or is not in M, we remove it from the packet and a new top edge pops up like a Kleenex.

§5.12 REMARK. This idea, of using up a packet's edges from lightest to heaviest, is a kind of localized version of Kruskal's insight that light edges should be considered before heavy ones (§3.6). The localization technique resembles that of the classic linear medians-of-5 algorithm for order statistics [1]. That algorithm crucially finds a near-median pivot element about which to partition m numbers. It divides the numbers into arbitrary packets of $p = 5$ numbers each; identifies each packet with its median element; and finds the median packet. (It is easy to see that this median-of-medians must fall between the 30th and the 70th percentile.) In much the same way, the Gabow *et al.* algorithm divides the edges into arbitrary packets of p edges each; identifies each packet with its *minimum* element; and (repeatedly) finds the *minimum* packet.

Relation to Kruskal's and medians-of-5

§5.13 DEVELOPMENT. Every packet contains an arbitrary group of p edges originating from the same vertex or cluster. Thus, when we initially create the packets, what we are doing is partitioning each vertex's adjacency list into groups of p edges. Throughout the new algorithm, we represent the adjacency list of each vertex not as a doubly-linked circular list of edges, but as a doubly-linked circular list of edge packets. When two vertices are merged (§4.4 line 16), these circular lists are combined in the usual way to give the adjacency list of the merged vertex—again a list of packets.

Manipulating edge packets

Where we used to iterate through a vertex's neighbor *edges*, throwing each into a bucket of the form $\ell[v]$, we now iterate through a vertex's neighbor *packets* in the same way. Where the bucket $\ell[v]$ used to maintain the minimum-weight edge from T to v, now it maintains the minimum-weight packet from T to v, where the weight and endpoint of a packet are determined by its top edge. So each top edge is like a handle: if it goes into a bucket or comes out of a bucket, it carries around the rest of the packet as so much baggage.

While our heap is now structured internally as a Fibonacci heap of sorted packets, it supports all the same operations as a Fibonacci heap of edges (as in Prim's). The lightest edge incident on T is simply the top edge of the lightest packet (a minimum of minima). Wherever the basic Fredman & Tarjan approach needs to extract the lightest incident edge, the Gabow *et al.* modification does so as follows:

1. Extract the lightest packet from the heap.

```
Remove its top edge uv (which we add to F).
    (This permanently alters the packet where it sits on u's adjacency list.)
Rebucket the shrunken packet so that its other edges get a chance.
```

Once we have added uv to F, we then as usual bucket all the packets incident on T's new vertex v.

Just as the extraction procedure must remove known good edges from their packets, salvaging the rest of the packet, the bucketing procedure must do the same with the known bad edges it discovers. We bucket a packet P as follows:

```
if P contains no edges
    return
else
    let e = uv be the top edge of P
    if e is a self-loop
        permanently delete e from P        (* it can't be in the MST *)
        BUCKET(P)        (* tail-recurse with new top edge *)
    elseif v is not yet marked as on the heap
        ℓ[v] := P        (* create a new bucket *)
        place v on the heap with key w(ℓ[v]) = w(e)
    else
        if e is lighter than the current ℓ[v]
            swap P with ℓ[v]
            DECREASE-MIN v's heap key to the new w(ℓ[v]) = w(e)
        permanently delete P's top edge        (* whichever edge lost by being too heavy can't be in the MST *)
        BUCKET(P)        (* tail-recurse with new top edge, which is even heavier but may fall in a different bucket *)
```

§5.14 Analysis. We must do the following work:

Now a pass need not consider all edges

- Initially, we need $O(m)$ time to partition the edges into packets and a total of $O(m \log p)$ time to sort each packet internally.
- On each of the $\leq \beta(m,n)$ passes, we take $O(n_i \log k_i)$ time to extract vertices, and $O(2m/p)$ time to bucket all the packets, exclusive of deletions and recursive calls within Bucket. On each pass, we still have exactly the $2m/p$ packets we had at the start (still ignoring roundoff), although some may have lost some or all of their edges.
- Since there are $2m$ edges mentioned in the packets, and each can die but once, the total number of deletions over all passes is at most $2m$. Each takes $O(1)$ time.
- Since every recursive call of Bucket immediately follows a deletion, we also have $\leq 2m$ recursive calls, each of which takes $O(1)$ additional time.

Summing up these times, we get

$$O\left(m + m\log p + \sum_{i=0}^{\text{max pass}} (n_i \log k_i + 2m/p) + 2m\right).$$

Our goal is to arrange for $\beta(m,n)$ passes of time $O(m/p)$ rather than $O(m)$ each. This requires choosing k_i on pass i such that $n_i \log k_i = O(m/p)$. Then we can choose $p = \beta(m,n)$, so that the expression reduces to

$$O\left(m + m\log p + \sum_{i=0}^{p-1}(m/p) + 2m\right) = O(m\log p) = O(m \log \beta(m,n)).$$

We must put $k_i = 2^{2m/pn_i}$ (rather than just $2^{2m/n_i}$) so that $n_i \log k_i$ will be m/p. Can we in fact still finish in $\beta(m,n)$ steps? In the old analysis (§5.5), we essentially argued that for each tree of F_i, more than k_i of the $2m$ directed edges originated in that tree, so there could be at most $2m/k_i$ of these trees. In the new algorithm, more than k_i of the $2m/p$ directed *packets* originate in each tree, so there are at most $2m/pk_i$ trees—an even smaller number. It follows that $n_{i+1} \leq 2m/pk_i$, so $k_{i+1} \geq 2^{k_i}$ as desired.

Since $k_0 = 2^{2m/pn}$, what we have actually shown is that we need $\leq \beta(2m/p, n)$ passes. To get the slightly stronger proposition that we need $\leq \beta(2m, n)$ passes (and *a fortiori* $\leq \beta(m, n)$), we exceptionally start with $k_0 = 2^{2m/n}$ (rather than $2^{2m/pn}$), so that the first pass alone takes time $O(m)$; this does not hurt the analysis.[7] Notice that the first pass could not possibly take time $O(m/p)$ under any choice of k_0, since it certainly takes time $\Omega(n)$, which is worse than m/p for very sparse graphs.

§5.15 Remark. Notice that the time is dominated by the $O(m \log \beta(m, n))$ cost of sorting the packets before we start. Once we begin to build the forest, the remaining time cost is only $O(m)$!

Only $O(m)$ total time wasted on repeats

Recall that this $O(m)$ cost is reduced from Fredman & Tarjan's $O(m\beta(m, n))$, which arose from the fact that they had to rebucket every edge up to $\beta(m, n)$ times (see §5.10). Should we conclude that the packet algorithm considers every edge only once? Not quite, since like the Fredman & Tarjan algorithm, it does discard full buckets if the heap overflows, only to reconsider the top edges of the packets in those buckets later. However, the total time wasted on repetition is limited to $O(m)$: on each of the p passes, we allow ourselves to bucket $2m/p$ edges that will be forgotten due to heap overflow. These are the top edges left in the $2m/p$ packets at the end of the pass. Other edges may have been bucketed as well, but those we were able to rule in or out of the tree rather than risk forgetting them. Over all passes, we waste time bucketing up to $p \cdot 2m/p = 2m$ edges that we will forget until a later pass, but we correctly determine the MST status of $2m$ edges, for a total of $O(m)$ time spent bucketing.

To see the effect, consider the example of §5.9. Fredman & Tarjan would bucket $\Theta(m)$ edges on each pass, including the heavy edges from each vertex. Gabow *et al.* initially bury most of those heavy edges deep in a packet, and do not pay any attention to them until later passes when the lighter edges from the vertex have already been considered.

§5.16 Analysis. We are forced to make one change to the Fredman & Tarjan implementation (§5.8), namely in how we deal with contraction (§4.6). We can no longer afford to spend $O(n)$ or $O(m_i)$ time per pass keeping track of contractions. (Each pass takes only time $O(m/p) = o(m)$, while both n and m_i may be $\Theta(m)$.) Instead, we use the solution from §4.6 that takes $O(\alpha(m, n))$ time per edge lookup, using Union-Find (§4.6). This means that bucketing an edge now takes time $O(\alpha(m, n))$, not $O(1)$, in order to determine which bucket to use. So the time to build the forest is really $O(m\alpha(m, n))$, not $O(m)$ as suggested. However, this is still dominated by the packet-sorting time of $O(m \log \beta(m, n))$.

We must use Union-Find when contracting

§5.17 Remark. Notice again the strong similarity to Kruskal's algorithm. Like Kruskal's, Gabow *et al.* begin with a kind of sort that dominates the running time. Again like Kruskal's, the rest of the algorithm is linear except for an $\alpha(m, n)$ factor needed to look up what trees each edge connects. The difference is that in Gabow *et al.*, the linear phase reflects a delicate tradeoff: we maintain a sorted heap of vertices while building the forest, but we compensate for the sort complexity by not allowing too many vertices on the heap.

Relation to Kruskal's

§5.18 Development. We now turn to the question of "roundoff error." Up till now, we

We must merge undersized packets

[7] The discipline for choosing k given in [7] is slightly different: it deviates more from [5] in that it simply puts $k_0 = 2^{2m/n}$, $k_{i+1} = 2^{k_i}$—the smallest rate of increase that will let us finish in $\beta(m, n)$ steps. From this choice it follows that $(\forall i \geq 0) n_{i+1} \log k_{i+1} \leq (2m/pk_i) \cdot k_i = 2m/p$ as desired. Again, the first pass is exceptional, taking $O(n_0 \log k_0) = O(2m)$ time.

have explicitly assumed that each adjacency list could be evenly divided into packets of exactly p edges, so that we have $2m/p$ packets. But if a vertex's degree d is not divisible by p, we must divide its adjacency list into $\lfloor d/p \rfloor$ packets of size p plus an undersized packet of size $d \bmod p$.

The trouble is that if many edges are in undersized packets, then we may end up with many more than $2m/p$ or even $O(m/p)$ packets, ruining our analysis. Consider a sparse graph many of whose vertices have degree much less than p. Each of the n vertices might contribute a packet of a few edges, and $n = \omega(m/p)$, so we have rather more packets than the above analysis bargained for.

In such an example, the solution is to consolidate those small packets on later passes, as their vertices merge and the graph gets denser. Thus, there will be fewer and fewer small extra packets on each successive pass.

Define the *size* of a packet to be the number of edges it had when it was created (i.e., removing edges from a packet does not reduce its size). Packets of size $\leq p/2$ are called "residual." When we initially divide adjacency lists into packets as above, any residual packet is marked as such and placed at the head of its adjacency list, where it is easily accessible. Clearly each vertex of G_0 contributes at most one residual packet—and this is a condition we can continue to enforce in $G_1, G_2, \ldots$ as vertices merge under contraction, as follows. When we merge the adjacency lists of two vertices v_1 and v_2, the new vertex v inherits the residual packets of both lists, if any. If there are two such, r_1 and r_2, they are merged into a new packet r, of size $\leq p$ (the sum of r_1's size and r_2's size). This new packet is marked residual, and put at the head of the new adjacency list, just if its size is also $\leq p/2$.

Now, how many packets are there in G_i? Each non-residual packet uses up at least $p/2$ of the $2m$ original edges, so there can be at most $2m/(p/2) = O(m/p)$ non-residual packets. In addition, each of the n_i vertices can contribute at most one residual packet. So on pass i, we must bucket not $O(m/p)$ but rather $O(m/p + n_i)$ packets. The total excess over our previous analysis is $\sum_i n_i = O(n)$ edges, taking an extra $O(n\alpha(m,n))$ time to bucket (see §5.16). This term becomes insignificant asymptotically. It is not difficult to check that the alternative analysis of footnote 7 is also preserved.

We are now stuck with the problem of merging residual packets rapidly. Two circular adjacency lists of packets can be merged in $O(1)$ time. If we represent each packet not as a sorted list of edges, but as a miniature Fibonacci heap of $\leq p$ edges, then we can merge the two residual packets from the adjacency lists in $O(1)$ time as well (see Appendix A). This requires only superficial changes to the method.[8]

Applying packets to Borůvka's

§5.19 REMARK. The packet approach seems to be more generally applicable. The discussion in §5.10, of how Fredman & Tarjan waste time reconsidering the same edges, is reminiscent of §4.14's similar criticism of Borůvka's algorithm. One might therefore expect that the same packet technique could speed up Borůvka's algorithm. Indeed this is the case: by using packets of size $p = \log n$, we can immediately reduce the time needed on sparse graphs from $O(m \log n)$ to $O(m \log\log n)$.[9] As in the Gabow *et al.* algorithm, we have chosen the packet size to match the number of passes.

To find the lightest edge incident on each vertex of G_i, as in §4.9, we now examine only

[8] In keeping with the nature of a heap, we no longer sort each packet initially, at cost $O(\log p)$ per edge. At the start of the algorithm, we INSERT all the edges into their packets, at $O(1)$ time apiece. We incur the $O(\log p)$ cost per edge only later, while building the forest, as we EXTRACT-MIN successive top edges of the packet, one by one. Notice that examining the top edge without extracting it still requires only $O(1)$ time via the MINIMUM operation.

[9] For dense graphs, our previous Borůvka's algorithm bound of $O(n^2)$ cannot be easily improved upon using packets. (Recall from §4.14 that it is sparse graphs where Borůvka's tends to reconsider edges, hence sparse graphs where packets should help.) That bound required reducing the per-pass time to $O(m_i)$ by spending $O(m_i)$ time eliminating multiple edges after every contraction—an elimination that is difficult to accomplish any faster, particularly when eliminating edges means deleting them from packets stored as Fibonacci heaps.

the top edge in each packet—with the exception that if the top edge is a self-loop, we delete it and examine the new top edge instead. As with Gabow *et al.*, there are at most $2m/p + n_i$ packets to consider on pass i, for a total of $O(m + n) = O(m)$ over all $p = \log n$ passes of Borůvka's algorithm. Like Gabow *et al.* we must use a Union-Find method to keep track of contracted vertices (see §5.16). So, excluding the time to throw out self-loops at the top of a packet, each packet we consider or reconsider requires $O(\alpha(m, n))$ time to confirm that its current top edge has *not* become a self-loop, and $O(1)$ time to consider the weight of the edge. The total time to identify lightest edges is therefore $O(m\alpha(m, n))$. Removing MST edges (those so identified) and non-edges (self-loops) from their packets, using EXTRACT-MIN, takes time $O(\log p)$ per edge, plus time $O(\alpha(m, n))$ to detect each self-loop, for a total of $O(m(\log p + \alpha(m, n))) = O(m \log \log n)$.

For sparse graphs ($m = o(n \log n / \log \log n)$) this improved Borůvka method actually beats Prim's algorithm implemented with a Fibonacci heap. Of course the new method also uses Fibonacci heaps, but differently—many small ones (the packets) instead of one large one. Moreover, it exploits a different property of these wonderful heaps—their $O(1)$ UNION rather than their $O(1)$ DECREASE-KEY.

Gabow *et al.* get their admirable performance by combining the two uses of Fibonacci heaps. They employ a large Fibonacci heap of buckets of small Fibonacci heaps (packets) of edges, where the large heap takes advantage of $O(1)$ DECREASE-KEY and the small ones take advantage of $O(1)$ UNION. Notice that using such a heap of packets would have given no advantage for the original Prim's algorithm, which requires only one pass and considers each edge only once. It is useful only for a multi-pass approach like that of Fredman & Tarjan or Borůvka.

Chapter §6

A linear-time verification algorithm

This section describes the $O(m)$ MST verification algorithm of King [9]. (A previous such algorithm was due to Dixon [4].) The description falls into two parts. The first is a clever application of general principles that is due to Komlós [10]: it shows how to perform verification using only $O(m)$ edge-weight comparisons. The second part is much more finicky. It shows how the additional bookkeeping required can also be kept to $O(m)$ time, thanks to the fact that it merely performs constant-time operations on a very small set of objects—a set of size $\leq \log n$. The idea here is to avoid iterating through these objects individually, by processing them in batches (i.e., tuples of objects). If the batch size is chosen such that there are $\leq \sqrt{n}$ distinct batches, each batch can be processed in constant time by looking it up in a table (array) with $O(\sqrt{n})$ entries. Each entry takes time $O(\text{batch size}) = O(\log \sqrt{n})$ to precompute, for a total precomputation time that is $o(n)$.

King's innovation is to use Borůvka's algorithm in service of Komlós's basic idea for verification with $O(m)$ comparisons. The idea of using table-lookup to achieve linear-time overhead was first used in a more complicated $O(m)$ verification algorithm [4]. King's application of this idea to Komlós's algorithm makes use of a labeling scheme of Schieber & Vishkin [15].

Achieving $O(m)$ comparisons

Reduction to finding heaviest edges on tree paths

§6.1 Overview. Given a spanning tree $M^?$ of G, there are two basic simple strategies to check whether it is minimal, based respectively on the Strong Cut Property and the Strong Cycle Property of §3.3. We could check that each edge e in $M^?$ is lightest across the cut defined by $M^? - e$: if not, $M^?$ cannot be the MST (see §3.3, $\Rightarrow$), and if so, $M^?$ must be the MST, since it consists of just those $n-1$ edges that do belong in the MST (see §3.3, $\Leftarrow$). On the other hand, similarly, we could check that each edge e *outside* $M^?$ is heaviest across the cycle defined by $M^? + e$: again, if not, $M^?$ cannot be the MST, and if so, $M^?$ must be the MST, since it excludes just those $m-(n-1)$ edges that do belong outside the MST.

The second strategy appears more promising, at least naively, since there can be $O(m)$ edges to compare across a cut but only $O(n)$ on a cycle. This is indeed the strategy King uses. Hence, for every edge uv not in $M^?$, she must check that its weight exceeds that of heaviest edge on $M^?$'s u-v path. This requires $< m$ comparisons of tree edges with non-tree edges. (See §7.14 below for further discussion.) The difficult part is to find the heaviest edges on all these $m-n-1$ paths, using only $O(m)$ comparisons of tree edges with each other. Looking at each path separately would take too long.

Thus, the problem to be solved is to spend a total of $O(m)$ time answering a set of $O(m)$ *queries* of the form: "What is the heaviest edge on the path between u and v in $M^?$?" We refer to this query as $\textsc{MaxEdge}(u,v)$. The path it asks about is called a *query path*.

§6.2 APPROACH. The basic clever idea in King's solution is to exploit the fact that if we run any standard MST-finding algorithm on the tree graph $M^?$ itself, it will reconstruct $M^?$ edge by edge, adding the light edges of paths first and the heavy edges only later. All the standard algorithms have the property that they defer heavy edges. King uses Borůvka's algorithm, because it runs quickly when the input graph is a tree and (as will prove important) because it completes in $O(\log n)$ passes. However, the crucial properties that let her use Borůvka's algorithm to detect heavy edges of $M^?$ turn out not to rely on any special magic about that algorithm, as we will see in §6.6.

Construct a tree as its own MST to find heaviest edges

§6.3 DEVELOPMENT. Indeed, for purposes of exposition, let us start by considering Prim's algorithm rather than Borůvka's. Suppose we wish to answer the query MAXEDGE(u, v). We run Prim's algorithm on $M^?$ with u as the start vertex, and keep a running maximum over all the edges we add to the growing tree. Just after v gets connected to the tree, the running maximum holds the answer to the query. The obvious proof: The heaviest edge added is at least as heavy as $\bar{e} =$ MAXEDGE(u, v) because all the edges on the u-v path have been added. It is also at most as heavy, for if we added any edges of $M^?$ not on the u-v path before completing the path just now, it was because they were lighter than the next edge of the u-v path that we could have added at the time, and *a fortiori* lighter than $\bar{e}$.

Answering MAXEDGE *queries via Prim's*

Ignoring running time for the moment, how well does this technique generalize to multiple queries? If u is fixed, a single run of Prim's algorithm, with u as the start vertex, will answer all queries of the form MAXEDGE(u, v), by checking the running maximum when we add each v. Indeed, the situation is even better. It is a consequence of §6.6 below that this single run starting at u can answer *arbitrary* queries: MAXEDGE(u', v) is the heaviest edge added between the addition of u' and the addition of v.

§6.4 DEVELOPMENT. Let us continue with this Prim-based approach to see that it takes too long. We need to keep track of the heaviest edges added since various times. Suppose we number the vertices $v_1, v_2, \ldots$ in the order that Prim's discovers them. Just after we add vertex v_j, we will construct an array A_j such that $A_j[1], A_j[2], \ldots A_j[j-1]$ are (so far) the heaviest edges that have been added since the respective vertices $v_1, v_2, \ldots v_{j-1}$ joined the tree. Thus $A_j[i]$ holds MAXEDGE(v_i, v_j). When v_{j+1} is added to the tree via some edge e, it is easy to create A_{j+1} from A_j: $A_{j+1}[i] = \max(A_j[i], e)$ for $i \le j$, and $A_{j+1}[j] = e$.

Binary search of sparse vectors isn't fast enough

Maintaining the edges in this way would require $O(n^2)$ array entries $A_j[i]$ $(i < j)$, and $O(n^2)$ comparisons to create them (the step $\max(A_j[i], e)$). For very dense graphs this is $O(m)$ as desired, but in general we need to do better. The two tricks we use foreshadow those employed in King's actual algorithm, which is based on Borůvka's method rather than Prim's.

(a) First, we can drop to $O(n \lg n)$ comparisons by using binary search. Observe that each array A_j lists $O(n)$ edges in *decreasing weight order*. Creating A_{j+1} from A_j is a matter of updating some tail of the array so that all its cells hold e: this tail consists of those edges of A_j that are lighter than e, and its starting position can be found in $O(\lg n)$ comparisons by binary-searching A_j for the weight $\mathbf{w}(e)$.

(b) Second, we do not really need to keep track of all of A_j. We only need to store $A_j[i]$ if one of our queries is MAXEDGE(v_i, v_k) for some $k \ge j > i$. We can store the relevant entries of A_j compactly as a sparse vector, i.e., as an array of pairs of the form $\langle i, A_j[i]\rangle$; to the extent that it has fewer than n entries, it can be binary-searched even faster than $\Theta(\lg n)$ time.

Unfortunately, the second trick does not give an asymptotic improvement. It is easy to come up with sparse graphs in which many of the vertices must still maintain large arrays. For example, imagine that we have the queries (corresponding to edges of $G - M^?$) MAXEDGE(v_1, v_n), MaxEdge(v_2, v_{n-1}), ... Then $O(n)$ of the vertices (the middle third) must create arrays that keep track of $O(n)$ query paths each, so we still need $O(n \lg n)$ comparisons.

Moreover, simply running Prim's algorithm on the n-vertex tree $M^?$ takes time $O(n \log n)$. Fortunately this algorithm—which takes too long and which adds edges in a long sequence to a single growing tree—is not our only option, as we now see.

§6.5 Definitions. Given an arbitrary weighted connected graph G, of which u and v are vertices. G has one or more u-v paths, each of which has a heaviest edge. Define Minimax(u, v) to be the lightest of these heaviest edges. *Minimax(u, v) and $F(w)$*

Recall that the generalized greedy algorithm (§3.14) constructs the MST of G by repeatedly adding edges to a forest F. Let $F(w)$ always denote the component tree of F containing the vertex w. At each step, some tree in F grows by selecting the lightest edge leaving it, causing this edge to be added to F.

§6.6 Theorem. Let G be an arbitrary weighted connected graph, and construct the MST using any algorithm that observes greedy discipline (§3.14). Then for any vertices u, v of G, Minimax(u, v) is the heaviest edge having the property that either $F(u)$ or $F(v)$ selected it while $F(u) \neq F(v)$. *All greedy MST algorithms find Minimax edges*

§6.7 Remark. This interesting theorem crucially gives us a way to find Minimax(u, v). Notice that it cannot be affected by tree contraction (§4.1), which is just a way of regarding trees like $F(u)$ as single vertices without changing the edges they add.

The theorem is actually stronger than we need. For present purposes we only care about the case where G is a tree, in which case Minimax(u, v) = MaxEdge(u, v). However, it is nearly as easy to prove the stronger form. We begin with two lemmas.

§6.8 Lemma. If $F(w)$ ever selects edge e during the generalized greedy algorithm, it already contains all paths from w consisting solely of edges lighter than e.

This follows trivially from greedy discipline: if it were false, then $F(w)$ would be able to select some edge lighter than e instead, extending such a path.

§6.9 Lemma. Let $e = xy$ be the heaviest edge on a particular u-v path, $u \ldots xy \ldots v$. Then:

(a) If e is ever selected, it is selected by $F(u)$ or $F(v)$.

Proof: It is necessarily selected by either $F(x)$ or $F(y)$. (These are the only trees that it leaves.) If by $F(x)$, then lemma §6.8 implies that $F(x)$ already contains the whole $u \ldots x$ path (whose edges are lighter than e). So $F(x) = F(u)$. Similarly, if it is selected by $F(y)$, then $F(y) = F(v)$.

(b) Nothing heavier than e is ever selected by $F(u)$ or $F(v)$ while $F(u) \neq F(v)$.

Proof: If $F(u)$ ever selected an edge heavier than e, then by lemma §6.8 it would already contain the entire $u \ldots xy \ldots v$ path (whose edges are lighter), so $F(u) = F(v)$.

§6.10 Proof of theorem §6.6. Since the algorithm starts with u and v isolated in F and ends with them in the same component, it must select and add all edges of some u-v path while $F(u) \neq F(v)$. Let e_1 be the heaviest edge on such a path. Then e_1 is selected while $F(u) \neq F(v)$, and by §6.9.a, it is selected by either $F(u)$ or $F(v)$. Let e_2 = Minimax(u, v): by §6.9.b, nothing heavier than e_2 is ever selected while $F(u) \neq F(v)$. But e_1 is at least as heavy as e_2 and is so selected. We conclude that $e_1 = e_2$. The theorem now follows from the various stated properties of e_1 and e_2.

Minimax, Cut, and Cycle are equivalent MST characterizations

§6.11 REMARK. Theorem §6.6 implies, *inter alia*, that the those edges of G expressible as $\text{Minimax}(u, v)$ are exactly those edges selected during a greedy construction of the MST. So the MST consists of these "minimax edges" of G—a moderately interesting characterization.

It is amusing to show this minimax characterization equivalent to the Strong Cut Property, which is expressed in terms of minima, and the Strong Cycle Property, which is expressed in terms of maxima (§3.3). The equivalence makes no mention of spanning trees.

For the Cut Property, we verify that the minimax edges are exactly those that are lightest across some cut. If xy is lightest across some cut, then xy must be the minimax edge $\text{Minimax}(x, y)$, since any other x-y path has a heavier edge where it crosses the cut. Conversely, if xy is the minimax edge $\text{Minimax}(u, v)$, then it is lightest across the cut formed by removing the heaviest edge from every u-v path.

For the Cycle Property, we verify that the non-minimax edges are exactly those that are heaviest on some cycle. If xy is a non-minimax edge, then in particular it is not $\text{Minimax}(x, y)$, so it is heaviest on the cycle consisting of xy itself plus an x-y path made of lighter edges. Conversely, if xy is heaviest on some cycle C, then we can replace it on any u-v path with lighter edges from $C - xy$, so it cannot be a minimax edge $\text{Minimax}(u, v)$.

Borůvka's beats Prim's for MAXEDGE

§6.12 DISCUSSION. We now return to the insufficiently fast method of §6.3–§6.4. We may replace Prim's algorithm by any algorithm that observes greedy discipline. For every query $\text{MAXEDGE}(u, v)$, we will endeavor to keep running maxima of the edges selected by $F(u)$ as it grows and likewise those selected by $F(v)$, at least until these trees merge. At that point, theorem §6.6 tells us that the greater of these two running maxima is $\text{MAXEDGE}(u, v)$.

Suppose that at some stage in the growth of the forest, $F(v_1) = F(v_2) = F(v_3) = \cdots$. That is, a certain tree T in F contains many vertices, including v_1, v_2, v_3, and so on. Suppose further that there are query paths from all these vertices. When T next selects an edge, it must update the records of the heaviest edge selected so far by each of $F(v_1), F(v_2), F(v_3), \ldots$ (all of which are now the same tree T). To minimize such updates, it is desirable for most edges to be selected by trees containing few rather than many vertices.

Thus Prim's algorithm is therefore exactly the wrong one to choose: on average, the tree doing the selection contains fully $\Theta(n)$ vertices. (As we saw in §6.4, this means that each edge selection can require $\Theta(n)$ updates to our heaviest-added-edge records, at a cost of $\Omega(\lg n)$ comparisons.) The correct algorithm is Borůvka's algorithm (§4.7), in which the task of edge selection on each pass is distributed equitably among the current (contracted or uncontracted) trees, each tree selecting one edge. Some edges may be selected by two trees at once, but this does not alter the point or affect theorem §6.6.

King therefore uses Borůvka's algorithm on $M^?$ to find the maximum edges on paths of $M^?$. An additional advantage is that actually running Borůvka's algorithm does not violate the desired $O(m)$ time bound for the verification task. Borůvka's algorithm takes only $O(m)$ time, indeed $O(n)$ time, when run on a tree such as $M^?$. In this case the contracted graphs G_i considered on successive passes are all trees: so $m_i = n_i - 1$. (This is far better than the bound $m_i = O(n_i^2)$ that §4.10–§4.11 used for the general case.) Borůvka's algorithm takes time proportional to $\sum_i m_i \leq \sum_i n_i \leq \sum_i n/2^i \leq 2n$.

Why Borůvka's runs fast on trees

§6.13 REMARK. Borůvka's algorithm takes linear time on trees, but not on arbitrary sparse graphs with $m = O(n)$. Why the difference? Borůvka's algorithm takes time $\sum m_i$. Our earlier analyses (§4.10–§4.11) bounded m_i with m and n_i^2. We are now considering a third bound, $m-(n-n_i)$, where $n-n_i$ is the number of edges actually contracted so far. This bound is not helpful unless m is very close to n, so that $m-(n-n_i)$ is a significant reduction in m. To put this another way, if m is close to n, then the $n/2$ or more edges actually contracted on the first pass constitute a significant fraction of the total edges. The same is true for subsequent passes; so noticing the loss of those edges makes a difference to the analysis.

§6.14 REMARK. Under Borůvka's algorithm, even if there is a long path Φ from u to v, neither $F(u)$ nor $F(v)$ will select more than $\log n$ edges as it grows: each selects just one edge per pass, which may or may not be on Φ. The rest of Φ is assembled by its internal vertices' finding each other. Theorem §6.6 nonetheless guarantees that the heaviest edge of Φ will be among the $2\log n$ edges that $F(u)$ and $F(v)$ do select! This provides another way of thinking about the advantage of Borůvka's algorithm: no running maximum (e.g., "heaviest edge from u") need be updated more than $\log n$ times. By comparison, the Prim-based approach could update each running maximum $\Theta(n)$ times.

Most edges on a path are never considered as possible maxima

§6.15 DEVELOPMENT. King's algorithm begins by running Borůvka's algorithm on $M^?$. We may implement it as described in §4.8. Recall that at the start of pass i, we have a contracted graph G_i whose vertices represent trees in the growing forest F. Each vertex (tree) v of G_i will select a single edge on the pass; for purposes of King's algorithm, we record this edge in a field $Sel(v)$. Two vertices may select the same edge.

Storing the selected edges

Now let v_0 be a vertex of the original graph. When Borůvka's algorithm ends, the edges selected by $F(v_0)$ will be stored as $Sel(v_0), Sel(P(v_0)), Sel(P(P(v_0))), \ldots$. Recall that the vertices $v_0 \in G_0, P(v_0) \in G_1, P(P(v_0)) \in G_2, \ldots$ represent (contracted versions of) the tree $F(v_0)$ as it grows from pass to pass.

§6.16 NOTATION. For convenience, King regards our data structure's P and C fields (parent and child-list) as implicitly defining a weighted, rooted tree $\mathbf{B}$. The leaves of $\mathbf{B}$ are the vertices V_0; their parents are V_1, and so on. The edge from v to its parent $P(v)$ is weighted by $\mathbf{w}(Sel(v))$. The weights of the edges selected by $F(v_0)$ as it grows can now be read off the path from v_0 to the root of $\mathbf{B}$.[1]

Reducing to MAXEDGE *queries on a balanced tree*

Now that we have run the Borůvka algorithm, the remainder of the problem can be reframed as follows. We are given a rooted tree $\mathbf{B}$ with n leaves. The tree is balanced in that all leaves are the same distance from the root (equal to the number of passes, $O(\log n)$). Every internal node has at least two children (possibly many more), so there are at most $n/2^i$ nodes at level i, and the height of the tree is at most $\log n$ edges.

We are permitted $O(m)$ comparisons to answer m batched queries of the form $\textsc{MaxEdge}(u_0, v_0)$, where u_0 and v_0 are leaves of $\mathbf{B}$. Each such query determines two *half-paths* in $\mathbf{B}$, from u_0 to $\textsc{lca}(u_0, v_0)$ and from v_0 to $\textsc{lca}(u_0, v_0)$. The edges on these half-paths correspond to the edges of $M^?$ selected by $F(u_0)$ and $F(v_0)$, respectively, while $F(u_0) \neq F(v_0)$. So our main goal is to find the heaviest edge on each half-path, using $O(m)$ comparisons. Then theorem §6.6 says that for each MAXEDGE query, we can simply compare the heaviest edges of its two half-paths, and return the heavier. This requires one additional comparison for each of the m edges.

§6.17 DEVELOPMENT. The solution is essentially that of §6.4, with the twist that we will descend the tree $\mathbf{B}$ from the root rather than ascending it. This corresponds to maintaining our running maxima backwards in time.

An array of running maxima at each node of $\mathbf{B}$

At each node v of $\mathbf{B}$ we will create an array A_v. We postpone discussion of how A_v is implemented, but conceptually it is a sparse array indexed by certain ancestors of v as follows.[2] Suppose u is a proper ancestor of v, and there are half-paths starting at u and passing down through v to one or more leaves. Then $A_v[u]$ exists and records the heaviest edge on the u-v path in $\mathbf{B}$. This u-v path is an initial segment of the aforementioned half-path(s). If v is a leaf,

[1] The weights do not necessarily increase upward on such a path: when $F(u)$ selects its lightest incident edge, thereby merging with some $F(v)$, a lighter edge may become available. All we can say is that a node's parent edge in $\mathbf{B}$ is heavier than at least one of its child edges. (This fact is not used.)

[2] To achieve the goal of the present section, $O(m)$ comparisons but non-linear overhead, the data structure suggested in §6.4.b will suffice. Thus if A_v is defined at a descending sequence of nodes $u_1, u_2, \ldots u_k$, we can store a length-k array whose ith element is $\langle u_i, A_v[u_i]\rangle$.

the u-v path is the entire half-path, and $A_v[u]$ records the heaviest edge on it as desired.

Like A_j in §6.4, each A_v is necessarily a decreasing array, in the sense that $A_v[u]$ decreases (or stays the same) for successively lower ancestors u.

We create the arrays A_v from the top of **B** down. A_{root} is empty since the root has no ancestors. For v not the root, we compute A_v from the parent array A_p, where p is the v's parent in **B**:

(* *Compute a version of A_v that does not take the additional pv edge into account.* *)
  for each element $A_p[u]$ of A_p in descending order of u
    if at least one of the half-paths from u to a leaf passes through v
      append a new element $A_v[u]$ to A_v, and set it to $A_p[u]$
(* *Update the running argmaxima in A_v with the pv edge, using $\leq \lg |A_v|$ comparisons.* *)
  using binary search, find the tail of A_v whose elements are lighter than pv (* *as in §6.4.a* *)
  replace every element in this tail with pv
  if there is a path from p to a leaf that passes through v
    append a new element $A_v[p]$ to A_v, and set it to pv

§6.18 Analysis. For each v in **B**, we use up to $\lg |A_v|$ comparisons to create A_v.[3] How many comparisons are needed *in toto*, over all v? King says only $O(n \log \frac{m+n}{n})$ $(= O(m))$, citing [10].

Let us check this. The intuition is that each node of **B** uses its array to keep track of several half-paths that will terminate at descendants of the node. Each level of the tree is keeping track of at most $2m$ half-paths in total, so the total size of arrays at level i of the tree is at most $2m$.[4] At high levels of the tree, these $2m$ half-paths are distributed among just a few large arrays. Binary search on each of these large arrays lets us complete the update in substantially less than $\Theta(m)$ time. At lower levels of the tree, where the half-paths might be split over many $O(1)$-size arrays, binary search is less effective and $\Theta(m)$ time may be required.

For the formal analysis, consider how many comparisons are needed at the n_i nodes at level i of the tree:

$$\begin{aligned}
\sum_v \lg |A_v| &= n_i \operatorname*{Avg}_v \lg |A_v| \\
&= n_i \operatorname*{Avg}_v \log(1 + |A_v|) \\
&\leq n_i \log \operatorname*{Avg}_v (1 + |A_v|) \qquad \text{(* using concavity of log, i.e., consider worst case of many small } |A_v| \text{ *)} \\
&= n_i \log \frac{\sum_v 1 + |A_v|}{n_i} \\
&\leq n_i \log \frac{n + \sum_v |A_v|}{n_i} \\
&\leq n_i \log \frac{n + 2m}{n_i} \qquad \text{(* see above for discussion *)} \\
&= n_i \left(\log \frac{n+2m}{n} + \log \frac{n}{n_i} \right)
\end{aligned}$$

We now sum the comparisons needed at all levels of the tree. The sum is at most

$$\sum_{i=0}^{\infty} n_i \log \frac{n+2m}{n} + \sum_{i=0}^{\infty} n_i \log \frac{n}{n_i}$$

[3] $\lg x = \lceil \log(1+x) \rceil$. It is important to use lg here rather than log, since $|A_v|$ may be 0, and also since if $|A_v|$ is 1 we still need 1 comparison to determine whether the desired tail of A_v has length 0 or 1.

[4] Specifically, each entry $A_v[u]$ corresponds to the set of half-paths starting at u and passing through the vertex v at level i. For a fixed i, each half-path appears in at most one such set, according to its uppermost vertex u and its level-i vertex v. Since there are only $2m$ half-paths, there can be at most $2m$ such sets—or equivalently, array entries—at level i.

Recall that $n_i < n/2^i$. The first summation is therefore $O(n \log \frac{m+n}{n})$. For the second summation, we observe that $x \log \frac{n}{x}$ is an increasing function for $x < n/(2^{1/\ln 2})$ and in particular for $x \leq n/2^2$. (Its derivative with respect to x is $\log(n/x) - 1/\ln 2$.) Therefore, the second summation is only

$$\begin{aligned}
\sum_{i=0}^{\infty} n_i \log \frac{n}{n_i} &= n_0 \log \frac{n}{n_0} + n_1 \log \frac{n}{n_1} + \sum_{i=2}^{\infty} n_i \log \frac{n}{n_i} \\
&\leq 0 + n_1 \cdot \frac{n}{n_1} + \sum_{i=2}^{\infty} 2^{-i} n \log \frac{n}{2^{-i} n} \\
&= 0 + n + \sum_{i=2}^{\infty} 2^{-i} n i \\
&= 0 + n + 3n/2 = O(n).
\end{aligned}$$

The total number of comparisons is therefore $O(n \log \frac{m+n}{n})$ as claimed. This is really a tighter bound than necessary; the important fact is that $n \log \frac{m+n}{n} < n \frac{m+n}{n} = m + n = O(m)$.

§6.19 REMARK. It is clear that if $M^?$ is the MST of G, then there exists a proof of that fact—a certificate for $M^?$—that can be checked using $m-1$ weight comparisons and in $O(m)$ time. One such proof has the following form: "Here is a sorted list $e_1 < e_2 < \cdots < e_m$ of the edges in G [notice the $m-1$ weight comparisons to check], and here is a trace of Kruskal's algorithm when run on that list." Another such proof follows King's algorithm more closely: "Here is a sorted list $e_1 < e_2 < \cdots < e_{n-1}$ of the edges in $M^?$ [$n-2$ comparisons to check]. Now for each edge $uv \notin M^?$, here is a lower bound $e_i < uv$ on uv's position relative to that list [a total of $m-n+1$ comparisons to check], and here is a u-v path Φ_{uv} made up of edges drawn from $\{e_1, e_2, \ldots e_i\}$ and therefore lighter than uv; so uv is not in the MST by the Strong Cycle Property." *Comparisons that witness the verification*

The interest of King's algorithm is therefore not that an $O(m)$ proof exists, but that one can be found with only $O(m)$ comparisons (and, as we shall see, in $O(m)$ time). Neither of the above proofs can be found that quickly. No general $O(m)$ algorithm can sort even the $n-1$ edges that fall in $M^?$, as this would take $\Omega(n \log n)$ comparisons.

Suppose we modify King's $O(m)$ algorithm to output its proof that $M^?$ is the MST (when this is true). This proof resembles the second proof above, but is slightly different in that it only partially sorts $e_1 < \cdots < e_{n-1}$. Thus, while the path Φ_{uv} does consist of edges of $M^?$ that are known to be no heavier than e_i, the relative order of these edges may not be otherwise known. (e_i was the heaviest edge added by $F(u)$ or $F(v)$ during the call to Borůvka's algorithm, and appears in Φ_{uv}. The other edges of Φ_{uv}, some of which were also added by $F(u)$ or $F(v)$, were determined to be lighter than e_i at various stages of the algorithm.)

How many of the algorithm's $O(m)$ comparisons must actually be mentioned in the proof? The proof must first show that the Borůvka tree $\mathbf{B}$ observes greedy discipline. It can do so by citing the $\leq 2n$ comparisons used by the Borůvka algorithm on $M^?$. Next, the algorithm spends several comparisons at each node of $\mathbf{B}$, doing a binary-search to see where an edge e whould fall in an edge sequence. The proof, however, can demonstrate each search's correctness merely by showing that e falls between two adjacent edges in the sequence. This adds $\leq 4n$ comparisons to the proof (two per node), some of which are redundant. Finally, the $m-(n-1)$ edges not in $M^?$ must each be compared to an edge in $M^?$, which is itself the heavier edge from a comparison over two half-paths. The total is $\leq 2m + 4n - 2$ comparisons, which bound is rather worse than the $m-1$ comparisons possible with proofs found via sorting.

Notice that King's algorithm runs faster than Kruskal's yet produces a longer proof. In general, partial orders can be found more quickly than total ones but may require longer to verify. (Example: The "diamond" partial order $(a < b < c) \wedge (a < b' < c)$ states four comparisons that must be verified independently. The total order $a < b < b' < c$ contains more information, and is harder to discover, but can be verified with only three comparisons.) Partial orders can be found with fewer comparisons because the decision tree for distinguishing among partial orders is shorter: its leaves correspond to internal nodes of the total order decision tree. But their proofs must make more use of comparisons because they can make less use of transitivity.

Reducing overhead to $O(m)$

§6.20 Data Structure. King's strategy can be implemented within the desired time bounds, without assuming the availability of $O(1)$ operations that do arithmetic or parallel bit manipulation on arbitrarily large numbers. To make this fact clear and concrete, I will introduce a simple data structure called the *bitstring*. This data structure will figure heavily in my presentation of the algorithm.

Precomputing functions on bitstrings

Let s be a fixed positive integer to be defined later. A bitstring b stands for a string of s or fewer bits: 11 is a different bitstring from 0011 or 1100, and 0 is distinct from the null bitstring ε. There are $S = 2^{s+1} - 1$ different bitstrings. Each is represented as an arbitrary integer in $[1, S]$, which means that it can be used as an array index into a lookup table.

Given a linked list of bits, we can find the corresponding bitstring, and vice-versa. Both these operations take time proportional to the length of the bitstring in question; in particular they are $O(s)$. For the correspondence we rely on an auxiliary data structure, a trie that can be prebuilt in $O(S)$ time. The trie is a balanced, perfectly binary tree of depth s and has S nodes. We number the nodes arbitrarily, e.g., consecutively as we create them depth-first. Given the list of bits 01101, we find the bitstring by following a left-right-right-left-right sequence of children from the root of the trie and returning the number of the resulting node, say 75. Conversely, given this bitstring 75, we can look up a pointer to the node, using an array that we built when we originally numbered the nodes. Then we can follow parent links up to the root of the trie, constructing the list 01101 starting with its rightmost bit.[5]

It is simple to compute functions such as $length(b)$ (the number of bits in a given bitstring), $concat(b_1, b_2)$ (the concatenation of two bitstrings), $substring(b, i, \ell)$ (the ℓ-bit substring of a bitstring b, starting at position i), and so forth. All the functions we are interested in can be computed in time $O(s)$ per call. The simplest implementation is to convert the bitstrings to lists of (s or fewer) bits, do the computation on the lists, and convert back to a bitstring as necessary. The results can be cached in lookup tables. For example, the lookup table for *substring* would be a three-dimensional array of bitstrings, indexed by b, i, and ℓ, which has $O(S \cdot s^2)$ entries and takes time $O(S \cdot s^3)$ to precompute in its entirety. Some entries of the table are undefined or hold an error value. Once the table is precomputed, entry lookup is $O(1)$.

§6.21 Strategy. The simple algorithm of §6.17 constructs A_v at a node from A_p at its parent node. The work of this construction is dominated by two simple loops. First, we copy selected elements of A_p to A_v; then we overwrite some tail of A_v (selected by binary search). These loops take time $O(|A_p|)$ and $O(|A_v|)$ respectively. Our goal is to make both of them take time $O(\lg |A_v|)$, the same as the time for the binary search. Then the work at each node will be proportional to the number of comparisons at that node. The total work will be therefore be proportional to the total number of comparisons, which we have seen to be $O(m)$.

Make overhead proportional to comparisons

[5] A standard way of storing such a binary tree is in an array a of length S, where $a[1]$ is the root and $a[i]$ has children $a[2i]$ and $a[2i+1]$. If we number the node $a[i]$ with i, there is no need to actually store the array. We can carry out the above procedure merely by doubling, halving, and incrementing. However, that would require us to do arithmetic on potentially large numbers rather than just using them as array indices. Note that the numbering scheme here represents the bitstring 01101 with the integer $45 = 101101_{\mathrm{two}}$ (i.e., a 1 is prefixed). The trie could be numbered according to this scheme in time $O(S \cdot s)$.

§6.22 DATA STRUCTURE. How do we represent the array A_v? Number the vertices on the root-to-v path by their distance from the root. Edges can be numbered similarly. Every number is $\leq \lg n$. If efficiency were not a concern, we would store A_v directly: $A_v[0]$ for the heaviest edge from 0 to v, $A_v[1]$ for the heaviest edge from 1 to v, and so forth. However, recall that we are going to leave some of these entries undefined: $A_v[u]$ is maintained only if there is a half-path starting at u that passes down through v.

A particular sparse array scheme

We therefore represent A_v as a pair of data structures, C_v ("contents") and I_v ("indices"). C_v is an "compacted" array containing just the defined entries of A_v, so $C_v[0]$ is the first defined entry, $C_v[1]$ is the second defined entry, and so forth. Notice that C_v can be binary-searched in time $\lg |A_v|$, as desired. One might expect that I_v would be a parallel array giving the indices for $C_v[0], C_v[1], \ldots$. However, we represent the same information more concisely as a bit vector of length $\leq \log n$ specifying the defined indices: $I_v[u] = 1$ just if $A_v[u]$ is defined. This representation better supports the operations we need.

§6.23 STRATEGY. Let us first see how to do the operations of §6.21 in time $O(\lg |A_p|)$. We will subsequently improve this to our goal of $O(\lg |A_v|)$, which is more ambitious since $|A_v| \leq |A_p| + 1$ and is possibly much smaller.

Process a batch of edges at a time

The basic strategy was sketched at the start of §6. Recall that C_v is an array of some of the edges on the root-to-v path, each of which can be stored as a small number $\leq \log n$ (called an *edge tag*) according to its distance from the root.[6] We will process these tags in *batches* (tuples) of about $\log_{(\log n)}(n^{1/3})$ tags each. This choice of batch size means there are at most $n^{1/3}$ possible batches, and only $n^{2/3}$ possible pairs of batches. We can therefore compute (for example) any unary or binary operation on batches in constant time, by looking up the operand(s) in a table of $n^{2/3}$ or fewer entries.

How long does it take to precompute such a table, assuming that each entry can be computed in time $O(\log n)$?[7] (This assumption is often reasonable: there are fewer than $\log n$ tags in all of $|C_v|$, so there are certainly fewer than $\log n$ tags per batch of C_v's tags, so we just need to spend $O(1)$ time per tag when computing a function on batches.) We need only $O(n^{2/3} \log n) = O(n)$ to precompute the whole table. The advantage to precomputation is that the same entries will be accessed repeatedly.

§6.24 IMPLEMENTATION. It is convenient to switch to the log domain and use bitstrings. To recast part of the above discussion, C_v consists of an array of some number $p \leq \log n$ edge tags of $\lg \log n$ bits each, for a total of $p \lg \log n$ bits. We divide this list into batches of tags, where each batch holds enough tags to consume $s \approx \frac{1}{3} \log n$ bits. (s is rounded up to a multiple of $\lg \log n$.) Each batch is represented as a single bitstring having the maximum length s, except that the last batch may be a shorter bitstring. These bitstrings are stored in a random-access array.

Sizes and quantities of bitstrings

We will never use any bitstrings longer than s. So in the terminology of §6.20, $S = 2^{s+1} - 1 = O(n^{1/3})$. We can therefore precompute an $O(s)$-time function of up to two bitstrings and k integer indices in $[0, s]$ in time $O(S^2 \cdot s^{k+1}) = O(n^{2/3} (\log n)^{k+1})$, which is $O(n)$ for any fixed k. For example, the *substring* example from §6.20 takes one bitstring and two indices. Looking up values of such functions is $O(1)$ thereafter. The idea is similar to doing base arithmetic using a large base that is determined at runtime and increases with the size n of the problem.

The total time required to iterate through all of C_v, performing such operations by $O(1)$ lookup one batch at a time, is proportional to the number of batches. This is the total number of bits in C_v divided by s, or at most $p \lg \log n / \frac{1}{3} \log n$. It is easy to see that this is less than

[6] In §6.27 we will revise the edge-numbering scheme.

[7] King does not call attention to the fact that each table entry requires $\omega(1)$ time (here $O(\log n)$) to precompute, although if this were not the case there would be no advantage to using a table. My choice of $n^{1/3}$ possible batches rather than $n^{1/2}$ ensures that lookup tables can be computed in $O(n)$ time.

$3\lg p = 3\lg |C_v| = O(\lg |C_v|)$ as desired by §6.21.[8]

We will also have to do some work on I_v. I_v is a vector of only $\log n$ bits, which we can store as a list of just three bitstrings of length $s = \frac{1}{3}\log n$. Iterating down the three bistrings on this list takes only $O(1)$ time. I will refer to these s-bit bitstrings as batches, too, although they are conceptually batches of bits rather than of edge tags.

Appending and removing bits quickly

§6.25 DEFINITIONS. Before giving the algorithm, we need to clarify the definitions of some bitstring functions. The concatenation of two bitstrings might result in more than our maximum of s bits. So $concat(b_1, b_2)$ is defined to return a list of either one or two bitstrings. If this list holds two bitstrings, the first has length s and the second holds the overflow. (Example: if $s = 6$, $concat(100, 11) = \langle 10011\rangle$, but $concat(100, 11101) = \langle 100111, 01\rangle$.) A lookup table for $concat$ can be computed in time $O(S^2 \cdot s) = O(n)$.

This definition of *concat*, with its table-lookup implementation, helps us maintain sequences of more than s bits, such as C_v. Suppose L is an array or list of one or more bitstrings, all of length s except possibly the last, representing a long sequence of bits. b is another bitstring whose bits are to be added to the end of L. Define an $O(1)$ procedure $Append(L, b)$ that destructively removes the last bitstring b' from L and appends $concat(b', b)$ instead. (Example: if $s = 6$, $L = \langle 101010, 000000, 100\rangle$, then $Append(L, 11101)$ lengthens L to $\langle 101010, 000000, 100111, 01\rangle$.)

Similarly, if L is an array or list of bitstrings, all of length s except possibly the first and last, then we can define an $O(1)$ procedure $RemoveHead(L, \ell)$ that destructively removes that first $\ell \le s$ bits of L. *RemoveHead* replaces the first two bitstrings of ℓ with one or two bitstrings, which are found with a looked-up function *remove* of the first two bitstrings. (Example: if $s = 6$, $L = \langle 010, 000000, 100111, 01\rangle$, then $RemoveHead(L, 5)$ shortens L to $\langle 0000, 100111, 01\rangle$.)

Some other precomputed functions on bitstrings will be needed below; we will give brief definitions as necessary.

Fast algorithm for computing each node's array of heavy ancestors

§6.26 DEVELOPMENT. We begin by spending $O(m + n)$ time to create the vectors I_v at the leaves v of $\mathbf{B}$, as follows. We iterate through the m queries $\textsc{MaxEdge}(u, v)$. For each query, we set $I_u[\textsc{lca}(u, v)]$ and $I_v[\textsc{lca}(u, v)]$ to 1, indicating which half-paths we must track. (This requires a prior computation of the least common ancestor $\textsc{lca}(u, v)$ for each of the m queries. Such a (batch) computation can be done in total time $O(m + n)$ [15].)

We now spend an additional $O(n)$ time creating I_v for all internal nodes v, from the bottom of the tree upward. For each v, we bitwise-OR together the already-created I vectors of v's children, since v will have to maintain running maxima on behalf of all its children. We then clear bit i of the result to say that v need not maintain the heaviest edge from itself. Each bitwise-OR operation would require iterating over $\log n$ bits of I_v if we were not using lookup tables. With a lookup table for bitwise-OR, we iterate over just three pairs of bitstrings in $O(1)$ time each, as described above. Clearing a bit in I_v using a lookup table similarly takes time $O(1)$. Thus, we spend $O(1)$ time on each of the $O(n)$ internal nodes.

Having created the I vectors bottom-up, we now create the C vectors top-down, at each node v determining C_v in $O(\lg |C_v|)$ time from its parent C_p. The pseudocode below recasts the method of §6.17 to use the new data structures. The presentation is substantially briefer and more precise than the one in [9], thanks to the use of variable-length bitstrings. By convention, italicized functions are precomputed functions on bitstrings, except for the capitalized ones such as *Append* (§6.25), which are $O(1)$ functions or procedures that handle sequences of bitstrings.

```
(* Compute a version of C_v that does not take the additional pv edge into account; cf. §6.17. *)
    (* Collect those bits of I_v that have corresponding 1 bits in I_p; call this compacted bit vector I =select(I_p, I_v). *)
    (* Thus I is a parallel array to the compacted array C_p, indicating which bits of C_p should be copied to C_v. *)
        I := ⟨ε⟩      (* initialize to list containing just the empty bitstring; we maintain pointers to head and tail *)
```

[8] We know that $p \le \log n$. Since $\frac{x}{\lg x}$ is an increasing function, we have $\frac{p}{\lg p} \le \frac{\log n}{\lg \log n}$, which gives the result.

4. **for** each batch b_{I_p} of I_p and the corresponding batch b_{I_v} of I_v
5. $b_I := select(b_{I_p}, b_{I_v})$ (* *by lookup, of course! compacts one batch of I_v as defined in line 2* *)
6. $Append(I, b_I)$ (* *as defined in §6.25* *)
7. (* *Now copy certain edges from C_p to C_v as directed by I, one batch of C_p at a time.* *)
8. $C_v := \langle \varepsilon \rangle$ (* *extensible array containing just the empty bitstring* *)
9. **for** each batch b_{C_p} of edge tags in C_p
10. $\ell :=$ number of tags in b_{C_p}, i.e., $length(b_{C_p})/\lg\log n$ (* *$\ell \leq length \leq s$; last batch might be shorter* *)
11. $b_I := Head(I, \ell)$ (* *first ℓ bits of the list I—computed from the first two bitstrings in the list* *)
12. $RemoveHead(I, \ell)$ (* *destructively remove those first ℓ bits, as defined in §6.25* *)
13. $b_{C_v} = select\text{-}tags(b_I, b_{C_p})$ (* *$\forall j$, collect the jth tag from b_{C_p} if the jth bit of b_I is 1* *)
14. $Append(C_v, b_{C_v})$
15. (* *Update the running argmaxima in C_v with the pv edge, using $\leq \lg|C_v|$ comparisons; cf. §6.17.* *)
16. $b_{pv} :=$ bitstring representing the tag of pv
17. binary-search C_v to find $j_{\min} = \min\{j : \mathbf{w}(C_v[j]) < \mathbf{w}(pv)\}$ (* *$C_v[j]$ denotes jth edge tag, not jth batch* *)
18. (* *Replace all tags from $j_{\min}$ on with the tag of pv* *)
19. $j := 0$
20. **for** each batch b_{C_v} of edge tags in C_v
21. $\ell :=$ number of tags in b_{C_p} (* *as before* *)
22. $j := j + \ell$ (* *number of tags seen so far* *)
23. $r := \min(\max(j - j_{\min}, 0), \ell)$ (* *number of tags to overwrite at end of b_{C_p}* *)
24. replace b_{C_v} in the array C_v with $replace\text{-}tail(b_{C_v}, concatcopies(b_{pv}, r))$ (* *glue r copies together* *)
25. **if** $testbit(I_v$,distance of p from root) (* *check status of bit telling us whether $A_v[p]$ should be defined* *)
26. Append(C_v, b_{pv})

Ignoring the binary search of line 17 for the moment, every step here is handled with an $O(1)$ lookup operation. The only cost is looping through batches. We loop through the $O(1)$ batches of I_p and I_v, the $O(\log|C_p|)$ batches of C_p, and the $O(\log|C_v|)$ batches of C_v. Recall that $|C_v| \leq |C_p| + 1$. So in short, by the argument of §6.24, the above algorithm creates C_v from C_p in total time $O(\log|C_p|)$—as promised in §6.23—with the possible exception of line 17. We turn now to this binary search.

Finding edges from edge tags

§6.27 Development. The binary search of the edge tags of C_v must make each comparison with $O(1)$ overhead. One issue is looking up the array elements $C_v[j]$, given that they are stored in batches. But this is not difficult. C_v is "really" represented as an array of batches. We know which batch holds the tag $C_v[j]$; we simply look up that batch and extract the tag $C_v[j]$ from it, in $O(1)$ time.

A more serious issue is the need to look up an edge weight on each comparison. Conceptually, $C_v[j]$ (once we have extracted it) stands for an edge e. However, it is represented as a short bitstring that has some value $d \leq \log n$ as a binary number. This is not enough to determine the edge or the edge weight; it merely indicates that e is the dth edge on the root-to-v path. We could trace up through the ancestors of v to find the edge itself—but not in constant time.

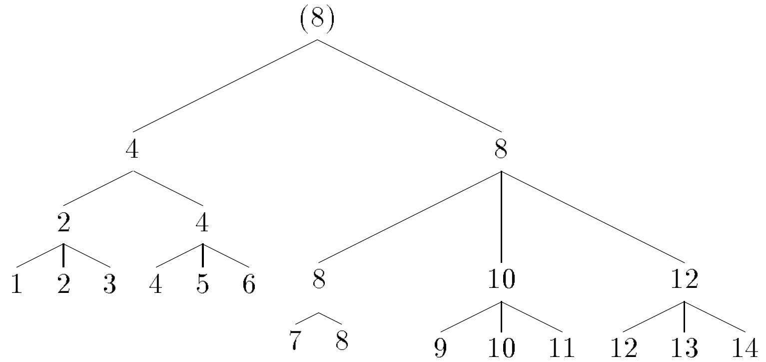
Let us develop a constant-time method by successive refinement. One idea is to look up the edge in a precomputed array. For example, every node of $\mathbf{B}$ could store an array of its ancestors, indexed by their distance d from the root. Unfortunately, these arrays would have $O(n\log n)$ entries total, so would take too long to precompute. We could avoid some duplication by storing such arrays only at the leaves of $\mathbf{B}$ and having each node point to a canonical leaf descendant. However, this does not work either. Each leaf needs to store its own $\log n$ ancestors, for a total of $O(n\log n)$ entries.

The solution is to maintain arrays of *different* lengths at the leaves. No edge will be stored in more than one array. We will partition the edges of $\mathbf{B}$ into n disjoint paths of various lengths: each path ascends from a leaf part-way to the root, and its remaining ancestors are stored on other paths. The leaf stores an array of the edges on its path, indexed by the edge heights. To find an edge quickly, we need to know both its height *and* which path it is on. Then we can look it up in $O(1)$ time at the appropriate leaf.

King doubles the size of edge tags so that they store not only the height of the desired edge, but also a hint as to which path the edge is on. (We can now fit only half as many edge tags into each each s-bit batch, so we need twice as many batches to store C_v, and the loops in §6.26 take twice as long.)

King's scheme is modified from [15]. We will label all nodes of **B** with integers in $[1, n]$. Label the leaves 1 through n in the order encountered by depth-first search. Thus the descendant leaves of any internal node v are labeled with consecutive integers. One (and only one) of these integers k has maximum *rank*, and the internal node v inherits the label k. The rank of a positive integer is defined as the highest power of 2 that divides it (e.g., the rank of 28 is 2, since it is divisible by 2^2 but not 2^3).

Example:



The disjoint paths of **B** are now defined by edges whose lower endpoints share a label. For example, the "10" path has length 2. Leaf 10 stores its ancestor edges of heights 0 and 1, but its ancestor edge at height 2 rises from a node labeled 8, so is stored in the array at leaf 8 instead, together with other ancestors of that leaf.

Simply knowing what path a node is on does not let us determine its ancestor nodes, unless we search the tree. For example, leaf 11 has ancestors numbered 11, 10, 8, and 8, but with a slightly different tree topology, its ancestors would have been 11, 12, 8, and 8. However, we can determine the ancestors if we also know their ranks. (The only possible ancestors of 11 having ranks 0, 1, 2, and 3 are 11, 10, 12, and 8 respectively. In the example shown, there is no rank-2 ancestor.) Thus, we identify an edge by its height (which determines the edge) and the rank of its lower endpoint (which enables us to find it quickly).

This method is based on some nice properties of ranks. The ranks of the numbers 1, 2, 3, 4, 5, ... are 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4 ... Between any two distinct numbers of the same rank, there is one of higher rank; this is an inductive generalization of the fact that between any two distinct odd numbers is an even number. We would like to know that a sequence of consecutive numbers, in particular, the leaves descending from a given node, will have a unique element of maximum rank. This is true, for if there were two such, an element of even higher rank would intervene. We would also like to know that if i is in such a sequence, of which the maximum-rank element is known to have rank r, then that element (call it j) is uniquely determined. Let $j_1 \leq i$ be maximal of rank r, and $j_2 \geq i$ be minimal of rank r, and k be the number of higher rank between them. k is known not to be in the sequence, so if $k > i$ then necessarily $j = j_1$, and if $k < i$ then necessarily $j = j_2$.

§6.28 IMPLEMENTATION DETAILS. We represent each node label as a vector of $\log n$ bits, stored at the node in three batches (bitstrings of length s). Our first job is to number the leaves of **B**. We can use a lookup table to implement *increment*(b), which returns the bitstring denoting $b+1$ together with a 0 or 1 carry bit; then we can increment a leaf label via three calls to *increment* in $O(1)$ time. We must also label the internal nodes of **B**. The rank of a label is the number of terminal zeroes in its bit vector; we design lookup tables that will compare the ranks of bit vectors, so that we can label an internal node with the maximum-rank label of its children.

Implementing more informative edge tags

An edge tag now consists of not one but two bitstrings of length $\lg \log n$ each, each representing a number in $[0, \log n)$. One gives the edge height (determining the edge), and the other gives the rank of the lower endpoint (enabling us to find it quickly). The rank-r ancestor of a node with label i is found by removing the final $r + 1$ bits of i and replacing them with a 1 and r zeroes. If C_v contains an edge tag $\langle \textit{height}, \textit{rank} \rangle$, we use the label of v together with *rank* to discover the label of the lower endpoint of the desired edge. We then look up the edge by *height*, in the array at the leaf of that label.

§6.29 DEVELOPMENT. Finally, as promised in §6.23, we must revise the method of §6.26 so that deriving C_v from C_p takes time only $O(\lg |C_v|)$, not $O(\lg |C_p|)$. The $O(\lg |C_p|)$ component of §6.26 is the extraction of tags from C_p to copy to C_v, for which we iterate in batches through C_p, guided by the parallel bit array I.

Use pointers whenever we don't have time to copy

Recall that there are always fewer batches per array ($O(\lg \log n)$) than tags per full-sized batch ($\Theta(\log n)$). Suppose that the child C_v is more than one batch long. Then $|C_p|$ (the number of tags in C_p) is less than $|C_v|$ times as long as $|C_v|$ itself, whence $\lg |C_p| < 2 \lg |C_v|$—so we have nothing to fix. Moreover, if the parent C_p is one batch or less, then again there is nothing to fix because the procedure is only $O(1)$. Thus, the problem only arises in the context of deriving a small child C_v from a big parent C_p. Here "small" has the technical meaning "no more than one batch's worth of tags."

King's technique is quite clever. Before creating C_v, we determine how many edge tags it will contain, using a lookup function that counts the number of bits in I_v. If C_v will be longer than one batch of tags, it is called *small*, otherwise *big*. The crucial case is where a small C_v must be created from a big C_p. Specially for this case, rather than iterate through all of C_p to extract a small list of edge tags as specified by I_v (§6.26, line 13), we use I simply to create a list of "pointers" to the relevant tags in C_p. For example, if C_p is conceptually a list of the edge tags 3, 5, 8, 9, and 10 (denoting edges at these distances from the root), and I is the indexing bit vector 10100, then instead of storing copies of the edge tags 3 and 8 in C_v we will store 0 and 2, which indicate at which positions of C_p the real edge tags can be found.

Pointers are the same size as edge tags, namely $\leq \log n$ as integers or $\lg \log n$ bits as bitstrings. It is a simple exercise with lookup tables to create our list of pointers directly from I, in time at most proportional to the number of batches in the resulting list, i.e., $O(\lg |C_v|)$ as desired.[9] C_p need not be consulted at all.

Like edge tags, pointers to edge tags can be used to look up edge weights in constant time. This lookup just takes two operations rather than one, the additional operation being a "dereference"—an array access into C_p in which one bitstring is used to index another. However, it is important to keep track of when a field is to be interpreted as a pointer requiring dereference, rather than as an edge tag! Our list of pointers C_v will have its tail overwritten with genuine edge tags (§6.26.18), and additional edge tags may be appended to it (§6.26.26). Therefore, at every node v of $\mathbf{B}$, we maintain an integer $\textit{numpointers}_v$ that indicates how many (perhaps 0) of the initial $O(\lg \log n)$-bit fields in C_v are to be interpreted as pointers rather than edge tags. (An alternative solution does not use *numpointers*, but rather extends each field with a type bit that indicates whether it is a pointer.) We also have v store p in a variable $\textit{pointee}_v$. This tells us where to go to dereference pointers in C_v.

Now that small C_v has been created from big C_p, what happens to the pointers at C_v's descendants?

- To propagate a small parent list down to a small child list, we follow the usual procedure of §6.26. That is, we copy the selected fields of the parent, in batches, without regard to whether they are edge tags or pointers. This is the same procedure that is used for

[9] One solution (not the only one) involves a lookup table whose entries are lists of $\Theta(\lg \log n)$ bitstrings. Each entry takes longer than $O(s)$ time to compute, but the table can still be easily precomputed in $O(n)$ time.

propagating big lists to big lists. The only twist is that we must set the *pointee* of the child to that of the parent, and compute *numpointers* at the child according to how many of the parent's pointers were selected.

- Small lists can also have big descendants, thanks to the accumulation of new edges as we descend through the tree (§6.26.26). These big descendants may have small children. If we are not careful, we will end up with multiple levels of pointers, damaging our $O(1)$ edge lookup. Therefore, when we create a big child C_v from a small parent C_p, we first create a version $\hat{C}_p$ of the parent in which all the pointers have been dereferenced. Then we create C_v from $\hat{C}_p$ in the usual way. In this way, big lists never contain any pointers.

How do we create $\hat{C}_p$? We certainly do not have enough time to dereference all the pointers in C_p individually, nor is there any apparent way to dereference in batches; but we must somehow use batches to get the dereferenced versions. Suppose it is an ancestor C_a to which the pointers in C_p point. (C_a is a big list and contains no pointers itself.) We create $\hat{C}_p$ from C_a just as if p were the child of a rather than an arbitrary descendant, using I_p and I_a to decide which entries of C_a to propagate to $\hat{C}_p$.[10] However, when creating $\hat{C}_p$, we only run the algorithm of §6.26 up to line 15. To complete the construction of $\hat{C}_p$, it would not be enough to take just one new edge into account: we must take account of all edges on the path from a down to p. Fortunately this is work we have already done. The tail of C_p that consists of edge tags, rather than pointers, records precisely the set of updates involving those edges. We therefore delete all but the initial $numpointers_p$ fields of $\hat{C}_p$, and replace them with this tail, which starts at position $numpointers_p$ in C_p.

[10] Notice that this involves copying a big list to a small list—which takes more time than we have to spend on a small list (the reason for using pointers in the first place). We are allowed to do it here because we are creating $\hat{C}_p$ in the process of creating a big list C_v.

Chapter §7

A linear-time randomized algorithm

Karger, Klein, & Tarjan [8] have exhibited a Las Vegas algorithm that solves the MST problem in expected linear time $O(m)$. Indeed, the algorithm violates a certain $O(m)$ bound with only exponentially small probability $p = e^{-\Omega(m)}$.

This result is remarkable, as no linear deterministic algorithm is known. It suggests that the MST problem is not intrinsically bound by the cost of Union-Find (assuming there is no randomized linear algorithm for that problem). In the same way, the Fredman & Tarjan result [5] showed that MST was not bound by sorting (§5.2).

The original presentation in [8] (given in terms of coin-flips) is an admirably clear introduction to the algorithm. It is largely for the sake of variety, and some extra perspective, that I organize my exposition differently. I also offer some remarks about how the algorithm might have been inspired and how it exploits randomness.

§7.1 Remark. Karger *et al.*'s algorithm uses randomness in the service of divide-and-conquer. A strength of randomized algorithms is that they can give us a reasonably even division of a large problem into subproblems, with high probability. This is necessary for divide-and-conquer strategies to work well. *Randomized divide & conquer*

Classic examples are Quicksort and Order Selection, in which we partition a set of m numbers according to whether they are greater or lesser than some pivot value. The problem must then be solved recursively for one or both sets of the partition. For this strategy to achieve good time bounds, we must stay away from extreme pivot values, so that both the partition sets will be appreciably smaller than the original (i.e., size $< cm$ for some fixed $c < 1$) and we therefore do not have to recurse too many times. However, simple deterministic strategies for choosing a good pivot can be foiled by an adversary. Randomized versions of these algorithms [12, p. 15] choose the pivot randomly from the set, thereby assuring a fairly even division on average.[1] As we will see in §7.13, Karger *et al.* use randomness to partition the set of graph edges in a strikingly different way.

§7.2 Motivation. One might attempt to construct an MST deterministically by successively revising an initial guess. Use DFS to find an arbitrary spanning tree T of G. Now iterate through the edges of $G - T$. If an edge uv is heavier than $\textsc{MaxEdge}(u, v)$ in T (as defined in §6.1), uv cannot be in the MST and we discard it; otherwise we use uv to replace $\textsc{MaxEdge}(u, v)$, which cannot be in the MST. To see correctness, observe that each of the $m - (n - 1)$ iterations discards one edge that cannot be in the MST. At the end of the loop, T consists of the only $n - 1$ edges that we have not discarded, so it is the MST. *Building an MST by successive revision*

[1] Randomization is not essential for pivot selection, since there is a complex but asymptotically $O(m)$ deterministic algorithm that will select the median [1]. In the case of MST, randomization is essential for all we know at present.

The difficulty with this algorithm is that T keeps changing, so it is hard to answer the MaxEdge queries rapidly. One response is to modify the algorithm so that we do not change T at all until after the loop through edges of $G - T$, and then make a batch of changes. The next section explores this approach.

§7.3 Motivation. King's verification method (described above in §6) makes MaxEdge queries efficient if T is held constant: the queries can share processing. If King's method is asked whether a spanning tree T is the MST of G, it does not merely answer yes or no. The outer loop of the algorithm (see §6.1) also computes some rather useful information that can help us revise T into the correct MST, M. In particular, it identifies all edges $uv \notin T$ that are heavier than MaxEdge(u, v) in T. Such edges cannot possibly be in M.[2]

Batching queries using King's

If $T = M$, then King's algorithm will rule out all edges of $G-T$ in this way, thereby verifying that $T = M$. If T is close to M, then King's algorithm will still rule out most edges of $G - T$. We can then confine our search for M to the edges that have not been ruled out (including those in T). Indeed, we can find M simply by computing the MST of the few surviving edges.

To exploit this idea, we must be able to find a spanning tree T that is "close to correct," a problem to which we now turn.

§7.4 Scenario. Imagine that for the sake of authenticity, we have implemented Kruskal's 1956 algorithm on equally venerable hardware. The machine always manages to creak through the edges of the input graph G from lightest to heaviest, correctly determining whether each edge e completes a cycle in the growing forest F. If so, e is called *heavy* (with respect to this run of the algorithm), otherwise *light*. Whenever the algorithm finds a light edge e, it attempts to add it to F using a procedure Add. Unfortunately, due to an intermittent hardware failure, each call to Add has an independent 1/2 chance of failing and not adding anything to F. Note that whether or not a given light edge is added may affect the classification of subsequent edges as light or heavy.

Kruskal's with random errors

§7.5 Discussion. How fault-tolerant is this system? Some fraction of the light edges were effectively rendered *invisible* by hardware failures. Thus, the device has actually executed Kruskal's algorithm perfectly, but on the wrong graph: it has found the MST of $G' = G - \{\text{invisible light edges}\}$. More precisely, since G' might not be connected, what the device has found is its minimum spanning *forest* (MSF).[3] We write $M(G')$ for the MSF of G', and likewise for other graphs.

False positives, not false negatives

When a light edge of G is rendered invisible through a hardware error, the classification of subsequent edges is affected. However, this effect is entirely in one direction: some edges *outside* $M(G)$ are "incorrectly" classified as light. That is, if a light edge is not added, a subsequent edge not in $M(G)$ but across the same cut may also be deemed light. But no edge *inside* $M(G)$ is ever classified as heavy. Indeed, each time the faulty machine leaves a heavy edge e out of F, it is justified in doing so by a reliable witness that $e \notin M(G)$. This witness takes the form of a cycle in G (in fact in $F + e \subseteq G' \subseteq G$) on which e is heaviest.

[2]The successive revision algorithm suggested in §7.2 would also make good use of the remaining cases, where $uv \notin T$ is lighter than MaxEdge(u, v) in T, indicating that MaxEdge$(u, v) \notin M$. King's algorithm obviously tells us about those, too: that is, in addition to identifying edges of $G-T$ that cannot be in M, it also identifies some edges *in* T that cannot be in M either. However, Karger *et al.* ignore this property because it is not in general very helpful at identifying bad edges of T. Even if all T's edges are wrong, and $G-T$ has many edges and hence many queries, it is possible for every query to pick out the *same* bad edge of T. This bad edge never gets deleted while we are considering T, because we are holding T constant, and since there may therefore be only one such edge there is little point in deleting it at the end of the pass.

[3]Defined as the spanning forest of G' having minimum weight, among just those forests with as many components as G'.

§7.6 DISCUSSION. Thus Kruskal's algorithm is quite useful even on faulty hardware. It lets us eliminate all but about $2n$ edges as candidate edges of $M(G)$, the rest being exposed as heavy and unsuitable during the construction of F. The only edges not eliminated are the visible and invisible light edges, i.e., the edges sent to the faulty ADD routine.

Kruskal's with random errors is an edge filter

§7.7 JUSTIFICATION. Why won't there be many more than $2n$ light edges? Because for F to find $3n$, or $4n$, or $5n$ light edges, ADD must randomly fail on an inordinate number of these edges. After all, once ADD has randomly succeeded $n-1$ times (if it ever does), F is a tree and all subsequent edges are heavy. So the number of light edges we actually send to ADD is certainly fewer than the number we need to get n successes, which is a random variable with a negative binomial distribution $NegBin_{n,1/2}$. This variable has expected value $2n$ and low variance.

§7.8 DEVELOPMENT. Thus, if we run an input graph through our antiquated Kruskal's device, we may not get quite the correct result, but we do manage to reduce the problem drastically. All that remains is to find the MST of the $\approx 2n$ visible and invisible light edges that the device did not explicitly reject.

Recovering the result of the filter

Of course, to solve the reduced problem of finding the MST on light edges, we must first identify which $\approx 2n$ edges were light. The device—a notional black box—produces only F, a forest of the $\leq n-1$ *visible* light edges. It does not output any indication as to which of the other edges of G were heavy, and which were invisible light edges.

Fortunately we can reconstruct this distinction after the fact, given F, in $O(m)$ time. An edge $uv \notin F$ will have been rejected as heavy just if the part of F that the device had constructed before considering uv already had a u-v path. Equivalently, $uv \notin F$ is heavy iff F has a u-v path consisting of lighter edges. That is, $uv \notin F$ is heavy iff it is heavier than MAXEDGE(u, v) in F. King's verification method (§6) is precisely suited to sift through all the edges of $G - F$ and determine which ones have this property.

§7.9 IMPLEMENTATION DETAIL. The above is exactly the application of King's algorithm foreshadowed in §7.3. We produce a tree $F \subseteq G$ that is close to correct (using a black box equivalent to the randomized Kruskal's method), use it to exclude many edges of G in $O(m)$ time via King's algorithm, and then find the MST of the surviving edges.

Handling disconnected graphs

There is just one hitch: because of the use of randomness, the F we produce is in general not a tree but a forest. Indeed, not even G need be connected: the recursive call in §7.12.2 below may request the MSF of a disconnected graph. But King's algorithm expects F to be connected.

A straightforward remedy has $O(n)$ overhead, for a total time of $O(m+n)$. (This does not damage the analysis in §7.16, even though for disconnected input G it may not be $O(m)$.) We simply connect the components of F before passing it to King's algorithm. Pick one vertex in each component of F, using depth-first search ($O(n)$), and connect these vertices into a path by adding $O(n)$ new, infinitely heavy edges. We use King's algorithm on the resulting tree $\bar{F}$ to find MAXEDGE(u, v) in $\bar{F}$ for each of the $O(m)$ edges uv of $G - F$. Note that when we use this procedure in §7.8, any (finite) edge of G that connects components of F will be correctly classified as an invisible light edge, and hence a candidate for the true tree.[4]

§7.10 DISCUSSION. Alas, running our device (or simulating one on modern hardware) is expensive. It means running Kruskal's algorithm, with all the sorting overhead that implies.

How to implement random Kruskal's?

One might try to mimic the output of the device (i.e., find F) using a different algorithm. The device finds the MSF of $G' = G - \{\text{invisible light edges}\}$. We could try to do the same by

[4]This method seems simpler than using "an adaptation" of the verification method of [4] or [9], as Karger *et al.* suggest. They note that [4, p. 1188] describes such an adaptation, but do not give one for King's method [9].

finding G' and recursively finding its MSF. But it is hard to see how to find G' without running Kruskal's to identify the light edges; and even if we can find it, computing its MST is nearly as hard as computing the MST of G (since G' is nearly as big as G), so we have not reduced the problem much.

§7.11 DEVELOPMENT. Fortunately, there is a clever indirect way to find $M(G')$ without finding G' itself! We will strike our device a precise blow with a crowbar, damaging it further, so that each heavy edge is now also ignored with independent probability 1/2, just like light edges. But this blow really has no discernible effect. Heavy edges were never added to F even when seen; they have no influence on the behavior of the algorithm, so it makes no difference to ignore some of them. So the stricken device still finds $M(G')$.

MSF of a random subgraph

In short, Karger *et al.*'s insight is that by making *every* edge of G invisible with independent probability 1/2, we obtain a graph

$$H = G - \{\text{invisible light and heavy edges}\} = G' - \{\text{invisible heavy edges of } G'\}$$

that has the same MSF as G' does. The MSF is the same because we can eliminate any number of non-MSF edges from G' (or any graph) without changing its MSF. Yet H is much easier to work with than G'. We can easily find H in time $O(m)$ with the help of a random bit generator, since there is no need to differentiate between light and heavy edges. Moreover, H is expected to have only half as many edges as G, so it is cheap to recursively find the MSF of H.

§7.12 DEVELOPMENT. We have now sketched the following algorithm for finding the MSF of an arbitrary graph G.

Recursion with Borůvka contraction

1. Construct H by randomly selecting edges of G with probability 1/2.
2. Recursively find F, the MSF of H, where H has expected $m/2$ edges and need not be connected.
3. Connect F's components into a tree $\bar{F}$ by adding infinitely heavy edges.
4. Run King's algorithm on $\bar{F}$, querying edges of G to find those too heavy to be in $M(G)$.
5. Remove these unwanted edges from G, leaving expected $2n$ or fewer edges.
6. Recursively find the MSF of this pruned G, and return it.

We must specify termination conditions for the recursion: say we return G if $m = 0$. But the resulting algorithm cannot be quite right, for two reasons. First, m does not decrease rapidly to 0: the second recursive call is always on a graph where $m \approx 2n$, so the algorithm gets stuck on that value of m and recurses more-or-less forever. Second, the algorithm never actually discovers any edges: the termination condition means that only the empty graph can ever be returned!

To solve these problems, we begin every recursive call by contracting the argument graph G along some edges of its MSF: we make two passes of Borůvka's algorithm (§4.7). We then proceed with steps 1–6. The contractions force n to drop by a factor of at least 4 with every level of recursion. In particular, the algorithm no longer gets stuck at $m \approx 2n$, since n keeps falling. Our termination condition is now that we return the argument G if $n = 1$. In line 6, rather than return the MSF of the contracted graph G, we "undo" the two contractions so that, as desired, we are returning the MSF of the graph that was passed to us. To do this we replace each vertex of the contracted MSF with the tree of order ≥ 4 that was contracted to form that vertex.

§7.13 ANALOGY. Before we turn to the complexity analysis, let us ask: What is the advantage that randomness confers on us here? One way of formulating the MST problem is that we are given m distinctly weighted edges and wish to find the smallest n, subject to the constraint that they not cause cycles. It is instructive to compare several strategies for a simplified, graph-free version: given m distinct weights, find the smallest n.

Understanding "blind split" on a simpler problem

$O(m \log m)$ **"Kruskal's":** Sort all the weights, then take the smallest n.

$O(m + n \log m)$ **"Prim's":** INSERT all m weights into a Fibonacci heap, in amortized $O(1)$ time each; or simply construct a binary heap of the m weights in a single $O(m)$ pass. Then perform n EXTRACT-MIN operations, in amortized $O(\log m)$ time each. (Note that the real Prim's algorithm "improves" this to $O(m+n \log n)$, though that is asymptotically the same, by exploiting the graph structure and DECREASE-KEY to limit the heap to n rather than m elements.)

$O(m)$**:** We find the nth smallest weight using a recursive procedure. (Then we finish up by scanning the dataset once to find all smaller weights.) The procedure is to pick a pivot element that is expected to be close to the median, partition the dataset about this element, and then recurse on the appropriate half of the partition. We can choose the pivot either randomly or deterministically in $O(m)$ time (see §7.1).

$O(m)$ **"Karger** *et al.***":** As above, we will find the nth smallest weight. Randomly select roughly half the dataset, selecting each of the m weights with independent probability 1/2 (cf. §7.12.1). Recursively find the nth smallest weight in this half-set (cf. §7.12.2). Now remove from the original dataset all weights larger than this (cf. §7.12.4–5). What remains is the smallest $2n$ (on average) weights. The final step is to find the nth smallest of these (using the previous method, so that the algorithm terminates).[5]

Notice how the last two solutions differ. Both recurse on about a set of size $m/2$. The first solution *carefully* splits the dataset into two very *different* halves, so that the desired element is known to be in one half and the other half can be definitively eliminated. This would presumably be difficult for the MST problem. The second solution *blindly* splits the dataset into two very *similar* halves, so that the answer on one half gives a good guess as to the answer on the other half.

The first solution does additional work up front in order to support its careful split. In particular it carries out $O(m)$ comparisons against a pivot element (which in the deterministic algorithm is itself chosen by a small recursion). Only then does it recurse. By contrast, the second solution does its additional work at the end. Once it has recursed on the first half, it deploys $O(m)$ comparisons to "filter" the second half using the answer to the first half.

What does "filter" mean here? Light sets of weights chase out heavy weights, and light paths chase out heavy edges. Put another way, if we identify the weight of a set or path with its heaviest element or MAXEDGE:

- Any set S of n weights establishes an upper bound on the lightest such set, $\hat{S}$. The lighter the set S, the stronger the bound it provides. In particular, every element of $\hat{S}$ is bounded by $\max(S)$. Heavier weights cannot be in $\hat{S}$, which we wish to find.

- Any path Φ from u to v establishes an upper bound on the lightest such path, $\hat{\Phi}$. The lighter the path Φ, the stronger the bound it provides. In particular, every edge of $\hat{\Phi}$ is bounded by the maximum edge of Φ. Heavier edges cannot be on $\hat{\Phi}$, which (by the cycle property of §3.3) is the unique u-v path in the MST, which we wish to find. It follows that if uv is such a heavier edge, it cannot be in the MST.

[5]To make the algorithm stand on its own (merely as a point of interest), we would like to search these final $\approx 2n$ weights recursively. I see no analogy to the Borůvka step introduced for this purpose in §7.12. A different strategy to make n decrease is as follows. When recursing on the half-set, we find not the nth but the $(2n/3)$th smallest weight. As before, we search the full set to find all the weights smaller than this. We now recurse on these k smallest weights to find the $(n-k+1)$th *largest* weight, where on average $k \approx 4n/3$ and $n-k+1 \approx n/3$. Or if we are unlucky enough to have $k < n$, we try again with a new division into half-sets. For n above some small constant c, the chance of $k < n$ is small enough that the cost of such retries is tolerable. Our base cases for the recursion include not just $n = 1$ but rather any case $n \leq c$; for these cases, we bottom-out with the $O(m \log n)$ strategy given above, which takes $O(m)$ time since $n \leq c$.

The Karger-style solution is to locate a set of n weights that has a relatively small maximum, or a tree of $\approx n$ edges whose paths have relatively light maxima. We locate this set or tree in one half of a blind partition, recursing on a size-$m/2$ problem, so that we know it provides a strong enough bound to chase away the rest of that half. We then spend linear time (courtesy of King's algorithm) using it to chase away most of the other half. Since the two halves are similar, the set or tree should be an equally strong bound on both halves, and chase away all but about n elements of each. This leaves us with only a shrunken, size-$2n$ problem.

Edge competitions for eliminating edges

§7.14 REMARK. This "blind split" strategy may generalize to other problems. The essence of the strategy: some comparisons are more useful than others, so recurse on part of the input in order to figure out what comparisons are likely to be most beneficial on the rest of the input.

This leads to the question: For the MST problem, which comparisons are the useful ones? We know from §6.19 that an MST can be determined with few comparisons, if we are lucky enough to choose the right ones.

Let us say that two edges *compete* if they are known to be the two heaviest edges on some cycle. The Strong Cycle Property (§3.3) can be restated as follows: "Non-edges of the MST are exactly those that are heavier than some competitor." (Equivalently, "Edges of the MST are exactly those that are lighter than all of their competitors.")

It is always useful to compare two competing edges directly: for in such a *direct competition*, the *loser* (heavier edge) can immediately be ruled out of the tree. Assuming that a loser never competes again, $m - (n - 1)$ direct competitions necessarily determine the MST. Other comparisons are necessary to find the competitors.

King's algorithm is more gladiatorial than most. It is set up specifically to stage direct competitions between edges in a given tree and edges outside it. For every edge e outside the tree, King uses average $O(1)$ comparisons to find a worthy competing edge e' in the tree, and then compares e with e'. (Determining this competing edge is analogous to determining $\max(S)$, the maximum of a known set of weights, in §7.13.)

Karger *et al.*'s algorithm stages exactly the same competitions, but in the service of constructing an MST rather than verifying one. They find F from one half H of the dataset, and then use the edges in F to knock out most edges (or so they hope) from $G - H$, the other half. Typically F contains some bad edges as well as good ones. Useful comparisons are just those direct competitions in which the edge from $G - H$ loses to the edge from F.[6]

Intuitively, why should the winning edges F from the first half of the dataset tend to yield useful comparisons in the second half? Put differently, why should we expect cycles from $F \subseteq H$ to suffice to witness the bad edges in $G - H$? After all, many of the cycles in G are not completely contained in H, so the witnesses we need may not be available.

Suppose $e \in G - H$ is not in the MST, but F does not witness that fact. Then if we had run our original Kruskal's device (§7.5), e would have been one of those non-MST edges "mistakenly" classified as light (but not added). We concluded in §7.7 that there could not be more than about n such edges, because if the device considered many light edges it would quickly find a tree. (We might have to consider many edges, but not light ones.) Let us translate this into a story about $G - H$. Suppose a particular light path in the MST is needed to witness the badness of $e \in G - H$, but part of this path is missing from H and hence from F. If there are many edges like e that rely on the path, at least one of the lightest such will surely appear in H. It will be added to F, where it will help complete a similar path that can rule out other such edges, like $e \in G - H$. On the other hand, if there are few edges like e, then not much harm is done if we fail to rule them out, since there are not many.

[6] If the edge from F happens to lose, then we have gained nothing, since we do not eliminate it from consideration. More precisely, the first time that an edge from F loses a direct competition, we have gained information that we do not use. The next time it loses a direct competition, we have not gained even that—and the same edge in F may lose many times during a call to King's algorithm. See footnote 2.

§7.15 REMARK. Direct competitions are efficient, but are by no means the only way to discover that an edge e is heavier than one of its competitors e'. There are also indirect ways of discovering this, and thereby eliminating e from the MST:

Alternatives to edge competitions

- If Kruskal's algorithm uses Quicksort, it may discover that $e > p$ and that $p > e'$. But p is just a pivot element used for partitioning, and in general competes with neither e nor e'.
- When Borůvka's algorithm scans for the lightest edge e' incident on a vertex, its comparisons are not in general direct competitions. They merely prove by contradiction that if e' has any competitors it can beat them.
- When Prim's algorithm eliminates all but the lightest edge in a bucket, its comparisons are not in general direct competitions. (One of the edges being compared is indeed the heaviest edge in the cycle, but the other need not be the second-heaviest.) The same applies when Borůvka's algorithm eliminates all but the lightest edge between two vertices.

§7.16 ANALYSIS. I use $\langle m, n\rangle$ to denote a call of the MSF function on a graph with size m and order n. The initial call is $\langle m_0, n_0\rangle$ It is clear from §7.12 that a call $\langle m, n\rangle$ takes time $O(m + n)$, exclusive of recursive calls. It is not quite right to reduce this to $O(m)$, as [8] does, since G may be horribly disconnected, with $n > m$. (After all, it may be a randomly chosen subgraph on a recursive call.) For this general case each Borůvka pass takes time $O(m + n)$, not $O(m)$, as do each of lines 1, 3, and 4 in §7.12.

The binary tree of recursive calls

So the total running time is proportional to the total number of edges plus the total number of vertices over all calls. Following [8], we consider the binary tree of recursive calls: each call $\langle m, n\rangle$ (with $n > 1$) spawns both a left-child call in §7.12.2 and a right-child call in §7.12.6. These child calls have at most $n/4$ vertices apiece, thanks to the two Borůvka passes. So the call tree has $O(n_0)$ vertices total. We now consider the total number of edges in the call tree, which will dominate the analyses.

§7.17 ANALYSIS (EXPECTED TIME ON ARBITRARY INPUT). This is straightforward. The call $\langle m, n\rangle$ has a left child of expected size $\langle m/2, n/4\rangle$ or smaller, and a right child of expected size $\langle n/2, n/4\rangle$ or smaller. We draw two conclusions:

Expected time $O(m)$

- At depth d in the tree, there are 2^{d-1} right children of expected size at most $\left\langle 2n_0/4^d, n_0/4^d\right\rangle$.
- The call $\langle m, n\rangle$ together with all its left-spine descendants has at most $m + m/2 + m/4 + \cdots = 2m$ expected edges.

Every call is a left-spine descendant of the root or of a right child. The root and its left-spine descendants expect at most $2m_0$ total edges. A right child at depth d and its left-spine descendants expect at most $2 \cdot 2n_0/4^d = n_0/4^{d-1}$ total edges. Summing these bounds over the root and all right children, we obtain a bound on expected edges of $2m_0 + \sum_{d=1}^{\infty} 2^{d-1} n_0/4^{d-1} = 2m_0 + 2n_0$. (Not $2m_0 + n_0$ as stated in [8], which forgets momentarily that right children also have left descendants.) So the expected time is $O(m_0)$.

§7.18 ANALYSIS (PROBABILITY OF RUNNING IN $O(m_0)$ TIME). The nub of the argument is that a call $\langle m, n\rangle$ is very unlikely to have a left child with many more than $m/2$ edges (since each edge is selected with probability 1/2), or a right child with many more than $2n$ edges (see §7.7). "Very unlikely" means $e^{-\Omega(m)}$ and $e^{-\Omega(n)}$ respectively, by Chernoff bounds.

Actual time $O(m)$ exponentially likely

However, there are many recursive calls in the tree, so some of them will probably violate such bounds. Moreover, calls $\langle m, n\rangle$ with small m or n—of which there are many—are not so unlikely to violate the bounds anyway. For this reason we do not attempt to prove edge bounds for individual calls to the MSF function. Rather we consider the *total* number of edges in right children, and show that it is $O(m_0)$ with probability $1 - e^{-\Omega(m_0)}$. We then extend this analysis to cover the total number of edges in all calls.

First consider right children. Each right child consists of the visible and invisible light edges encountered when running the faulty Kruskal's device on (a contracted version of) its parent. That is, it is a forest consisting of the visible light edges, plus about the same number of invisible light edges. The number of visible light edges is limited by the space available for forests on right children: the right children have only $\sum_d 2^{d-1} n_0/4^d = n_0/2 \leq m_0$ vertices total, so there are at most this many visible light edges. If we encountered *more* than $3m_0$ light edges while running the device, then ADD must have succeeded less than $1/3$ of the time in its first $3m_0$ edges. The probability of this is $e^{-\Omega(m_0)}$.

Let $m' \geq m_0$ be the total number of edges in the root and all right children. We have just shown that $m' \leq m_0 + 3m_0 = 4m_0$ except for $e^{-\Omega(m_0)}$ of the time. By considering the left-spine descendants of the root and right children, we will now see that the total number of edges is $\leq 3m'$ except for $e^{-\Omega(m_0)}$ of the time. These results together imply that with probability $1 - e^{-\Omega(m_0)}$, the total number of edges is at most $\leq 12m'$.

Each left child consists of just the visible light and heavy edges encountered when running the device on (a contracted version of) its parent. Put another way, it is a random subgraph of the contracted parent (see §7.12.1). The root, or a right child, gives rise to a sequence of left descendants through successive subgraph operations. Each of the m' "source edges" in the root and right children is selected on some number of these subgraph operations before it is unlucky enough to be contracted or explicitly not selected (i.e., invisiblized).

Suppose the left children contain more than $2m'$ edges, so that we have more than $3m'$ edges total. (This is the event we are claiming is unlikely.) Then the average source edge was selected by left children more than twice before it was contracted or not selected. Since we actually selected $2m'$ edges, we must have made at least $2m'$ selection decisions (though not necessarily $3m'$, as [8] claims, because of the possibility of contraction). Moreover, at least $2/3$ of such decisions came out in the affirmative. The probability of this happening is only $e^{-\Omega(m')}$, and *a fortiori* $e^{-\Omega(m_0)}$.

§7.19 ANALYSIS (WORST-CASE RUNNING TIME). Karger *et al.* show that the algorithm has the same asymptotic worst-case bounds as Borůvka's algorithm. For Borůvka's algorithm, those bounds derive from the fact that the number of edges on pass d is bounded both by $(n_0/2^d)^2$ and by m_0 (§4.11). [8] shows straightforwardly that the total number of edges at depth d in the Karger *et al.* call tree also meets these both bounds.

Worst-case $O(m \cdot (1 + \log(n^2/m)))$, like Borůvka's

Chapter §8

A dynamic algorithm

§8.1 MOTIVATION. Imagine that we must frequently broadcast messages on a communications network (a preindustrial example was sketched in §2). The network is represented by a graph G. Each edge of G represents a bidirectional communications link, which is cheap or expensive to use according to the weight of the edge. To broadcast a message from $v \in G$ to all vertices of G, we send it along the edges of a spanning tree in the obvious way. Using the minimum spanning tree minimizes the cost of the broadcast.

A changing communications network

Edges may change weight often, due to hardware problems or fluctuating system load. Frederickson [6] describes a data structure that can efficiently maintain the MST under such weight changes. Given the initial MST, the data structure may be initialized in time $O(m)$; each weight update then takes $O(\sqrt{m})$ time. Frederickson's scheme extends to handle edge insertions and deletions, also in $O(\sqrt{m})$ time. (Note that true edge deletions reduce m, speeding subsequent operations. For this reason they are preferable to the "pseudo-deletion" of increasing edge weights to ∞.) We will add this feature in §8.18.

Frederickson makes the intuitions fairly clear at each step of his exposition, but the series of steps is complicated (consisting of successive revisions) and the data structures are unfamiliar. I have attempted to compress and clarify the ideas, in part by accomodating the ideas to the notation and data structures of §4.3. These data structures differ only slightly from Frederickson's final structure, a "two-dimensional topology tree." I have also attempted to fill in some details that appear to be missing in [6]—in particular, the implementation of the data structures and some details of the bookkeeping.

§8.2 OVERVIEW. Our main problem is to maintain the MST of a graph G when edge weights change dynamically but the topology does not change. More precisely, the data structure we maintain will always be some subforest of the current MST, since in the course of manipulating the MST while weights are changing, we may temporarily remove an edge.

Operations to support dynamically

So long as we are dealing with forests, I generalize the problem so that G need not be connected. Thus the structure we are maintaining is a subforest of the MSF. We will construct a representation for such a forest, $F \subseteq G$, that supports the following operations (as well as the ability to look up the weight of a given edge):

TEST(e): Determine whether an edge e is in F.

INCWEIGHT($e, \Delta w$): Increase by Δw the weight of edge $e \in G - F$, but without changing F.

REMOVE(e): Remove e from F (without changing G). This increases the number of components of F.

ADD(e): Add e to F (without changing G). This decreases the number of components of F.

MAXEDGEIN(u, v): Return the heaviest edge on the u-v path in F.

MINEDGEOUT(u,v): Return the lightest edge of $G-F$ that connects u's component in F to v's component in F.

We are now able to increase the weight of edge uv by Δw, while maintaining the invariant that we always store a subforest of the correct MSF:[1]

```
if TEST(uv) and Δw > 0        (* edge in F getting heavier *)
    REMOVE(uv)        (* disconnects a component of F *)
    INCWEIGHT(uv, Δw)
    ADD(MINEDGEOUT(u, v))        (* reconnects F with some edge, possibly uv again *)
elseif ¬TEST(uv) and (Δw < 0) and (w(uv) + Δw < w(MAXEDGEIN(u, v)))        (* edge out of F getting lighter *)
    REMOVE(MAXEDGEIN(u, v))
    INCWEIGHT(uv, Δw)
    ADD(uv)
else
    INCWEIGHT(uv, Δw)
```

Dynamic trees plus a new data structure

§8.3 DEVELOPMENT. Our representation of $F \subseteq G$ really consists of *two* data structures, each of which represents F in its own way. One structure is responsible for answering MAXEDGEIN queries; it supports all the operations except for MINEDGEOUT. The complementary structure is responsible for answering MINEDGEOUT queries; it supports all the operations except for MAXEDGEIN. Modifications to F or G (i.e., ADD, REMOVE, INCWEIGHT) are carried out on both structures simultaneously.

The first structure represents the forest F via the *dynamic trees* of Sleator & Tarjan [14], a paper that I will not dwell on here. The dynamic tree structure treats F as a set of vertex-disjoint, edge-weighted, rooted trees that support several $O(\log n)$ operations. So far as we are concerned, the choice of root in each tree is arbitrary. Indeed, Sleator & Tarjan provide an operation *evert*(u) that lets us make u the new root of its tree.

It is trivial to implement the operations of §8.2 using dynamic trees. Our TEST(uv) is true just if $v = \textit{parent}(u)$ or vice-versa. REMOVE(uv) is implemented in the dynamic tree structure as $\textit{cut}(u)$ if $v = \textit{parent}(u)$ and $\textit{cut}(v)$ otherwise. ADD(uv) is implemented as $\textit{evert}(u)$ followed by $\textit{link}(u, v, \mathbf{w}(uv))$. MAXEDGEIN$(u,v)$ is $\textit{evert}(u)$ followed by $\textit{maxcost}(v)$, which finds the maximum edge on the path from v to its new root u. In the same vein, INCWEIGHT$(uv, \Delta w)$ is $\textit{evert}(u)$ followed by $\textit{update}(v, \Delta w)$, which increments all weights on that path.

The second structure must be able to find the lightest edge across a cut, in order to answer MINEDGEOUT. In the worst case there are $O(m)$ such edges. We want to make sure we are never in the position of considering all those edges individually. Frederickson's approach is to group most of the edges into bundles (related to the buckets of §3.10), each of which maintains its minimum edge. Then to find the lightest edge across a cut (MINEDGEOUT), we only have to consider the minima of the bundles crossing that cut. We now develop this idea in detail, completing Frederickson's method.

Dividing F into vertex clusters connected by edge bundles

§8.4 STRATEGY. Let us begin with a sketch, deferring details and implementation. Suppose we partition the vertices of G into clusters of intermediate size. Given distinct clusters C_i and C_j, we keep track of G's lightest edge (if any) from C_i to C_j. Conceptually we are dealing with bundles of the form {edges from C_i to C_j}.[2] But we need not maintain a list of all the edges in each bundle, only the minimum edge of each bundle.

To find MINEDGEOUT(u,v), we look at all bundles that cross the cut. We only have to consider the minimum edge of each bundle; so we can find the minimum edge across the cut

[1] In the data structure of §8.8 below, this invariant ensures that F_i is a subgraph of G_i, where G_i has eliminated all but the lightest edges between various subtrees of F.

[2] Edges within a cluster C_i are not in any bundle, unless for aesthetic reasons we choose to maintain the bundle of edges from C_i to itself. Maintaining such a bundle is unnecessary, since we never need to know the minimum edge within C_i.

more quickly than if we looked at every edge. Specifically, we consider bundles that run from any cluster in u's component to any cluster in v's component. For this to make sense, we must choose our vertex clusters in G so that each does fall within a single component of F. More strongly, we will explicitly choose them by dividing F into subtrees. In general, each component of F will be divided into several subtrees, which remain linked to each other by edges of F.

How long do our operations now take? The expensive operations are MINEDGEOUT and REMOVE. MINEDGEOUT clearly requires time $O(\text{number of clusters}^2)$. If REMOVE($e$) removes an edge of F that was within some cluster C_i, we must split C_i into C_{i_1} and C_{i_2} so that it does not straddle two components of F.[3] All edge bundles from C_i to other clusters are destroyed; we must create new bundles from C_{i_1} and C_{i_2} to these other clusters. We do so by iterating through all edges of G incident on vertices of C_i, bucketing them into new bundles of which we maintain the minima. This requires time $O(\text{degree of } C_i)$, where the degree of a cluster is the total number of edges in the adjacency lists of its vertices.

The other operations are less critical and do not dominate update time. ADD(e) comes for free; it merely marks some edge between clusters as being in F. (Notice that given our use of ADD in §8.2, e is always minimal in its bundle.) INCWEIGHT($e, \Delta w$), an operation Frederickson does not mention, requires us to recompute the minimum edge in e's bundle, which may have changed. Suppose e runs from C_i to C_j. A brute-force solution is to examine *all* edges incident on vertices of C_i and keep a running minimum of those that lead to C_j. This has complexity no worse than that of REMOVE.

§8.5 ROUGH ANALYSIS. These implementations of MINEDGEOUT and REMOVE exert opposing forces on cluster size. MINEDGEOUT prefers that we have few clusters (so that searching the set of bundles is fast). REMOVE prefers that we have many clusters, each with only a small number of vertices (so that splitting is fast) and a low degree in G (so that rebundling is fast). Of these two criteria for REMOVE, low degree is the stronger one: a low degree guarantees an equally low number of vertices, since a large cluster with few incident edges could not be connected in F.

Choosing an intermediate cluster size

This tension between few clusters and low cluster degree is best resolved by having $O(m^{1/3})$ clusters, each with degree $O(m^{2/3})$ in G, for a total of $2m$ directed edges. Then MINEDGEOUT and REMOVE now take $O(m^{2/3})$ time apiece. These operations determine the time cost of our data structure's fundamental update operation, the edge-weight update in §8.2. We will soon improve this basic result to $O(m^{1/2})$. We postpone till §8.11 the question of how to find and maintain the balanced partition into clusters.

§8.6 REMARK. The edge bundles here resemble the buckets used in Prim's algorithm (§3.10). Those buckets maintain the edges that would result from contracting the single growing Prim's tree, if we define contraction to eliminate all but the lightest edge from every set (or "bundle") of multiple edges. In the same way, Frederickson maintains the edges that would result from contracting various subtrees of F (namely, those induced by the vertex clusters). So Frederickson is applying Prim-style bucketing to a Kruskal-style forest of simultaneous trees! The goal is the same: to cheapen subsequent searches for a light cut-crossing edge.

Dynamic algorithms need a balanced data structure

To elaborate, both Frederickson's and Prim's algorithms maintain k trees and the minimum edge between each pair of these trees. (In the case of Prim's, most of these $O(k^2)$ edges are the unique edges between isolated vertices.) For the static algorithm, Prim's, we do not yet know

[3]This split operation is the reason for requiring each vertex cluster to induce a subtree of F. If the subtree loses an edge, we can run DFS on it to determine the split rapidly (in time proportional to the number of vertices in the cluster).

What is the alternative? In principle, we could have chosen each vertex cluster to be a set of vertices scattered throughout a component of F. In that case, removing an edge of F might have split not one but several clusters across the new component boundary. It would have been prohibitively expensive to determine which clusters were split and how the vertices were to be allocated to the new clusters.

which of the $O(k^2)$ edges between trees will be in the MST; while for the dynamic algorithm, Frederickson's, which already has a tree but is maintaining it, such edges *are* marked as to whether they are in the current MSF, F. The key operation that the two algorithms share is to pick a light edge from across the $O(k^2)$ minimum edges that they have been maintaining between trees, in order to bridge a cut in the current forest.

Given this similarity, why do the two algorithms choose their trees so differently? Frederickson insists on a "balanced" forest in which all the trees are about the same size and have about the same degree. Prim is at the other extreme, one large tree and many isolated vertices. Indeed, none of the static algorithms insist on a balanced forest. Although Fredman & Tarjan find it advantageous to limit the degree of Prim's trees grown using the heap, and Borůvka tries to grow all trees in the forest in parallel, both those algorithms allow trees to run into each other and merge as they grow. The result is that some trees may end up much bigger than others.

There is a deep difference between static and dynamic algorithms that add edges according to the Strong Cut property (§3.3): a static algorithm can choose which cuts it will search across, whereas a dynamic algorithm is at the mercy of its caller. In a dynamic graph, any edge could vanish or (equivalently) become very heavy and get removed, creating an arbitrary cut in F that the algorithm must repair. In the worst case there are $k/2$ trees linked together by edges of F on each side of the cut, so the algorithm must consider $k^2/4$ bundles in order to find the lightest crossing edge. A static algorithm such as Prim's beats this performance by always bridging the cut between a single tree and the other $k-1$ trees, meaning that there are only $k-1$ bundles for it to consider. That is why the static algorithm is faster.

Absolute performance aside, why does the dynamic case favor equal trees and the static case favor unequal ones? In Frederickson's algorithm, any of the k trees (subtrees of F induced by the vertex clusters) may be split if one of its edges is removed. It is more expensive to split large trees than small ones, because a large tree has more incident edges that have to be rebundled after splitting. (Remember, we are maintaining the minimum edge between each pair of trees; see the discussion of REMOVE in §8.4.) So for the sake of worst-case performance we avoid having any large trees. Thus dynamic graphs strongly favor equal trees. Static graphs have no strong preference for tree size, so long any edge they add is across an cut that separates one tree from all the others. The reason that Prim's uneven forest turns out to work well is that since Prim's algorithm uses essentially the same cut every time it adds an edge, it does not have to search anew through the $O(n)$ cut-crossing bundles for every edge it adds.[4] Rather it can take advantage of the fact that the cut does not change very much—using a heap, which is just a data structure that rapidly takes account of small changes, percolating minima to the top in $\log m$ time. We now turn to a similar idea of Frederickson's—a kind of heap that percolates minima to the top in the more difficult case of arbitrary *dynamic* changes to the graph.

§8.7 STRATEGY. The solution of §8.4 reduces the search of all cut-crossing edges to a search of only the bundle-minimal ones—still a considerable problem. Fortunately it is a smaller problem of the same form. Hence we can repeat the solution, creating clusters of clusters in order to group the bundle-minimal edges into their own bundles. When an edge weight in G changes, say, the change propagates up through successive levels of bundles, much as in a heap.

Recursively bundling edges, heap-style

We may think of this idea as contracting the clusters used in §8.4 and then recursing. We adapt the notation and data structures of §4.3–§4.6.

[4] An alternative strategy for avoiding such an $O(n)$ search is Borůvka's algorithm (§4.8), which moves from tree to tree, making a *different* uneven cut every time. This has the quite different advantage that it can do a single sweep through all m edges and choose n edges across n different cuts, at least $n/2$ of the chosen edges being distinct. It thereby takes advantage of the fact that only $O(m/n)$ edges will cross an average uneven cut: it takes time $O(m/n)$ to pick each edge on the first pass (and then twice as long on each successive pass, since n drops). By comparison, Prim's algorithm would take $O(n)$ rather than $O(m/n)$ per edge if it were not for the heap, which brings the per-edge cost down to $O(\log n)$.

Initial build of the recursive data structure

§8.8 DEVELOPMENT. Let G_0 be the current state of the dynamic graph, and $F_0 \subseteq G_0$ its minimum spanning forest, represented in our usual way (§4.4). Initial versions of these objects are given to us, and we use them to build our initial data structure from the bottom up. When updates cause us to change F_0 and G_0, the changes will propagate from the bottom up. The resulting structure is one that could have been built in the first place from the revised versions of F_0 and G_0.

Given G_i and F_i, we can initially construct G_{i+1} and F_{i+1} from scratch as follows. We partition F_i evenly into subtrees of size $\approx z_{i+1}$, for some number $z_{i+1} \geq 2$, in a manner to be described in §8.11. We then repeat our contraction maneuver from §4.4:

```
(* create G_{i+1}, F_{i+1} from G_i, F_i *)
V_{i+1} := ∅
for each subtree T in the partition of F_i      (* differs from §4.4, which contracted entire components *)
    add a new vertex v̂ to V_{i+1}, with empty adjacency lists
    C(v̂) := ∅
    for each v ∈ T
        P(v) := v̂;  C(v̂) := C(v̂) ∪ {v};  Γ_{G_{i+1}}(v̂) := Γ_{G_{i+1}}(v̂) ∪ Γ_{G_i}(v)
        Γ_{F_{i+1}}(v̂) := Γ_{F_{i+1}}(v̂) ∪ Γ_{F_i}(v)      (* was not necessary in §4.4, which contracted all edges of F_i so F_{i+1} had none *)
```

We follow this with an $O(m_i)$ step that eliminates self-loops from G_{i+1} and F_{i+1}, and eliminates all but the lightest edge from every bundle of multiple edges. (See §4.12 for one implementation, a radix sort with two passes. If $n_i \leq \sqrt{m}$, as Frederickson arranges throughout by putting $z_1 \geq \sqrt{m}$, a one-pass $O(n_i^2)$ bucket sort will suffice to preserve the analysis.)

We repeat these contractions until we arrive at a graph F_N that is trivial (that is, $m_N = 0, n_N =$ number of components of F_0, so every vertex is isolated). Since n_i falls by about $z_i \geq 2$ on each pass, $N = O(\log n)$.

Pass i takes time $O(m_i)$. The total time for the initial construction is then $O(\sum_{i=0}^{N-1} m_i)$, which as we will see is $O(m)$ given appropriate choices of the z_i.

Storage of edges

§8.9 IMPLEMENTATION DETAILS. There is an important difference between this data structure and the one that §4.1 used for discovering spanning trees: F_0, F_1, F_2, F_3 are all genuinely contractions of the *completed* spanning forest F. In particular, $F_0 = F$ itself, and since F_{i+1} is merely the result of contracting some edges of F_i and eliminating some of the multiple edges that remain, all the edges of F_{i+1} are contained in F_i. The subtrees of F_i that we will contract ("vertex clusters") are not in general separate components of F_i.[5]

We maintain all the graphs $G_0, F_0;\ G_1, F_1;\ \ldots G_N, F_N$ simultaneously. Within each graph, we want our adjacency list to specify neighbors in that graph. Therefore we represent adjacency lists as in the $O(m_i)$ merge of §4.6. There is one extra twist: the entry for uv in $\Gamma(u)$ should specify not only v and the edge, but also point to the entry for vu in $\Gamma(v)$. This means that if we delete the edge uv from u's adjacency list, we can also delete it from v's, in constant time. An alternative storage option, which Frederickson essentially adopts though with different terminology, is to use an adjacency *matrix* in the representations of $G_1, G_1, \ldots G_N$ (but not G_0 or $F_0, F_1, \ldots F_N$). The matrix element $M[u][v]$ holds the edge uv, if any, and null otherwise.

Trivial to bridge cuts on the fully contracted graph

§8.10 DEVELOPMENT. The broad use of the data structure should now become clear: we can always answer MINEDGEOUT(x, y) as the edge from $P^N(x)$ to $P^N(y)$ in G_N.

F_N and G_N are the result of completely contracting F_0 and G_0 over the *components* of F_0. For the usual case, suppose that F_0 currently holds the MSF of G (and not a subforest thereof). Then $F_N = G_N =$ the trivial graph on V_N; each vertex represents a contracted component of $F_0 = G_0$. If we REMOVE an edge from F_0 and propagate the changes up, a vertex $v \in V_N$ will

[5] This is important because for the clustering to be maintained correctly under changes (§8.14), each cluster in F_i must know its outdegree, i.e., how many other clusters of F_i it is connected to in F. Under our representation, this can be determined by considering just the adjacency lists in F_i of vertices in the cluster.

split into two vertices v_1, v_2. This reflects the splitting of a tree in F_0 into two components with vertex sets $C^\infty(v_1)$ and $C^\infty(v_2)$, as defined in §4.3. These components remain connected in G_0, so v_1 and v_2 are connected in G_N by a single edge—namely, the lightest edge of G between the new trees in F_0.

§8.11 DEVELOPMENT. All that is left to do is to explain how to partition a forest into clusters in a balanced way, and how to fix up the partition rapidly for every change to the forest.

Exploding and partitioning the graph

As discussed in §8.5, we would like to limit the degree of each cluster. This is difficult if the degree of the graph is unbounded: any vertex of degree k must be in a cluster of degree $\geq k$. Frederickson simply assumes that the degree is bounded: specifically, $\Delta(G_0) \leq 3$, where $\Delta(G_0)$ denotes the maximum degree of any vertex (length of any adjacency list) in G_0. If this is not true, Frederickson notes, the input graph can be transformed in a standard way: each vertex of degree $k > 3$ may be "exploded" into a path of k vertices, each of which is assigned one of the original vertices' neighbors. The edges in the path have weight $-\infty$ so that they will be part of the MSF. This graph has at most three times as many edges as the original, and has $m = O(n)$. Frederickson's method maintains the MSF of this exploded graph, which supports the usual queries on the MSF of the original graph, at only a constant-time penalty. Notice that the graph gets denser with contraction, but even the denser graphs are guaranteed to respect certain degree bounds: supposing for simplicity that each cluster in F_i contains exactly z_{i+1} vertices, then $\Delta(G_0) \leq 3$, $\Delta(G_1) \leq 3z_1$, $\Delta(G_2) \leq 3z_1z_2, \ldots \Delta(G_i) \leq z_i \cdot \Delta(G_{i-1})$.

In particular, $\Delta(G_0) \leq 3$ implies that $\Delta(F_0) \leq 3$. The method for partitioning F_{i-1} into subtrees expects that $\Delta(F_{i-1}) \leq 3$, and finds subtrees (clusters) in such a way that $\Delta(F_i) \leq 3$ as well (after contraction). Moreover, the method guarantees that every cluster is connected and has from z_i to $3z_i - 2$ vertices, with the exception that a cluster with outdegree 3 (meaning its degree in F_i after contraction) is allowed to have fewer vertices.

The partitioning method, for which [6] gives brief and simple pseudocode, takes time $O(n_{i-1})$. We partition each component of F_{i-1} separately. We root the component of F_{i-1} at an arbitrary node, so each node has at most 2 children except possibly the root, which may have 3. The idea is to repeatedly break minimal legal clusters off the bottom of this tree. Legal clusters are defined as rooted subtrees with outdegree = 3 or $\geq z$ vertices. Every minimal legal cluster has at most $2z_i - 1$ vertices, since its two children are too small to have been broken off as clusters of their own, i.e., they have $\leq z_i - 1$ vertices each. (There is one exception: there may be a minimal cluster of size $3z_i - 2$ at the root, if the root has three children.)

Once all the minimal clusters have been broken off, we may be left with an subminimal cluster of $\leq z_i - 1$ vertices and outdegree < 3 at the root. If so, we merge this tree with any cluster adjacent to it, to get a cluster of $\leq 3z_i - 2$ vertices. It is not hard to see that this cannot increase the outdegree of the adjacent cluster.

This last step fails if we run the partitioning method on any component of F_{i-1} that is so small it has $< z$ vertices, for in that case, there is no adjacent cluster to merge with. In that case, we simply put the whole component into one cluster. It is possible that some components of F shrink down to a single vertex in this way before others, so that F_i contains some isolated vertices for $i < N$. (Frederickson's scheme would treat the components as "topology trees" of different heights.)

§8.12 DEVELOPMENT. Let us now describe how changes propagate up the data structure (not fully spelled out in [6]).

Repairing the structure after changes

We process an ADD(uv) or REMOVE(uv) request in three sequential stages:

(a) Go through $F_0, F_1, \ldots F_N$ in order, and in each, add or remove the edge in question. For each i, we must modify $\Gamma_{F_i}(P^i(u))$ and $\Gamma_{F_i}(P^i(v))$, unless $P^i(u) = P^i(v)$. Each modification takes constant time because these adjacency lists have length ≤ 3.

These modifications may lead to "instabilities" in the clusters containing u and v. Adding an edge from $P^i(u)$ to $P^i(v)$ in F_i may leave either or both with degree 4, which is bad. If $P^i(u)$ and $P^i(v)$ are in the same cluster of F_i, then removing the edge between them will leave the cluster disconnected, which is bad.

(b) Go through $F_0, F_1, \ldots F_N$ in order again, and reorganize the clusters of each F_i to fix up any instabilities created during stage (a). This involves merging and splitting a constant number of clusters of F_i, taking $O(z_i)$ time each, as described in §8.14. These merges and splits in F_i may create additional instabilities in the parent clusters containing u and v in F_{i+1}, but we will fix these up too later in this stage, when we get to F_{i+1}.

N may increase or decrease on this pass. If we ever finish fixing up a graph F_i and discover that every vertex is isolated, we set $N = i$ and delete $F_{i+1}, F_{i+2}, \ldots$. Conversely, if we have fixed up F_N only to discover that some vertices are *not* isolated, we create F_{N+1} and G_{N+1} from F_N using the method of §8.8, increment N, and repeat as necessary.

(c) Go through $G_0, G_1, \ldots G_N$ in order to recompute lightest edges under the new topology. In each G_i, a constant number of vertices u have been marked as having out-of-date adjacency lists, because they were newly created or changed their child sets $C(u)$. We delete the edges in those lists from G_i, and recreate updated $\Gamma_{G_i}(u)$ by scanning and bucketing all the edges incident on $C(u)$ in G_{i-1}. This takes time proportional to the number of such edges, so is $O(z_i \Delta(G_{i-1}))$.[6]

§8.13 REMARK. One can optionally interleave the operations of the stages of §8.12. Instead of running each stage on all the levels $0, 1, \ldots N$ before proceeding to the next, one can run all three stages in sequence on level 0, then on level 1, etc.

Note also that we can handle INCWEIGHT$(uv, \Delta w)$ by running stage §8.12.c alone, applying it to update $\Gamma_{G_i} P^i(u)$ and $\Gamma_{G_i} P^i(v)$. Recall that INCWEIGHT is only called on $uv \notin F$.

Repairing unstable clusters

§8.14 DEVELOPMENT. We now give the details of §8.12.b, which are not fully specified in [6]. Define a vertex $u \in F_i$, $i \geq 1$, to be *unstable* (or to represent an unstable cluster $C(u)$ of F_{i-1}) if any of the following conditions hold.

[6] Actually, the naive implementation takes time $O(z_i \Delta(G_{i-1}) + n_i)$, where the $O(n_i)$ term is for maintaining the n_i buckets. Frederickson can tolerate this, since his choice of z_i values ensures that each $n_i = O(\sqrt{m})$) for each $i \geq 1$. If we wished to deal with some larger, sparser instances of G_i, it would be desirable to eliminate the n_i term. This is possible if mildly tricky:

```
for each edge uv in Γ_{G_i}(u)
    remove uv from Γ_{G_i}(u) and Γ_{G_i}(v)        (* see §8.9 *)
A := a preallocated, reuseable, timestamped array of length n       (* buckets *)
N := an empty list      (* list of buckets that are actually in use *)
for x ∈ C(v)
    for e = xy ∈ Γ_{G_{i-1}}(x)
        v := P(y)      (* so after contraction, the edge is uv in G_i *)
        if A[v] is empty      (* i.e., timestamp is out of date *)
            add v to the list N
            A[v] := e
        elseif e is lighter than A[v]
            A[v] := e
for v in N
    if v ≠ u
        add the edge A[v] to Γ_{G_i}(u) and Γ_{G_i}(v)
```

- $C(u)$ is not connected in F_{i-1}. (In this case it will always have just 2 components.)
- $C(u)$ contains fewer than z_i vertices, and u has degree 1 or 2 (not 0 or 3) in F_i.
- $C(u)$ contains more than $3z_i - 2$ vertices. (In this case it will still always contain at most $4z_i - 3$.)
- u has degree > 3. (In this case it will always have degree 4.)

These are exactly the conditions that are avoided by the method for finding an initial partition (§8.11); that is, the clusters we form initially are stable.

Suppose $u = P^i(u_0) \in F_i$; the following procedure will ensure u's stability, but possibly leave $P^{i+1}(u_0) \in F_{i+1}$ unstable. So to carry out step §8.12.b, we simply call the following procedure with u bound to $P^i(u_0)$ and $P^i(v_0)$ in F_i, for each i in turn. Such a call takes time $O(z_i)$.

```
1.  if C(u) has two components      (* we'll split it into two clusters *)
2.      replace u in V_i with two initially isolated vertices u_1 and u_2
3.      set C(u_1) and C(u_2) to be the vertex sets of the components      (* and set backpointers P *)
4.      set P(u_1) and P(u_2) to the former value of P(u)      (* and modify backpointer C(P(u)) *)
5.      determine Γ_{F_i}(u_1) from Γ_{F_{i-1}}(C(u_1)), and likewise for u_2
6.      (* now P^{i+1}(u_0) is unstable by virtue of having two components *)
7.      (* also, u_1 and u_2 may have too few vertices, so fall through to next case *)
8.  if C(u) contains fewer than z vertices and u has degree 1 or 2 in F_i      (* we'll merge it with an adjacent cluster *)
9.      let w be an arbitrary neighbor of u in F_i      (* exists since u has degree ≥ 1 in F_i *)
10.     merge u with w: C(w) := C(w) ∪ C(u), remove u from P(u) and V_i, etc.
11.     (* merging u into w cannot increase degree of w, since u has degree ≤ 2 in F_i *)
12.     (* now w may have too many vertices, so fall through to next case *)
13. if C(u) contains 3z_i − 1 to 4z_i − 3 vertices or u has degree 4 in F_i      (* we'll repartition it *)
14.     run §8.11 on just C(u) to split it into two clusters C_1 and C_2
15.     replace u in V_i with two initially isolated vertices u_1 and u_2
16.     set C(u_1) := C_1;  C(u_2) := C_2      (* and set backpointers P *)
17.     set P(u_1) and P(u_2) to the former value of P(u)      (* and modify backpointer C(P(u)) *)
18.     determine Γ_{F_i}(u_1) from Γ_{F_{i-1}}(C(u_1)), and likewise for u_2
19.     (* u_1 and u_2 are unconnected, so now P^{i+1}(u_0) may be unstable by virtue of having two components *)
```

The code at lines 9–10 is not quite right, because when $P(u)$ loses u as a child, it may become unstable by virtue of having too few vertices. Then we have made something unstable that is (no longer) an ancestor of u_0, and it will not get properly stabilized on the next pass. We can avoid this problem by choosing w specifically to be a neighbor of u that is also a sibling. Then $P(u)$ loses a child u and may become unstable, but it remains an ancestor of u_0 through w so will get fixed when we stabilize F_{i+1}. What if there is no sibling, i.e., u is the only vertex in its cluster? Then we choose w to be a first cousin. Then $P(u)$ loses its only child u and we delete it; so $P^2(u)$ has lost a child $P(u)$, and may become unstable, but remains an ancestor of u_0 through $P(w)$ and w. What if $P(u)$ has no sibling either, so there is no first cousin? Then we look for a sibling of $P^2(u)$ so that we can get a second cousin, and so forth. The overhead of this scanning will vanish in the runtime analysis.

§8.15 Analysis. Given the partitioning method of §8.11, how much do F_i and G_i shrink as i grows? If every cluster in F_{i-1} really did have from z_i to $3z_i - 2$ vertices, then $|F_i|$ would be at most $|F_{i-1}|/z_i$. This is not quite the case, since we allow clusters as small as one vertex; however, such small clusters are required to use up more than their share of edges (outdegree $= 3$), so there cannot be too many of them.

Recursive structure has $O(\log n)$ levels

We will quantify this argument. Suppose F_i has only one component. Then each vertex of F_i has degree ≥ 1, and acyclicity of F_i means the average vertex has degree < 2. It follows that at most half the vertices can have degree 3. These correspond to clusters in F_{i-1} of at least one vertex each, and the other half have degree < 3 and correspond to clusters of at least z_i vertices each. So the average cluster in F_{i-1} has at least $(z_i + 1)/2$ vertices. $|F_i|$ shrinks from $|F_{i-1}|$ by at least that factor. Even if $z_i = 2$ for all i, each graph is at most $2/3$ as large as the one before, which is enough to make $N = O(\log n)$.

In the more general case, where F_i has multiple components, each with fewer than n vertices, the algorithm will be even faster: the components shrink (in parallel) more quickly because they are smaller, and updates are faster because the degree of a cluster is more tightly bounded. For the sake of simplicity, we will assume below the worst case of a connected graph.

§8.16 ANALYSIS. As sketched in §8.12, the runtime for an update to F_i and G_i $(i \geq 1)$ is dominated by the $O(z_i\Delta(G_{i-1}))$ term in step §8.12.c. Again, this time is proportional to the number of edges emanating from a single cluster of G_{i-1}, which edges must be rebundled. We are also interested in the time to initially create the structure (§8.8): the runtime to create G_i in the first place is n_i times as long as to update it (thus $O(m_{i-1})$), since it must bundle the edges from *all* clusters of G_{i-1}.

$O(m)$ to create, $O(m^{1/2})$ to update

To bound these expressions, we may write:

$$\begin{aligned}
n_0 &\leq m \quad \text{(* since connected; indeed, } n_0 = O(m)\text{, because we forced } G_0 \text{ to have limited degree (§8.11) *)} \\
n_i &\leq n_{i-1}/((z_i + 1)/2) \\
\Delta(G_0) &\leq 3 \quad \text{(* again because } G_0 \text{ was forced to have limited degree *)} \\
\Delta(G_i) &\leq \min(3z_i\Delta(G_{i-1}), n_i) \quad \text{(* recall } 3z_i - 2 \text{ is maximum size of a cluster in } G_{i-1} \text{ *)}
\end{aligned}$$

In the final inequality, the two bounds are respectively "intrinsic" and "extrinsic" (see §4.11 for such bounds on a pass of Borůvka's algorithm). The first bound comes from keeping track of the number of G_0's vertices that could have been shrunk together into a vertex of G_i, and counting the original edges of G_0 incident on these. The second bound comes from noting that there are not too many *other* vertices in G_i for a given vertex to connect to.

Frederickson puts $z_1 = Z =$ a largish number to be chosen below, and $z_2 = z_3 = \cdots = z_N = 2$. Thus the number of vertices in the graph shrinks on the first contraction by a factor of more than $Z/2$, and on subsequent contractions by a factor of at least 1.5. It follows that $n_i < 2m/(1.5)^{i-1}Z$ for $i \geq 1$; this is a closed-form bound for n_i.

Using the extrinsic bound for $\Delta(G_i)$, G_i can be updated in time $O(z_i n_{i-1})$, which is $O(4m/(1.5)^{i-2}Z)$ for $i \geq 2$. As for the case $i = 1$, the intrinsic bound says that updates to G_1 take $O(z_i\Delta(G_0)) = O(Z)$. The total update time summed over all i is therefore $O(Z + m/Z)$. Frederickson chooses $Z = \sqrt{m}$ so that this update time is $O(\sqrt{m})$. As for the time to *create* G_i, it is $O(z_i n_i n_{i-1}) = O(8m^2/(1.5)^{2i-3}Z^2)$ for $i \geq 2$, and $O(6m)$ for $i = 1$. The total creation time is therefore $O(m + (m/Z)^2) = O(m)$.

§8.17 REMARK. The above scheme beats that of §8.5, namely $z_1 = m^{2/3}, z_2 = m^{1/3}$. Why is there not an even better scheme for choosing the z_i? We might hope for faster performance if we chose z_1 to be less and the later z_i to be greater, so that the approach looked more like a true recursive solution. Alternatively, the algorithm could be simplified considerably—but at what cost?—if we took $z_1 = z_2 = \cdots = z_N = 2$. Then the methods for partitioning (§8.11) and stabilization (§8.14) could be specialized for clusters of size ≈ 2.

Frederickson's choice of the z_i appears optimal

It is clear that we cannot improve the $O(\sqrt{m})$ update time without decreasing z_1 below $\sqrt{m}$, since the total update time is at least the $O(z_1\Delta(G_0)) = O(3z_1)$ time to update G_1. But decreasing z_1 does not give a speedup either. To so much as match Frederickson's $O(\sqrt{m})$ per update, we need to require that no cluster in any G_i can have more than $c\sqrt{m}$ incident edges (for some c), since we might have to iterate over all those edges. Let $G_{\hat{\imath}}$ be the first graph in the contracted series to have $\leq c\sqrt{m}$ vertices. (For Frederickson, $\hat{\imath} \approx 1$; by decreasing z_1 we can contract more slowly and raise $\hat{\imath}$.) How many edges are incident on clusters of $G_{\hat{\imath}-1}$?

The extrinsic bound $z_{\hat{\imath}} n_{\hat{\imath}-1} > c\sqrt{m}$ by the definition of $\hat{\imath}$. So our only hope is to rely on the intrinsic bound, which asks how many endpoints of edges of G_0 could have ended up within a cluster of $G_{\hat{\imath}-1}$. Notice that the average cluster of $G_{\hat{\imath}-1}$ must have $2m/n_{\hat{\imath}}$ such edges, which is at least $2\sqrt{m}/c$. The worst-case cluster will be worse than that by a factor of up to about $6^{\hat{\imath}}$: at each contraction, the biggest clusters (size $3z_i - 2$) can be about 6 times as large as the average cluster (size $(z_i + 1)/2$). So unless $\hat{\imath}$ and hence $6^{\hat{\imath}}$ is a constant independent of m, as in Frederickson's method, the intrinsic bound will also allow more than $c\sqrt{m}$ incident edges for some cluster of $G_{\hat{\imath}}$. Provided that we cannot find any better bounds, we therefore run more slowly than Frederickson—$\omega(\sqrt{m})$ time per update—under any discipline where $\hat{\imath}$ increases with m, e.g., $z_1 = z_2 = \cdots = 2$.

§8.18 DEVELOPMENT. To adapt the method for insertion or deletion of edges, remember that G_0 is an "exploded" version of the input graph (§8.11). To add an edge uv of the input graph, we must add it to the exploded version G_0: we may have to split a vertex in V_0. (We also add uv to F_0, provided that u and v are in different components (i.e., $P^N(u) \neq P^N(v)$); otherwise we simply add it to G_0 with infinite weight that we will decrease later.) These operations may render clusters in F_0 unstable; we stabilize them and propagate the changes up through the F_i and G_i as usual. Edge deletion is handled similarly. Each insertion or deletion takes time $O(\sqrt{m})$.

Updating the graph topology

Frederickson notes that as m grows, $Z = \lceil\sqrt{m}\rceil$ might change. The solution is to gradually carry out such adjustments as are due to changes in Z. We keep a work list of vertices of F_1 that have not been stabilized since the last time Z changed. For each edge insertion or deletion, we take some vertices of F_1 off the work list and stabilize any that are unstable, recursively stabilizing their ancestors in F_2, F_3, etc. How many vertices do we need to take off? The work list starts with length $|V_1| = O(m/Z) = O(Z)$. For Z to increase or decrease by 1, m must increase or decrease by $O(Z)$, since $m \approx Z^2$. So each time an edge is inserted or deleted, we only need to take $O(1)$ vertices of F_1 off the work list, and we will have exhausted the work list by the time Z changes again. (In the meantime, some clusters may be slightly too big or too small because they reflect a value of Z that is off by one, but that is not enough to affect the performance analysis.)

Frederickson does not discuss the addition of new vertices, but it follows naturally from the generalization to maintaining MSFs that I have presented here. Isolated vertices can be straightforwardly added or deleted by changing all the F_i and G_i, in time $O(N) = O(\log m)$.

Chapter §9

Lessons Learned

Let us close by reviewing some motifs of the work reviewed here. Throughout the paper, I have tried to motivate each algorithmic technique as a solution to some specific problem. Some useful general algorithm speedup techniques emerge:

- Amdahl's Law: attack the bad term in the complexity. If the algorithm is spending time on DECREASE-KEY, for example, then either make DECREASE-KEY faster (§3.11) or call it less often (§5.10).
- Try to introduce new degrees of freedom in the algorithm and choose these free parameters so as to minimize the complexity. (heap size bound—§5.3; packet size—§5.14; cluster size—§8.5, §8.16)
- Don't waste time computing information you don't need. In particular, don't find a minimum or other partial order by sorting. (§3.13, §4.13, §6.19)
- Postponing work can lead to economies of scale as work piles up. (packets—§5.10; Fibonacci heaps—§A.9; also note the economy of scale from batched queries—§6.18)
- When you may need an expensive piece of information multiple times, store it for later lookup. (precomputed lookup tables—§6.23; presorted packets—§5.11, §5.19; not to mention memoization and dynamic programming!)
- A fast algorithm for a related task may discover useful information along the way. (greedy MST locates path-maximal edges—§6.6, §6.14; verification locates many such in batch mode—§7.3)
- Loose invariants for data structures reduce maintenance time when the structure is modified. (edge tags *or* pointers—§6.29; variable size vertex clusters—§8.11, §8.14, §8.18; variable number of children in Fibonacci trees—§A.10, §A.11, §A.12)
- Divide and conquer: the solution to a subproblem may be used in solving the whole problem.[1] (minimum of minima (not limited to heaps)—§3.10, §5.12, §8.7, §A.17; find subtrees of the MST and contract them to reduce the original problem—§4.2; solution to a random subproblem gives information about the original problem—§7.13, §7.14)

[1]The subproblem might be chosen on the fly rather than identified in advance: e.g., Fredman & Tarjan's algorithm (§3.11) grows subtrees of the true MST. Each of these is the minimum subgraph connecting its own vertices, hence solves a subproblem not known in advance.

Appendix §A

Fibonacci Heaps

§A.1 PROBLEM. A *heap* or priority queue is a data structure used to maintain a collection of *nodes*. Each node appears in at most one heap, and stores some arbitrary data. In addition, each node exports a *key* that determines its order relative to other nodes. Keys may be drawn from any ordered set, such as real numbers. *Operations supported by heaps*

In the presentation of [3], a heap must support the operations shown below. It is not *a priori* obvious that DECREASE-KEY should be regarded as a separate operation, since it appears to be equivalent to deleting the node and inserting a version with lower key. One innovation of Fibonacci heaps is that they show that DECREASE-KEY can be made faster than DELETE; this is crucial for good performance on Prim's algorithm (§3.11).

MAKE-HEAP() : Return a new heap with no nodes.

INSERT(x, h) : Add node x to heap h, modifying h.

MINIMUM(h) : Return a minimal node of heap h.

EXTRACT-MIN(h) : Delete a minimal node from heap h and return it.

UNION(h_1, h_2) : Create and return the union of two heaps, destroying the originals.

DECREASE-KEY(x, k) : Notify node x's heap that node x's key has changed to k.

DELETE(x) : Delete node x from its heap.

[3] gives pseudocode for three kinds of heap—the standard binary kind, the binomial heaps of [17], and the Fibonacci heaps that Fredman & Tarjan [5] introduced for the MST problem. As the table below (adapted from [3]) shows, the asymptotic performance on a heap of n nodes gets steadily better with these innovations, although the constant factors involved get steadily worse.

	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$O(1)$	$O(1)$	$O(1)$
INSERT	$O(\lg n)$	$O(\lg n)$	$O(1)$
MINIMUM	$O(1)$	$O(1)$[a]	$O(1)$
EXTRACT-MIN	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
UNION	$O(n)$	$O(\lg n)$	$O(1)$
DECREASE-KEY	$O(\lg n)$	$O(\lg n)$	$O(1)$
DELETE	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$

[a] Trivially improved from [3]'s $O(\lg n)$. See §A.2.

This appendix offers the reader a more intuitive description of how and why Fibonacci heaps work, and which details of their implementation are critical. We will begin with their predecessors, binomial heaps. (It is assumed that the reader already possesses a good understanding of binary heaps.)

UNION, Locate-Min, and DECREASE-KEY are central

§A.2 SIMPLIFICATION. Some heap operations can be implemented in terms of others. For all three kinds of heaps, it is easy to implement DELETE(x) in $O(\lg n)$ time in terms of the other operations: we DECREASE-KEY node x sharply, so that it becomes the minimum node and bubbles up to the top of the heap, and then call EXTRACT-MIN to remove it. So the time for DELETE is dominated by EXTRACT-MIN.

For binomial and Fibonacci heaps, INSERT(x, h) is implemented by creating a singleton heap holding x—an $O(1)$ procedure we may call *Make-Singleton*—and UNIONing it into h. Thus INSERT takes the same time as UNION.

Finally, it is easy to implement MINIMUM in $O(1)$ time, given a procedure *Locate-Min* that does the same in $O(\lg n)$ time. (The $O(1)$ MINIMUM operation is useful if we repeatedly query the minimum of the same heap without doing any other $O(\lg n)$ operations in between; §5.18 depends on this $O(1)$ behavior for Fibonacci heaps.) All that is necessary is for the heap h to maintain a pointer $m(h)$ to its minimum node, which MINIMUM can then look up. Specifically, MAKE-HEAP initializes $m(h)$ to reference a dummy node of infinite weight; INSERT and DECREASE-KEY replace it whenever they create a smaller node on the heap. $h = \text{UNION}(h_1, h_2)$ essentially sets $m(h)$ to $\min(m(h_1), m(h_2))$. Finally, EXTRACT-MIN would ordinarily call *Locate-Min* and delete the minimal node thus found; we switch the order of these operations so that it first deletes the known minimal node $m(h)$, and then uses *Locate-Min* to set $m(h)$ to a new minimal node that subsequent operations can use.

Since MAKE-HEAP and *Make-Singleton* are always trivial, our main concerns will be UNION, EXTRACT-MIN (which calls *Locate-Min*), and DECREASE-KEY.

A short list of heap trees

§A.3 BINOMIAL HEAPS. Binomial heaps were developed from binary heaps to improve the time of UNION. The problem is easily solved—by fiat!—if the heap consists of a *list* of binary heaps (i.e., heap-ordered trees) rather than just a single such tree. Then UNION is merely a matter of list concatenation. Unfortunately, if the list gets too long we have problems: not with UNION, which can be kept fast with (say) circular lists, but with EXTRACT-MIN, which must search through all the trees to discover which has the smallest minimum. In the extreme case, every tree has just one node and this search (called *Locate-Min* above) examines all n nodes.

We therefore wish to ensure that the list contains only $O(\lg n)$ heap trees. The secret is to control the size of the trees on the list. If $n = 10110_{\text{two}} = 16 + 4 + 2$, then the list will have trees of sizes 2, 4, and 16 (in that order). Essentially a binomial heap of n elements is stored as a linked list of $\leq \lg n$ heap trees. To merge two such lists under UNION (or its special case INSERT), we essentially do binary addition to get a new list in the correct format.

For example, suppose we have two heaps of sizes 22 and 23, necessarily represented as lists of trees of sizes 2, 4, 16 and 1, 2, 4, 16. We merge them (à la mergesort) into a list 1, 4, 8, 32, starting at the units place, in a manner that is parallel to the binary addition problem $22 + 23 = 45$:

$$
\begin{array}{rl} 10110 & {\scriptstyle 2,4,16} \\ 10111 & {\scriptstyle 1,2,4,16} \\ \hline & \end{array}
\longrightarrow
\begin{array}{rl} 1011\cancel{0} & {\scriptstyle 2,4,16} \\ 1011\cancel{1} & {\scriptstyle 2,4,16} \\ \hline 1 & {\scriptstyle 1} \end{array}
\longrightarrow
\begin{array}{rl} 1 & {\scriptstyle 4} \\ 101\cancel{1}\cancel{0} & {\scriptstyle 4,16} \\ 101\cancel{1}\cancel{1} & {\scriptstyle 4,16} \\ \hline 01 & {\scriptstyle 1} \end{array}
\longrightarrow
\begin{array}{rl} 1\cancel{1} & {\scriptstyle 8} \\ 10\cancel{1}\cancel{1}\cancel{0} & {\scriptstyle 16} \\ 10\cancel{1}\cancel{1}\cancel{1} & {\scriptstyle 16} \\ \hline 101 & {\scriptstyle 1,4} \end{array}
\longrightarrow
\begin{array}{rl} \cancel{1}\cancel{1} & \\ 1\cancel{0}\cancel{1}\cancel{1}\cancel{0} & {\scriptstyle 16} \\ 1\cancel{0}\cancel{1}\cancel{1}\cancel{1} & {\scriptstyle 16} \\ \hline 1101 & {\scriptstyle 1,4,8} \end{array}
$$

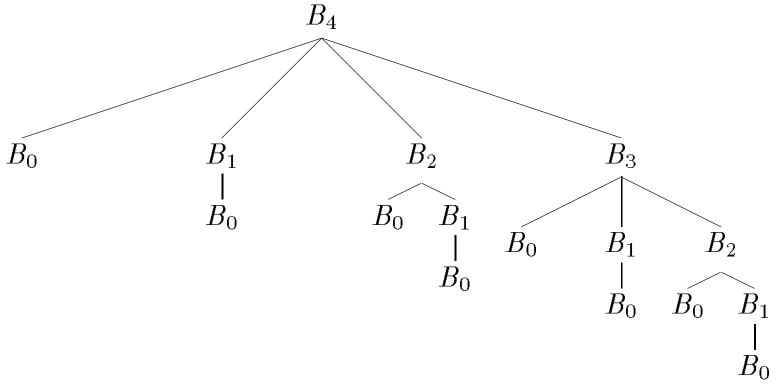
$$
\longrightarrow
\begin{array}{rl} 1\ \cancel{1}\cancel{1} & {\scriptstyle 32} \\ \cancel{1}\cancel{0}\cancel{1}\cancel{1}\cancel{0} & \\ \cancel{1}\cancel{0}\cancel{1}\cancel{1}\cancel{1} & \\ \hline 01101 & {\scriptstyle 1,4,8} \end{array}
\longrightarrow
\begin{array}{rl} \cancel{1}\ \cancel{1}\cancel{1} & \\ \cancel{1}\cancel{0}\cancel{1}\cancel{1}\cancel{0} & \\ \cancel{1}\cancel{0}\cancel{1}\cancel{1}\cancel{1} & \\ \hline 101101 & {\scriptstyle 1,4,8,32} \end{array}
$$

To handle the case of a carry, we must be able to link two trees of size 8 (say) into one of size 16, in constant time. Moreover, if we later EXTRACT-MIN the root from the tree of size 16, we need to organize the remaining 15 elements into trees of sizes 1, 2, 4, and 8 that can be UNIONed in with the rest of the heap.[1]

To make these operations fast, we use binomial trees instead of binary trees. A binomial tree of $16 = 2^4$ nodes, denoted B_4, has the topology of a B_3 with an extra B_3 attached to the root. Given two B_3 trees, it takes only constant time to link them together into a B_4 while preserving the heap property, by attaching the root of larger key to the root of smaller key. Furthermore, the children of the root of B_4 are B_0, B_1, B_2, and B_3, so we can simply UNION these back into the heap if we happen to delete the root.

B_4 = B_3 with another B_3 attached as its rightmost child

=



Now EXTRACT-MIN can be implemented with two $O(\lg n)$ calls: to find the minimum (*Locate-Min*) we scan the roots of all the $\leq \lg n$ trees, and to extract a known minimum at a root, we delete the root and UNION its $\leq \lg n$ children back into the heap. DECREASE-KEY(x, h) may be handled within x's tree in the usual manner of (not necessarily binary) heap trees, by bubbling x up the tree as necessary. This too takes time $O(\lg n)$, since B_k contains nodes at depth $k - 1$.

§A.4 IMPLEMENTATION DETAILS. We implement the binomial heap h by connecting all the tree roots into a singly-linked list, called the *root list*, ordered so that the size of the trees increases (strictly) as one moves down the list. Such ordered lists enable UNION to use the "binary-addition" procedure.

It is convenient to represent each node's children in the same way, by linking them into a *child list* of identical format, using the same pointer fields. In this way, when a root node of h is deleted (via EXTRACT-MIN), its child list may be regarded without any ado as the root list of a new heap h', which we fold back into h simply by calling UNION(h', h).

It is also convenient for each child list to maintain a tail pointer as well as a head pointer. Thus, when two instances of B_k are linked together, one can be appended in $O(1)$ time to the child list of the other's root.[2]

Each node in the heap records its degree. In particular, the binary addition procedure can immediately identify a tree as B_i (corresponding to a 1 in bit i of the binary representation of

[1] One might instead try to replace the deleted root with an element of a smaller tree, and then restore the heap property on the tree, but this is not possible if there is no smaller tree on the list.

[2] Neither of the above convenient choices is actually essential. Imagine that child lists were stored in a different format from root lists, and that they allowed only prepending (not appending) of new children. Then we would have to maintain a node's children in *decreasing* rank order. When EXTRACT-MIN needed to call UNION(h', h), the heap h' would have to be constructed by reversing and reformatting a child list. Since no sorting is required, this would need only $O(\lg n)$ time, which EXTRACT-MIN could afford.

n), by noting that its root has degree i. We will refer to the degree of a tree's root as the *rank* of the tree.

Binomial trees balance depth and breadth

§A.5 REMARKS. The largest tree on the binomial heap is necessarily $B_{\lg n-1}$, which has $2^{\lg n-1} = 2^{\lceil\log(1+n)\rceil-1} = 2^{\lfloor\log n\rfloor}$ nodes. We may regard the root list as the child list of an "overall" root node that holds the minimum of the whole heap. This root node is essentially $m(h)$ of §A.2, and is a copy of its smallest child. In this way we regard the heap h as a single heap-ordered tree $T(h)$. If $n = 2^k - 1$, so that the trees on the heap are $B_0, B_1, \ldots B_{k-1}$, then $T(h)$ is simply B_k.[3]

The above picture makes it clear how binomial heaps balance the number of trees (small for *Locate-Min*'s sake) against the depth of the trees (small for DECREASE-KEY's sake). The heap is in some sense just a single balanced tree $T(h)$, maintaining its minimum in just the same way as an ordinary binary heap. The only difference is that a node may have more than two children. In particular, $T(h)$ is roughly a binomial tree, which is *defined* to balance depth and breadth: a property of binomial trees is that the root of a height-d subtree has d children. ($T(h)$ relaxes this property very slightly in that the root may have fewer children.)

To better understand how binomial heaps differ from binary heaps (on operations other than UNION), let us imagine a generalized version of the binary heap, where nodes of the heap tree may have more than two children. ($T(h)$ is such a tree with added restrictions, which we ignore here, on its form and root node.) DECREASE-KEY(x, k) is just as for a binary heap: it bubbles node x up the heap in the usual way, placing it into its ordered list of ancestors à la insertion sort. This takes time proportional to the depth of the tree. A binary-heap-like implementation of EXTRACT-MIN would replace the root with an arbitrary node x and then restore the heap property by forcing x down the tree, repeatedly exchanging it with its smallest child. But this requires a search for the smallest child at each level. Such a search may take more than constant time if nodes have high degree. For example, if the tree is shaped like B_k, this version of EXTRACT-MIN can take up to $\Theta(k^2) = \Theta((\lg n)^2)$ steps.

Fortunately EXTRACT-MIN allows a cheaper $O(\lg n)$ method that turns the possibility of high degree into a benefit. The smallest child x of the deleted root becomes the new root. x keeps all its existing children, and it also acquires the other children of the deleted root as new children. In the event that this has burdened x with too many children, we can now distribute the burden by reattaching some of the children to each other rather than to x.

For binomial heaps, the binary-addition discipline arranges for exactly such reattachments. It effectively connects the children of x into a set of trees that respect the heap property. The tallest possible such trees are chains of the form "B_i reattaches to B_i which reattaches to B_{i+1} which reattaches to B_{i+2} which reattaches to B_{i+3} which stays attached to x." Notice that the trees that hang lowest in such a chain (viz. B_i) are also the shortest trees, so that the heap remains balanced. It is instructive to contrast what would happen if we applied this EXTRACT-MIN method in the case of binary trees, which are more balanced than binomial trees. Here the new root x would have two old children plus a new one, making three, an intolerable number.

[3] Indeed, if we were lucky enough to have n in the form 2^k, we could simply store the binomial heap as a single B_k and implement MINIMUM in $O(1)$ time by treating the root node as $m(h)$. For other values of n, we could still use a single binomial tree, padding it with up to $n-1$ extra nodes of infinite weight. However, the details of tracking the padding nodes make this construction much clumsier than that of §A.3.

How, for example, can we rapidly handle the transition from $n = 2^k$ (stored as B_k with no padding) to $n = 2^k + 1$ (stored as B_{k+1} with $2^k - 1$ padding nodes, a much larger tree)? We could borrow Frederickson's "work list" technique (§8.18): maintain a separate tree consisting entirely of padding nodes, such that if the main heap B_k has $2^k - i$ occupied nodes, this separate tree has $2^k - 2i$ nodes overall and is a "partially grown" B_k. If B_k becomes fully occupied, the separate tree then has the size and shape of a second B_k; we combine it with the main heap to create B_{k+1}, and start a new separate tree with 0 nodes. To go backwards if this B_{k+1} becomes only half-occupied, it is necessary to split off an entire B_k-shaped tree of padding nodes. To make this work, we must arrange for padding nodes to be segregated in their own child trees of the root. The child trees consisting of real nodes must therefore be maintained just like a standard binomial heap, only more clumsily.

One child would therefore be reattached to another, but this would just shift the problem one level down the tree, so that eventually one of the three child subtrees of x would reattach to a leaf. The resulting tree would be quite unbalanced!

Deferred consolidation of binomial trees

§A.6 Precursor to Fibonacci heaps. It is possible to improve the time of Union (hence also Insert) even further, to $O(1)$. We do so by abandoning the strict discipline of binomial heaps, where the trees on the list have distinct sizes prescribed by n, and fall in a particular order. Instead, we represent the heap as an arbitrary, unordered list of binomial trees that may contain duplicates. Now Union is just list concatenation. This is $O(1)$ if we use circular lists or lists with tail pointers, just as suggested at the start of §A.3.

To deal with the problem that long lists handicap Extract-Min, we perform a *consolidation* step at the start of each Extract-Min, which turns the long list back into a proper binomial heap. This will take more than $\Theta(\lg n)$ time, of course. The time is proportional to the length of the list—but in an amortized analysis, the extra time can be charged to the operations that lengthened the list beyond $\lg n$ trees in the first place, such as Insert. So Extract-Min can still find the minimum in only $O(\lg n)$ amortized time.

The consolidation step is like adding a long list of binary numbers to a running total, where each number on the list is a power of 2. A different perspective is that it resembles bucket sort, just as the "binary addition" step of §A.3 resembles one pass of mergesort. We set up $\lg n$ buckets to hold trees of ranks $0, 1, 2, 3, \ldots \lg n{-}1$, i.e., trees of the form $B_0, B_1, B_2, B_3, \ldots B_{\lg n-1}$. Each bucket can hold just one tree.

We now consider the k trees on the heap in their arbitrary order, attempting to throw each tree into the appropriate bucket. A B_i tree wants to go into bucket i. If bucket i already holds another B_i, however, it cannot hold the new tree as well. In this case we need a "carry" operation: we withdraw the second B_i from the i bucket and link it together with the first B_i to form a B_{i+1}. We then attempt to put this new tree into bucket $i+1$ in exactly the same way (iterating as necessary if that bucket is also full). Once we have disposed of the tree we were holding, we proceed to the next tree on the heap until we have disposed of all k. To complete the consolidation, we make a list of the final $b \leq \lg n$ bucketed trees to obtain a proper binomial heap.

§A.7 Analysis. How long did this consolidation take? Initialization and termination were $O(\lg n)$. There were $k - b$ linkages, each of which reduced the number of trees by one; so bucketing the k trees took time $O(k)$ plus $O(k - b)$ for the linkages. We charge $O(b)$ of this time to the consolidation step, for a total amortized time of $O(\lg n)$ including initialization and termination. The other $O(k - b)$ is charged against the *potential* of the heap, which is equal to the number of trees on it. The heap acquires potential from operations that lengthen it: thus each Insert contributes an additional $O(1)$, and each Extract-Min contributes an additional $O(\lg n)$ for the several trees that it adds to the heap. These contributions do not damage the analysis of Insert or Extract-Min. Notice that Union adds no potential; the output heap inherits its potential from the work previously done on the two input heaps.

§A.8 Implementation details. To implement the modified binomial heap of §A.6, exactly the same data structures will do as §A.4 used for standard binomial heaps. However, it has now become crucial for the root list to maintain a tail pointer: this is what lets Union concatenate two root lists in $O(1)$ time. (Recall that in the case of standard binomial heaps, the only reason for the root list to maintain a tail pointer was that it was convenient, though inessential, for root lists to look like child lists and for child lists to maintain tail pointers.)

Postponing work creates economies of scale

§A.9 Discussion. The heaps of §A.6 appear to improve on the standard binomial heaps of §A.3: they reduce Union and Insert to constant-time. But how much of this apparent improvement is merely the result of amortized analysis?

In fact we have not really improved INSERT: standard binomial heaps, too, require only $O(1)$ *amortized* time for INSERT. (Compute potential as above, and observe that the time spent on a single-node insertion is proportional to the number of carries, i.e., the number of prepaid tree linkages.[4])

On the other hand, we have genuinely improved UNION, not just found a more favorable analysis. UNION of standard binomial heaps can take time $\Theta(\lg n)$ simply to merge the tree lists. This time cannot in general be charged to previous operations. To see the problem, suppose we start with a binomial heap of the form 111111_{two}, and union it with arbitrarily many heaps of the form 100000_{two}. Every UNION wastes time skipping anew over B_0, B_1, B_2, B_3, B_4, which are never eliminated but always survive till the next UNION.[5] The time spent on these trees is genuinely wasted in that it consists only of iterating over them and looking at their ranks; no keys are compared and no information is gained.

The deferred-consolidation approach of §A.6 actually does less *work* on this example: the 111111 heap and many 100000 heaps are concatenated in $O(1)$ time per UNION, and if a consolidation step is necessary at the end of all these concatenations (e.g., in order to find a minimum), it considers each tree only once. In particular, each of B_0, B_1, B_2, B_3, B_4 is considered only once overall (during the consolidation step), rather than once for each UNION.

Thus, the deferred approach of §A.6 takes advantage of an economy of scale: it is more efficient to postpone work until there is a lot of it. Precisely the same observation was used in §5.10 to motivate the use of packets in Gabow *et al.* In that algorithm, we waited to bucket heavy edges until the number of buckets *decreased*; in §A.6, we wait to bucket binomial trees until the number of trees *increases*. Both postponements serve to increase the number of objects per bucket or tree.

The analogy is worth making in more detail. The common idea is that all but one object per bucket can be eliminated permanently in $O(1)$ time; only the single survivor of a bucket stays around to create more work later. Many objects per bucket is a good strategy, because it means a high ratio of cheap eliminated objects to expensive surviving objects.

Thus in §5.10, only the *lightest* edge in each bucket survived to be considered on the next pass; the others were eliminated from the MST immediately. We may think of such eliminations as having been charged to an $O(m)$ budget that paid for each non-MST edge to be eliminated once. Thus, the surplus cost of Gabow *et al.*, in excess of this budget, was determined by the number of surviving edges over all passes. In precisely the same way, in §A.6, only the *minimum* binomial-tree root in each bucket survives as a root, to be examined again by *Locate-Min* or future consolidation operations. The other roots are "eliminated" in that they become internal nodes of the surviving trees, thanks to linkages. These eliminations are charged against the potential of the tree—a kind of escrow fund whereby each inserted tree pays for itself to be linked later. Thus, again, the amortized cost of each consolidation is determined by the number of survivors, which is at most $\lg n$.

To take a different perspective, a bucket with s objects results in $s-1$ comparisons. If s is small, a substantial fraction of the bucketing work does not result in any comparisons, so it takes more work to gain a given amount of information about the MST or the heap minimum. In particular we want s to average above $1+\epsilon$ for some $\epsilon > 0$, so that the overhead per comparison is constant.

§A.10 FIBONACCI HEAPS. Recall that we obtained the heaps of §A.6 by modifying binomial heaps to have constant-time UNION. We now obtain Fibonacci heaps by making a further

DECREASE-KEY *by detaching a subtree*

[4] More generally, the worst-case *actual* cost of UNION(h_1, h_2) is $O(\lg \max(|h_1|, |h_2|))$, whereas the worst-case *amortized* cost is $O(\lg \min(|h_1|, |h_2|))$.

[5] Obviously, if we tried to pay for this work by drawing on heap potential accumulated from previous operations, the potential would run out after some finite number of such UNIONs.

modification, designed to give us constant-time DECREASE-KEY(x, k) as well.

Again we can begin with a naive idea. Rather than bubbling the node x of decreased key up to the top of its tree T, suppose we simply cut it off from its parent in T. In this way x becomes the root of a new tree on the heap, where it can be examined by LOCATE-MIN in the usual way. This *detachment* takes $O(1)$ actual time (see §A.15). Since it increases the heap by one tree, it is also charged an additional $O(1)$ to increase the heap potential; this will be used later to reattach the detached node.

However, T is left stunted by this amputation. Removing arbitrary subtrees means that the heap no longer consists of binomial trees. We had relied on the fact that trees were binomial to tell us that they all had rank $< \lg n$—trees of higher rank would have more than n nodes—and therefore consolidation required only $\lg n$ buckets and $O(\lg n)$ amortized time. Now that the binomial property is gone, a succession of key-decreases and consolidations might leave us with small trees of many more different ranks. Consolidation could therefore require far more buckets and be far slower.

All we really need to ensure is one important property of binomial trees: that a tree with high rank has many descendants. This ensures that rank stays small relative to n, and there cannot be many distinct ranks. In the manner of 2-3-4 trees, we will enforce the desired property while being somewhat more flexible about how many children each node actually has. Instead of using binomial trees we will use *Fibonacci trees*: these have the desired property that their ranks will be $O(\lg n)$, but they have somewhat weaker invariants that we can maintain with less computation.

§A.11 DEFINITION. Fredman & Tarjan write that they "impose no explicit constraints on the number or structure of the trees; the only constraints are implicit in the way the trees are manipulated" [5, p. 598]. For the sake of understanding, however, it is valuable to state explicit constraints from the start.

Fibonacci trees balance depth and breadth, too

Recall that a binomial tree of rank k, denoted B_k, is defined as a root with k ordered child trees, which are binomial trees of respective ranks $0, 1, 2, 3, \ldots k-1$. Let us define a Fibonacci tree of rank k, denoted F_k, to likewise consist of a root with k ordered child trees, which are Fibonacci trees of respective ranks *at least* $0, 0, 1, 2, \ldots k-2$.[6] Thus, the ith child of a root has degree $= i-1$ in B_k and degree $\geq i-2$ in F_k. (In F_k the rank may be as large as i, so deep narrow trees are possible, including (sorted) vertical chains. In fact the rank may be as large as k, or even larger.)

How many nodes must a tree of form F_k have? We can show a tight lower bound of f_k nodes, where $f_0 = 1$, $f_1 = 2$, and $f_{k+1} = f_k + f_{k-1}$ for $k \geq 1$. By a simple induction,

$$f_k = 1 + f_0 + f_0 + f_1 + f_2 + \cdots + f_{k-2} \text{ for } k \geq 2$$

(For the inductive step, merely add f_{k-1} to both sides of the equation.) Now observe that $|F_0| = 1 = f_0$, $|F_1| \geq 2 = f_1$, and by induction for $k \geq 2$

$$|F_k| \geq 1 + |F_0| + |F_0| + |F_1| + |F_2| + \cdots + |F_{k-2}| \geq 1 + f_0 + f_0 + f_1 + f_2 + \cdots + f_{k-2} = f_k$$

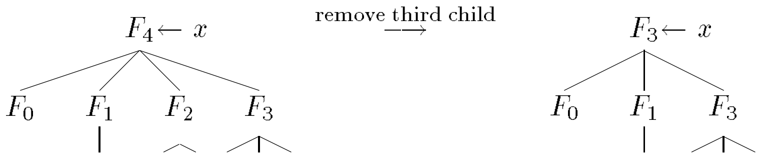
as desired. Since f denotes the Fibonacci series, we see that $|F_k| = \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^k\right)$. While this grows a little more slowly than $|B_k| = 2^k$, it is still exponential, and thus quite adequate to ensure that all Fibonacci trees of n or fewer nodes have bounded rank $O(\lg n)$.

[6] There actually is no need in the implementation for the children to be stored in this order, as long as some child has rank at least $k-2$, another has rank at least $k-1$, etc. In fact the child list of a Fibonacci tree is conventionally presented as unordered [5, 3], in contrast to the (standard) binomial trees of §A.3, which rely on keeping lists sorted because they merge lists rather than implictly bucket-sorting their elements. However, ordering the children just as for binomial trees costs nothing and is quite convenient for exposition.

It is convenient to say that a node x satisfies the *Fibonacci property* if its ith child has degree $\geq i-2$. Thus, a Fibonacci tree is precisely a tree all of whose nodes satisfy the Fibonacci property.

The Fibonacci property has some "give"

§A.12 Observation. Fibonacci trees can sustain a certain amount of "damage" and still qualify as Fibonacci trees. Removing a child of node x will reduce the degree of x, but does not affect whether x satisfies the Fibonacci property. (This is not hard to see from the definitions of §A.11.) Thus if x formerly rooted an F_4 it now roots an F_3:



However, x may lose the Fibonacci property if any of its children diminish too severely in degree.

§A.13 Development. How do we maintain the invariant that all trees of the heap are Fibonacci trees? We must consider how nodes acquire and lose children. Nodes acquire children only during consolidation steps: thanks to bucket $i-1$, the root of an F_{i-1} may acquire another F_{i-1} as its new, ith child tree. Thus when a node x is attached to a parent p as the ith child, it invariably has degree $i-1$. The Fibonacci property at p requires x to have degree $\geq i-2$.

So x is allowed to lose one child "for free," since its degree may drop to $i-2$ without damaging p's Fibonacci property. If x loses a second child, however, we detach it from p and put it onto the heap's main list of trees, thereby protecting p's Fibonacci property. Now p has lost a child, so may possibly have to be detached from *its* parent (if any), and so on. A single Decrease-Key operation that detaches a child of x may therefore cause a cascade of detachments.

§A.14 Remark. If x started out as the ith child of p and has lost two children, it certainly has degree $i-3$ now. Notice that this has not *necessarily* destroyed the Fibonacci property at p, since x may no longer be the ith child. (Lower-numbered children of p may have been deleted.) For example, if x is only the 1st or 2nd child now, the Fibonacci property at p now only requires x to have degree ≥ 0. But to be safe, the discipline above goes ahead and detaches x anyway as soon as it has lost two children.

Deathbed bits

§A.15 Implementation. The data structures used differ only slightly from those in §A.8. Each child list must now be *doubly* linked, so that an arbitrary node on the list can be detached from its parent in $O(1)$ time. Moreover, each node now records not only its degree, but also a "deathbed bit" indicating whether its degree has decreased since it was attached to its current parent. Deathbed nodes need to be detached if their degrees decrease again.

An amortized analysis

§A.16 Analysis. It is crucial that a node is left alone the first time it loses a child. Otherwise, each Decrease-Key would cause a cascade of detachments all the way up to the root of its tree: by detaching a child from its parent, it would cause the parent to be detached from its grandparent and so on up. This would actually force trees to stay "at least binomial"—the ith child of a node would have degree $\geq i-1$—and the cost of Decrease-Key would be up to $\Theta(\lg n)$, just as for binomial trees.

Under the more forgiving Fibonacci discipline, detachments only cascade through nodes that are already on their deathbeds. They do not cascade past a node that is only now losing its first child. In the worst case, Decrease-Key still cascades up to the top of the tree and takes $\Theta(\lg n)$ time. However, this worst case cannot occur often. For DECREASE-KEY to take a long time deleting nodes, there must have been many previous DECREASE-KEY operations that put nodes on their deathbeds.

To turn this observation into an amortized analysis, we simply require prepaid funerals. Any operation that marks a node p as being on its deathbed must prepay for that node to be detached later. DECREASE-KEY is the only operation that marks nodes in this way. Each call DECREASE-KEY(x,k) puts at most one node z on its deathbed, where z is some proper ancestor of x. Hence the total amortized cost of DECREASE-KEY(x,k) consists of an $O(1)$ prepayment for z's subsequent detachment, together with $O(1)$ to detach x itself, which was not necessarily on its deathbed.[7] All other work performed by DECREASE-KEY is a prepaid cascade of detachments of deathbed nodes on the path between x and z.

§A.17 REMARK. The introductory remarks in §A.1 observed that it was surprising that DECREASE-KEY could be made faster than DELETE. Why is this possible and how did we accomplish it?

Why only DELETE *is intrinsically slow*

A heap is essentially a device for maintaining some function f on a changing set S of n elements. It works by maintaining the value of f on various subsets of S, called *local sets*, which include the singletons of S. Some local sets are subsets of others; the inclusion lattice on local sets, also called the *local topology*, is typically a tree. The function f must have the property that $f(S_1 \cup S_2)$ can be determined quickly from $f(S_1)$ and $f(S_2)$; this is called an *accumulation*. (For binary, binomial, and Fibonacci heaps, the function is "minimum," and the elements of S are arranged in a forest whose subtrees are used as the local sets of the heap. Frederickson's algorithm (§8.7) provides another example.)

The asymmetry between insertion and deletion hinges on the fact that it is $f(S_1 \cup S_2)$ that can be determined quickly, not $f(S_1 \cap S_2)$.

If we delete an element x from S, then we must recompute $f(S - \{x\})$, either from scratch or from our saved values of f on the local sets of S that are subsets not just of S but of $S - \{x\}$. In general the worst case must take at least $\log n$ accumulations, no matter how we define the local subsets. Otherwise we would be able to do a comparison sort in better than $n \log n$ comparisons. DELETE can be no faster than this if it determines the new value $f(S - \{x\})$ (e.g., the new minimum).

By contrast, if we *add* an element x to S, then we can determine $f(S \cup \{x\})$ with a single accumulation. Similarly, if we combine two heaps, which represent S and S', a single accumulation gives us $f(S \cup S')$. We therefore expect that INSERT and UNION can in principle take constant time. In the case where f is the "minimum" function, DECREASE-KEY(x,k) can also in principle take constant time, since its effect on the minimum is just as if we had inserted a new element of key k.

§A.18 REMARK. The difficult part about achieving INSERT, UNION, and DECREASE-KEY in constant time is that they must ensure that the data structure still maintains f on the local sets of §A.17. Otherwise subsequent calls to DELETE or *Locate-Min*—the difficult operations—might take *more* than logarithmic time.

The strategic difference from binary heaps

Binary heaps preserve the size and local topology of the local sets, to the extent that this is possible as S grows and shrinks; but they spend up to $\Theta(\lg n)$ time per operation to swap elements among local sets, changing the minima of the local sets affected by this.

Binomial and Fibonacci heaps rely instead on changing the local topology. Thus they move entire local sets around rather than individual elements, which allows them to change fewer local sets per operation. They rely specifically on three cheap operations: (a) expanding one local set with all the elements of another, (b) removing a local set from the topology, and

[7] Each of these two $O(1)$ detachments, as mentioned in §A.10, consists of the actual work to detach the node plus a contribution to heap potential that will pay for the node to be reattached later.

(c) shrinking a local set so that it no longer includes any of the elements of one of its local subsets. This last operation is cheap only if it is guaranteed not to affect the value of f on the shrunken set. The operations correspond to (a) attaching one tree root to another as a child, (b) detaching all the children from a root, and (c) detaching a subtree from a tree.

Bibliography

[1] Manuel Blum, Robert W. Floyd, Vaughan Pratt, Ronald L. Rivest, and Robert E. Tarjan. 1973. Time bounds for selection. *Journal of Computer and System Sciences,* 7(4):448–461.

[2] O. Borůvka. 1926. O jistém problému minimálním. *Práca Moravské Přirodovědecké Společnosi 3*, 37–58, (In Czech.)

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to Algorithms.* MIT Press, Cambridge, MA.

[4] B. Dixon, M. Rauch, and R. Tarjan. 1992. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal of Computing*, 21(6):1184–1192.

[5] Michael L. Fredman and Robert E. Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615

[6] Greg Frederickson. 1985. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal of Computing,* 14(4):781–798.

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

[8] D. R. Karger, P. N. Klein, and R. E. Tarjan. 1995. A randomized linear-time algorithm for finding minimum spanning trees. *Journal of the ACM,* 42(2):321–328.

[9] Valerie King. 1993. A simpler minimum spanning tree verification algorithm. Unpublished manuscript,
`ftp://godot.uvic.ca/pub/Publications/King/Algorithmica-MSTverif.ps`. (To appear in *Proceedings of the Workshop on Algorithms and Data Structures.*)

[10] J. Komlós. 1985. Linear verification for spanning trees. *Combinatorica,* 5:57–65.

[11] J. B. Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society,* 7:48–50.

[12] Rajeev Motwani and Prabhakar Raghavan. 1995. *Randomized Algorithms.* Cambridge University Press.

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

[14] D. D. Sleator and R. E. Tarjan. 1983. A data structure for dynamic trees. *Journal of Computing System Science,* 26:362–391.

[15] B. Schieber and U. Vishkin. 1988. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal of Computing*, 17:1253–1262.

[16] Robert E. Tarjan. 1975. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225.

[17] J. Vuillemin. 1978. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309–315.