The Dissertation Committee for Seth Pettie
certifies that this is the approved version of the following dissertation:

On the Shortest Path and Minimum Spanning Tree Problems

Committee:

Vijaya Ramachandran, Supervisor

Harold Gabow

Anna Gál

Tandy Warnow

David Zuckerman

On the Shortest Path and Minimum Spanning Tree Problems

by

Seth Pettie, B.A.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2003

for my mother and my muse

Acknowledgments

This thesis would not have been possible without the support and guidance of my advisor, Vijaya Ramachandran. She encouraged all of my research endeavors and has been my staunchest advocate.

I must also thank my committee members, Hal Gabow, Anna Gál, Tandy Warnow, and David Zuckerman, as well as the theory group at UT–Austin. Their comments greatly helped to improve the presentation of this thesis. I would also like to thank Jay Misra. We have had too few conversations, but for those I am grateful. (One cannot be in the presence of Jay without learning — as if by osmosis — that clear presentation and clear thinking are inseparable — and that both are in short supply.)

Grad school could not have been as enjoyable without my fellow grad students. I would like to thank, in particular, Lisa Kaczmarczyk for taking the coffee break to the next level, and my office-mate Eunjin Jung, a good friend and conversationalist and a great chef.

I have failed, on many occasions, to properly explain to my family what algorithms are and why they're worth studying. Nonetheless, they supported my pursuits simply because they were mine. I would especially like to thank my mother and father, Andy, grandma Betty, and uncle Tim, who treated me like a grad student since I was 5 years old. I cannot thank my mother enough. This thesis, my grad school career, and everything else would not have been possible without her active support.

Lastly, I must thank my muse, Elizabeth Harr. Her joie de vivre has made the last $2\frac{1}{2}$ years the best of my life. Without her life would be nothing but an empty void.

SETH PETTIE

The University of Texas at Austin
August 2003

On the Shortest Path and Minimum Spanning Tree Problems

Publication No. ____________________

Seth Pettie, Ph.D.
The University of Texas at Austin, 2003

Supervisor: Vijaya Ramachandran

The shortest path and minimum spanning tree problems are two of the classic textbook problems in combinatorial optimization. They are simple to describe and admit simple polynomial-time algorithms. However, despite years of concerted research effort, the asymptotic complexity of these problems remains unresolved.

The main contributions of this dissertation are a number of asymptotically faster algorithms for the minimum spanning tree and shortest path problems. Of equal interest, we provide some clues as to *why* these problems are so difficult. In particular, we show why certain modern approaches to the problems are doomed to have super-linear complexity.

A sampling of our results are listed below. We emphasize that all of our algorithms work with *general* graphs, and make no restrictive assumptions on the numerical representation of edge-lengths.

- A provably optimal deterministic minimum spanning tree algorithm. (We give a constructive proof that the algorithmic complexity of the minimum spanning tree problem is equivalent to its decision-tree complexity.)
- An all-pairs shortest path algorithm for general graphs running in time $O(mn + n^2 \log\log n)$, where m and n are the number of edges and vertices. This provides the first improvement over approaches based on Dijkstra's algorithm.
- An all-pairs shortest path algorithm for *undirected* graphs running in $O(mn \log \alpha)$ time, where $\alpha = \alpha(m, n)$ is the inverse-Ackermann function.

- A single-source shortest path algorithm running in $O(m\alpha + \min\{n \log\log r,\ n \log n\})$ time, where r bounds the ratio of any two edge lengths. For r polynomial in n this is $O(m + n\log\log n)$, an improvement over Dijkstra's algorithm.

- An inverse-Ackermann style lower bound for the *online* minimum spanning tree verification problem. This is the first inverse-Ackermann type lower bound for a comparison-based problem.

- An $\Omega(m + n\log n)$ lower bound on any *hierarchy-type* single-source shortest path algorithm, implying that this type of algorithm cannot improve upon Dijkstra's algorithm. (All of our shortest path algorithms are of the hierarchy type.)

- The first *parallel* minimum spanning tree algorithm that is optimal w.r.t. to both time and work. Our algorithm is for the EREW PRAM model.

- A parallel, expected linear-work minimum spanning tree algorithm using only a polylogarithmic number of random bits.

- An $O(mn\log\alpha)$ bound on the *comparison-addition* complexity of all-pairs shortest paths. This is within a tiny $\log\alpha$ factor of optimal when $m = O(n)$.

Contents

Chapter 1

Introduction

As optimization problems go, the *minimum spanning tree* and *shortest path* problems are as old as the hills. They are so firmly established in the canon of computer science education that today no student can avoid learning the algorithms of Dijkstra, Prim, Bellman-Ford, Floyd-Warshall, Kruskal, and Borůvka. Given the rich history of both problems (the minimum spanning tree problem dates back 75 years) and the vigor of recent research efforts, it is thoroughly surprising that neither problem is solved. In particular, the question of their *inherent* algorithmic complexity has yet to be fully answered.

The primary focus of this dissertation is obtaining asymptotically faster algorithms for three classical graph optimization problems: single-source shortest paths, all-pairs shortest paths, and minimum spanning trees. For each problem we offer algorithms that achieve optimality, or make substantial strides toward optimality. Highlights of our results include a provably optimal minimum spanning tree algorithm (with unknown running time) and an all-pairs shortest paths algorithm that improves on Dijkstra's textbook algorithm from 1959. We survey our results in more detail in Section 1.1.1 (shortest paths) and Section 1.1.2 (minimum spanning trees). Before delving into details, we would like to highlight our assumptions concerning the model of computation.

Any discussion of an algorithmic result must begin with the answers to two fundamental questions: What does the input to the algorithm *look like*? and what can our (imaginary) computer *do* (and at what cost)? The answers to these questions define the computational model, or simply *model*. Most researchers choose a model by considering aesthetic simplicity, historical precedent, realism, convenience, or some combination thereof. In this dissertation we study the shortest path and minimum spanning tree problems under the *traditional* textbook model. The input is assumed to be given as a *real*-weighted general graph, either directed or undirected, and the defining characteristic of the machine model is that real numbers are only subject to a specific set of unit-time operations, e.g., addition, subtraction, and comparison. (See Sections 2.4 and

7.2.1 for the specifics.)

The strength of the traditional model is its *weakness*. It is weak in that it makes minimal assumptions about the form of the input, and minimal assumptions about how the abstract computer can manipulate the input. As a consequence, algorithms designed for the traditional model map easily onto actual physical computers, usually without modification. The traditional model also forces us, as theoreticians, to concentrate on the problem at hand. A number of algorithms these days — even for shortest paths and minimum spanning trees — apply very *model-specific* techniques and, as such, reveal less about the problem than they do about the *power* of the underlying machine.

1.1 Overview of the Results

1.1.1 Shortest Paths

In 1997 Thorup invented what we dub the *hierarchy-based* approach to shortest paths. Thorup's original algorithm was designed for *integer*-weighted undirected graphs, and the powerful RAM model, or random access machine. Because the hierarchy approach seemed to depend on all kinds of model-specific techniques, it was unclear whether the more general problem — shortest paths on real-weighted graphs — would admit an efficient hierarchy-based algorithm. In Chapters 2–6 we develop a number of faster shortest path algorithms, all hierarchy-based, and explore the inherent limitations of the approach.

In Chapter 3 we define a large class of *hierarchy-type* algorithms, and prove that, in general, no hierarchy-type algorithm can improve on Dijkstra's classical single-source shortest path (SSSP) algorithm. Basically, we show that there is an inherent "sorting bottleneck" in the approach, just as there is in Dijkstra's algorithm. However our lower bound does not scale up well. For instance it does *not* say that computing SSSP 5 times from different sources is 5 times as hard as SSSP. This is because shortest paths on the same graph are, by their nature, highly *dependent*. Knowing *some* shortest paths might give you a great deal of information about others.

The main theoretical contributions of our shortest path algorithms are some new techniques for *identifying* and *exploiting* the dependencies among shortest paths in the same graph. In Chapter 4 we give a new all-pairs shortest path (APSP) algorithm that runs in time $O(mn + n^2 \log\log n)$, where m and n are the number of edges and vertices respectively. This is the first theoretical improvement over Dijkstra's 1959 algorithm, which runs in $O(mn + n^2 \log n)$ time if implemented with a Fibonacci heap. In Chapter 4 we also address the *non-uniform* complexity of APSP. In particular we give an APSP algorithm making $O(mn \log \alpha(m, n))$ numerical operations, where α is the inverse-Ackermann function. Due to the trivial lower bound of $\Omega(n^2)$, our algorithm is within a tiny $\log \alpha(n, n)$ factor of *optimal* when $m = O(n)$.

In Chapter 5 we give a faster shortest path algorithm for *undirected* graphs. As an undirected *APSP* algorithm, it runs in $O(mn \log \alpha(m, n))$ time — again, nearly optimal for $m = O(n)$. As an undirected *SSSP* algorithm it runs in $O(m\alpha(m, n) + n \log \log r)$ time, where r bounds the ratio of any two edge lengths. Thus for $r = \text{poly}(n)$, our undirected SSSP algorithm runs in $O(m + n \log \log n)$ time, an improvement over Dijkstra's. In Chapter 6 we present the results of some experiments with a simplified version of our undirected shortest path algorithm. It consistently outperforms Dijkstra's on a variety of sparse graph types, and comes surprisingly close to the speed of breadth first search, which we use as a benchmark linear-time algorithm.

1.1.2 Minimum Spanning Trees

The minimum spanning tree problem (MST) has been studied for over 75 years, though it was only in recent years that sophisticated techniques were applied to the problem. In 1994 Karger, Klein, and Tarjan [127] developed a *randomized* expected linear time algorithm based on two key techniques: random sampling and minimum spanning tree *verification*. In 1997 Chazelle [28] addressed the *deterministic* complexity of the MST problem. The running time of his algorithm was slightly super-linear (of the inverse-Ackermann variety) and was based on a new approximate priority queue called the Soft Heap [29].

In Chapter 8 we solve *part* of the MST problem. We give, in particular, a provably optimal MST algorithm, and show that the decision-tree (comparison) complexity of the problem is equivalent to its algorithmic complexity. Thus, we have separated the issues of *finding* an optimal algorithm with *analyzing* its complexity. Our algorithm, like Chazelle's [28], is based on the Soft Heap.

In [28] Chazelle wondered what sort of data structure might be the key to an *explicit* linear-time MST algorithm. Clearly inspired by the success of MST verification in the randomized algorithm of Karger et al. [127], he proposed a "dynamic equivalent" to MST verification. In Chapter 9 we give an inverse-Ackermann type lower bound for the *online* MST verification problem, which may be considered the simplest dynamic equivalent. Our lower bound seems to rule out a faster explicit MST algorithm based on online MST verification. Parenthetically, this is the first inverse-Ackermann type lower bound for any comparison-based problem.

In Chapter 10 we give the first randomized *time-work* optimal parallel MST algorithm. Our algorithm improves on a long line of results, some time-optimal and some work-optimal.

One disadvantage of the *randomized* MST algorithms is that they use a number of random bits that is linear in the size of the problem. In reality however random bits are usually considered a scarce resource. In Chapter 11 we develop a new randomized MST algorithm that runs in expected linear-time, even if only a polylogarithmic number of random bits are available. It is parallelizable, and also gives an efficient parallel *connectivity* algorithm using polylogarithmic random bits. (A simple tweak of our optimal MST algorithm yields one that runs in expected linear time using $o(\log^* n)$ random bits. However this algorithm is not parallelizable.)

1.2 Preliminaries

We assume no specialized background knowledge. However the reader should be familiar with asymptotic notation $(O, \Omega, \Theta, \omega, o)$, graph terminology (tree, path, vertex, edge, cycle, etc.), and a little probability for the latter chapters. Refer to any standard algorithms textbook [47] for the necessary definitions.

We have summarized the standard asymptotic notation in Section 1.2.1. In Section 1.2.2 we summarize the chapter dependencies.

1.2.1 Asymptotic Notation

We use the standard asymptotic notations. Below, f and g are functions from naturals to naturals.

$$\begin{aligned}
f(n) = O(g(n)) &\quad\equiv\quad \exists c_1, c_2\, \forall n > 0 \ :\ f(n) \ \leq\ c_1 \cdot g(n) + c_2 \\
f(n) = \Omega(g(n)) &\quad\equiv\quad g(n) \ =\ O(f(n)) \\
f(n) = \Theta(g(n)) &\quad\equiv\quad f(n) = O(g(n)) \ \text{ and } \ f(n) = \Omega(g(n)) \\
f(n) = \omega(g(n)) &\quad\equiv\quad \lim_{n\to\infty} g(n)/f(n) \ =\ 0 \\
f(n) = o(g(n)) &\quad\equiv\quad g(n) \ =\ \omega(f(n))
\end{aligned}$$

Remark. Some sources in the literature use the asymmetric definition: $f(n) = \Omega(g(n))$ if there exists a constant c such that $c \cdot f(n) \geq g(n)$ for infinitely many integers n.

1.2.2 Chapter Dependencies

Parts I and II, on shortest paths and minimum spanning trees, respectively, are entirely independent.

Chapters 4 (directed shortest paths) and 5 (undirected shortest paths) are both built on the foundation of Chapters 2 and 3. Chapter 6 (experimental shortest paths) may be read separately, though it does frequently refer to the algorithm from Chapter 5.

Chapters 8–11 (results on minimum spanning trees) are independent of one another, though each should be read following the introduction to minimum spanning trees, in Chapter 7.

Part I

Shortest Paths

Chapter 2

Introduction to Shortest Paths

2.1 History

In Sections 2.1.1 and 2.1.2 we survey the history of the *single-source* and *all-pairs* shortest path problems, which are the "textbook" shortest path problems and the subject of subsequent chapters. In Section 2.1.3 we attempt to survey a slew of results extending the shortest path problem in various directions.

2.1.1 Single-Source Shortest Paths

The single-source shortest path problem, or SSSP, is a deceptively difficult problem. As early as 1960 there were two algorithmic solutions: Bellman and Ford's [17, 65, 47], which worked on arbitrarily weighted graphs, and Dijkstra's [52], which was a bit faster but assumed non-negatively weighted graphs. To date *neither* of these algorithms have been improved in the context of general real-weighted graphs. However there have been a number of qualified successes, as we shall see.

The Bellman-Ford algorithm runs in $O(mn)$ time, where m and n are the number of edges and vertices respectively. However this cost is generally very pessimistic; a finer analysis shows it runs in $O(hm)$ time, where h is the maximum number of edges in any shortest path. Goldberg [87], improving very slightly on Gabow and Tarjan's work [77, 80], gave an SSSP algorithm for *integer*-weighted graphs running in $O(\sqrt{n}m \log N)$ time, where N bounds the magnitude of the negative edge-lengths.

Dijkstra's 1959 SSSP algorithm [52] runs in $O(n^2)$ time if implemented in a straightforward fashion; this is optimal for dense graphs. It was quickly observed that speeding up Dijkstra's algorithm is tantamount to implementing a fast *priority queue*. Using Johnson's d-ary heap [118, 119], a generalization of Williams' binary heap [205], Dijkstra's algorithm runs in $O(m \log_{2+m/n} n)$ time, which is optimal for moderately dense graphs, say when $m/n = n^{\Omega(1)}$. The fastest implementation of Dijkstra's algorithm to date runs in $O(m + n \log n)$ time, making it optimal for $m/n = \Omega(\log n)$. It uses Fredman and Tarjan's Fibonacci heap [73]. In a *comparison-based* model of computation, one can easily show that Fibonacci heaps are asymptotically optimal, and that in the worst case Dijkstra's algorithm requires $\Omega(m + n \log n)$ time to solve. Thus any research on the SSSP problem must depart from the general comparison-based model, or keep the comparison model and depart from Dijkstra's algorithm. We take the latter approach. Efforts on the former have focused on implementations of Dijkstra's algorithm for *integer*-weighted graphs in the unit-cost RAM (random access machine) model of computation.[1]

Fredman and Willard [74, 75] showed that in the RAM model it is possible to sort n integers in $o(n \log n)$ time, and to implement priority queue operations in $o(\log n)$ time. (In other words the information-theoretic bottlenecks inherent in a comparison-based model do not apply here.) To date the best implementations of Dijkstra's algorithm on integer-weighted graphs run in time $O(m\sqrt{\log\log n})$ [102] (expected) and time $O(m + n \log\log n)$ [199].

In 1997, Thorup [196] invented the *hierarchy*-based approach to shortest paths — a clean break from Dijkstra's algorithm — and gave a linear-time SSSP algorithm for the restricted case of non-negative integer-weighted *undirected* graphs. The question of whether the hierarchy-based approach could be adapted to directed graphs and/or a comparison-based model of computation was left unanswered. Hagerup [98], in 2000, showed that indeed the hierarchy approach can be applied to directed integer-weighted graphs. His SSSP algorithm ran in $O(m \log\log N)$ time, where N is the largest edge length. Hagerup's algorithm provided no speedup over existing RAM-based SSSP algorithms, though it was deterministic and used only linear space.

2.1.2 All-Pairs Shortest Paths

The *APSP* problem — find the shortest path from every vertex to every other — can easily be solved with n SSSP computations. Thus, Bellman-Ford solves APSP in $O(mn^2)$ time and Dijkstra solves APSP (on non-negative edge lengths) in $O(mn + n^2 \log n) = O(n^3)$ time. However a more direct approach to APSP can give better bounds.

Dense Graphs

The Floyd-Warshall algorithm [47] computes APSP in $O(n^3)$ time, and has the practical advantages of being simple and streamlined. It is well known that a $(\min, +)$ matrix multiplier can be used to solve the all-pairs *distance* problem (APD), which does not ask for shortest paths per se. This gives an obvious $O(n^3 \log n)$-time APD/APSP

[1]The phrase *unit-cost* here emphasizes that all operations take unit time, even non-AC^0 ones like multiplication, and that all memory accesses take unit time, i.e., there is no cache in the model.

algorithm. What is less obvious is that the complexity of APD is asymptotically equivalent to $(\min, +)$ matrix multiplication — see Aho et al. [4]. Fredman [69] gave a min-plus multiplier that performs $O(n^{2.5})$ *numerical operations*; however there is no known polynomial-time implementation of Fredman's algorithm. The fastest min-plus algorithm to date is due to Takaoka [188], who uses Fredman's approach on small subproblems. Takaoka's algorithm runs in time $O(n^3\sqrt{\frac{\log\log n}{\log n}})$, which is a sub-logarithmic improvement over standard matrix multiplication.

One cannot directly apply the "fast" matrix multipliers, such as those of Strassen [186] or Coppersmith and Winograd [45], because $(\min, +)$ is not a ring: min has no inverse. However, ring-based matrix multiplication can be used in less obvious ways to compute APSP. The algorithms of [180, 82, 182, 9, 189, 209] take this approach, and yield improved, $o(n^3)$ APSP algorithms on integer-weighted graphs, provided that the magnitude of the integers is sufficiently small — always sublinear in n.

Sparse Graphs

Johnson [119] gave an interesting solution to the problem of negative edge-lengths. Assuming that no negative-length cycles exist, he showed that the shortest path problem is reducible in $O(mn)$ time to one of the same size, but having only non-negative edge lengths. Combined with Dijkstra's algorithm this immediately yields an APSP algorithm for arbitrarily weighted graphs running in $O(mn + n^2 \log n)$ time. Surprisingly Dijkstra's algorithm (with or without Johnson's reduction) remained the fastest general APSP algorithm for many years. (Refer to Chapters 4 and 5 for our improved APSP algorithms.)

In the context of integer-weighted graphs and the RAM model, the existing implementations of Dijkstra's SSSP algorithm [102, 199] imply some bounds on APSP: $O(\min\{mn\sqrt{\log\log n},\ mn + n^2\log\log n\})$. The hierarchy-type algorithms of Thorup [196] and Hagerup [98] also give bounds on APSP. Hagerup's algorithm solves APSP in $O(mn + n^2 \log\log n)$ time,[2] and Thorup's algorithm [196] solves *undirected* APSP in $O(mn)$ time.

2.1.3 Variations

Due to the practical significance of shortest paths, a number of variations on the problem have been proposed, each restricting or generalizing some aspect of the SSSP or APSP problems.

[2] Although their running times are identical, Hagerup's APSP algorithm is theoretically cleaner than the one derived from an implementation of Dijkstra's algorithm with Thorup's recent integer priority queue [199]. Thorup uses multiplication whereas Hagerup only uses standard AC^0 operations.

The case of planar graphs has been studied extensively [151, 152, 66, 67, 105, 61]. Interestingly the SSSP problem on planar graphs is only slightly more difficult under arbitrary edge-lengths [61] as opposed to positive edge lengths [105]. A number of algorithms have been analyzed under the assumption of a complete graph with randomly chosen edge lengths [184, 165, 128, 140, 187, 44], and two SSSP algorithms were presented recently [160, 89] that run in expected linear time when the edge-lengths are selected uniformly from some interval. There are shortest path algorithms guaranteeing approximate solutions (see Zwick's survey [208]), dynamic shortest path algorithms (see Demetrescu and Italiano [50] for more references), preprocessing schemes for answering (approximate) shortest path queries [200, 197, 136, 96, 144, 51], parallel shortest path algorithms [201, 137, 101, 161], cache-efficient shortest path algorithms [155, 156, 162], geometric shortest path algorithms [164], and a zillion others. (We have only sampled the available literature and make no claim to completeness.)

2.1.4 Organization

In Section 2.2 we summarize our contributions to the shortest path problem, which are revealed in merciless detail in Chapters 3–6. In Section 2.3 we give a formal definition of the problem and introduce some notational conventions. In Section 2.4 we define the *comparison-addition* model, and discuss various aspects of the model. In Section 2.5 we describe Dijkstra's algorithm and discuss a class of *Dijkstra-like* algorithms. In Section 2.6 we give a gentle introduction to the hierarchy-based approach to shortest paths.

2.2 Our Contributions

Thorup's hierarchy approach [196] to shortest paths is designed for integer-weighted graphs, and at first glance, *seems* to depend essentially on the RAM model and the assumption of integral edge-lengths. Indeed, any straightforward "port" of Thorup's SSSP algorithm to the comparison-addition model (see Section 2.4) will incur a sorting bottleneck, that is, a running time of $\Omega(n \log n)$. In Section 3.6 we give a fairly strong lower bound showing that any *hierarchy-type* SSSP algorithm must, in the worst case, perform $\Omega(m + n \log n)$ numerical operations, even if the graph is undirected. The implications for hierarchy-type *APSP* algorithms are less severe. Our lower bound shows that solving APSP with n independent executions of a hierarchy-type SSSP algorithm is sure to lead to running times of at least $\Omega(mn + n^2 \log n)$ — no improvement over Dijkstra — since each SSSP computation is subject to the lower bound.

The way out of this bind is to exploit the strong *dependencies* that exist among shortest paths in the same graph. Our undirected shortest path algorithm [Chapter 5], for instance, constructs a linear-space *hierarchy* structure that encodes useful information about every shortest path in the graph. Once the hierarchy structure is built we can compute SSSP from any source in $O(m \log \alpha(m,n))$ time — essentially linear — with a relatively simple and streamlined algorithm. This leads directly to an $O(mn \log \alpha(m,n))$ APSP algorithm for undirected graphs. In the context of computing APSP, or even SSSP multiple times, the cost of computing the hierarchy structure is insignificant. However it may be the dominant cost when computing SSSP exactly once. Our best bound on SSSP is $O(m\alpha(m,n) + \min\{n \log\log r,\, n \log n\})$, where r bounds the ratio of any two edge lengths. For $r = \text{poly}(n)$ — a fairly reasonable assumption — the bound becomes $O(m + n \log\log n)$, which is an improvement over Dijkstra's algorithm.

Directed graphs are a different beast. At a high level our directed shortest path algorithms [Chapter 4] are applying the same general technique: trimming costs by exploiting certain dependencies among shortest paths. However the techniques we develop for directed graphs are significantly more sophisticated than those for undirected graphs. In Section 4.1 we present a directed APSP algorithm that runs in time $O(mn + n^2 \log\log n)$; this is the first improvement over Dijkstra's APSP algorithm on real-weighted graphs. We cannot find a *faster* directed APSP algorithm, but in Section 4.2 we give a *non-uniform* APSP algorithm performing $O(mn \log \alpha(m,n))$ numerical operations. Notice that for $m = O(n)$, this bound is only a miniscule $\log \alpha(n,n)$ factor from optimal complexity. (This is very encouraging. It suggests that some part of the APSP problem is actually soluble with *existing* techniques.)

In Chapter 6 we present the results of some experiments with a simplified version of our undirected shortest path algorithm [Chapter 5]. The results are fairly impressive. After the hierarchy structure is built, our algorithm consistently outperforms Dijkstra's algorithm on a variety of graph classes and sizes. It also performs between 1.81 and 2.77 times the speed of breadth first search, which can be considered a reasonable *lower bound* on the practical limits of any shortest path algorithm.

2.3 Problem Definition

The input is a weighted directed graph $G = (V, E, \ell)$ where $|V| = n$, $|E| = m$, and $\ell : E \to \mathbb{R}$ assigns a real *length* to every edge. It was mentioned in Section 2.1.2 that the shortest path problem is reducible in $O(mn)$ time to one of the same size but having only non-negative edge lengths, assuming that no negative length cycles exist. We will assume henceforth that $\ell : E \to \mathbb{R}^+$ assigns only non-negative lengths.

The length of a path is defined to be the sum of its constituent edge lengths, and a *shortest* path, from one specified vertex to another, is one having minimum length. The *distance* from u to v, denoted $d(u,v)$ is the length of a shortest path from u to v, or ∞ if none exists. The APSP problem is to compute the values $d(u,v)$, for all $(u,v) \in V \times V$, and the SSSP problem is to compute the values $d(s,u)$ for a fixed *source*

$s \in V$ and all $u \in V$. The SSSP problem is sometimes defined to be that of finding shortest paths, not distances. However, one can easily show that given one — shortest paths or distances — the other is computable in linear time. For the sake of simplicity we focus on distances.

We frequently extend the distance notation to include objects other than vertices. For instance, if H is a subgraph, a set of vertices, or any object identified with a set of vertices, we let $d(u, H)$ denote the minimum distance from u to any vertex in H.

2.4 The Comparison-Addition Model

Many computational models, such as the Turing machine and the word RAM, have the property that data is finite, discrete, and *inspectible*. That is, the representation of an elemental piece of data (a symbol on the tape of a Turing machine or the bits of a word in a word RAM) can be fully known. For problems whose input consists of *real*-weighted elements, such as the shortest path problem, it is impossible to work within a model whose data is both finite and inspectible. In the comparison-addition model we sacrifice inspectibility in order to retain the full generality of real-weighted data. Real numbers are represented in special variables of type real. The *only* operations allowed on reals are additions and comparisons, of the form:

$$a \;:=\; b + c$$

and

```
if $a < b$ then . . . else . . .
```

The comparison-addition model is not really complete because we have yet to define what happens on non-real data. All of our algorithms work under the RAM model (random access machine). Specifically, we assume the existence of a type *integer*, which, like reals, is subject to comparisons and additions. We also assume that integers can be used to index arrays. That is, if A is an array and i an integer, the element $A[i]$ can be retrieved in unit-time. We assume *no* primitive operations that convert reals to integers or vice versa.

A realist may argue that since real-life machines have finite, discrete, and inspectible data, one should study optimization problems (e.g., shortest paths) whose weighted elements are assumed to be integers. In the abstract this has certainly been a very successful endeavor. For several important optimization problems, such as maximum flow [91], maximum weight matching [80, 81], and single-source shortest paths [87, 80], the fastest algorithms for integer-weighted inputs can be faster than their counterparts for real-weighted inputs by up to a polynomial factor, so long as the magnitude of the integers does not get too large. These theoretical improvements are significant, though they do not always result in corresponding real-world improvements. In practice it is not unusual for an algorithm to have wildly differing worst-case and typical-case running times (Bellman-Ford and nearly all maximum flow algorithms come to mind). Depending on the problem, there may be no practical benefit to assuming integer-weighted graphs.

An often overlooked aspect of the comparison-addition model is that its restrictive, algebraic framework is actually useful in practice. By not meddling with the internal representation of numbers, algorithms in the comparison-addition model naturally work with a variety of numerical types.[3] Moreover, it is possible to prove the correctness of such algorithms with clean mathematical arguments.

2.4.1 Non-Uniform Complexity

We will use the term *comparison-addition complexity* to refer to the number of real-number operations performed by an algorithm. This is a *non-uniform* complexity measure, in the sense that an algorithm with a certain comparison-addition complexity will not, in general, have the same running time asymptotically. The difference between uniform and non-uniform computation is usually understood as the difference between Turing machine complexity and circuit complexity. Our situation is basically analogous to this one, where our souped-up RAM takes the place of the Turing machine and algebraic decision trees replace circuits.[4]

2.4.2 Basic Techniques

We frequently make use of real number operations not included in the comparison-addition model, such as subtraction, multiplication by an integer, division and the floor operation. We show below how these operations can be simulated in the comparison-addition model, sometimes without asymptotic penalty.

To simulate subtraction we represent each abstract real number a by two actual real numbers a_1 and a_2 such that $a = a_1 - a_2$. Both abstract addition and abstract subtraction are accomplished with two actual additions, since $a+b = (a_1+b_1)-(a_2+b_2)$ and $a - b = (a_1 + b_2) - (a_2 + b_1)$. An abstract comparison between a and b translates into an actual comparison between $(a_1 + b_2)$ and $(a_2 + b_1)$.

Multiplication by an integer is also not difficult. Suppose a is a real and N an integer. We can calculate Na in $O(\log N)$ time as follows. Produce the set of reals $B = \{a, 2a, 4a, 8a, \ldots, 2^{\lfloor \log N \rfloor} a\}$, using $\log N$ additions, then produce Nx by summing

[3]LEDA [157], for instance, has a number of numerical data types beyond the usual `int` and `float`, as do the Java & C# programming languages.

[4]This analogy is not entirely tight. A family of circuits solving a problem would have one circuit per problem size, whereas in the shortest path problem we would have one algebraic decision tree for each distinct input graph.

up the appropriate subset of B. Division by an integer is accomplished in a similar fashion. Suppose we set $a := b/N$. If we want to compare a with another number, say c, we can substitute the equivalent comparison between b and Nc. Here b/N is not calculated but represented symbolically. (In general division can be very inefficient; it can cause a large blow-up in the time to simulate future comparisons.)

An operation that comes in very handy is taking the floor (or ceiling) of the ratio of two reals, i.e., computing the *integer* $\lfloor \frac{a}{b} \rfloor$. This operation is different from the ones discussed above because the result is an integer rather than a real number. We compute the floor of a ratio using a method similar to our simulation of multiplication. To compute $\lfloor \frac{a}{b} \rfloor$ we first produce the set $B = \{b, 2b, 4b, 8b, \ldots, 2^{\lceil \log \frac{a}{b} \rceil} b\}$, then use the elements of B to implement a binary search to find the integer $\lfloor \frac{a}{b} \rfloor$. This takes $O(1 + \log \frac{a}{b})$ time

2.4.3 Lower Bounds

There are several known lower bounds on various shortest path problems in the comparison-addition model. However, they are all very weak. Spira and Pan [185] showed that, regardless of additions, $\Omega(n^2)$ comparisons are necessary to solve SSSP on the complete graph. Karger et al. [128] proved that all-pairs shortest paths requires $\Omega(mn)$ comparisons if all summations correspond to paths in the graph. However, this assumption is restrictive: the Fredman and Takaoka algorithms [69, 188] are not path-based, and neither are ours. Kerr [132] showed that any straight-line (oblivious) APSP algorithm performs $\Omega(n^3)$ operations, and Kolliopoulos and Stein [140] proved that any fixed sequence of edge relaxations solving SSSP must have length $\Omega(mn)$. By "fixed sequence" they mean one which depends on m and n but *not* the graph topology. Graham et al. [95] did not give a lower bound but showed that the standard information-theoretic argument cannot yield a non-trivial, $\omega(n^2)$ lower bound in the APSP problem. Similarly, no information-theoretic argument can provide an interesting lower bound on SSSP.

2.5 Dijkstra's Algorithm

It is sometimes useful to think about the SSSP problem as that of simulating a physical process. Suppose that the graph represents a network of water pipes, and that at time zero we begin injecting water into the network at a specific place: the source. The SSSP problem is to compute when the water reaches each place in the network. Other network optimization problems correspond to certain physical processes (network flow and minimum spanning trees come to mind). Dijkstra's algorithm is one of the few that actually simulates the physical process directly. That is, the states of Dijkstra's algorithm correspond to states in the physical system.

Recall that the source vertex is represented by s. Dijkstra's algorithm maintains a set of *visited vertices* S, which, from the point of view of the simulation, corresponds exactly to the places in the pipe network already reaches by the water. Therefore, at any point in Dijkstra's algorithm we are implicitly at time $\max_{v \in S} d(s, v)$. Dijkstra's algorithm maintains a *tentative distance* $D(v)$ for each $v \in V$, satisfying the following invariant.

Invariant 1 *(Dijkstra's Invariant) For $v \in S$, $D(v) = d(s, v)$ and for $v \notin S$, $D(v)$ is the distance from s to v in the subgraph induced by $S \cup \{v\}$.*

In the simulation $D(v)$ represents the estimated time when water will reach v, based on when water reached vertices in S. $D(v)$ is an upper bound on $d(s, v)$ and is not equal to $d(s, v)$ precisely when the shortest path to v passes through some vertex in $V - S$. Dijkstra's algorithm adds vertices to the set S one by one, which implies, since it is a physical simulation, that the next vertex added is always the $v \in (V - S)$ minimizing $d(s, v)$. This is the same $v \in (V - S)$ minimizing $D(v)$ since edge lengths are assumed to be non-negative. Once we set $S := S \cup \{v\}$, the D-values may not satisfy Dijkstra's Invariant. To restore Invariant 1 we *relax* each outgoing edge (v, w) of v, setting $D(w) := \min\{D(w), D(v) + \ell(v, w)\}$. Eventually $S = V$, implying that $D(v) = d(s, v)$ for all $v \in V$.

The only complicated part of Dijkstra's algorithm is deciding which vertex to visit next. Dijkstra [52], more concerned with space than time, proposed examining $D(v)$ for all $v \in (V - S)$. This gives an SSSP algorithm with overall running time $O(n^2)$. Using Fibonacci heaps [73], Dijkstra's algorithm can be made to run much faster — in $O(m + n \log n)$ time — with only a small constant factor increase in space usage.

It is important to notice that Dijkstra's algorithm represents only one method for maintaining Invariant 1 and that, in principle, there are many "Dijkstra-like" algorithms that grow the set S while preserving Invariant 1. When such an algorithm adds a vertex to S, say v, it must have a certificate that $D(v) = d(s, v)$, in particular that for all $u \notin S$, $D(u) + d(u, v) \geq D(v)$. Dijkstra's certificate is simply that $D(u) \geq D(v)$ by choice of v, and that $d(u, v) \geq 0$ by the assumption that edge-lengths are non-negative. To depart from Dijkstra's algorithm one must be able to find a better lower bound on $d(u, v)$ than the trivial $d(u, v) \geq 0$.

Our shortest path algorithms are all Dijkstra-like, according to the definition above. Therefore, the meaning of D, S, and s will be preserved in later chapters, as will the meaning of the terms "visit" and "relax." We may refer to Invariant 1 as simply Dijkstra's Invariant.

2.6 The Hierarchy Approach

The main limitation of Dijkstra's algorithm is that it visits vertices in order of increasing distance from the source. If we view the set S as the state, Dijkstra's algorithm passes through n distinct states corresponding to n physical states. Dinic [56] observed that in general, not every state of the SSSP algorithm must correspond to a physical state. Let $t > 0$ be the minimum edge length in the graph. In Dinic's variation on Dijkstra's algorithm, rather than visiting $v \in (V - S)$ minimizing $D(v)$, we visit any $v \in (V - S)$ minimizing $\lfloor D(v)/t \rfloor$, or indeed, every such v minimizing $\lfloor D(v)/t \rfloor$ simultaneously. In other words, we are setting up checkpoints at "time" $0, t, 2t, 3t, \ldots$ where the physical and algorithmic states are in alignment. Between these checkpoints the algorithm passes through states that have no physical equivalent.

Generally speaking Dinic's algorithm provides no improvement over Dijkstra's algorithm. However, it is the kernel of the hierarchy-based approach, which was invented by Thorup [196] for the special case of integer-weighted undirected graphs. Thorup's insight was that Dinic's algorithm can be generalized to arbitrary (and even multiple) values of t; it need not fix t at the minimum edge length. Consider a simplified, but illustrative example.

Suppose that $t > 0$ is arbitrary and the vertex set V is partitioned into disjoint sets $V_1, V_2, \ldots, V_k$ where any edge from V_i to V_j, $i \neq j$, has length at least t. Let G^c be derived from the input graph G by contracting $V_1, \ldots, V_k$ to single vertices, denoted $v_1, \ldots, v_k$. On such a graph one can think of a hierarchy-type SSSP algorithm as being composed of (at least) $k + 1$ processes, one that operates on G^c, and k that operate on the graphs induced by $V_1, \ldots, V_k$. The process operating on G^c basically runs Dinic's SSSP algorithm. It needs a slight modification because a vertex $v_i \in V(G^c)$ is really a subgraph on V_i, not an actual vertex. Therefore, rather than v_i being either visited or not, it can be *partially* visited if V_i is only partially contained in S. The process operating on G^c proceeds as follows. It visits, by delegating responsibility to the other processes, all vertices whose distances lie in the interval $[0, t)$, followed by those that lie in $[t, 2t)$, $[2t, 3t)$, etc. Suppose that the process governing V_i is told to visit all vertices in V_i whose distances lie in $[jt, (j + 1)t)$. This process is given what in later sections is called an *independent subproblem*, meaning that it can be solved by looking only at V_i and the current tentative distances, i.e. D-values. (Proving independence is not difficult; the argument is essentially the same as that found in the proof of correctness of Dinic's algorithm.) The process governing V_i could solve its subproblems using Dijkstra's algorithm, where the heap would contain the D-values of just those vertices in V_i. However, there is no reason why we cannot apply the same scheme recursively. We would simply choose a new threshold t_i and partition V_i into $V_{i,1}, V_{i,2}, \ldots, V_{i,k_i}$ such that all edges crossing the partition have length at least t_i. We refer to this recursive partitioning of the vertices as a hierarchy.

It is certainly not obvious how to implement this algorithm efficiently. There is the question of whether a *good* hierarchy can be computed efficiently, and — this is a separate issue — whether the algorithm admits a fast implementation, given a sufficiently good hierarchy. One of our primary concerns is whether there is an inherent *sorting bottleneck* in the approach. If there is such a bottleneck, then all hierarchy-based algorithms are doomed to have running times of $\Omega(m + n \log n)$, the same as Dijkstra's. Of course, the absence of any kind of information-theoretic bottleneck does not imply a faster hierarchy-based shortest path algorithm, but it would suggest the existence of one.

In subsequent chapters we give a nearly-complete answer to the sorting bottleneck question, though it is more complicated than simply *yes* or *no*. Several factors influence the complexity of the hierarchy-type shortest path algorithms, including:

- Whether the graph is directed or undirected.
- Whether the ratio of the maximum-to-minimum edge length is large, as a function of the number of vertices.
- Whether a good hierarchy is given or needs to be computed from scratch. (Computing it from scratch can involve a sorting bottleneck.)
- Whether SSSP is to be computed once, or repeatedly on the same graph.
- Whether the topology and edge-length distribution of the input graph is typical. Typical graphs are very different than our worst-case examples.

Chapter 3

Hierarchies & Shortest Paths

The central idea in hierarchy-type algorithms is that of dividing the SSSP problem into a series of *independent subproblems.* In this chapter we define precisely this notion of independence, and show how independent subproblems can be created and manipulated.[1]

3.1 Independent Subproblems

Recall that s denotes the source of the SSSP problem. Let $X \subseteq V$ denote a set of vertices. We define $d_X(s,v)$ to be the distance from s to v in the subgraph induced by X (or ∞ if X does not contain both s and v.) If I is a real interval, we define X^I to be the set $\{v \in X \ : \ d(s,v) \in I\}$, that is, those vertices in X whose distances from the source lie in I.

Definition 1 *Let X and S be sets of vertices and I be a real interval. We will call X (S,I)-independent if for all $v \in X^I$, $d(s,v) = d_{S \cup X^I}(s,v)$*

To paraphrase Definition 1, if X is (S,I)-independent then one can determine the set X^I by examining only the subgraph induced by $S \cup X^I$. Suppose that we discover that X is (S,I)-independent in the context of a Dijkstra-like algorithm, i.e. one satisfying Invariant 1. Now we can say something stronger: because the D-values for vertices in $X^I - S$ encode all the relevant information about the subgraph induced on S, one can determine X^I by examining only the subgraph induced by $X^I - S$ and the D-values of those vertices.

[1]This chapter's notation and exposition are taken largely from two papers: (1) S. Pettie, A faster all-pairs shortest path algorithm for real-weighted sparse graphs, Proc. 29th Int'l Colloq. on Automata, Languages, and Programming (ICALP), pp. 85–97, 2002, full version to appear in *Theoretical Computer Science*, and (2) S. Pettie and V. Ramachandran, Computing shortest paths with comparisons and additions, Proc. 13th ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 267–276, 2002. The results of Section 3.6.3 will appear in the journal version of (2).

A *t-partition*, defined below, is a key tool for creating new, smaller independent subproblems given a larger one.

Definition 2 *Let X be a set of vertices. The sequence $(X_1, X_2, \ldots, X_k)$ is a t-*partition *of X if $\{X_i\}_i$ is a partition of X and for every edge (u, v) where $u \in X_i$, $v \in X_j$, and $j < i$, we have $\ell(u, v) \geq t$.*

Note the asymmetry in Definition 2. In a t-partition only "backward" edges crossing the partition have length at least t; "forward" edges can have any length. Lemma 1 shows the relationship between t-partitions and independent subproblems. It generalizes some of the Lemmas given by Thorup [196].

Lemma 1 *Suppose that X is $(S, [a, b))$-independent. Let $(X_1, \ldots, X_k)$ be a t-partition of X, let I be the interval $[a, \min\{a + t, b\})$, and let $S_i = S \cup X_1^I \cup X_2^I \cup \cdots \cup X_i^I$. Then*

1. *X_{i+1} is (S_i, I)-independent*
2. *X is $(S_k, [a + t, b))$-independent*

Proof: First consider Part (2). The assumption is that X is $(S, [a, b))$-independent, meaning that for $v \in X^{[a,b)}$, $d_{S \cup X^{[a,b)}}(s, v) = d(s, v)$. Since $S_k = S \cup X^I$, we have $S \cup X^{[a,b)} = S_k \cup X^{[a+t,b)}$, which immediately implies that X is $(S_k, [a+t, b))$-independent as well. Note that the interval $[a + t, b)$ may be empty if $b \leq a + t$.

Now consider Part (1). The set X_{i+1} is (S_i, I) independent if for any $v \in X_{i+1}^I$, $d(s, v) = d_{S_i \cup X_{i+1}^I}(s, v)$. Suppose that this is not the case, that is, that every shortest s–to–v path is not contained in $S_i \cup X_{i+1}^I = S_{i+1}$. Let w be the last vertex on such a shortest path which is not in S_{i+1}. The independence of X w.r.t. $(S, [a, b))$ implies $w \in X$, and the inequalities $d(s, w) \leq d(s, v) < \min\{a + t, b\}$ further imply $w \in (S_k - S_{i+1})$. By the definition of a t-partition we have that $d(w, v) \geq t$. Together with the inequality $d(s, v) = d(s, w) + d(w, v) < \min\{a + t, b\}$ we also have that $d(s, w) < a$. We now have enough to obtain a contradiction. For any shortest s–to–v path we proved the existence of a w on this path that is neither in S nor in $X^{[a,b)}$, implying that $d(s, v) < d_{S \cup X^{[a,b)}}(s, v)$. This directly contradicts our initial assumption that X is $(S, [a, b))$-independent.

□

Lemma 1 is essentially describing a divide and conquer scheme for SSSP. The idea is to find an independent subproblem on the vertex set X, divide it into a series of smaller independent subproblems, with the aid of a t-partition, then solve the smaller problems recursively. There are several major obstacles to implementing this general algorithm efficiently, which we will address in subsequent chapters. The first order of business is computing and representing t-partitions. All of our shortest path algorithms have the property that the choice of t-partitions does not depend on the source vertex. Therefore, for any input graph we shall compute, once and for all, a single set of t-partitions, which we represent using a rooted tree, or *hierarchy*.

3.2 A Stratified Hierarchy

A *hierarchy* is a rooted tree where there is a one-to-one correspondence between its leaves and the graph's vertices. There is a natural correspondence between hierarchy nodes and graph objects. We will frequently use the same notation to refer to leaf-nodes and graph vertices, and will treat internal nodes as representing either sets of vertices or the induced subgraphs of those vertices. If x is an internal node we let $V(x)$ be the vertices represented by x, i.e. the set of leaf-nodes descending from x. We denote the parent of x in the hierarchy by $p(x)$, and let $\textsc{child}(x) = (x_1, x_2, \ldots, x_{\textsc{deg}(x)})$ denote the children of x, from left to right, where $\textsc{deg}(x) = |\textsc{child}(x)|$. The $|V(x)|$ and $\textsc{deg}(x)$ statistics provide two ways to measure how "big" a node x is. Two others will come in handy. We let $\textsc{diam}(x)$ represent an upper bound on the diameter of $V(x)$, where diameter is defined as $\max_{u,v \in V(x)} \{d(u,v)\}$. We associate with x a real number $\textsc{norm}(x)$, and refer to the ratio $\textsc{diam}(x)/\textsc{norm}(x)$ as the *normalized diameter* of x. We assign $\textsc{norm}$-values to hierarchy nodes with several objectives in mind, namely the correctness, speed, and simplicity of our shortest path algorithms. Since the main concerns of this chapter are only correctness and simplicity, we can say that $\textsc{norm}$-values are assigned to satisfy two conditions.

1. Either $\textsc{norm}(p(x))$ is an integer multiple of $\textsc{norm}(x)$ or $\textsc{norm}(p(x)) > \textsc{diam}(x)$.
2. Let $\textsc{child}(x) = (x_1, \ldots, x_{\textsc{deg}(x)})$. Then $(V(x_1), \ldots V(x_{\textsc{deg}(x)}))$ is a $\textsc{norm}(x)$-partition of $V(x)$.

Item (1) allows us to avoid great complications in our shortest path algorithms, but is otherwise of no interest. Item (2), in conjunction with Lemma 1, will clearly be useful in the creation of independent shortest path subproblems.

Our system for assigning $\textsc{norm}$-values is best explained by demonstrating why the simple schemes used by Thorup and Hagerup [196, 98] do not work in the comparison-addition model. Thorup and Hagerup always choose their $\textsc{norm}$-values from the set $\{2^i\}_{i \geq 0}$; a node with $\textsc{norm}$-value 2^i then corresponds to a connected component [196] (or strongly connected component [98]) in the graph restricted to edges with length less than 2^{i+1}. In the comparison-addition model, however, the set $\{2^i\}_{i \geq 0}$ cannot be generated because there is no sequence of operations (that is, additions) that generates the constant 1. This, of course, is no great obstacle. We can simply choose our $\textsc{norm}$-values from the set $\{\ell_1 \cdot 2^i\}_{i \geq 0}$, where ℓ_1 denotes the minimum non-zero edge length in the graph. In other words, we are just using the old system, under the irrefutable assumption that $\ell_1 = 1$. Although this system should work well in practice, there is a theoretical objection to it that must be addressed. In the comparison-addition model the time required to generate $\ell_1 \cdot 2^i$ is exactly i, so if the ratio of the maximum-to-minimum edge length is r, generating the largest $\textsc{norm}$-value could take $\log r$ time, which is *unbounded*[2] in terms of m and n. Our solution is to build a *stratified hierarchy* $\mathcal{SH}$, where each stratum corresponds to a different normalizing edge length. For example, the scheme with NORM-values from $\{\ell_1 \cdot 2^i\}_{i \geq 0}$ would have one stratum, with ℓ_1 as its normalizing edge length. We ensure that the ratio of two NORM-values within a stratum is bounded as a function of n, and that the strata are well-separated in a certain sense.

We now define the structure of our stratified hierarchy $\mathcal{SH}$. First, let $\ell_1, \ldots, \ell_m$ be the non-zero edge lengths of the graph in sorted order. We choose, as our set of normalizing lengths,

$$\{\ell_1\} \cup \{\ell_j \,:\, \ell_j > 2n \cdot \ell_{j-1}\} \cup \{\infty\}$$

That is, every normalizing length is much larger than any shorter edge lengths. Let ℓ_{r_k} be the kth smallest normalizing length. The nodes of $\mathcal{SH}$ are indexed by their stratum and level within the stratum. For stratum k the levels run from 0 to the maximum i such that $\ell_{r_k} \cdot 2^i < \ell_{r_{k+1}}$. The stratum k, level i nodes of $\mathcal{SH}$ correspond to the strongly connected components [3] (SCCs) in the graph restricted to edges with length less than $\ell_{r_k} \cdot 2^i$. If x is such an $\mathcal{SH}$-node then $\text{NORM}(x)$ is defined as:

$$\text{NORM}(x) \overset{\textbf{def}}{=} \ell_{r_k} \cdot 2^{i-1}, \quad \text{where } x \text{ is at stratum } k\text{, level } i \tag{3.1}$$

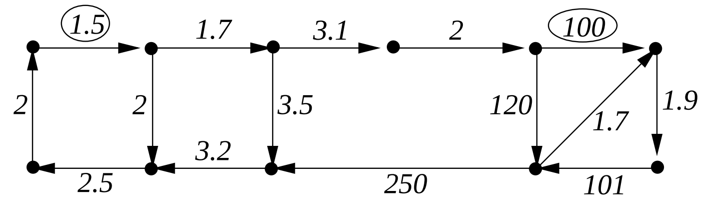
A node x is an ancestor of y if $V(x) \supseteq V(y)$ and x is higher in $\mathcal{SH}$ than y (higher stratum of same stratum and higher level). If $V(x) = V(y)$, where y is a descendant of x, then we will call x *irrelevant*. In the tree representation of $\mathcal{SH}$ we shall ignore irrelevant nodes, that is, nodes with one child. Henceforth, "$x \in \mathcal{SH}$" means x is a relevant node in $\mathcal{SH}$. The notation $p(x)$ and $\text{CHILD}(x)$ should be interpreted with respect to the tree of relevant $\mathcal{SH}$ nodes. That is, $p(x)$ is the nearest relevant ancestor, and $\text{CHILD}(x)$ is a sequence of nodes $(x_i)_i$ for which $p(x_i) = x$. Figure 3.1 gives an example input graph and its associated $\mathcal{SH}$.

If $\{x_i\}_{1 \leq i \leq \text{DEG}(x)}$ is the set of x's children, we set $\text{CHILD}(x) = (x_1, x_2, \ldots, x_{\text{DEG}(x)})$ so that $(V(x_1), V(x_2), \ldots, V(x_{\text{DEG}(x)}))$ is a $\text{NORM}(x)$-partition of $V(x)$. Lemma 2 guarantees that such a left-to-right ordering of x's children always exists.

Lemma 2 *Let $x \in \mathcal{SH}$ and $\{x_i\}_i$ be the children of x. Then for at least one permutation π, $(V(x_{\pi(1)}), V(x_{\pi(2)}), \ldots, V(x_{\pi(\text{DEG}(x))}))$ is a $\text{NORM}(x)$-partition of $V(x)$. Moreover, if the graph is undirected then then all such permutations give $\text{NORM}(x)$-partitions of $V(x)$.*

[2] In the algorithms of Thorup and Hagerup [196, 98] $\log r$ is also unbounded in terms of m and n, but, by assumption, *not* in terms of the machine's word size. Therefore the [196, 98] algorithms get around this issue by assuming that the power of the machine scales with the largest edge-length, not with m or n.

[3] A strongly connected component is a maximal subgraph such that any vertex in the subgraph is reachable from any other.



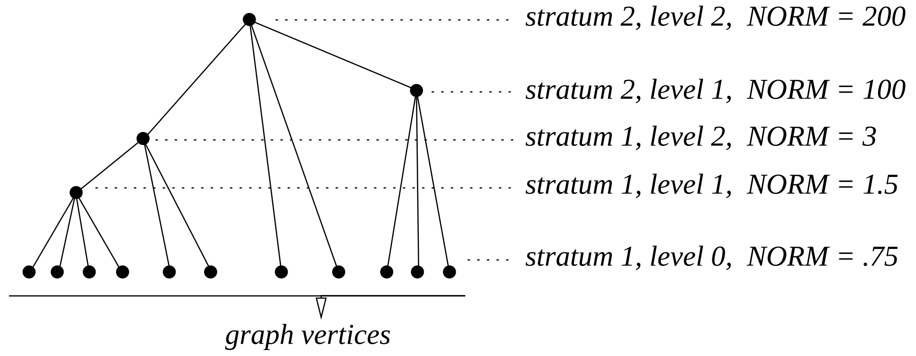

stratum 2, level 2, NORM = 200
stratum 2, level 1, NORM = 100
stratum 1, level 2, NORM = 3
stratum 1, level 1, NORM = 1.5
stratum 1, level 0, NORM = .75
graph vertices

Figure 3.1: Above: the input graph. Circled edge lengths represent "normalizing" lengths. Below: the associated $\mathcal{SH}$. It has two strata, based on the normalizing lengths $\ell_{r_1} = 1.5$ and $\ell_{r_2} = 100$. A stratum k, level i node x has $\text{NORM}(x) = \ell_{r_k} \cdot 2^{i-1}$, and represents a strongly connected component of the graph, when restricted to edges with length less than $2 \cdot \text{NORM}(x)$. Irrelevant $\mathcal{SH}$-nodes (those having one child) are not shown in the figure.

Proof: Let $G(x)$ be the subgraph of G induced by $V(x)$. By definition $G(x)$ is strongly connected, even when restricted to edges with length less than $2 \cdot \text{NORM}(x)$. Let $G^c(x)$ be the graph derived from $G(x)$ by contracting $G(x_1), G(x_2), \ldots, G(x_{\text{DEG}(x)})$ and retaining only edges with length less than $\text{NORM}(x)$. There is a natural correspondence between vertices in $G^c(x)$ and the children of x. We claim that (a) $G^c(x)$ is acyclic and (b) If we let $\pi(i)$ be the index of the ith vertex in a topological sort of $G^c(x)$, then $(V(x_{\pi(1)}), \ldots, V(x_{\pi(\text{DEG}(x))}))$ is a $\text{NORM}(x)$-partition of $V(x)$.

Consider claim (a). By definition $V(x_i)$ is a *maximal* strongly connected set, in the graph restricted to edges with length less than $\text{NORM}(x)$. If x_i were contained in a cycle in $G^c(x)$, then the maximality of $V(x_i)$ would be violated, since all edges in $G^c(x)$ have length less than $\text{NORM}(x)$.

We turn to claim (b). Assume w.l.o.g. that $\pi(i) = i$. If the claim were not true then by the definition of t-partition (Definition 2) there must be an edge $e = (x_j, x_i)$ where $i < j$ and $\ell(e) < \text{NORM}(x)$. However, $\ell(e) < \text{NORM}(x)$ implies e was included in

$G^c(x)$, which implies that $j < i$ – a contradiction – since x_j must precede x_i in every topological sort.

Now suppose that G were undirected, or rather, G is a directed graph where the existence of an edge (u, v) implies an edge (v, u) with equal length. Claim (a) above states that $G^c(x)$ is acyclic. This implies that $G^c(x)$ has no edges, since the existence of one edge immediately implies the existence of a cycle of length 2. Therefore, any permutation π corresponds to a topological sort of $G^c(x)$.

□

Recall that $\textsc{diam}(x)$ represented an upper bound on the diameter of $V(x)$. For any leaf-node z, setting $\textsc{diam}(z) = 0$ is clearly satisfactory. We compute $\textsc{diam}(x)$ for all internal $\mathcal{SH}$-nodes with the following recursive definition.

$$\textsc{diam}(x) \;=\; 2\,\textsc{norm}(x) \cdot (\textsc{deg}(x) - 1) \;+\; \sum_{y \,\in\, \textsc{child}(x)} \textsc{diam}(y) \tag{3.2}$$

Lemma 3, given below, summarizes all the relevant properties of $\mathcal{SH}$ used in our algorithm's analysis and proof of correctness. Parts 2 and 4 are implicit in [196, 98]; weaker versions of Part 6 were also used in [196, 98].

Lemma 3 *$\mathcal{SH}$ has the following properties:*

1. *$\mathcal{SH}$ has a single root, denoted $\textsc{root}(\mathcal{SH})$.*

2. *Let $\textsc{child}(x) = (x_1, x_2, \ldots, x_{\textsc{deg}(x)})$. Then $(V(x_1), \ldots, V(x_{\textsc{deg}(x)}))$ is a $\textsc{norm}(x)$-partition of $V(x)$.*

3. *Either $\textsc{norm}(p(x))$ is an integer multiple of $\textsc{norm}(x)$ or $\textsc{diam}(x) < \textsc{norm}(p(x))$.*

4. $$\sum_{x \,\in\, \mathcal{SH}} \textsc{deg}(x) \;<\; 2n - 1$$

5. $$\text{For any } x \in \mathcal{SH},\; \frac{\textsc{diam}(x)}{\textsc{norm}(x)} \;<\; 2n$$

6. $$\sum_{x \,\in\, \mathcal{SH}} \frac{\textsc{diam}(x)}{\textsc{norm}(x)} \;<\; 4n$$

7. $$\left|\left\{x \,\in\, \mathcal{SH} \;:\; \frac{\textsc{diam}(x)}{\textsc{norm}(x)} \;\geq\; k\right\}\right| \;<\; \frac{4n}{k}$$

8. $\mathcal{SH}$ is constructible in $O(m \log n)$ time.

Proof:
(1) The input graph may or may not be strongly connected. However, we will interpret the graph as being complete: any edges not appearing in the input implicitly have length ∞. Since we included ∞ as one of the normalizing lengths, there is some (possibly irrelevant) node x such that $\textsc{norm}(x) = \infty$ and $V(x) = V$.
(2) See Lemma 2.
(3) If $p(x)$ and x are in the same stratum, then clearly $\textsc{norm}(p(x))$ is a multiple of $\textsc{norm}(x)$. If $\textsc{norm}(p(x)) = \ell_{r_k} \cdot 2^i$, where $i \geq -1$, and x is not in stratum k, then $\textsc{diam}(x) < (|V(x)| - 1) \cdot 2\textsc{norm}(x) < n \cdot \ell_{r_k}/2n \leq \textsc{norm}(p(x))$.
(4) Every relevant $\mathcal{SH}$-node has at least two children. The sum counts every relevant $\mathcal{SH}$-node (except the root) exactly once.
(5) $V(x)$ is a strongly connected set, even when restricted to edges with length less than $2\textsc{norm}(x)$. Therefore, $\textsc{diam}(x) < |V(x) - 1| \cdot 2\textsc{norm}(x) < 2n \cdot \textsc{norm}(x)$.
(6) Let z^j denote the j^{th} ancestor of $z \in \mathcal{SH}$. Since the $\textsc{norm}$-value of a node is no more than half that of its parent (see Equation 3.1), we have $\textsc{norm}(z)/\textsc{norm}(z^j) \leq 2^{-j}$. We write z desc. x to mean z is a (not necessarily proper) descendant of x in $\mathcal{SH}$. Using the definition of $\textsc{diam}$ from Equation 3.2 we can bound the sum as follows.

$$\begin{aligned}
\sum_{x \in \mathcal{SH}} \frac{\text{DIAM}(x)}{\text{NORM}(x)} &= \sum_{x \in \mathcal{SH}} \frac{2\,\textsc{norm}(x) \cdot (\textsc{deg}(x) - 1) + \sum_{y \in \textsc{child}(x)} \text{DIAM}(y)}{\textsc{norm}(x)} \\
&= \sum_{x \in \mathcal{SH}} \sum_{z \text{ desc. } x} \frac{2\,\textsc{norm}(z) \cdot (\textsc{deg}(z) - 1)}{\textsc{norm}(x)} \\
&= \sum_{z \in \mathcal{SH}} \sum_{j \geq 0} \frac{2\,\textsc{norm}(z) \cdot (\textsc{deg}(z) - 1)}{\textsc{norm}(z^j)} \\
&< \sum_{z \in \mathcal{SH}} \sum_{j=0}^{\infty} \frac{\textsc{deg}(z) - 1}{2^{j-1}} \\
&= \sum_{z \in \mathcal{SH}} 4 \cdot (\textsc{deg}(z) - 1) \;<\; 4n
\end{aligned}$$

(7) Follows from Part 6.
(8) We construct $\mathcal{SH}$ using essentially the same algorithm found in [98]. The idea is to determine those nodes in the "middle" level of $\mathcal{SH}$, then find those nodes above the middle and below the middle recursively. As in [98] we use Tarjan's linear-time algorithm for finding SCCs. We first sort the edge-lengths and determine the $O(m \log n)$ possible $\textsc{norm}$-values in $O(m \log n)$ time. Let $\textsc{norm}_1 < \textsc{norm}_2 < \cdots < \textsc{norm}_k$ be the possible $\textsc{norm}$-values and G' be the input graph G restricted to edges with length less than

$2\textsc{norm}_{\lfloor k/2 \rfloor}$. We find the SCCs of G' in $O(m+n)$ time; let $\{C_i\}_i$ be the set of SCCs and G^c be derived from G by contracting the $\{C_i\}_i$ into single vertices. The $\{C_i\}_i$ correspond to $\mathcal{SH}$-nodes with NORM-values equal to $\textsc{norm}_{\lfloor k/2 \rfloor}$. We proceed recursively on the $\{C_i\}_i$ (finding $\mathcal{SH}$-nodes with NORM-values in the range $\textsc{norm}_1..\textsc{norm}_{\lfloor k/2 \rfloor -1}$) and on the graph G^c (for NORM-values in the range $\textsc{norm}_{\lfloor k/2 \rfloor +1}..\textsc{norm}_k$). There are $\log(m \log n) = O(\log n)$ levels of recursion and for each level the number of edges and vertices for subgraphs at that level is no more than m and $2n$, respectively. Therefore, the total time required is $O(m \log n)$.

□

3.3 A Generalized Hierarchy-Type Algorithm

The hierarchy-type algorithms are Dijkstra-like in the sense that they fix the distance of, or *visit*, vertices one by one, while maintaining Invariant 1. We generalize, somewhat, the notions of *visit* and *tentative distance* used in Dijkstra's algorithm. Recall that the D-value of a vertex is its tentative distance from the source. We define the D-value of an $\mathcal{SH}$-node as the minimum over its constituent vertices:

$$D(x) \overset{\textbf{def}}{=} \min_{v \in V(x)} \{D(v)\}, \quad \text{where } x \in SH$$

Note that the D-value of a leaf node is the same as its corresponding vertex.

We compute SSSP with a recursive algorithm called GENERALIZED-VISIT, given in Figure 3.2. Applied to a leaf-node of $\mathcal{SH}$, GENERALIZED-VISIT works just like the usual visit routine: it visits the leaf's associated vertex, and updates tentative distances to accord with Dijkstra's Invariant 1. However, GENERALIZED-VISIT can be used to solve any independent subproblem of SSSP. It takes two arguments: an $\mathcal{SH}$-node x and an interval I with the guarantee that $V(x)$ is (S, I)-independent, where S is the current set of visited vertices. Its only task is to visit the vertices in $V(x)^I$ and update the tentative distances, restoring Invariant 1. Using the GENERALIZED-VISIT procedure, we can compute SSSP from source s as follows. We set $S := \emptyset$, $D(s) := 0$, and $D(v) := \infty$ for all $v \neq s$, then call GENERALIZED-VISIT(ROOT, $[0, \infty)$), where ROOT = ROOT($\mathcal{SH}$). Invariant 1 is clearly satisfied w.r.t. $S = \emptyset$, and $V(\textsc{root}) = V$ is clearly $(\emptyset, [0, \infty))$-independent, so the input guarantees for the initial call to GENERALIZED-VISIT are met. After the call to GENERALIZED-VISIT(ROOT, $[0, \infty)$), Invariant 1 will hold w.r.t. $S \supseteq V(\textsc{root})^{[0,\infty)} = V$, implying $D(v) = d(s, v)$ for all $v \in S = V$.

In each call to VISIT there are two cases, depending on whether x is a leaf node or an internal node of $\mathcal{SH}$. Suppose x is a leaf and $V(x) = \{v\}$. Because we maintain Invariant 1, deciding whether $v \in V(x)^I$ is equivalent to deciding if $D(v) \in I$, which is simple to do. In the general case x is an internal node. We determine $V(x)^I$ by making a series of recursive calls to children of x, using subintervals of I of width $\textsc{norm}(x)$.

Generalized-Visit(x, $[a, b)$)

Specifications: It is assumed that $V(x)$ is $(S, [a, b))$-independent, where S is the set of visited vertices at the time of the call, and that Dijkstra's Invariant 1 is satisfied. Upon completion all vertices in $V(x)^{[a,b)}$ will have been visited.

1. If x is a leaf and $D(x) \in [a, b)$, then set $S := S \cup \{x\}$ and relax all of x's outgoing edges.

2. If Visit$(x, \cdot)$ is being called for the first time, assign intervals to x's buckets. Bucket i is labeled

$$[t_x + i \cdot \textsc{norm}(x),\ t_x + (i+1)\textsc{norm}(x))$$

where t_x is set to

$$t_x = \begin{cases} D(x) & \text{if } D(x) + \textsc{diam}(x) < b \\ b - \textsc{norm}(x) \left\lceil \frac{b - D(x)}{\textsc{norm}(x)} \right\rceil & \text{otherwise} \end{cases}$$

3. Set $a_x = \begin{cases} t_x & \text{if this is the first call to Visit}(x, \cdot) \\ a & \text{otherwise} \end{cases}$

```
While $a_x < b$ and $V(x) \not\subseteq S$
    While bucket $[a_x,\ a_x + \textsc{norm}(x))$ is not empty
        Let $y$ be the leftmost child of $x$ in bucket $[a_x,\ a_x + \textsc{norm}(x))$
        Visit$(y, [a_x,\ a_x + \textsc{norm}(x)))$
        Remove $y$ from its bucket
        If $V(y) \not\subseteq S$, put $y$ in bucket $[a_x + \textsc{norm}(x),\ a_x + 2\,\textsc{norm}(x))$
    $a_x := a_x + \textsc{norm}(x)$
```

Figure 3.2: A general divide-and-conquer algorithm for single-source shortest paths.

The crucial property of $\mathcal{SH}$ that we use is that the ordered set $\textsc{child}(x)$ represents a $\textsc{norm}(x)$-partition of $V(x)$ — see Lemma 3(2). Together with Lemma 1 we are able to guarantee that each recursive call represents an independent subproblem.

To bound the number of recursive calls, it is important not to make too many trivial ones, that is, calls which cause no vertex to be visited. To that end we associate with x an array of buckets that will contain the children of x. The buckets represent consecutive real intervals of width $\textsc{norm}(x)$ and the bucket array represents an interval spanning $[d(s,x), d(s,x) + \textsc{diam}(x)]$ where $d(s,x) = d(s, V(x))$ is the distance to any node in $V(x)$. When $\textsc{Generalized-Visit}(x, \cdot)$ is called for the first time we choose a suitable starting point t_x and label each bucket with its associated interval: the i^{th} bucket is assigned the interval $[t_x + i\textsc{norm}(x), t_x + (i+1)\textsc{norm}(x))$. We will choose t_x such that $t_x \leq d(s,x) < t_x + \textsc{norm}(x)$. Therefore, at most $\left\lceil \frac{\textsc{diam}(x)}{\textsc{norm}(x)} \right\rceil + 1$ buckets are required. For notational convenience we may refer to a bucket by its associated interval.

We will say x is *inactive* until $\textsc{Generalized-Visit}(x, \cdot)$ is called, and *active* afterward. We will assume, for the time being, that Invariant 2 is maintained.

Invariant 2 *(Bucket Invariant) Let x be an active $\mathcal{SH}$-node. A child y of x appears in one of x's buckets, unless $D(y) = \infty$ or $V(y) \subseteq S$, in which case y appears in no bucket. Every node y appearing in bucket $[q, q + \textsc{norm}(x))$ is either an inactive child such that $D(y) \in [q, q + \textsc{norm}(x))$, or an active child such that $V(y)^{[0,q)} \subseteq S$, but $V(y)^{[q,q+\textsc{norm}(x))} \not\subseteq S$.*

Suppose that in the call to $\textsc{Generalized-Visit}(x, I)$, I spans the intervals of k of x's buckets, say, buckets $b_{j+1}, b_{j+2}, \ldots, b_{j+k}$. $\textsc{Generalized-Visit}$ performs up to k iterations. In the i^{th} iteration it repeatedly locates the leftmost[4] child y of x in bucket b_{j+i}, performs a recursive call on y, whose interval argument is the same interval associated with b_{j+i}, then restores the Bucket Invariant 2. This involves either moving y to the next bucket if $V(y)$ is not yet contained in S, or removing y from the bucket array altogether if $V(y) \subseteq S$. If, after processing some bucket, $V(x) \subseteq S$, the current call to $\textsc{Generalized-Visit}(x, \cdot)$ halts. In the next section we prove the correctness of this algorithm. Many of the finer points in the analysis revolve around our choice of t_x in Step 2 of $\textsc{Generalized-Visit}$.

3.4 Correctness of Generalized-Visit

In this section we prove that $\textsc{Generalized-Visit}$ works correctly. Specifically, we show that $\textsc{Generalized-Visit}(x, I)$ visits (adds to the set S) all vertices in $V(x)^I$. We assume that Dijkstra's Invariant and the Bucket Invariant (1 and 2) are magically updated

[4]Recall that the set $\textsc{child}(x)$ has some left-to-right ordering.

behind the scenes. That is, adding a vertex to S causes the D-values of all vertices and $\mathcal{SH}$-nodes to be updated, restoring Dijkstra's Invariant, and causes some number of $\mathcal{SH}$-nodes to be moved to different buckets in accordance with the Bucket Invariant. In Section 3.5 we discuss the problem of efficiently implementing GENERALIZED-VISIT; off-the-shelf data structures and techniques seem inadequate. In Chapters 4 and 5 we develop shortest path algorithms for directed and undirected graphs, respectively, based on more sophisticated implementations of GENERALIZED-VISIT.

The following lemmas look at GENERALIZED-VISIT from the perspective of some $\mathcal{SH}$-node x. They assume implicitly that at the call GENERALIZED-VISIT(x, I), $V(x)$ is (S, I)-independent. They also assume that the initial call was GENERALIZED-VISIT(ROOT, $[0, \infty)$).

Lemma 4 *In any two calls* GENERALIZED-VISIT(x, I_1) *and* GENERALIZED-VISIT(x, I_2), $|I_1| = |I_2| = \text{NORM}(p(x))$.

Proof: All recursive calls on x are made from calls on $p(x)$. Moreover, all recursive calls from $p(x)$ have interval arguments of width $\text{NORM}(p(x))$.

□

Lemma 5 *If* GENERALIZED-VISIT(x, I) *is the first call to an* $\mathcal{SH}$*-node* x*, then we have* $D(x) = d(s, x) \in I$.

Proof: The lemma clearly holds for the initial call GENERALIZED-VISIT(ROOT, $[0, \infty)$), so consider the case when $x \neq \text{ROOT}$. Before the recursive call GENERALIZED-VISIT(x, I), x must have been in $p(x)$'s bucket spanning the interval I. Since x was inactive before the call, the Bucket Invariant 2 guarantees that $D(x) \in I$. Together with the assumption that $V(x)$ is (S, I)-independent we have the equality $D(x) = d(s, x)$.

□

Lemma 6 *Consider the variables* a_x *and* b *in any call to* GENERALIZED-VISIT$(x, [a, b))$. *Either* $\text{NORM}(x)$ *divides* $b - a_x$ *or* $V(x)^{[0,b)} = V(x)$.

Proof: In the first call to GENERALIZED-VISIT$(x, [a, b))$, a_x is set to t_x. Suppose that $t_x = D(x)$, because $D(x) + \text{DIAM}(x) < b$. By Lemma 5, $D(x) = d(s, x)$, implying that $V(x)^{[0,b)} = V(x)$. If, on the other hand, t_x is set to $b - \text{NORM}(x) \left\lceil \frac{b - D(x)}{\text{NORM}(x)} \right\rceil$, then $\text{NORM}(x)$ divides $b - t_x$ and, at least initially, $b - a_x$ as well. Since a_x is only incremented in units of $\text{NORM}(x)$, $b - a_x$ remains divisible by $\text{NORM}(x)$. We have proved the lemma for the *first* recursive call on x.

Now suppose that GENERALIZED-VISIT$(x, [a, b))$ is *not* the first recursive call on x, hence we set $a_x := a$ initially. According to Lemma 3(3) either $\text{NORM}(x)$ divides $\text{NORM}(p(x))$ or $\text{DIAM}(x) < \text{NORM}(p(x))$. Suppose $\text{NORM}(x)$ divides $\text{NORM}(p(x))$. By Lemma 4, $\text{NORM}(p(x)) = b - a$ and therefore $\text{NORM}(x)$ divides $b - a_x$ initially, and, with the observation that a_x is incremented in units of $\textsc{norm}(x)$, ever after. Now suppose $\textsc{diam}(x) < \textsc{norm}(p(x))$. Since this is not the first recursive call on x, we know, by Lemma 5, that $d(s,x) < a$ and therefore that $d(s,x) + \textsc{diam}(x) < b$, implying $V(x)^{[0,b)} = V(x)$.

□

Lemma 6 is a little technical. We use it to show that the intervals generated by a node and its parent are properly *aligned.* Consider $\mathcal{I}_1$, the set of intervals passed in recursive calls from $p(x)$ to x, and $\mathcal{I}_2$, the set of intervals passed from x to its children. We require that intervals in $\mathcal{I}_1$ and $\mathcal{I}_2$ have widths $\textsc{norm}(p(x))$ and $\textsc{norm}(x)$ respectively, and that they each cover the interval $[d(s,x),\ d(s,x) + \textsc{diam}(x)]$. Furthermore, each interval in $\mathcal{I}_2$ must be wholly contained in one interval from $\mathcal{I}_1$. Because we use a *stratified* hierarchy, $\textsc{norm}(p(x))$ is not necessarily a multiple of $\textsc{norm}(x)$. Therefore, these requirements can only be satisfied if $\textsc{diam}(x) < \textsc{norm}(p(x))$, i.e., if $\textsc{norm}(x)$ does not divide $\textsc{norm}(p(x))$ then it is impossible for $\mathcal{I}_1$ to contain more than two intervals. Our choice of t_x in Step 2 of Generalized-Visit is certainly not profound, but it does greatly simplify the algorithm's analysis and proof of correctness.

The following Lemma proves that Generalized-Visit works as advertised. We point out, since it may not be obvious on the first reading, that the proof of Lemma 7 is composed of three induction arguments. There is an induction over *time*, where we assume previous recursive calls behaved properly. There is an induction over *problem size*, where we assume certain future recursive calls behave properly, and finally, a double-induction over the two while-loops in Step 3 of Generalized-Visit, addressing the current recursive call.

Lemma 7 *After the call to* Generalized-Visit$(x,[a,b))$*,* $V(x)^{[a,b)} \subseteq S$*.*

Proof: We assume inductively that $V(x)$ is $(S,[a,b))$-independent when Generalized-Visit$(x,[a,b))$ is called. This clearly holds for the first recursive call, when $x = \textsc{root}$, $[a,b) = [0,\infty)$, and $S = \emptyset$.

Consider the case when x is a leaf in $\mathcal{SH}$, that is, a vertex. Generalized-Visit includes x in S precisely when $D(x) \in [a,b)$. According to the definition of independence $D(x) \in [a,b)$ implies $D(x) = d(s,x)$, so in this case the lemma is satisfied.

Suppose, now, that x is an internal node in $\mathcal{SH}$. We will assume, inductively, that each time through the outer while loop in Step 3 of Generalized-Visit, $V(x)^{[0,a_x)} \subseteq S$ and $V(x)$ is $(S,[a_x,b))$-independent w.r.t. the current values for a_x and S. Let us examine the base cases, concerning the first entry into the outer while loop. If a_x is set to t_x initially, then $a_x \leq D(x) = d(s,x)$, implying that $V(x)^{[0,a_x)} = \emptyset \subseteq S$. Furthermore, since $V(x)$ is $(S,[a,b))$-independent, it is $(S,[a_x,b))$-independent as well. The other case is when a_x is set to a on entry into the outer while loop. In this case $V(x)^{[0,a_x)} \subseteq S$ follows from our inductive assumption (w.r.t. the parent of x in $\mathcal{SH}$) and the $(S, [a, b))$-independence of $V(x)$ has already been assumed. Since a_x is incremented by precisely $\textsc{norm}(x)$ after each iteration of the outer while loop, to complete the induction we will show that the recursive calls in the inner while loop cause all vertices in $V(x)^{[a_x, a_x + \textsc{norm}(x))}$ to be visited.

Consider the entry into the inner while loop in Generalized-Visit, and let $I = [a_x, a_x + \textsc{norm}(x))$, that is, the current bucket is labeled I. Imagine that we consider each node in $\textsc{child}(x) = (x_j)_j$ in left-to-right order. We will show two things: first, that when x_j is considered $V(x_j)$ is (S, I)-independent for the current value of S. Therefore, *if* the recursive call Generalized-Visit(x_j, I) is made, we can assume inductively that it visits all vertices in $V(x_j)^I$. Second, if no recursive call is made on x_j (meaning x_j never appears in the bucket labeled I) then $V(x_j)^I - S = \emptyset$. This will establish the correctness of the inner while loop.

We claim that when x_j is considered $V(x_j)$ is (S, I)-independent. Let S' be the set S just before this iteration of the outer while loop, and assume inductively that when x_j is considered $S = S' \cup V(x_1)^I \cup \cdots \cup V(x_{j-1})^I$. Lemma 3(2) states that $(V(x_i))_i$ is a $\textsc{norm}(x)$-partition of $V(x)$. Together with the assumption that $V(x)$ is $(S', [a_x, b))$-independent and Lemma 1(1), we have that $V(x_j)$ is $(S, [a_x, \min\{a_x + \textsc{norm}(x),\, b\}))$-independent. However, we need to show that it is (S, I)-independent, since it is the interval $I = [a_x, a_x + \textsc{norm}(x))$ that would be passed to the recursive call. By Lemma 6, either $\textsc{norm}(x)$ divides $b - a_x$ or $V(x)^{[0,b)} = V(x)$. If $\textsc{norm}(x)$ divides $b - a_x$ then $I = [a_x, \min\{a_x + \textsc{norm}(x), b\})$ since we only entered the outer while loop if $a_x < b$, implying $a_x \le b - \textsc{norm}(x)$. On the other hand, if $V(x)^{[0,b)} = V(x)$, then $V(x_j)$ being $(S, [a_x, \min\{a_x + \textsc{norm}(x), b\}))$-independent implies that it is (S, I)-independent as well. To complete the induction we must show that *after* x_j is considered, $S = S' \cup V(x_1)^I \cup \cdots \cup V(x_j)^I$. If we perform the recursive call Generalized-Visit(x_j, I) then we can assume inductively that vertices in $V(x_j)^I$ are visited. Therefore, we must only prove that if no such recursive call is made, then $V(x_j)^I - S = \emptyset$. We perform recursive calls on all children that end up in bucket I. By Invariant 2, if x_j is not in bucket I when it is considered, then either $D(x_j) \ge a_x + \textsc{norm}(x)$ (implying $V(x_j)^I = \emptyset$) or $V(x_j) \subseteq S$; in either case $V(C_{x_j})^I - S = \emptyset$. This completes the induction for the inner and outer while loops.

The outer while loop in Step 3 terminates either because $a_x \ge b$ or $V(x) \subseteq S$, both of which imply $V(x)^{[0,b)} \subseteq S$. Therefore, after the call to Generalized-Visit$(x, [a, b))$, all vertices in $V(x)^{[a,b)}$ are visited. This establishes the lemma.

□

3.5 Implementation Details

An efficient implementation of the Generalized-Visit routine must solve two data structural problems, corresponding to Dijkstra's Invariant 1 and the Bucket Invariant 2. Whereas Dijkstra's algorithm only has to maintain the D-values (tentative distances) of vertices, which is trivial, we must maintain the D-values of hierarchy nodes as well, which is no longer trivial. The problem of maintaining the Bucket Invariant is not difficult, but maintaining (or simulating) it *efficiently* is quite tricky. Each of our shortest path algorithms uses a different technique for simulating the Bucket Invariant.

We first show that the costs of implementing Generalized-Visit are linear in the number of vertices, assuming Invariants 1 and 2 are maintained behind the scenes. We must account for two costs: that of performing some number of recursive calls, and that of computing t_x in Step 2, for all $x \in \mathcal{SH}$.

Lemma 8 *For each SSSP computation, the total number of recursive calls to* Generalized-Visit *is less than $5n$.*

Proof: By Lemma 5, if Generalized-Visit(x, I) is the first recursive call on x, then $D(x) = d(s, x) \in I$. Together with Invariant 2 and Lemma 4, this implies that each node $x \in \mathcal{SH}$ is passed to at most $\left\lceil \frac{\text{DIAM}(x)}{\text{NORM}(p(x))} \right\rceil + 1$ recursive calls, where $p(x)$ is the parent of x in $\mathcal{SH}$. The total number of recursive calls is then

$$\begin{aligned}
\sum_x \left\lceil \frac{\text{DIAM}(x)}{\text{NORM}(p(x))} \right\rceil + 1 \quad &\leq \quad |\mathcal{SH}| + \sum_x \left\lceil \frac{\text{DIAM}(x)}{2\text{NORM}(x)} \right\rceil && (3.3)\\
&< \quad |\mathcal{SH}| + n - 1 + \tfrac{1}{2} \cdot \sum_x \frac{\text{DIAM}(x)}{\text{NORM}(x)} && (3.4)\\
&< \quad 5n && (3.5)
\end{aligned}$$

Line 3.3 follows from the inequality $\text{NORM}(p(x)) \geq 2\text{NORM}(x)$. Line 3.4 follows since $\left\lceil \frac{\text{DIAM}(x)}{\text{NORM}(x)} \right\rceil$ is only strictly greater than $\frac{\text{DIAM}(x)}{\text{NORM}(x)}$ if x is an internal node of $\mathcal{SH}$, of which there are no more than $n-1$. (If x were a leaf, then $\text{DIAM}(x) = 0$.) Line 3.5 follows from the bounds $|\mathcal{SH}| < 2n$ and, by Lemma 3(6), $\sum_x \frac{\text{DIAM}(x)}{\text{NORM}(x)} < 4n$.
□

Lemma 9 *The total time required to find $\{t_x\}_{x \in \mathcal{SH}}$ is $O(n)$.*

Proof: In Step 2 of Generalized-Visit, t_x is set to $D(x)$ if $D(x) + \text{DIAM}(x) < b$ and $b - \text{NORM}(x) \left\lceil \frac{b - D(x)}{\text{NORM}(x)} \right\rceil$ otherwise. Checking whether $D(x) + \text{DIAM}(x) < b$ takes $O(1)$ time, and computing $b - \text{NORM}(x) \left\lceil \frac{b-D(x)}{\text{NORM}(x)} \right\rceil$ takes $O(\frac{b-D(x)}{\text{NORM}(x)})$ time: one simply counts back from b in units of $\textsc{norm}(x)$ in order to find $\min\{j \,:\, b - j \cdot \textsc{norm}(x) \leq D(x)\}$. Given that $b - D(x) \leq \textsc{diam}(x)$, the total time to find all $\{t_x\}_{x \in \mathcal{SH}}$ is $\sum_x O(\frac{\textsc{diam}(x)}{\textsc{norm}(x)})$, which is $O(n)$ by Lemma 3(6).

□

We support an implementation of Generalized-Visit with two abstract data structures, denoted $\mathcal{D}$ and $\mathcal{B}$. $\mathcal{D}$ updates the D-values of $\mathcal{SH}$-nodes as dictated by Invariant 1, and $\mathcal{B}$ maintains the bucket arrays of active $\mathcal{SH}$-nodes in accordance with Invariant 2. Although it is typical to assume that data structures do not talk to each other, it is conceptually simpler here to think of $\mathcal{D}$ and $\mathcal{B}$ making queries to each other. We describe their interactions below, then bound their complexity.

When an edge (u, v) is relaxed in Step 1 of Generalized-Visit, we tell $\mathcal{D}$ to set $D(v) := \min\{D(v), D(u) + \ell(u, v)\}$. If this decreases $D(v)$ then it may decrease the D-values of many ancestors of v in $\mathcal{SH}$ as well. Let y be the unique ancestor of v which is an inactive child of an active node. If $D(y)$ is also decreased then to restore Invariant 2 y may have to be moved to a different bucket. If this is the case then $\mathcal{D}$ notifies $\mathcal{B}$ that $D(y)$ has changed. $\mathcal{D}$ also accepts *queries* to D-values. In particular, when an $\mathcal{SH}$-node x becomes active $\mathcal{B}$ files each child y of x in its bucket array based on the value of $D(y)$. The bucketing structure $\mathcal{B}$ must also fulfill the needs of Generalized-Visit. Specifically, in a call to Generalized-Visit$(x, \cdot)$, Generalized-Visit repeatedly requests the leftmost child of x in the current bucket labeled $[a_x,\, a_x + \textsc{norm}(x))$, and possibly moves that node to the next bucket, labeled $[a_x + \textsc{norm}(x),\, a_x + 2\,\textsc{norm}(x))$. Lemmas 10 and 11 bound the complexities of $\mathcal{D}$ and $\mathcal{B}$, respectively.

Lemma 10 *$\mathcal{D}$ can be implemented to run in time $\Theta(\textsc{split-findmin}(m, n)) = O(m \log \alpha(m, n))$, where $\textsc{split-findmin}(m, n)$ is the decision-tree complexity of the split-findmin problem on m operations on an n-element sequence.*

We show below how the split-findmin data structure can be used to implement $\mathcal{D}$. The complexity bounds on split-findmin claimed in Lemma 10 are proved in Appendix A.

The split-findmin data structure operates on a collection of disjoint sequences of elements. Initially, there is one sequence containing all n elements, and each element has key ∞. The following operations are supported.

split(u) Splits the sequence containing u into two sequences, one consisting of those elements up to and including u, the other sequence taking the rest.

findmin(u) Returns the element in u's sequence with minimum key.

decrease-key(u, κ) sets $\text{key}(u) := \min\{\text{key}(u),\, \kappa\}$.

The elements in the split-findmin structure correspond to the leaves of $\mathcal{SH}$ and the keys correspond to D-values. Thus, edge relaxations can be implemented with decrease-key operations: if (u,v) is to be relaxed, we tell the split-findmin structure to decrease-key(v, $D(u)+\ell(u,v)$). The sequences in the split-findmin structure correspond to inactive $\mathcal{SH}$-nodes that are the children of active parents. One can readily verify that Generalized-Visit only queries the D-values of such nodes; thus, requesting $D(x)$ translates into the operation findmin(u), where u is any leaf in $V(x)$. Whenever a node x becomes active, we perform splits on the sequence representing x so that the resulting sub-sequences correspond to x's children. There are clearly no more than m decrease-keys and $O(m+n)$ splits and findmins. In Appendix A we show that the complexity of split-findmin on a RAM is asymptotically equivalent to its decision-tree complexity, which is $O(m \log \alpha(m,n))$.

Lemma 11 *Suppose $\mathcal{B}$ is assigned to maintain the bucket arrays of just those nodes in $X \subseteq \mathcal{SH}$. Then $\mathcal{B}$ can be implemented in time*

$$O\left(m + n\log\log n + \sum_{x\in X} \textsc{deg}(x)\cdot \log \frac{\textsc{diam}(C_x)}{\textsc{norm}(x)}\right)$$

Proof: Fix some $\mathcal{SH}$-node $x \in X$. The Bucket Invariant 2 says that all inactive children of x are bucketed by their D-values. However, in Generalized-Visit we only extract x's children from the "current" bucket, hence any structure that places the correct contents in the current bucket can be said to simulate Invariant 2. We use the hierarchical bucketing structure from Section 5.1.3 to simulate Invariant 2. The amortized cost of a decrease-key and an insert are, respectively, $O(1)$ and $O(\log \frac{\textsc{diam}(x)}{\textsc{norm}(x)})$, where $\frac{\textsc{diam}(x)}{\textsc{norm}(x)}$ represents the maximum number of buckets associated with x. This structure accounts for the first and third term in the claimed running time. The second term arises out of our need to enumerate the contents of the current bucket in left-to-right order. We use a van Emde Boas heap [203] to prioritize nodes in the current bucket. For any child of x the amortized cost of all van Emde Boas operations is $O(\log\log \textsc{deg}(x))$, which is $O(n\log\log n)$ over all $x \in X$ and all children of x.

□

Let us make a few observations. First, the $O(n\log\log n)$ term in the running time of Lemma 11 reflects the cost of sorting siblings in left-to-right order. However, by Lemma 2 all such orderings are equally good on undirected graphs. Therefore, no van Emde Boas heaps are used in the undirected version of Generalized-Visit. Moreover, the cost of van Emde Boas heaps can be ignored when analyzing the non-uniform complexity of shortest paths, since they are used to sort discrete data, not real data.

The third term in Lemma 11's running time is certainly the most interesting: The sum $\sum_{x \in X} \textsc{deg}(x) \log \frac{\textsc{diam}(x)}{\textsc{norm}(x)}$ can be thought of as a measure of the entropy of a specific hierarchy, under two strong assumptions: first, that each $y \in \textsc{child}(x)$ can appear in each of x's $\textsc{diam}(x)/\textsc{norm}(x)$ buckets with (more or less) equal probability, and second, that which bucket y appears in is independent of which buckets other nodes appear in. For $X = \mathcal{SH}$ it is fairly easy to force the time bound of Lemma 11 to be $\Omega(m + n \log n)$. To improve upon it, we must either derive a hierarchy with lower entropy (see Chapter 5) or circumvent the entropy lower bound by exploiting the dependencies among shortest paths.

Lemma 11 is more useful than it may first appear. For instance, if we let X be the set of hierarchy nodes with small normalized diameter, say all x with $\textsc{diam}(x)/\textsc{norm}(x) < (\log n)^{O(1)}$, then the bound from Lemma 11 is $O(m + n \log \log n)$. Thus, with low-diameter nodes being handled by Lemma 11, we are free to deal with high-diameter nodes by other means. This is exactly the strategy taken by the directed shortest path algorithm of Section 4.1.

3.6 Lower Bounds

In a comparison-based model of computation, the easiest way to lower bound the complexity of a problem is by a simple information-theoretic argument. In particular, the logarithm of the number of distinct solutions to the problem gives an immediate lower bound on the number of comparison operations required to solve it. Unfortunately, counting distinct solutions does not lead to any non-trivial lower bounds on the SSSP problem. Indeed, it seems quite plausible that there are no non-trivial lower bounds for SSSP. Nonetheless, it is still useful to lower bound the complexities of specific algorithms or approaches to SSSP. Such lower bounds can tell us *why* a certain algorithm or approach is doomed to be suboptimal, and, perhaps, how the bottleneck in such an approach could be overcome.

We lower bound the complexity of an algorithm in two steps. First, we characterize the *extra information* derived by running the algorithm. Second, we lower bound the complexity of computing that extra information from scratch. The robustness of this approach depends, of course, on how crucial the extra information is to the algorithm in question. Consider Dijkstra's algorithm. It computes, besides shortest paths, a permutation π_s of the vertices satisfying Property 1.

Property 1 *π_s satisfies:*

$$\textit{For all } u, v \in V, \quad \pi_s(u) < \pi_s(v) \implies d(s,u) \le d(s,v)$$

Any lower bound on the time to compute a π_s from scratch that satisfies Property 1 effectively lower bounds the complexity of Dijkstra's algorithm. The star graph in Figure

3.3, for instance, provides a very simple worst-case scenario for Dijkstra's algorithm. Visiting the vertices in order of distance necessarily involves sorting the edge lengths — that is, sorting $n-1$ arbitrary numbers.

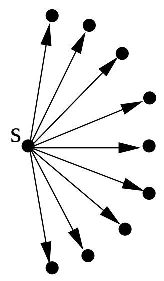


Figure 3.3: The star graph. If edge-lengths are permuted at random, finding a π_s satisfying Property 1 takes $\log((n-1)!) = \Omega(n \log n)$ comparisons.

One is tempted to say that this is a *weak* lower bound, because it can be circumvented by an algorithm that does not satisfy Property 1 but is, but any reasonable person's estimate, an implementation of Dijkstra's algorithm. The algorithm is, namely, to contract edges not on any cycle and run Dijkstra's algorithm on whatever is left.

The refutation to this argument is that the star graph is *not* claimed to be a hard instance of SSSP but the *kernel* of hard instances for Dijkstra's algorithm. Therefore, the lower bound applies not to one graph but any graph that has, embedded in it in some way, a small set of large star graphs. It is often the case that simple worst-case graphs translate into strong lower bounds and complicated ones into weaker lower bounds.

In this Section we give a characterization of all hierarchy-type algorithms that parallels Property 1's characterization of Dijkstra's algorithm. Using slightly more complicated hard kernel graphs than the star graph of Figure 3.3, we show that such algorithms cannot compute SSSP in $o(n \log n)$ time. This lower bound also holds for undirected graphs, though it can only be attained on unusually weighted graphs, where the ratio of the maximum to minimum edge-length is large.

3.6.1 Characterization of Hierarchy-Type Algorithms

The permutation π_s from Property 1 simply corresponds to the order in which vertices are visited in Dijkstra's algorithm. All Dijkstra-like algorithms (those maintaining Dijkstra's Invariant 1) can therefore be characterized by the restrictions placed on their allowable permutations. Property 2, given below, defines one such restriction that is intrinsic to all existing hierarchy-based algorithms. Before stating it we need some additional notation.

Let $\textsc{cycles}(u,v)$ be the set of all cycles, not necessarily simple, containing vertices u and v. For instance, on an undirected graph the cycle could follow a path from u to v then retrace its steps from v to u. We define $\textsc{sep}(u,v)$ as:

$$\textsc{sep}(u,v) \quad = \quad \min_{\mathcal{C} \,\in\, \textsc{cycles}(u,v)} \quad \max_{e \,\in\, \mathcal{C}} \quad \ell(e)$$

To see the connection between the $\textsc{sep}$-values and $\mathcal{SH}$, notice that $R_t(u,v) \equiv (\textsc{sep}(u,v) \leq t)$ is an equivalence relation, and that the equivalence classes of R_t correspond to the strongly connected components of the graph restricted to edges with length at most t. Moreover, as t varies R_t defines a set of laminar relations. That is, $R_t(u,v) \Rightarrow R_{t'}(u,v)$ if $t' > t$. Therefore, any set of relations $\{R_{t_i}\}_i$, can be represented by a rooted tree, or hierarchy.

Observation 1 gives us a cleaner interpretation of $\textsc{sep}$-values when the graph is undirected. Thorup [196] makes a similar observation, although he never uses the idea of a $\textsc{sep}$ function.

Observation 1 *If the graph is undirected,* $\textsc{sep}(u,v)$ *equals the length of the longest edge on the minimum spanning tree path connecting u and v.*

Regardless of whether the graph is undirected or directed, all hierarchy-based algorithms generate a permutation π_s satisfying Property 2, given below. We prove that Generalized-Visit satisfies Property 2 in Lemma 12.

Property 2 *If* $\textsc{sep}(u,v) > 0$ *then* π_s *satisfies:*

$$d(s,v) \geq d(s,u) + \textsc{sep}(u,v) \quad \Rightarrow \quad \pi_s(u) < \pi_s(v)$$

Is there a sorting bottleneck inherent in Property 2? The short answer is *yes*. However, the nature of the sorting bottleneck depends, to a large extent, on the little details. For instance, suppose we consider, besides m and n, a new parameter r representing a bound on the ratio of any two edge lengths. In Sections 3.6.2 and 3.6.3 we show that our lower bounds for directed and undirected graphs become, respectively, $\Omega(\min\{n \log n,\, n \log r\})$ and $\Omega(\min\{n \log n,\, n \log\log r\})$. In other words, to induce an $\Omega(n \log n)$ lower bound r must be exponential in n for undirected graphs, but only polynomial for directed ones. As we show in Chapter 5, both of these bounds are, somewhat surprisingly, tight.[5]

[5]Actually, the undirected bound is tight only if r is not in the vicinity of $\alpha(m,n)$, which is exceptionally small.

We show that undirected graphs are qualitatively easier in another respect. In Property 2, notice that the $\textsc{sep}(u,v)$ term is independent of the source s. From the perspective of an algorithm computing many shortest paths on the same graph,[6] computation relating to $\textsc{sep}$-values may be considered a *one-time* cost, whereas computing SSSP given the $\textsc{sep}$-values represents the *marginal* cost of computing SSSP.[7]. For directed graphs, we show that our lower bound holds even if all $\textsc{sep}$-values (and any functions thereof) are known a priori. This is in contrast to undirected graphs, where the only obstacle to computing SSSP in near-linear time is computing (or approximating) the $\textsc{sep}$ function.

	$\textsc{sep}$ known	$\textsc{sep}$ unknown
Undirected SSSP	$\Omega(\ m\ )$	$\Omega(\ m\ +\ \min\{\,n\log\log r,\ \ n\log n\,\}\ )$
Directed SSSP	$\Omega(\ m\ +\ \min\ \{\,n\log r,\ \ n\log n\,\}\ )$	

Figure 3.4: Lower bounds on SSSP algorithms satisfying Property 2 in the comparison-addition model. The parameter r bounds the ratio of any two non-zero edge lengths.

Lemma 12 $\textsc{Generalized-Visit}$ *generates a permutation of the vertices satisfying Property 2.*

Proof: The permutation named in the lemma is, of course, the order in which vertices are visited by $\textsc{Generalized-Visit}$. Let u, v be leaves of $\mathcal{SH}$ (i.e. graph vertices), let $x = LCA(u,v)$, and let u', v' be the children of x that are ancestors of u and v, respectively. By the definition of $\mathcal{SH}$, $\textsc{norm}(x) \leq \textsc{sep}(u,v)$. Now consider the recursive calls on u' and v' that caused u and v to be visited, say $\textsc{Generalized-Visit}(u', I_u)$ and $\textsc{Generalized-Visit}(v', I_v)$, where $|I_u| = |I_v| = \textsc{norm}(x)$. If $d(s,v) \geq d(s,u) + \textsc{sep}(u,v) \geq d(s,u) + \textsc{norm}(x)$ then $I_u \neq I_v$, implying $\textsc{Generalized-Visit}$ visits u before v.

□

We present our directed and undirected lower bounds in Sections 3.6.2 and 3.6.3, respectively. Figure 3.4 summarizes these results.

[6]As a concrete example, the website *MapQuest* claims to serve 10 million requests a day (many shortest path queries) on a graph (the US road network) that rarely changes.

[7]One may read “compute $\textsc{sep}$-values” as “compute $\mathcal{SH}$” or “compute a good hierarchy” since $\mathcal{SH}$ is just a very compact structure for representing (approximate) $\textsc{sep}$-values. In particular, if u, v are leaf-nodes in $\mathcal{SH}$ and $x = LCA(u,v)$ then $\textsc{sep}(u,v) \in [\textsc{norm}(x), 2\textsc{norm}(x))$.

3.6.2 Lower Bound: Directed Graphs

We will say that an SSSP algorithm satisfies Property 2 if, in addition to computing SSSP, it computes a permutation π_s satisfying Property 2. In this section we will also assume a slightly more powerful computation model. Besides comparisons, we will assume that any operation mapping tuples of reals to tuples of reals can be performed at unit cost.

Theorem 1 *Suppose* $\textsc{sep}(u,v)$ *is already known, for all vertices* u, v. *Any directed SSSP algorithm satisfying Property 2 performs* $\Omega(m + \min\{n \log r,\, n \log n\})$ *operations, where the source can be any of* $n - o(n)$ *vertices and* r *bounds the ratio of any two non-zero edge-lengths.*

Proof: Clearly every edge length must participate in at least one operation. This gives us the $\Omega(m)$ lower bound. The rest of the proof is devoted to showing that $\min\{n \log r, n \log n\}$ comparisons are required. In particular, we give a fixed graph (depending on n and r) and a set of possible edge-length functions $\mathcal{L}$. We show that any SSSP algorithm satisfying Property 2 must decide which length function was chosen, implying a lower bound of $\log |\mathcal{L}|$.

A permutation of the vertices is said to be *compatible* with a certain edge-length function if it satisfies Property 2.

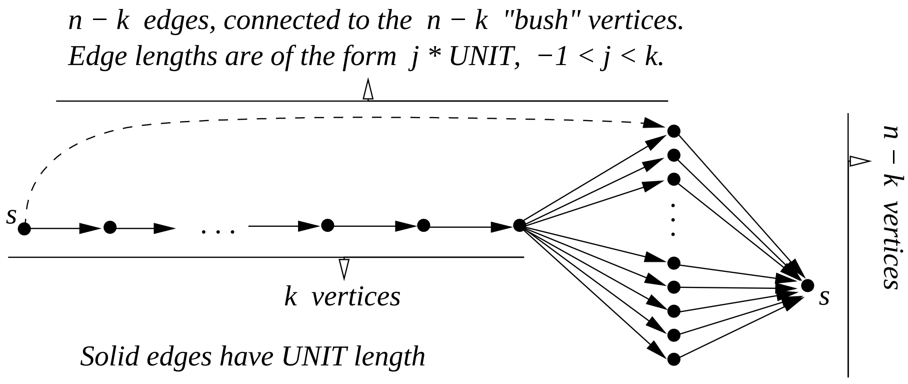

n − k edges, connected to the n − k "bush" vertices.
Edge lengths are of the form j * UNIT, −1 < j < k.
s
. . .
k vertices
Solid edges have UNIT length
s
n − k vertices

Figure 3.5: The "broom" graph.

Our fixed graph, depicted in Figure 3.5, is organized a little like a broom. It has a "broom stick" of $k \geq 2$ vertices, whose head is the source s and whose tail connects to the remaining $n - k$ vertices (the "bush"), each of which is connected back to s by an edge (s appears twice to simplify the figure). All these edges have equal length *UNIT*, which is an arbitrary positive real. Additionally, there are $n - k$ edges directed from s

to each of the vertices in the bush, having lengths of the form $j \cdot \mathit{UNIT}$, where j, chosen below, is a non-negative integer. One may easily confirm that $\textsc{sep}(u,v) = \mathit{UNIT}$ for all distinct u, v. (Our lower bound holds even if the SSSP algorithm is assumed to know this.) Assuming without loss of generality that k divides n, we define $\mathcal{L}$ to be the set of length functions that assign the edge length $j \cdot \text{UNIT}$ to exactly $(n-k)/k = n/k - 1$ edges from s to the "bush", for $0 \le j < k$. Consider the following claims:

1. For v in the "bush", $d(s,v) = \ell(s,v) < k \cdot \text{UNIT}$. (Recall that d and ℓ are the distance and length functions.)

2. $|\mathcal{L}| = (n-k)!/(\frac{n}{k}-1)!^k$ and $\log|\mathcal{L}| = \Omega(n \log k)$

3. For $\ell^1, \ell^2 \in \mathcal{L}$, there always exists u, v in the "bush" such that $d^1(s,u) < d^1(s,v)$ but $d^2(s,v) < d^2(s,u)$, where d^i is distance w.r.t. ℓ^i.

4. No permutation of the vertices can be compatible with two distinct length functions in $\mathcal{L}$.

(1) follows because the path from s to v along the "broomstick" has length $k \cdot \text{UNIT}$. (2) is simple counting. (3) follows from the pidgeonhole principle: because $\ell^1, \ell^2 \in \mathcal{L}$ assign each length to an equal number of edges, $d^1(s,u) < d^2(s,u)$ implies the existance of a v such that $\{d^1(s,u),\, d^2(s,v)\} < \{d^1(s,v),\, d^2(s,u)\}$. (4) follows from (3). To see this, notice that for any two vertices u, v, $d(s,u) < d(s,v)$ implies $d(s,u) \le d(s,v) + \text{UNIT} = d(s,v) + \textsc{sep}(u,v)$, which implies that if π_s is a compatible permutation, $\pi_s(u) < \pi_s(v)$. Along with (3) we can conclude that no two length functions in $\mathcal{L}$ are compatible with the same permutation. Therefore, at least $\log|\mathcal{L}| = \Omega(n \log k)$ comparisons are required to decide which $\ell \in \mathcal{L}$ is the actual length function.

The above argument can be repeated with little modification if the source vertex lies in the broom's bush. Together with the observation that $r = k - 1$, the Theorem follows.

$\square$

3.6.3 Lower Bound: Undirected Graphs

Theorem 2 *Any undirected single-source shortest path algorithm for real-weighted graphs satisfying Property 2 makes $\Omega(m + \min\{n \log\log r,\, n \log n\})$ operations in the worst case, where r bounds the ratio of any two non-zero edge lengths.*

Proof: The minimum spanning tree of the input graph is as depicted in Figure 3.6. It consists of the source vertex s which is connected to $p = (n-1)/2$ vertices in the top row, each of which is paired with one vertex in the bottom row. We divide the pairs into $q \ge 2$ disjoint *groups* and assign edge lengths based on group. Group i,

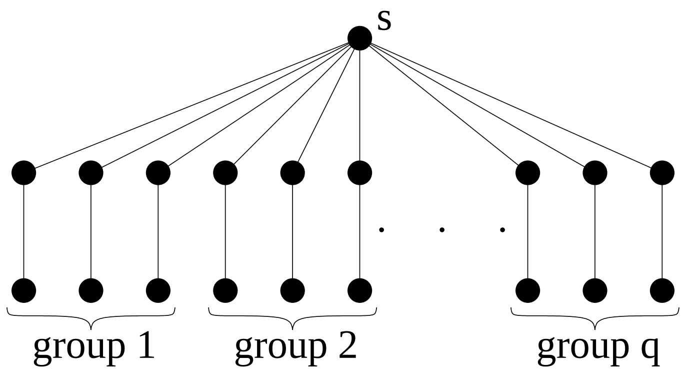


Figure 3.6: The minimum spanning tree of the graph

where $1 \le i \le q$, consists of exactly p/q pairs of vertices. Edges in group i have length $2^i \cdot$ UNIT, where UNIT is an arbitrary positive real. This includes edges connecting s to a top-row vertex and edges connecting the two rows. All non-MST edges are assigned any lengths less than $2^{O(q)} \cdot$ UNIT such that the shortest path tree from s coincides with the MST. Assuming, without loss of generality, that q divides p, the number of group arrangements is $p!/(p/q)!^q = q^{\Omega(p)}$. We will show that any SSSP algorithm satisfying Property 2 must sort the vertices by group number. Because the groups are of equal size, by the pidgeonhole principle no permutation of the vertices can be compatible with two distinct group arrangements. This implies a lower bound of $\Omega(p \log q)$ on such an SSSP algorithm. Since $\log r = \Theta(q)$, this also implies a bound of $\Omega(n \log\log r)$.

Let v_i denote some vertex in the bottom row of group i. Then $d(s, v_i) = 2 \cdot 2^i \cdot$ UNIT and $\textsc{sep}(v_i, v_j) = 2^{max\{i,j\}} \cdot$ UNIT. By Property 2, $\pi_s(v_i)$ must be less than $\pi_s(v_j)$ if $d(s, v_i) + \textsc{sep}(v_i, v_j) \le d(s, v_j)$. This is equivalent to $(2 \cdot 2^i + 2^j) \cdot \text{UNIT} \le 2 \cdot 2^j \cdot \text{UNIT}$, which holds precisely when $i < j$. Therefore, any SSSP algorithm satisfying Property 2 must sort the vertices by group number.

□

Remark. Note that in the proof of Theorem 2, we are essentially bounding the time to compute the SEP function (equivalently, the group arrangement), whereas in Theorem 1 we assume the SEP function is common knowledge.

Chapter 4

Shortest Paths on Directed Graphs

In Section 3.5 we showed that in order to implement GENERALIZED-VISIT, it suffices to solve certain abstract data structuring problems, all of which, save for $\mathcal{B}$, admit relatively simple near-linear time solutions. The primary focus of each of our shortest path algorithms is an efficient implementation of $\mathcal{B}$, the bucketing structure.[1]

The structure $\mathcal{B}$ is really just a restricted form of priority queue. Indeed, one obvious way to implement $\mathcal{B}$ is with an off-the-shelf data structure, such as a Fibonacci heap [73]. Unfortunately, any general data structure implementing $\mathcal{B}$ will invariably incur a sorting bottleneck. In order to implement $\mathcal{B}$ more efficiently it is crucial that we take into account the underlying graph. In particular, we must exploit the highly redundant nature of the distance function. After all, the distances, if represented explicitly, occupy $\Theta(n^2)$ space, whereas they are represented *implicitly* by the graph itself, which occupies just $\Theta(m)$ space.

The most straightforward correlations in the distance function are the pair-wise sibling correlations: for any $y, z \in \text{CHILD}(x)$, and any source vertex s, we have:

$$|d(s,y) - d(s,z)| \leq \text{DIAM}(x)$$

which is just a rephrasing of the parent-child correlation: $d(s,y) - d(s,x) \leq \text{DIAM}(x)$ for any $y \in \text{CHILD}(x)$. These correlations are trivial. One interpretation of Theorem 1 is that, in the worst case, there are essentially *no* non-trivial correlations, assuming a directed graph with fixed source vertex. As we will see in Chapter 5, undirected graphs

[1]The algorithms presented in this chapter were originally published as: S. Pettie, A faster all-pairs shortest path algorithm for real-weighted sparse graphs, Proc. 29th Int'l Colloq. on Automata, Languages, and Programming (ICALP), pp. 85–97, 2002, and S. Pettie, On the comparison-addition complexity of all-pairs shortest paths, Proc. 13th Int'l Symp. on Algorithms and Computation (ISAAC), pp. 32–43, 2002.

are an entirely different story, even when the source is fixed. In this Chapter we will study the correlations between elements of the set

$$\{\, d(s, y) \,\}_{s \,\in\, V,\, y \,\in\, \textsc{child}(x)}$$

In other words, we fix an $\mathcal{SH}$-node x and look at the sibling correlations among nodes in $\textsc{child}(x)$, ranging over all source vertices. Although the technical language we introduce in Sections 4.1 and 4.2 does not refer to sibling correlations and other intuitive ideas, correlation between distances is the principle that underlies our algorithms, and should always be kept in mind.

In Section 4.1 we give an APSP algorithm whose running time is $O(mn+n^2 \log\log n)$. The *running time* measure takes into account both real-number operations and data structural issues as well. In Section 4.2 we look at how far our techniques can be pushed if the only measure of efficiency is real-number operations. The result is a non-uniform APSP algorithm making $O(mn \log \alpha(m, n))$ comparison and addition operations.

4.1 A Faster APSP Algorithm

We have shown in Section 3.5 that an implementation of the Generalized-Visit algorithm amounts, essentially, to an implementation of $\mathcal{B}$, the bucketing structure. One might just as easily say that we have *reduced* Generalized-Visit to $\mathcal{B}$, and that the APSP problem is reducible to n runs of Generalized-Visit. We will show, in this section, that the problem of implementing $\mathcal{B}$ is itself reducible to a set of $O(n)$ SSSP problems. Each such problem is on a graph whose topology is basically the same as the original graph, but whose length function is source-dependent. This sequence of reductions does not seem profitable at first since APSP is trivially reducible to n SSSP computations on the original graph. However, not all SSSP problems are equal. Of our $O(n)$ derived SSSP problems, only $O(n/\log n)$ are on real-weighted graphs. The rest are on graphs whose lengths are relatively small *integers*. Because integer variables are not bound by the limitations of the comparison-addition model, we are able to solve these SSSP problems in amortized linear time.

In Section 4.1.1 we introduce the notions of relative distance and approximate relative distance. (These distances are the solutions to the derived SSSP problems mentioned above.) In Section 4.1.2 we show how approximate relative distances are useful in the implementation of Generalized-Visit, and in Section 4.1.3 we show how they can be computed cheaply.

4.1.1 Relative Distances and Their Approximations

Let x be an arbitrary internal $\mathcal{SH}$-node, and recall that $\textsc{child}(x)$ represents the children of x in $\mathcal{SH}$. For $y \in \textsc{child}(x)$ we let $\Delta_x(u, y)$ denote the *relative distance* from u to y, defined as:

$$\Delta_x(u,y) \overset{\textbf{def}}{=} d(u,y) - d(u,x)$$

Since $V(y) \subset V(x)$, it follows that Δ_x is always non-negative. Our algorithm does not deal with Δ_x directly but rather with a discrete approximation to it. We define $\hat{\Delta}_x$ as:

$$\hat{\Delta}_x(u,y) \overset{\textbf{def}}{=} \left\lfloor \frac{\Delta_x(u,y)}{\epsilon_x} \right\rfloor \quad \text{or} \quad \left\lceil \frac{\Delta_x(u,y)}{\epsilon_x} \right\rceil$$

where

$$\epsilon_x \overset{\textbf{def}}{=} \frac{\textsc{norm}(x)}{2}$$

It is crucial that $\hat{\Delta}_x$ be represented as an integer, *not* as a real. Lemma 13 and 14 capture the salient features of the $\hat{\Delta}$ function: that it is relatively cheap to compute, and that despite its approximate nature, it is useful in implementing the Generalized-Visit routine.

Lemma 13 *The $\hat{\Delta}_x$ function can be computed for every $\mathcal{SH}$ node x for which $\frac{\textsc{diam}(x)}{\textsc{norm}(x)} \geq \log n$, in $O(mn)$ time total.*

Lemma 14 *If $\hat{\Delta}_x$ is known for all $x \in \mathcal{SH}$ for which $\frac{\textsc{diam}(x)}{\textsc{norm}(x)} \geq \log n$, then we can compute SSSP in $O(m + n \log\log n)$ time using* Generalized-Visit.

Together with Lemma 3(8), stating that $\mathcal{SH}$ can be constructed in $O(m \log n)$ time, Lemmas 13 and 14 directly imply Theorem 3.

Theorem 3 *The all-pairs shortest path problem on real-weighted directed graphs can be solved in $O(mn + n^2 \log\log n)$ time, where the only operations allowed on reals are comparisons and additions.*

We prove Lemma 14 in Section 4.1.2. Lemma 13 is addressed in Section 4.1.3.

4.1.2 Generalized-Visit and Relative Distances

In this section we show how to implement the bucketing structure $\mathcal{B}$, assuming that $\hat{\Delta}_x$ is already computed for all $x \in \mathcal{SH}$ for which $\textsc{diam}(x)/\textsc{norm}(x) \geq \log n$. The remainder of this section will constitute a proof of Lemma 14. As it was observed in Section 3.5, managing the bucket arrays for all $\mathcal{SH}$-nodes x with $\textsc{diam}(x) \leq \log n \cdot \textsc{norm}(x)$ requires, by Lemma 11, only $O(m + n \log\log n)$ time. Therefore, we concentrate on an arbitrary $\mathcal{SH}$-node x for the case when $\hat{\Delta}_x$ is known.

We remarked earlier that maintaining the Bucket Invariant 2 is expensive. Consider the following weakened form of Invariant 2.

Invariant 3 *Suppose that y is a child of an active $\mathcal{SH}$-node x. Then y is either bucketed in accordance with Invariant 2, or it is known that $D(y)$ will decrease in the future, in which case y appears in no bucket.*

By Lemma 5, we only extract a node y from its bucket when $D(y)$ is finalized, that is, when $D(y) = d(s,y)$. Therefore, the correctness of GENERALIZED-VISIT w.r.t Invariant 2 implies its correctness w.r.t. Invariant 3. The only question is whether Invariant 3 is any easier to maintain, specifically, whether it is possible to tell if a node's D-value will decrease in the future. This is where the $\hat{\Delta}$ function comes into play.

Suppose that we are attempting to bucket an inactive node y by its D-value, either because its parent, x, just became active, or because we just relaxed an edge (u,v), where $v \in V(y)$. We know $d(s,x)$ lies in the interval of x's first bucket, that is, $t_x \le d(s,x) < t_x + \text{NORM}(x)$. According to Invariant 2, y belongs in bucket number

$$\left\lfloor \frac{D(y) - t_x}{\text{NORM}(x)} \right\rfloor \quad = \quad \left\lfloor \frac{D(y) - d(s,x)}{\text{NORM}(x)} \right\rfloor \quad \text{or} \quad \left\lfloor \frac{D(y) - d(s,x)}{\text{NORM}(x)} \right\rfloor + 1$$

Therefore, if $D(y)$ does not decrease in the future, then $D(y) = d(s,y)$ and $\Delta_x(s,y) = D(y) - d(s,x)$. This implies that y must be bucketed in either bucket number $\left\lfloor \frac{\Delta_x(s,y)}{\text{NORM}(x)} \right\rfloor$ or the following bucket. On the other hand, if $D(y)$ decreases in the future, we have, according to Invariant 3, the freedom not to bucket y at all.

The situation is made only slightly more complicated by the fact that we are not dealing with Δ_x but a discrete approximation to it. Recall that $\hat{\Delta}_x(s,y)$ is an integer and $|\epsilon_x \cdot \hat{\Delta}_x(s,y) - \Delta_x(s,y)| < \epsilon_x = \frac{\text{NORM}(x)}{2}$. Using the same argument as above, it follows that if $D(y) = d(s,y)$, that is, $D(y)$ will not decrease in the future, then y belongs in some bucket numbered in the interval

$$\begin{aligned} & \left[\left\lfloor \frac{\epsilon_x \cdot \hat{\Delta}_x(s,y) - \epsilon_x}{\text{NORM}(x)} \right\rfloor, \ \left\lfloor \frac{\epsilon_x \cdot \hat{\Delta}_x(s,y) + \epsilon_x + \text{NORM}(x)}{\text{NORM}(x)} \right\rfloor \right] \\ = \quad & \left[\left\lfloor \frac{(\hat{\Delta}_x(s,y) - 1)}{2} \right\rfloor, \ \left\lfloor \frac{(\hat{\Delta}_x(s,y) + 3)}{2} \right\rfloor \right] \end{aligned}$$

Thus, the number of eligible buckets is at most three. Since $\hat{\Delta}_x(s,y)$ is represented as an integer, we can identify the three eligible buckets in constant time, and, by checking $D(y)$ against the buckets' labels, we can determine which, if any, should contain y. To sum up, all insert and decrease-key operations on y take constant time, provided $\hat{\Delta}_x$ is known.

The other costs of implementing GENERALIZED-VISIT were discussed in Section 3.5. The $\mathcal{D}$ structure is implemented in $O(m \log \alpha(m,n)) = O(m + n \log\log n)$ time, and the cost of prioritizing nodes within the same bucket is $O(n \log\log n)$ using a van Emde Boas heap [203]. This concludes the proof of Lemma 14.

4.1.3 The Computation of $\hat{\Delta}$

We show in this section that for any $\mathcal{SH}$ node x, all $\hat{\Delta}_x(\cdot,\cdot)$-values can be computed in time $O(m \log n + m \cdot \textsc{deg}(x) + n \cdot \frac{\textsc{diam}(x)}{\textsc{norm}(x)})$. It turns out that this cost is affordable if the $m \log n$ term is not significantly larger than the others. It is for this reason that Lemma 13 only considers $\mathcal{SH}$ nodes x such that $\textsc{diam}(x)/\textsc{norm}(x) \geq \log n$.

Consider the two edge-labeling functions $\delta_x : E \to \mathbb{R}$ and $\hat{\delta}_x : E \to \mathbb{N}$, given below.

$$\begin{aligned}
\delta_x(u,v) &\overset{\textbf{def}}{=} \ell(u,v) + d(v,x) - d(u,x) \\
\hat{\delta}_x(u,v) &\overset{\textbf{def}}{=} \left\lfloor \frac{\delta_x(u,v)}{\epsilon'_x} \right\rfloor \quad \text{or} \quad \infty \quad \text{if} \quad \delta_x(u,v) > \textsc{diam}(x) \\
&\text{where } \epsilon'_x \overset{\textbf{def}}{=} \frac{\epsilon_x}{n} = \frac{\textsc{norm}(x)}{2n}
\end{aligned}$$

We let $G^\delta = (V(G), E(G), \delta)$ denote the graph G under a new length function δ, and let d^δ be the distance function for G^δ. We show that $\Delta_x(u,y)$ is equal to $d^{\delta_x}(u,y)$ and that $d^{\hat{\delta}_x}$ provides a sufficiently good approximation to Δ_x to satisfy the constraints put on $\hat{\Delta}_x$. Our method for computing $\hat{\Delta}_x$ is given in Figure 4.1. We spend the remainder of this section analyzing its complexity and proving its correctness.

```
Compute-$\hat{\Delta}_x$:

    (1)    Generate the graph $G^{\hat{\delta}_x}$

    (2)    For all $u \in V$ and $y \in \textsc{child}(x)$, compute $d^{\hat{\delta}_x}(u,y)$

    (3)    Set $\hat{\Delta}_x(u,y) := \left\lceil \frac{d^{\hat{\delta}_x}(u,y)}{n} \right\rceil$
```

Figure 4.1: A three-step method for computing $\hat{\Delta}_x$.

The following Lemma establishes the properties of Δ_x, δ_x, and $\hat{\delta}_x$ used in the analysis of Compute-$\hat{\Delta}_x$.

Lemma 15 *Suppose $x \in \mathcal{SH}$, $y \in \textsc{child}(x)$ and $u \in V$. Then*

1. $\Delta_x(u,y) = d^{\delta_x}(u,y)$
2. $d^{\delta_x}(u,y) \leq \textsc{diam}(x)$
3. $d^{\delta_x}(u,y) - \epsilon'_x \cdot d^{\hat{\delta}_x}(u,y) \in [0, \epsilon_x)$
4. $d^{\hat{\delta}_x}(u,y) < 2n\frac{\textsc{diam}(x)}{\textsc{norm}(x)}$

Proof: (1) Denote by $\langle u_1, u_2, \ldots, u_j\rangle$ a path from u_1 to u_j. Then

$$\begin{aligned}
d^{\delta_x}(u,y) &= \min_{j,\,\langle u=u_1,\ldots,u_j\in V(y)\rangle} \left\{\sum_{i=1}^{j-1} \delta_x(u_i,u_{i+1})\right\} && (4.1)\\
&= \min_{j,\,\langle u=u_1,\ldots,u_j\in V(y)\rangle} \Big\{\ell(\langle u_1,\ldots,u_j\rangle) + d(u_j,x) - d(u_1,x)\Big\} && (4.2)\\
&= d(u,y) - d(u,x) \;=\; \Delta_x(u,y) && (4.3)
\end{aligned}$$

Line 4.1 is simply the definition of d^{δ_x}. Line 4.2 is derived by cancelling terms in the telescoping sum. Note that $d(u_j,x)=0$ since $u_j \in V(y) \subseteq V(x)$, and that $d(u_1,x) = d(u,x)$. Line 4.3 then follows from the definition of d and Δ_x.

(2) From part (1) we have $d^{\delta_x}(u,y) = \Delta_x(u,y) = d(u,y) - d(u,x)$. The inequality $d(u,y) - d(u,x) \le \text{DIAM}(x)$ follows trivially from the fact that $V(y) \subset V(x)$.

(3) Let e be an arbitrary edge. By definition of δ_x and $\hat{\delta}_x$, we have that either $\delta_x(e) > \text{DIAM}(x)$ (i.e., $\hat{\delta}_x(e) = \infty$) or $\epsilon'_x \cdot \hat{\delta}_x(e) \le \delta_x(e) < \epsilon'_x \cdot (\hat{\delta}_x(e)+1)$. Let P_{uy} be the shortest path from u to y in G^{δ_x}, and denote by $|P_{uy}|$ the number of its edges. According to part (2), $d^{\delta_x}(u,y) \le \text{DIAM}(x)$, implying that for $e \in P_{uy}$, $\hat{\delta}_x(e) \ne \infty$, and

$$\epsilon'_x \cdot d^{\hat{\delta}_x}(u,y) \;\le\; d^{\delta_x}(u,y) \;<\; \epsilon'_x \cdot \left(d^{\hat{\delta}_x}(u,y) + |P_{uy}|\right) \;<\; \epsilon'_x \cdot d^{\hat{\delta}_x}(u,y) + \epsilon_x$$

The last inequality follows from the bound $|P_{uy}| < n$ and the definition of $\epsilon_x = n \cdot \epsilon'_x$. This proves part (3).

(4) From parts (2) and (3) we have

$$d^{\hat{\delta}_x}(u,y) \;\le\; \frac{d^{\delta_x}(u,y)}{\epsilon'_x} \;\le\; \frac{\text{DIAM}(x)}{\epsilon'_x} \;\le\; \frac{2n \cdot \text{DIAM}(x)}{\text{NORM}(x)}$$

which proves part (4).

□

Lemma 16 bounds the time to compute the $\hat{\delta}_x$ function in Step 1.

Lemma 16 *$G^{\hat{\delta}_x}$ is computable in $O(m \log n)$ time.*

Proof: Let (u,v) be an arbitrary edge. Recall that $\hat{\delta}_x(u,v)$ is either ∞ or:

$$\left\lfloor \frac{\ell(u,v) + d(v,x) - d(u,x)}{\epsilon'_x} \right\rfloor$$

The original length function ℓ is, of course, already known. We compute the other terms in the numerator with one Dijkstra computation. Let G_1 be derived from G by reversing the direction of all edges and contracting $V(x)$ into a single vertex. Computing SSSP from the source $V(x)$ in G_1 produces the $d(\cdot,x)$ distances. This takes $O(m + n\log n)$

time with Fibonacci heaps. However, we can afford to spend $O(m\log n)$ time using a simpler binary heap.

If $\delta_x(u,v) \le \textsc{diam}(x)$, which can be checked in constant time, then $\hat{\delta}_x(u,v)$ can be expressed as:

$$\hat{\delta}_x(u,v) = \max\{j \;:\; 2n \cdot d(u,x) + j \cdot \textsc{norm}(x) \le 2n \cdot (\ell(u,v) + d(v,x))\}$$

which follows from the definition of $\hat{\delta}_x$ and $\epsilon'_x = \textsc{norm}(x)/2n$. The terms $2n \cdot d(u,x)$ and $2n \cdot (\ell(u,v) + d(v,x))$ are easily computable in $O(\log n)$ time — see Section 2.4. We compute $\hat{\delta}_x(u,v)$ in $O(\log \frac{\textsc{diam}(x)}{\epsilon'_x}) = O(\log n)$ time by first generating the values

$$\left\{\textsc{norm}(x), 2\,\textsc{norm}(x), 4\,\textsc{norm}(x), \ldots, 2^{\left\lceil \log \frac{\textsc{diam}(x)}{\epsilon'_x} \right\rceil}\textsc{norm}(x)\right\}$$

using simple doubling, then using these values to perform a binary search to find the maximal j satisfying the inequality above. This binary search is performed once for each edge, taking $O(m\log n)$ time in total.

□

In Step 2 of Compute-$\hat{\Delta}_x$ we compute certain distances in the graph $G^{\hat{\delta}_x}$, using a variation on Dial's implementation of Dijkstra's algorithm. We are free to use Dial's algorithm here because $G^{\hat{\delta}_x}$ is an *integer*-weighted graph, whose shortest paths have bounded length.

Lemma 17 *Step 2 requires* $O(m \cdot \textsc{deg}(x) + n \cdot \frac{\textsc{diam}(x)}{\textsc{norm}(x)})$ *time.*

Proof: Let $y \in \textsc{child}(x)$ be a child of x and let N denote an upper bound on $d^{\hat{\delta}_x}(u,y)$. Let G_1 be the graph derived from $G^{\hat{\delta}_x}$ by reversing the direction of all edges in G. Clearly $d^{\hat{\delta}_x}(u,y)$ is equal to the distance from $V(y)$ to u in G_1. Therefore, we can perform Step 2 of Compute-$\hat{\Delta}_x$ by computing SSSP in G_1 from the source $V(y)$ (viewing it as a single vertex), for each $y \in \textsc{child}(x)$. To save time we solve each of these $\textsc{deg}(x)$ SSSP problems simultaneously, using Dial's implementation of Dijkstra's algorithm. The priority queue is implemented as a bucket array of length N. If the pair $\langle y,u\rangle$ appears in bucket b this indicates that in the SSSP computation with source $V(y)$, the tentative distance to u is b. Since $\hat{\delta}_x$ is an integer-valued function, edge relaxations take constant time. The overall running time is then $O(\#(\text{edge relaxations}) + \#(\text{buckets scanned})) = O(m \cdot \textsc{deg}(x) + N) = O\left(m \cdot \textsc{deg}(x) + n \cdot \frac{\textsc{diam}(x)}{\textsc{norm}(x)}\right)$. The bound on N follows from Lemma 15(4).

□

Lemmas 16 and 17 prove that Steps 1 and 2 take $O(m\log n + m\,\textsc{deg}(x) + n\,\frac{\textsc{diam}(x)}{\textsc{norm}(x)})$ time. Step 3 just involves dividing $d^{\hat{\delta}_x}(u,y)$ by n and rounding up. We did not assume a general integer division operation. However, Step 3 can easily be incorporated into

Step 2 by keeping track of the number $\lceil \frac{b}{n} \rceil$ where b is the current bucket number. In Lemma 18 we prove the correctness of COMPUTE-$\hat{\Delta}_x$.

Lemma 18 *Step 3 sets $\hat{\Delta}_x$ correctly, i.e.*

$$\hat{\Delta}_x(u,y) \text{ is an integer and } \quad \left| \epsilon_x \cdot \hat{\Delta}_x(u,y) \; - \; \Delta_x(u,y) \right| \; < \; \epsilon_x$$

Proof: It is clear from Step 3 that $\hat{\Delta}_x(u,y)$ is assigned an integer value. We turn to the second requirement, that $\left| \epsilon_x \cdot \hat{\Delta}_x(u,y) \; - \; \Delta_x(u,y) \right| \; < \; \epsilon_x$. Notice that $\frac{\epsilon'_x}{\epsilon_x} = \frac{1}{n}$. From the definition of the ceiling function we have:

$$\epsilon'_x \cdot d^{\hat{\delta}_x}(u,y) \quad \leq \quad \epsilon_x \cdot \left\lceil \frac{d^{\hat{\delta}_x}(u,y)}{n} \right\rceil \quad < \quad \epsilon'_x \cdot d^{\hat{\delta}_x}(u,y) + \epsilon_x \tag{4.4}$$

From Lemma 15 parts (1) and (3) we have that:

$$\epsilon'_x \cdot d^{\hat{\delta}_x}(u,y) \quad \leq \quad \Delta_x(u,y) \; = \; d^{\delta_x}(u,y) \quad < \quad \epsilon'_x \cdot d^{\hat{\delta}_x}(u,y) + \epsilon_x \tag{4.5}$$

Notice that in lines 4.4 and 4.5 the upper and lower bounds are identical, and that they are separated from each other by ϵ_x. Therefore,

$$\left| \epsilon_x \cdot \left\lceil \frac{d^{\hat{\delta}_x}(u,y)}{n} \right\rceil \; - \; \Delta_x(u,y) \right| \quad = \quad \left| \epsilon_x \cdot \hat{\Delta}_x(u,y) \; - \; \Delta_x(u,y) \right| \quad < \quad \epsilon_x$$

which proves the lemma.

□

Now that the correctness of this scheme is established, we are ready to prove the overall time bound of Lemma 13.

Proof: (Lemma 13) Let $T(m,n,k)$ be the time to compute $\hat{\Delta}_x$ for all $\mathcal{SH}$ nodes x for which $\frac{\text{DIAM}(x)}{\text{NORM}(x)} \geq k$. From Lemmas 16 and 17 we can bound T as follows.

$$\begin{aligned} T(m,n,k) \quad &= \quad \sum_{x \,:\, \frac{\text{DIAM}(x)}{\text{NORM}(x)} \geq k} O(m \log n + m\text{DEG}(x) + n\tfrac{\text{DIAM}(x)}{\text{NORM}(x)}) \\ &= \quad O(4mn\tfrac{\log n}{k} + 2mn + 4n^2) \qquad \{\text{Lemma 3(4), (6) \& (7)}\} \\ &= \quad O(mn \left\lceil \frac{\log n}{k} \right\rceil) \end{aligned}$$

hence $T(m,n,\log n) \; = \; O(mn)$

□

4.2 A Non-Uniform APSP Algorithm

The APSP algorithm from Section 4.1 has two distinct parts: a first pass for computing discrete, approximate distances and a subsequent pass for computing the exact distances. In this Section we show how to compute APSP with asymptotically fewer comparison and addition operations by basically running the two passes concurrently.

Our method for implementing the bucketing structure $\mathcal{B}$ is a hybrid of previous techniques. For every internal node $x \in SH$, we will simulate Invariant 2 with an *actual* bucket array and a heap, denoted H_x. The idea is to properly bucket nodes when we have enough information to do so (for instance, if we know the $\hat{\Delta}_x$-values) and to keep all unbucketed children of x in the heap H_x. When new information becomes available we may decide to migrate nodes from H_x to the bucket array. Consider the following bucketing invariant, which is weaker than both Invariants 2 and 3.

Invariant 4 *Let x be an active $\mathcal{SH}$-node. Active children of x appear in a bucket consistent with Invariant 2. An inactive node $y \in \text{CHILD}(x)$ either appears in a bucket numbered between $\left\lfloor \frac{d(s,y)-t_x}{\text{NORM}(x)} \right\rfloor - 2$ and $\left\lfloor \frac{D(y)-t_x}{\text{NORM}(x)} \right\rfloor$ inclusive, or in the heap H_x.*

We need to make a couple modifications to Generalized-Visit so that Invariant 4 can be said to simulate Invariant 2. Since Generalized-Visit only extracts nodes from the active bucket (the one labeled $[a_x, a_x + \text{NORM}(x))$ in Step 3 of Generalized-Visit), we will migrate the appropriate nodes from H_x to the active bucket, whenever the active bucket changes. Because of the conspicuous "-2" in Invariant 4 the active bucket may contain nodes that logically belong in later buckets. Whenever such a node is discovered (which can happen at most twice per node) we simply move it to the next bucket. One can easily see that under these modifications to Generalized-Visit, Invariant 4 simulates Invariant 2.

The simple method for maintaining Invariant 4 is to keep all inactive children of x in H_x. However, this sort of dependence on heaps leads inextricably to some kind of sorting bottleneck. The efficiency of our APSP algorithm depends on minimal use of the heaps.

In Section 4.2.2 we define functions Γ_x, $\hat{\Gamma}_x$, γ_x, and $\hat{\gamma}_x$ that closely parallel the functions Δ_x, $\hat{\Delta}_x$, δ_x, $\hat{\delta}_x$ from Sections 4.1.1 and 4.1.3. In Section 4.2.3 we show how the $\hat{\Gamma}_x$ and γ_x functions can be used to maintain Invariant 4 inexpensively.

4.2.1 Preliminaries

In our algorithm we use the phrase *is known* in a technical sense. The statement "it is known that $a < b$" means that the inequality $a < b$ could be inferred from the known set of linear inequalities, as revealed by previous comparison and addition operations. Similarly, "$\lfloor \frac{a}{b} \rfloor$ is known" means the integer $\lfloor \frac{a}{b} \rfloor$ could be inferred from previous operations, and "a is known", where a is a real, means a is actually stored in a specific real variable. As comparison-addition complexity is the only measure of interest in this section, we need not provide any method for deciding when something is known or not.

The sequence of operations performed by our algorithm is rather unpredictable. It depends, to a great extent, on what is known at a given time. We describe parts of our algorithm using *triggers*, which are of the form "Whenever some *(Precondition)* holds, perform some *(Action)*," where the *(Precondition)* typically depends on whether something is known. We assume that triggers are invoked at the earliest possible moment, and that for any two applicable triggers, the lower numbered one takes precedence. As a consequence of this policy, our high-level algorithm, Generalized-Visit, only proceeds if every trigger's precondition is unsatisfied.

4.2.2 Lengths, Distances, and Their Approximations

Define the edge-length function $\gamma_x : E \to \mathbb{R}$ as:

$$\gamma_x(u, v) \stackrel{\textbf{def}}{=} \ell(u, v) + w_x(v) - w_x(u)$$

where $w_x(v)$ and $w_x(u)$ are initially unspecified. Trigger 1 shows how $w_x(v)$ is assigned.

Trigger 1 *When the variable $w_x(u)$ is unspecified but $d(u, v)$ is known, for some $v \in V(x)$, set $w_x(u) := d(u, v)$.*

It follows from Trigger 1 that if $u \in V(x)$, $w_x(u) = 0$ holds initially since $d(u, u) = 0$ is known a priori. Note that if we set $w_x(\cdot) = d(\cdot, x)$ then γ_x would be identical to the δ_x function defined in Section 4.1.3.

We define the discrete approximation $\hat{\gamma}_x : E \to \mathbb{N}$ as:

$$\hat{\gamma}_x(u, v) \stackrel{\textbf{def}}{=} \left\lfloor \frac{\gamma_x(u, v)}{\rho_x} \right\rfloor \quad \text{or} \quad \infty \quad \text{if} \quad \gamma_x(u, v) > 2 \cdot \text{DIAM}(x)$$

where

$$\rho_x \stackrel{\textbf{def}}{=} \frac{\text{NORM}(x)}{4 \cdot \text{DEG}(x)}$$

Trigger 2, given below, updates the $\hat{\gamma}_x$ function whenever possible:

Trigger 2 *When $\gamma_x(u, v)$ is known but $\hat{\gamma}_x(u, v)$ is unknown, compute $\hat{\gamma}_x(u, v)$.*

Lemma 19 gives a couple properties of the γ_x and $\hat{\gamma}_x$ functions, and lets us bound the cost of Trigger 2.

Lemma 19 *Properties of $\hat{\gamma}_x$:*

1. *$\rho_x \cdot \hat{\gamma}_x(u,v) \in (-\textsc{diam}(x) - \rho_x,\, 2\textsc{diam}(x)] \cup \{\infty\}$. Moreover, if $\hat{\gamma}_x(u,v) = \infty$ then (u,v) is not on any shortest path from u to any vertex in $V(x)$.*

2. *The cost of computing $\hat{\gamma}_x(u,v)$ for all $x \in \mathcal{SH}$ and $(u,v) \in E$, is $O(mn)$.*

Proof: (1) By Trigger 1 we have $w_x(u) \in [d(u,x),\, d(u,x) + \textsc{diam}(x)]$. We also have that $\ell(u,v) + d(v,x) - d(u,x) \in [0,\, \infty)$, and furthermore, if (u,v) is on a shortest path to some vertex in $V(x)$, then $\ell(u,v) + d(v,x) - d(u,x) \in [0,\, \textsc{diam}(x)]$. Thus:

$$\begin{aligned}
\gamma_x(u,v) &= \ell(u,v) + w_x(v) - w_x(u) \\
&= \ell(u,v) + d(v,x) - d(u,x) + [-\textsc{diam}(x), \textsc{diam}(x)] \\
&= [-\textsc{diam}(x),\, \infty) \qquad\qquad \{\text{in general}\} \\
&= [-\textsc{diam}(x),\, 2 \cdot \textsc{diam}(x)] \qquad \{\text{if } (u,v) \text{ is relevant}\}
\end{aligned}$$

Therefore $\hat{\gamma}_x(u,v) = \infty$ only if (u,v) is not on any shortest path to a vertex in $V(x)$. Furthermore, if $\hat{\gamma}_x(u,v) \neq \infty$ then $\rho_x \cdot \hat{\gamma}_x(u,v) = \gamma_x(u,v) + (-\rho_x, 0] = (-\textsc{diam}(x) - \rho_x,\, 2\textsc{diam}(x)]$.

(2) Given $\gamma_x(u,v)$, we compute $\hat{\gamma}_x(u,v)$ using essentially the same algorithm from Lemma 16 in Section 4.1.3. It takes time logarithmic in the range, i.e.

$$\log\left(\frac{3 \cdot \textsc{diam}(x)}{\rho_x}\right) = \log\left(\frac{12 \cdot \textsc{deg}(x) \cdot \textsc{diam}(x)}{\textsc{norm}(x)}\right) \leq \textsc{deg}(x) + \frac{\textsc{diam}(x)}{\textsc{norm}(x)} + O(1)$$

By Lemma 3 parts (4) and (6), the cost of computing $\hat{\gamma}_x(e)$, for all $x \in \mathcal{SH}$ and $e \in E$, is $O(mn)$.

□

The γ_x and $\hat{\gamma}_x$ functions are clearly analogues of δ_x and $\hat{\delta}_x$ from Section 4.1.3. Below we define the functions $\Gamma_x, \hat{\Gamma}_x$, and $\tilde{\Gamma}_x$, where Γ_x is a real-valued function analogous to Δ_x and $\hat{\Gamma}_x$ and $\tilde{\Gamma}_x$ are certain integer-valued approximations of Γ_x.

$$\Gamma_x(u,y) = d(u,y) - w_x(u)$$

$$\tilde{\Gamma}_x(u,y) \overset{\textbf{def}}{=} \left\lfloor \frac{\Gamma_x(u,y)}{\rho_x} \right\rfloor$$

$$\rho_x \cdot \hat{\Gamma}_x(u,y) \overset{\textbf{def}}{=} \Gamma_x(u,y) - [0, \rho_x \cdot \textsc{deg}(x))$$

$\hat{\Gamma}_x$ is actually not completely defined. We use it to denote any integer-valued function satisfying the inequalities above.

The $\hat{\Gamma}_x$ function is the one we wish to compute. It is, however, a little too expensive to compute directly. Lemma 20, given below, shows how we might infer the $\hat{\Gamma}_x$ function by computing a few well-chosen $\tilde{\Gamma}_x$-values and the $\hat{\gamma}_x$ function.

Lemma 20 *Suppose $\langle v_1, \ldots, v_i, \ldots, v_j \in V(y)\rangle$ is known to be the shortest path from v_1 to $y \in \text{CHILD}(x)$, and suppose that $\tilde{\Gamma}_x(v_i, y)$ is known. If $i \leq \text{DEG}(x)$ then $\hat{\Gamma}_x(v_{i'}, y)$ is known as well, for $1 \leq i' \leq i$.*

Proof: Because $\langle v_1, \ldots, v_j\rangle$ is known to be a shortest path to $V(y) \subseteq V(x)$, it follows from Triggers 1 and 2 that the $\hat{\gamma}_x$-values are known for all edges in $\langle v_1, \ldots, v_j\rangle$. We claim that for $i' \leq i$, $\hat{\gamma}_x(\langle v_{i'}, \ldots, v_i\rangle) + \tilde{\Gamma}_x(v_i, y)$ is a good enough approximation to $\Gamma_x(v_{i'}, y)$ to satisfy the constraints put on $\hat{\Gamma}_x(v_{i'}, y)$. Note that in general, $\rho_x \cdot \hat{\gamma}_x(e) = \gamma_x(e) - [0, \rho_x)$ and $\rho_x \cdot \tilde{\Gamma}_x(u_i, y) = \Gamma_x(u_i, y) - [0, \rho_x)$. Therefore,

$$\begin{aligned} & \rho_x \left(\hat{\gamma}_x(\langle v_{i'}, \ldots, v_i\rangle) + \tilde{\Gamma}_x(v_i, y)\right) \\ = \;& \gamma_x(\langle v_{i'}, \ldots, v_i\rangle) + \Gamma_x(v_i, y) - [0, \rho_x \cdot (i - i' + 1)) \\ = \;& d(v_{i'}, v_i) + w_x(v_i) - w_x(v_{i'}) + d(v_i, y) - w_x(v_i) - [0, \rho_x \cdot \text{DEG}(x)) \\ = \;& \Gamma_x(v_{i'}, y) - [0, \rho_x \cdot \text{DEG}(x)) \quad = \quad \hat{\Gamma}_x(v_{i'}, y) \end{aligned}$$

□

Lemma 20 shows that we can infer a $\hat{\Gamma}_x$-value if a "nearby" $\tilde{\Gamma}_x$-value is already known. We will show that Trigger 3 computes a relatively small set of $\tilde{\Gamma}_x$-values at an affordable cost. Before giving Trigger 3 we have to introduce a little more notation. Let $\text{IN}(u)$ be the tree rooted at u of *known* shortest paths to u. Similarly, define $\text{OUT}(u)$ to be the known shortest paths out of u. (If u is an $\mathcal{SH}$-node then $\text{IN}(u)$ is actually an in-forest, whose roots are the vertices of $V(u)$.)

Trigger 3 *When the following hold: $y \in \text{CHILD}(x)$, $u \in \text{IN}(y)$, v is the nearest ancestor of u in $\text{IN}(y)$ for which $\tilde{\Gamma}_x(v, y)$ is known, and v is at (unweighted) distance at least $\text{DEG}(x)$ from u, we compute the value $\tilde{\Gamma}_x(w, y)$, where w is the ancestor of u at (unweighted) distance $\left\lfloor \frac{\text{DEG}(x)}{2} \right\rfloor$.*

Lemma 21 *Properties of $\hat{\Gamma}_x$:*

1. *If $u \in \text{IN}(y)$, where $y \in \text{CHILD}(x)$, then $\hat{\Gamma}_x(u, y)$ is known.*

2. *The cost of computing all $\tilde{\Gamma}$-values with Trigger 3 is $O(n^2)$.*

Proof: Trigger 3 ensures that every vertex in $\text{IN}(y)$ has an ancestor at distance at most $\text{DEG}(x) - 1$ (unweighted distance, that is) whose $\tilde{\Gamma}_x(\cdot, y)$-value is known. Part (1) then follows directly from Lemma 20. To prove Part (2) we first show that at most $3n/\text{DEG}(x)$ different $\tilde{\Gamma}_x(\cdot, y)$ values are ever computed by Trigger 3; we then bound the overall comparison-addition cost. When Trigger 3 is invoked we say u *claims* the edges between u and w. For the purpose of obtaining a contradiction, suppose an edge was claimed twice, say by u (with w) and subsequently by u' (with w'). Whether w' is an ancestor or descendant of w, the fact that u–w overlaps with u'–w' at one edge implies the (unweighted) length of u'–w is at most $2 \cdot \left\lfloor \frac{\text{DEG}(x)}{2} \right\rfloor - 1 < \text{DEG}(x)$. Therefore, Trigger 3 could not have been invoked at u', a contradiction, and consequently, at most $(n-1)/\lfloor \text{DEG}(x)/2 \rfloor < 3n/\text{DEG}(x)$ $\tilde{\Gamma}_x(\cdot, y)$-values were computed. The time required to compute a $\tilde{\Gamma}_x(\cdot, y)$-value is the same as a $\hat{\gamma}_x$-value: $O(\text{DEG}(x) + \frac{\text{DIAM}(x)}{\text{NORM}(x)})$ according to Lemma 19(2). Summing over all $x \in \mathcal{SH}, y \in \text{CHILD}(x)$, and $u \in V$, the total cost of Trigger 3 is:

$$\sum_x \text{DEG}(x) \cdot \frac{3n}{\text{DEG}(x)} \cdot \left(\text{DEG}(x) + \frac{\text{DIAM}(x)}{\text{NORM}(x)}\right) \;=\; O(n^2)$$

The $O(n^2)$ bound follows directly from Lemma 3 (4) and (6).
□

4.2.3 Buckets, Heaps, and Invariant 4

Recall that H_x is a heap associated with $x \in \mathcal{SH}$ that holds any unbucketed children of x. The main focus of this Section is how to keep nodes out of H_x while maintaining Invariant 4. We will analyze, in particular, Triggers 4, 5, and 6, given below.

Trigger 4 *Upon activation of x, for each $y \in \text{CHILD}(x)$, if possible, bucket y according to Invariant 4; otherwise put y in H_x.*

Trigger 5 *Whenever new $\hat{\gamma}_x$-values become known (Triggers 1 and 2) and x is active, for each $y \in \text{CHILD}(x)$, if possible, bucket y according to Invariant 4; otherwise keep y in H_x.*

Trigger 6 *Whenever $D(y)$ is decreased, where y is a bucketed child of x, if possible, keep y bucketed according to Invariant 4; otherwise, move y to H_x.*

We will clarify in due time what is meant by "if possible" in Triggers 4, 5, and 6. For the moment, let it suffice to say that successfully (or unsuccessfully) bucketing a node takes constant time. Therefore, each invocation of Triggers 4 and 5 takes $O(\text{DEG}(x))$ time and each invocation of Trigger 6 takes constant time. These times reflect some assumptions about the heap H_x. We assume, in particular, that heap inserts, decrease-keys, and find-mins take constant amortized time, and that deleting any subset of the heap takes $O(|H_x|) = O(\text{DEG}(x))$ time.[2]

The problem of bucketing y in constant time is that of finding a discrete approximation to the quantity $d(s, y) - d(s, x)$. Of course, since we do not know the shortest

[2]These are weak assumptions. For instance, H_x could be implemented as a singly linked list with a pointer pointing to the minimum element.

path from s–to–y, we have little certain information about $d(s, y)$. Our solution is to consider many hypothetically shortest s–y paths, and for each such path Q, estimate the quantity $\ell(Q) - d(s, x)$. In particular, we will examine all paths of the form $\langle P_h, P_b, P_t \rangle$, where P_h, the *head*, is a prefix of the known shortest path from s to x, P_t, the *tail*, is itself a known shortest path into y (and therefore part of $\textsc{in}(y)$), and P_b, the *bridge*, connects P_h to P_t — see Figure 4.2. If, in the actual shortest s–to–y path $P^* = \langle P_h^*, P_b^*, P_t^* \rangle$, the bridge P_b^* satisfies certain conditions, we show that y can always be bucketed in constant time.

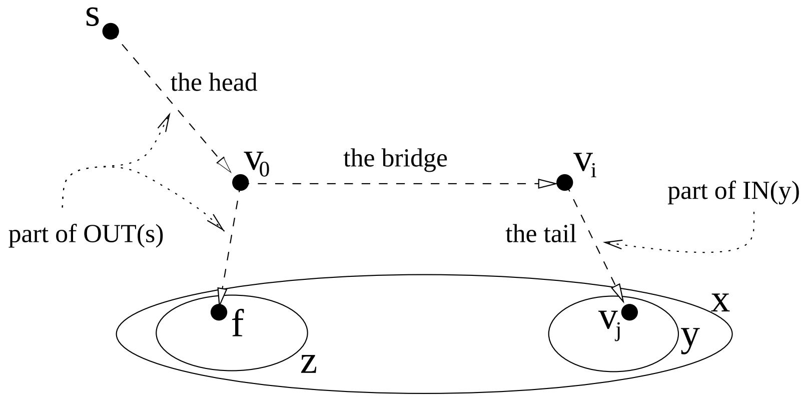


Figure 4.2: The path $\langle s, \ldots, v_j \rangle$, broken into a head $\langle s, \ldots, v_0 \rangle$, a bridge $\langle v_0, \ldots, v_i \rangle$, and a tail $\langle v_i, \ldots, v_j \rangle$.

When attempting to bucket y, we consider the paths in $\mathcal{Q}_y$ — see Definition 3. Paths in $\mathcal{Q}_y$ have no heads; they consist of a bridge and tail of a hypothetically shortest s–to–y path.

Definition 3 *Let $z \in \textsc{child}(x)$ and $f \in V(z) \subseteq V(x)$ be the vertex satisfying $d(s, f) = d(s, x)$. Let P_{sf} be the shortest path from s to f (and from s to x). We define $\mathcal{Q}_y$, where $y \in \textsc{child}(x)$, to be the set of paths of the form $\langle v_0, \ldots, v_i, \ldots, v_j \rangle$ that satisfy:*

1. $v_0 \in P_{sf} \subseteq \textsc{out}(s)$
2. $\hat{\gamma}_x(\langle v_0, \ldots, v_i \rangle)$ *is known*
3. $i \leq \textsc{deg}(x)$
4. $v_j \in V(y)$ *and* $\langle v_i, \ldots, v_j \rangle \subseteq \textsc{in}(y)$

We define the *integer* $\textsc{diff}(\mathcal{Q}_y)$ below. Under the assumption that $\mathcal{Q}_y$ contains a suffix of the shortest s–to–y path, we can place some interesting bounds on $\textsc{diff}(\mathcal{Q}_y)$ in terms of $d(s, y)$; however, in general $\textsc{diff}(\mathcal{Q}_y)$ might not approximate any useful quantity.

For $Q \in \mathcal{Q}_y$, where $Q = \langle v_0, \ldots, v_i, \ldots, v_j \rangle$ as in Definition 3, we define $\textsc{diff}(Q)$ and $\textsc{diff}(\mathcal{Q}_y)$ as:

$$\textsc{diff}(\mathcal{Q}_y) \overset{\textbf{def}}{=} \min_{Q \,\in\, \mathcal{Q}_y} \textsc{diff}(Q)$$

$$\textsc{diff}(Q) \overset{\textbf{def}}{=} \hat{\Gamma}_x(v_i, y) + \hat{\gamma}_x(\langle v_0, \ldots, v_i \rangle) - \hat{\Gamma}_x(v_0, z)$$

Lemma 22 $\textsc{diff}$ *has the following properties:*

1. $\textsc{diff}(\mathcal{Q}_y)$ *is an integer and its current value is known implicitly*
2. *At all times,* $\rho_x \cdot \textsc{diff}(\mathcal{Q}_y) > d(s,y) - d(s,x) - \frac{\textsc{norm}(x)}{2}$
3. *If some* $Q^* \in \mathcal{Q}_y$ *is a suffix of the shortest s–to–y path, then it holds that* $\rho_x \cdot \textsc{diff}(\mathcal{Q}_y) < d(s,y) - d(s,x) + \frac{\textsc{norm}(x)}{4}$

Proof: $\textsc{diff}(\mathcal{Q}_y)$ is an expression over integers, each of which is implicitly known according to Lemma 21(1) and Definition 3(2). This implies part (1). We turn to parts (2) and (3). Recall that by definition of $\hat{\Gamma}_x$ and $\hat{\gamma}_x$ we have the inequalities $\rho_x \cdot \hat{\Gamma}_x(u,y) = \Gamma_x(u,y) - [0,\, \rho_x \cdot \textsc{deg}(x))$ and $\rho_x \cdot \hat{\gamma}_x(u,v) = \gamma_x(u,v) - [0,\, \rho_x)$. Let $Q \in \mathcal{Q}_y$ be arbitrary, and, following the terms of Definition 3, we write Q as $\langle v_0, \ldots, v_i, \ldots, v_j \rangle$ and let $z \in \textsc{child}(x)$ be such that $d(s,z) = d(s,x)$. Let ξ be the interval $\left(-\frac{\textsc{norm}(x)}{2}, \frac{\textsc{norm}(x)}{4}\right)$.

$$\begin{aligned}
\rho_x \cdot \textsc{diff}(Q) &= \rho_x \cdot [\hat{\Gamma}_x(v_i, y) + \hat{\gamma}_x(\langle v_0, \ldots, v_i \rangle) - \hat{\Gamma}_x(v_0, z)] && (4.6)\\
&= \Gamma_x(v_i, y) + \gamma_x(\langle v_0, \ldots, v_i \rangle) - \Gamma_x(v_0, z) + \xi && (4.7)\\
&= d(v_i, y) + \ell(\langle v_0, \ldots, v_i \rangle) - w_x(v_0) - \Gamma_x(v_0, z) + \xi && (4.8)\\
&= \ell(Q) - d(v_0, x) + \xi && (4.9)
\end{aligned}$$

Line 4.6 is the definition of $\textsc{diff}$; Line 4.7 follows from the definitions of $\hat{\gamma}_x, \hat{\Gamma}_x$, and $\rho_x = \frac{\textsc{norm}(x)}{4 \cdot \textsc{deg}(x)}$, and Definition 3(3) stating that $i \leq \textsc{deg}(x)$. Line 4.8 is derived by expanding $\Gamma_x(v_i, y)$ and $\gamma_x(\langle v_0, \ldots, v_i \rangle)$ and cancelling terms. Line 4.9 follows from the definition of Γ_x and the identity $d(s,z) = d(s,x)$.

Consider Line 4.9. Clearly $\ell(Q) - d(v_0, x) = (d(s, v_0), +\ell(Q)) - (d(s, v_0) + d(v_0, x)) \geq d(s,y) - d(s,x)$, and that $\ell(Q) - d(v_0, x) = d(s,y) - d(s,x)$ only if Q is a suffix of a shortest s–to–y path. By taking into account the upper and lower bounds of $\xi = \left(-\frac{\textsc{norm}(x)}{2}, \frac{\textsc{norm}(x)}{4}\right)$, parts (2) and (3) immediately follow.
□

We use the $\textsc{diff}$-values to quickly decide if it is possible to bucket nodes in accordance with Invariant 4. Suppose that we are attempting to bucket a node $y \in \textsc{child}(x)$ due to either Trigger 4, 5, or 6. Our procedure is as follows:

1. Recall that x's first bucket spans the interval $[t_x,\, t_x + \textsc{norm}(x))$. Let $[\beta,\, \beta + \textsc{norm}(x))$ be the bucket in x's bucket array such that $t_x + \rho_x \cdot \textsc{diff}(\mathcal{Q}_y) \in [\beta,\, \beta + \textsc{norm}(x))$.
2. If $D(y) \geq \beta$, put y in bucket $[\beta,\, \beta + \textsc{norm}(x))$ and stop.
3. If $D(y) \geq \beta - \textsc{norm}(x)$, put y in bucket $[\beta - \textsc{norm}(x),\, \beta)$ and stop.
4. Otherwise, put or keep y in H_x.

Lemma 23 *The bucketing procedure does not violate Invariant 4 and if $\mathcal{Q}_y$ contains a suffix of a shortest s–to–y path, then y is successfully bucketed.*

Proof: Recall from Lemma 5 in Section 3.4 that t_x was chosen so that $d(s,x) \in [t_x, t_x + \textsc{norm}(x))$. Lines 2 and 3 of the bucketing procedure guarantee that y is never bucketed in a higher bucket than $\left\lfloor \frac{D(y)-t_x}{\textsc{norm}(x)} \right\rfloor$. To show that Invariant 4 is preserved, we need only prove that in Line 2, y is not bucketed before bucket $\left\lfloor \frac{d(s,y)-t_x}{\textsc{norm}(x)} \right\rfloor - 2$. Lemma 22(2) states that $\rho_x \cdot \textsc{diff}(\mathcal{Q}_y) > d(s,y) - d(s,x) - \frac{1}{2}\textsc{norm}(x)$, which implies that $\rho_x \cdot \textsc{diff}(\mathcal{Q}_y) > d(s,y) - t_x - \frac{3}{2}\textsc{norm}(x)$. So bucketing y according to $\rho_x \cdot \textsc{diff}(\mathcal{Q}_y)$ can put it at most $\lceil \frac{3}{2} \rceil = 2$ buckets before bucket $\left\lfloor \frac{d(s,y)-t_x}{\textsc{norm}(x)} \right\rfloor$, which is the slack tolerated by Invariant 4. For the second part of the Lemma, assume that some $Q \in \mathcal{Q}_y$ is a suffix of the shortest s–to–y path. It follows from Lemma 22(3) that $\rho_x \cdot \textsc{diff}(\mathcal{Q}_y) < d(s,y) - t_x + \frac{1}{4}\textsc{norm}(x)$. By choice of β, we have $\beta - t_x \leq \rho_x \cdot \textsc{diff}(\mathcal{Q}_y)$, which implies that $\beta - \frac{1}{4}\textsc{norm}(x) < d(s,y) \leq D(y)$ Therefore, y must have been bucketed in Step 2 or 3 or the bucketing procedure.

□

Lemma 24 *Suppose that we perform n SSSP computations with* Generalized-Visit. *Then the cost of all heap operations, including the cost of Triggers 4, 5, and 6, is $O(mn)$.*

Proof: Recall that attempting to bucket a node takes constant time, and that each invocation of Triggers 4, 5 take $O(\textsc{deg}(x))$ time, and that Trigger 6 takes constant time.

Trigger 4 is called once per $\mathcal{SH}$-node per SSSP computation. Thus the total cost for Trigger 4 is $\sum_{x \in \mathcal{SH}} O(\textsc{deg}(x)) \cdot n$, which is $O(n^2)$ by Lemma 3(4). Trigger 5 is invoked whenever new $\hat{\gamma}_x$-values become known (for any $x \in \mathcal{SH}$), which, by Triggers 1 and 2, means that for some vertex u, $w_x(u)$ was just fixed in Trigger 1. This can only happen n times (for x), for a total cost of $\sum_x O(\textsc{deg}(x)) \cdot n = O(n^2)$. Finally, Trigger 6 is called once per edge relaxation, of which there are no more than $O(mn)$.

We now account for the cost of extracting items from H_x Let $y \in \textsc{child}(x)$, and let P_{sy} and P_{sx} be the shortest paths from s–to–y and s–to–x, respectively. Now suppose that y is inserted into H_x. We can write P_{sy} as $\langle P_1, P_2, P_3\rangle$, where P_1 and P_3 are maximal such that $P_1 \subseteq P_{sx} \subseteq \textsc{out}(s)$ and $P_3 \subseteq \textsc{in}(y)$. By Lemma 23, y would have been bucketed (rather than inserted into H_x) if $\langle P_2, P_3\rangle \in \mathcal{Q}_y$. By Definition 3 $\langle P_2, P_3\rangle$ is not in $\mathcal{Q}_y$ either because (a) $|P_2| > \textsc{deg}(x)$ or (b) $w_x(u)$ is not known, for some $u \in P_2$. Case (a) can only happen $n/\textsc{deg}(x)$ times for y, because after the SSSP computation from source s, $\textsc{in}(y)$ will have absorbed P_2 (and P_1 for that matter). Thus the total cost for (a) is $\sum_x O(\textsc{deg}(x))^2 \cdot n/\textsc{deg}(x) = O(n^2)$. The cost of (b) has actually been accounted for, since once $w_x(u)$ is fixed, for all $u \in P_2$, y will be immediately bucketed by Trigger 5.

□

The only costs not covered by Lemma 24 are constructing the stratified hierarchy, which is $O(m \log n)$ by Lemma 3(8), computing the $\hat{\Gamma}$ and $\hat{\gamma}$ functions, which is $O(mn)$ by Lemmas 19(2) and 21(2) and implementing the $\mathcal{D}$ data structure, which, by Lemma 10, is $O(m \log \alpha(m,n))$ for each SSSP computation. Theorem 4 follows.

Theorem 4 *The all-pairs shortest path problem on arbitrarily-weighted, directed graphs can be solved with $O(mn \log \alpha(m,n))$ comparisons and additions, where m and n are the number of edges and vertices, respectively, and α is the inverse-Ackermann function.*

Chapter 5

Shortest Paths on Undirected Graphs

In this Chapter we give an implementation of Generalized-Visit for undirected graphs that is quantitatively and qualitatively superior to those algorithms for directed graphs presented in Chapter 4. Why are undirected graphs so much easier? The short answer is that undirected graphs can be effectively *clustered*, whereas directed graphs, in general, cannot. Consider a single edge (u, v). In an undirected graph we can claim that $|d(s,u) - d(s,v)| \leq \ell(u,v)$, regardless of the rest of the graph, whereas in a directed graph only the inequality $d(s,v) \leq d(s,u) + \ell(u,v)$ holds. Thus, the distance function for undirected graphs exhibits much stronger correlations.[1]

The particulars of our clustering scheme are a bit involved, though the overall idea is quite simple. Suppose that x is an $\mathcal{SH}$-node. Unless we know *something* about the input graph, the set $\{d(s,y) - d(s,x)\}_{y \in \text{CHILD}(x)}$ consists of more or less independent variables, each somewhere is the range $[0, \text{DIAM}(x))$. Therefore, *barring any extra information about the graph*, the set $\{\lfloor [d(s,y) - d(s,x)]/\text{NORM}(x) \rfloor\}_{y \in \text{CHILD}(x)}$ has about $\text{DEG}(x) \log(\text{DIAM}(x)/\text{NORM}(x))$ bits of information in it. In other words, we are imagining that the graph is chosen at random — though still consistent with the hierarchy $\mathcal{SH}$ — and asking about the entropy of certain variables. It is not difficult to show that the entropy of $\mathcal{SH}$ can be as much as $\Omega(n \log n)$. We show that by carefully introducing new layers of nodes into $\mathcal{SH}$, the overall entropy can be reduced to $O(n)$. Furthermore, we give a bucketing scheme (an implementation of the $\mathcal{B}$ structure) whose running time matches the entropy of the given hierarchy.

The running time of our algorithm is significantly more impressive than the algorithms from Chapter 4. The time required to compute a low-entropy hierarchy is

[1]The results of this chapter appeared in: S. Pettie and V. Ramachandran, Computing shortest paths with comparisons and additions, Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 267–276, 2002. The full version is under review.

only $O(m\alpha(m,n) + \min\{n \log n, n \log\log r\})$, where r bounds the ratio of any two edge-lengths. Once this hierarchy is given, we are able to compute SSSP from any source in $O(\text{SPLIT-FINDMIN}(m,n)) = O(m \log \alpha(m,n))$ time, which is nearly linear-time — perhaps even linear-time — and essentially unimprovable. As we will see in Chapter 6 the algorithm is streamlined and fares well in head-to-head comparisons with Dijkstra's algorithm. A stubborn bottleneck, both theoretically and practically, is the cost of computing a low-entropy hierarchy. Thus, for the problem of computing SSSP *exactly once*, our algorithm is only a theoretical improvement for reasonably-sized r. For instance, the asymptotic running time for $r = \text{poly}(n)$ is $O(m + n \log\log n)$.

5.1 An Undirected Shortest Path Algorithm

5.1.1 Refined Hierarchies

Let $\mathcal{H}_1$ and $\mathcal{H}_2$ be two hierarchies. We will say that $\mathcal{H}_2$ is a *refinement* of $\mathcal{H}_1$ if for every $x_1 \in \mathcal{H}_1$, there exists an $x_2 \in \mathcal{H}_2$ such that $V(x_1) = V(x_2)$ and $\text{NORM}(x_1) = \text{NORM}(x_2)$. Our undirected shortest path algorithm operates on a hierarchy called $\mathcal{RH}$, which is a refinement of $\mathcal{SH}$ having certain properties. We construct $\mathcal{RH}$ in Section 5.2.

We will exploit the correspondence between $\mathcal{SH}$-nodes and their counterparts in $\mathcal{RH}$. For instance, if x is known to be an $\mathcal{RH}$-node, the assertion that $x \in \mathcal{SH}$ is short for $\exists\, x' \in \mathcal{SH} : V(x) = V(x')$. The nodes in $\mathcal{RH} - \mathcal{SH}$ will be called *auxiliary*. Let $x \in \mathcal{SH}$ and let χ be the children of x in $\mathcal{SH}$. We define H_x to be the subtree of $\mathcal{RH}$ induced by x, χ, and all the auxiliary nodes between x and χ. For the moment we will only make two assumptions about H_x (and by extension $\mathcal{RH}$): that any auxiliary node $y \in H_x$ has at least two children (implying $|\mathcal{RH}| = O(n)$), and that $\text{NORM}(y) = \text{NORM}(x)$. It is easily shown that if $\mathcal{SH}$ satisfies Lemma 3 Parts (2) and (3) (the properties crucial for computing SSSP correctly) then $\mathcal{RH}$ satisfies these properties as well.

5.1.2 The Undirected-Visit Algorithm

Our shortest path algorithm for undirected graphs is given in Figure 5.1. It is nearly identical to the Generalized-Visit algorithm from Chapter 3, save for two small modifications. Since there is no distinction between *connected* and *strongly connected* components in undirected graphs, we can treat any t-partition as an unordered (rather than ordered) partition — see Lemma 2. In terms of the effect on our algorithm, rather than extracting the *leftmost* node from the current bucket, as we do in Generalized-Visit, we are free to extract any node in the current bucket.[2]

[2]Incidentally, this eliminates the need for the van Emde Boas heap [203] used in our implementation of Generalized-Visit.

```
UNDIRECTED-VISIT($x$, $[a, b)$)

    Input: $x \in \mathcal{SH}$ and $V(x)$ is $(S, [a,b))$-independent
    Output: All vertices in $V(x)^{[a,b)}$ are visited

  1. If $x$ is a leaf and $D(x) \in [a,b)$, then let $S := S \cup \{x\}$, relax all edges incident on
     $x$, restoring Invariant 1, and return.

  2. If UNDIRECTED-VISIT($x, \cdot$) is being called for the first time, create a bucket array
     of $\lceil \text{DIAM}(x)/\text{NORM}(x) \rceil + 1$ buckets. Bucket $i$ represents the interval

         $[t_x + i \cdot \text{NORM}(x),\ t_x + (i+1) \cdot \text{NORM}(x))$

     where $t_x$ is set to:

         $t_x = \begin{cases} D(x) & \text{if } D(x) + \text{DIAM}(x) < b \\ b - \left\lceil \frac{b - D(x)}{\text{NORM}(x)} \right\rceil \text{NORM}(x) & \text{otherwise} \end{cases}$

     Bucket the nodes in CHILD($x$) by their $D$-values

  3. Set $a_x = \begin{cases} t_x & \text{if this is the first call to VISIT}(x, \cdot) \\ a & \text{otherwise} \end{cases}$

     While $a_x < b$ and $V(x) \not\subseteq S$
         While bucket $[a_x,\ a_x + \text{NORM}(x))$ contains an auxiliary node $y$
             Remove $y$ from the bucket array
             Bucket the nodes in CHILD($y$)
         While bucket $[a_x,\ a_x + \text{NORM}(x))$ contains any node $y$
             UNDIRECTED-VISIT($y, [a_x,\ a_x + \text{NORM}(x))$)
             Remove $y$ from its bucket
             If $V(y) \not\subseteq S$, put $y$ in bucket $[a_x + \text{NORM}(x),\ a_x + 2\,\text{NORM}(x))$
         $a_x := a_x + \text{NORM}(x)$
```

Figure 5.1: The UNDIRECTED-VISIT procedure.

The UNDIRECTED-VISIT procedure is only called on $\mathcal{SH}$-nodes, never auxiliary nodes. Indeed, the pattern of recursive calls with UNDIRECTED-VISIT is identical to that of GENERALIZED-VISIT. We simply use the auxiliary nodes as representatives for multiple $\mathcal{SH}$-nodes in the bucket arrays. Specifically, we maintain that for any active $\mathcal{SH}$-node x, every leaf $y \in H_x$ that belongs in x's bucket array (according to Invariant 2) is represented in x's bucket array by some ancestor of y in H_x. Furthermore, if y is itself active, or if it belongs in the current bucket, then y is represented by itself. One can clearly see that UNDIRECTED-VISIT maintains this invariant. When x first becomes active, in Step 2, we bucket only x's children, a set that clearly represents the leaves of H_x. When a new bucket becomes the current bucket, in Step 3, we repeatedly replace an auxiliary node in the current bucket by its children, and proceed only after no auxiliary nodes remain. This bucketing regimen clearly simulates Invariant 2.

5.1.3 A Lazy Bucketing Structure

In this section we describe a simple abstract bucketing structure which is specially suited for use in UNDIRECTED-VISIT. However, it is still general enough to be used in other situations. The structure operates on an array of buckets and a set of elements with associated real-valued keys. The ith bucket represents a real interval I_i, which is adjacent to I_{i+1}, and an element with key κ belongs in the unique bucket i such that $\kappa \in I_i$. As a simplifying assumption, we assume that given i, I_i is computable in constant time. Buckets are either open or closed; only the contents of open buckets may change.
The Bucket-Heap:

create(f) Create a new Bucket-Heap, where $f(i) = I_i$ is constant time computable. All buckets are initially open.

insert(y, κ) Insert a new item y with key$(y) := \kappa$.

decrease-key(y, κ) Set key$(y) := \min\{\text{key}(y), \kappa\}$. It is guaranteed that y is not moved to a closed bucket.

close Close the first open bucket, and remove and enumerate its contents.

Lemma 25 *The Bucket-Heap can be implemented to run in time $O(N + \sum_y \log \Delta(y))$, where N is the total number of operations and $\Delta(y)$ is the number of* close *operations between y's insertion and its removal.*

Proof: Our bucketing structure simulates the logical specification given above; it actually consists of *levels* of bucket arrays. The level zero buckets are the ones referred to in the Bucket-Heap's specification, and the level i buckets preside over disjoint pairs of

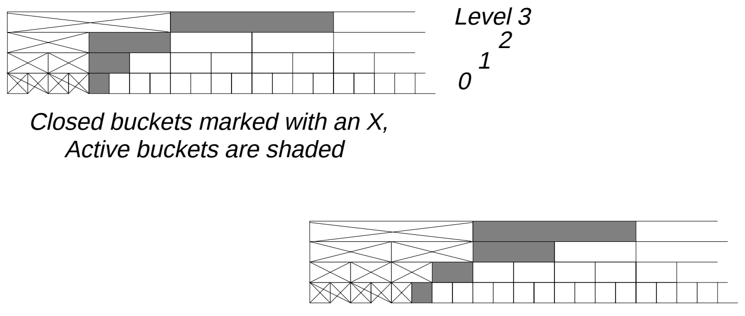

Level 3
2
1
0
Closed buckets marked with an X,
Active buckets are shaded
The effect of closing the first open bucket

Figure 5.2: Active buckets are shaded.

level $i-1$ buckets, and hence 2^i level zero buckets. The real interval associated with a higher-level bucket is the union of the intervals of its level zero buckets. Thus, it is computable is constant time with two calls to f.

Only one bucket at each level is *active*: it is the first one which presides over no closed level zero buckets. See Figure 5.2. Suppose that an item y should *logically* be in the level zero bucket B. We maintain the invariant that y is either *descending* and in the lowest active bucket presiding over B, or *ascending* and in some active bucket presiding over level zero buckets before B.

To insert a node we put it in the first open level zero bucket and label it ascending. This clearly satisfies the invariant. The result of a decrease-key depends on whether the node y is ascending or descending. Suppose y is ascending and in a bucket (at some level) spanning the interval $[a, b)$. If $\text{key}(y) < b$ we relabel it descending, otherwise we do nothing. If y is descending (or was just relabeled descending) we move it to the lowest-level active bucket consistent with the invariant. If y drops $j \geq 0$ levels we assume this is accomplished in $O(j+1)$ time, i.e., we search from its current level down, not from the bottom-up.

Suppose we close the first open level zero bucket B. According to the invariant all items that are logically in B are descending and actually in B, so enumerating them is no problem; there will, in general, be ascending items in B which do not logically belong there. In order to maintain the invariant we must deactivate all active buckets which preside over B (including B). Consider one such bucket at level i. If $i > 0$ we move each descending node in it to the new level $i-1$ active bucket. For each ascending node (at level $i \geq 0$), depending on its key we either move it to the new level $i+1$ active bucket and keep it ascending, or relabel it descending and move it to the proper active bucket at level $\leq i+1$.

It follows from the procedure above that no node y appears in a bucket at level greater than $\log \Delta(y) + 1$. Furthermore, every time a node is moved to a new bucket it either remains ascending and ascends in level, remains descending and descends, or makes the transition from ascending to descending. Therefore, the total cost of moving nodes between buckets (insert, decrease-key, and close operations) is $O(\sum_y \log \Delta(y))$. The other costs are clearly linear in the number of operations.

$\square$

Remark: Any implementation of the Bucket-Heap need not actually label the items. Whether an item is ascending or descending can be inferred from context.

5.1.4 Analysis of Undirected-Visit

In this section we bound the time required to compute SSSP with Undirected-Visit as a function of m, n, and the given hierarchy $\mathcal{RH}$. We will see later that the dominant term in this running time corresponds to the split-findmin structure.

Theorem 5 *Let $\mathcal{H}$ be any given refinement of $\mathcal{SH}$. Using $\mathcal{H}$, the SSSP problem can be solved in $O(\textsc{split-findmin}(m, n) + \phi(\mathcal{H}))$ time, where:*

$$\phi(\mathcal{H}) \overset{\mathbf{def}}{=} \sum_{z \in \mathcal{H}} \textsc{deg}(z) \cdot \log \frac{\textsc{diam}(z)}{\textsc{norm}(z)}$$

Proof: We compute SSSP with the Undirected-Visit procedure, using the same setup as the SSSP algorithm based on Generalized-Visit — see Chapter 3. The $\textsc{split-findmin}(m, n)$ term represents the time to implement the $\mathcal{D}$ data structure (see Section 3.5), which maintains Dijkstra's Invariant 1. We argue below that $\phi(\mathcal{H})$ represents the cost of implementing the abstract bucketing structure $\mathcal{B}$ with the Bucket-Heap from Section 5.1.3. As in our directed SSSP algorithm, the other costs (excluding $\mathcal{D}$ and $\mathcal{B}$) are $O(n)$. Refer to Lemmas 8 and 9.

Let $x \in \mathcal{SH}$, $z \in H_x$, and let y be a child of z. When y is inserted into x's bucket array, the current bucket must span an interval containing $d(s, z)$. (Either y was inserted immediately after z's removal – in Step 3 of Undirected-Visit – or $z = x$ and y was inserted in Step 2, just after the first call to Undirected-Visit$(x, \cdot)$.) Because $V(y) \subset V(z)$, we have the inequality $d(s, y) \leq d(s, z) + \textsc{diam}(z)$, which implies that y must be extracted from one of the next $\frac{\textsc{diam}(z)}{\textsc{norm}(x)} = \frac{\textsc{diam}(z)}{\textsc{norm}(z)}$ buckets. By Lemma 25 the amortized cost of inserting y is $O(\log \frac{\textsc{diam}(z)}{\textsc{norm}(z)})$, and over all $\mathcal{H}$-nodes the cost is $O(\sum_{z \in \mathcal{H}} \textsc{deg}(z) \cdot \log \frac{\textsc{diam}(z)}{\textsc{norm}(z)})$. By Lemma 25 the other costs of the Bucket-Heap are linear in the number of inserts, decrease-keys, and buckets, or $O(n + m + \sum_{x \in \mathcal{SH}} \frac{\textsc{diam}(x)}{\textsc{norm}(x)})$, which is $O(n + m)$ by Lemma 3(6). Since $O(n + m)$ is known to be dominated by $\textsc{split-findmin}(m, n)$, we conclude that SSSP are computed in time $O(\textsc{split-findmin}(m, n) + \phi(\mathcal{H}))$. This assumes, of course, that $\mathcal{H}$ is already available.

□

Our implementation of UNDIRECTED-VISIT is clearly not very complicated, certainly relative to the directed shortest path algorithms from Chapter 4. However, our overall SSSP algorithm is rather tricky because before the computation of shortest paths commences, a high-quality refinement of $\mathcal{SH}$ must first be found. In Section 5.2 we give a simple, but involved, method for computing a hierarchy $\mathcal{RH}$ such that $\phi(\mathcal{RH}) = O(n)$. It is unfortunate – but simply unavoidable – that the construction of $\mathcal{RH}$ requires superlinear time. Indeed, the lower bound in Theorem 2 is effectively a lower bound on computing $\mathcal{SH}$, and hence any refinement thereof.

5.2 An Algorithm for a Refined Hierarchy

Our algorithm for constructing $\mathcal{RH}$ has three distinct phases. In Phase 1 we compute the graph's minimum spanning tree (MST) and determine $\mathcal{SH}(MST(G))$, the stratified hierarchy for the MST. (Lemma 26, given shortly, justifies Phase 1 by showing that $\mathcal{SH}(MST(G))$ is basically the same as $\mathcal{SH}(G)$.) In Phase 2 we compute some compact structures that approximate certain subtrees of the MST. The results of Phase 2 are used to facilitate Phase 3, in which we compute a hierarchical clustering of the graph. This clustering leads directly to a refinement of $\mathcal{SH}$, denoted $\mathcal{RH}$, such that $\phi(\mathcal{RH}) = O(n)$. The specific goal of Phase 3 is to compute a clustering that is *balanced* in a certain sense. Ideally, we would like to guarantee that the *mass* of any cluster (a measure to be defined later) is never much smaller than the diameter of its parent cluster.

5.2.1 Phase 1: Computing the MST and $\mathcal{SH}$

Recall that the result of Phase 1 is the hierarchy $\mathcal{SH}(MST(G))$. In Lemmas 26 and 27 we establish the relevance of $\mathcal{SH}(MST(G))$ — it is just as useful as $\mathcal{SH}$ — and the time required to compute it.

Lemma 26 $\mathcal{SH}(MST(G))$ *satisfies the properties of Lemma 3.*

Remark. Recall that in our proof of correctness of GENERALIZED-VISIT and UNDIRECTED-VISIT, we assumed only that $\mathcal{SH}$satisfied Lemma 3.
Remark. Note that if $\mathcal{SH}(MST(G))$ were identical to $\mathcal{SH}(G)$ then we would have nothing to prove. Indeed, in [196] Thorup noted that his hierarchy on G was the same as that on $MST(G)$. However, in our case we cannot claim an identity because of the peculiar way NORM-values are chosen.

We now prove Lemma 26:
Proof: Lemma 3 Parts (1),(4), and (8) hold by construction, and Parts (3), (5), (6), and (7) hold because the diameter of a subgraph w.r.t. G is a lower bound on its

diameter w.r.t. $MST(G)$, or indeed any subgraph of G. To prove Part (2), we must show that for any $x \in \mathcal{SH}(MST(G))$ with $\textsc{child}(x) = (x_1, \ldots, x_{\textsc{deg}(x)})$, the fact that $(V(x_1), \ldots, V_{x_{\textsc{deg}(x)}})$ is a $\textsc{norm}(x)$-partition of $V(x)$ w.r.t. $MST(G)$ implies that it is also a $\textsc{norm}(x)$-partition w.r.t. G. By construction $\textsc{norm}(p(x)) \geq \textsc{norm}(x)$; therefore, any edge in $MST(G)$ crossing the cut $(V(x_i), V - V(x_i))$, for $x_i \in \textsc{child}(x)$, has length at least $\textsc{norm}(x)$. By the *cut property* of MSTs (see Chapter 7), some minimum-length edge crossing any cut is in the MST. Hence $\textsc{child}(x)$ represents a $\textsc{norm}(x)$-partition of $V(x)$ w.r.t. G as well.

□

Lemma 27 *For an undirected graph G, a hierarchy $\mathcal{SH}$ satisfying Lemma 3 is computable in time $O(\textsc{mst}^*(m, n) + \min\{n \log\log r,\ n \log n\})$, where m and n are the number of edges and vertices, respectively, $\textsc{mst}^*$ is the decision-tree complexity of the minimum spanning tree problem, and r bounds the ratio of any two non-zero edge lengths.*

Proof: We first compute $MST(G)$, in $\textsc{mst}^*(m, n)$ time (see Chapter 8), then proceed to determine $\mathcal{SH}(MST(G))$. By Lemma 26 $\mathcal{SH}(MST(G))$ will satisfy Lemma 3. The algorithm for computing stratified hierarchies, given in the proof of Lemma 3(8), takes $O(m \log n)$ time, which is $O(n \log n)$ in our case since $|MST(G)| \leq n - 1$. When $r < 2^n$, which can be confirmed in $O(n)$ time, the algorithm from Lemma 3(8) is easily modified to run in time $O(m \log\log r)$, or in our case, $O(n \log\log r)$. We select the faster of the two algorithms for computing $\mathcal{SH}$, which leads to an overall running time of $O(\textsc{mst}^*(m, n) + \min\{n \log\log r,\ n \log n\})$.

□

In Section 5.2.2 we introduce some new notation and concepts used in the construction of $\mathcal{RH}$. Phases 2 and 3 of the construction are given in Sections 5.2.3 and 5.2.4, respectively.

5.2.2 Definitions and Properties

Minimum Spanning Tree Notation

Let M denote the minimum spanning tree of G, which is w.l.o.g. assumed to be connected. If $X \subseteq V$ is a set of vertices, we define $M(X)$ to be the minimal subtree of M connecting X. In Phases 2 and 3 we will perform depth-first-search traversals of M and other trees. Therefore, it is convenient to think of M as being rooted at an arbitrary vertex. In any oriented tree or subtree T we let $\textsc{root}(T)$ be the root vertex of T. For instance, $\textsc{root}(M(X))$ would be the least common ancestor in M of all vertices in X. If v is a vertex in any oriented tree, $p(v)$ denotes the parent of v.

Definition of MASS

Let $X \subseteq V$ be an arbitrary set of vertices. We define $\textsc{mass}(X)$ as the total length of $M(X)$, i.e.,

$$\textsc{mass}(X) \overset{\textbf{def}}{=} \sum_{e \in M(X)} \ell(e)$$

We generally argue about the mass of hierarchy nodes, or sets of such nodes. To simplify things a bit, let $\textsc{mass}(x)$ be short for $\textsc{mass}(V(x))$, and $\textsc{mass}(\{x, y, \ldots, z\})$ be short for $\textsc{mass}(V(x) \cup V(y) \cup \cdots \cup V(z))$, where $x, y, \ldots, z$ are nodes in some hierarchy. Similarly, let $M(x)$ be short for $M(V(x))$.

We summarize, in Observation 2, what little can be said of about MASS in the absence of any specifics about the underlying graph.

Observation 2 *The* MASS *function has the following properties:*

1. *If* $X_1 \cap X_2 = \emptyset$ *then* $\textsc{mass}(X_1 \cup X_2) \geq \textsc{mass}(X_1) + \textsc{mass}(X_2)$.
2. *If* $X_1 \cap X_2 \neq \emptyset$ *then* $\textsc{mass}(X_1 \cup X_2) \leq \textsc{mass}(X_1) + \textsc{mass}(X_2)$.
3. $\textsc{mass}(X)$ *is an upper bound on the diameter of X*

The point of parts (1) and (2) is to show that before manipulating $\textsc{mass}(X_1)$ and $\textsc{mass}(X_2)$ algebraically, something should be known about X_1 and X_2.

Lambda Values

In Section 5.2.2 we will enumerate the properties satisfied by $\mathcal{RH}$. Those properties refer to the following sequence of λ-values:

$$\begin{aligned} \lambda_0 &= 0 \\ \lambda_1 &= 12 \\ \lambda_{j+1} &= 2^{(\lambda_j / 2^j)} \end{aligned}$$

Lemma 28 gives a lower bound on the growth of the λ-values.

Lemma 28 $\min\{j : \lambda_j \geq n\} \leq 2\log^* n$

Proof: Let $\mathcal{S}_j$ be a stack of j twos; for example, $\mathcal{S}_3 = 2^{2^2} = 16$. We will prove that $\lambda_j \geq \mathcal{S}_{\lfloor j/2 \rfloor}$, giving the lemma. One can verify that this statement holds for $j \leq 9$. Assume that it holds for all $j' \leq j$.

$$\begin{aligned} \lambda_{j+1} &= 2^{2^{\lambda_{j-1}\cdot 2^{-(j-1)}}2^{-j}} && \{\text{Definition of } \lambda_{j+1}\} \\ &\geq 2^{2^{\mathcal{S}_{\lfloor (j-1)/2\rfloor}\cdot 2^{-(j-1)}-j}} && \{\text{Inductive Assumption}\} \\ &\geq 2^{2^{\mathcal{S}_{\lfloor (j-1)/2\rfloor -1}}} = \mathcal{S}_{\lfloor (j+1)/2\rfloor} && \{\text{Holds for } j \geq 9\} \end{aligned}$$

The third line follows from the inequality $\mathcal{S}_{\lfloor (j-1)/2\rfloor} \cdot 2^{-(j-1)} - j \geq \mathcal{S}_{\lfloor (j-1)/2\rfloor -1}$, which holds for $j \geq 9$.

□

Properties of $\mathcal{RH}$

Recall that we derive the hierarchy $\mathcal{RH}$ by composing $\mathcal{SH}$ with the set of trees $\{H_x\}_{x\in\mathcal{SH}}$ where H_x is some tree rooted at x, whose leaves are the children of x in $\mathcal{SH}$. This is just a verbose way of saying $\mathcal{RH}$ is a refinement of $\mathcal{SH}$. In this section we state some properties satisfied by $\mathcal{RH}$.

Our discussion focusses mainly on some arbitrary H_x tree, where $x \in \mathcal{SH}$. We give all nodes in H_x a non-negative integer *rank*. In the ideal situation, every rank j node $y \in H_x$ would satisfy $\text{MASS}(y)/\text{NORM}(x) = \lambda_j$. Unfortunately, we are unable to place any non-trivial upper or lower bounds on $\text{MASS}(y)$. We assign ranks to satisfy Property 3, given below, which ensures us a sufficiently good approximation to the ideal situation.

We should point out that the assignment of ranks is entirely for the purpose of analysis. Rank information is never stored explicitly in the hierarchy nodes, nor is rank information used, implicitly or explicitly, in the computation of shortest paths. We only refer to ranks in the construction of of $\mathcal{RH}$ and in the analysis of $\phi(\mathcal{RH})$.

Property 3 *Let $x \in \mathcal{SH}$ and $y, z \in H_x$.*

1. *If y is an internal node of H_x then $\text{NORM}(y) = \text{NORM}(x)$ and $\text{DEG}(y) > 1$.*
2. *If y is a leaf of H_x (i.e., a child of x in $\mathcal{SH}$) then y has rank j, where j is maximal s.t. $\text{MASS}(y)/\text{NORM}(x) \geq \lambda_j$.*
3. *Let y be a child of a rank j node. We call y stunted if $\text{MASS}(y)/\text{NORM}(x) < \lambda_{j-1}/2$. Each node has at most one stunted child.*
4. *Let y be of rank j. The children of y can be divided into three sets: Y_1, Y_2, and a singleton $\{z\}$ such that $(\text{MASS}(Y_1) + \text{MASS}(Y_2))/\text{NORM}(x) < (2 + o(1)) \cdot \lambda_j$.*
5. *Let H_x^j be the nodes of H_x of rank j. Then $\sum_{y\in H_x^j} \text{MASS}(y) \leq 2 \cdot \text{MASS}(x)$.*

Before moving on let us examine some features of Property 3. Part (1) is restating the properties of $\mathcal{RH}$ assumed earlier. Part (2) shows how we set the rank of leaves of H_x. Part (3) says that at most one child of any node is less than half its ideal mass. Part (4) is a little technical but basically says that for a rank j node y, although $\textsc{mass}(y)$ may be very large relative to $\lambda_j\textsc{norm}(x)$ (the ideal mass), the children of y can be divided into sets $Y_1, Y_2, \{z\}$ such that Y_1 and Y_2 have mass at least close to the ideal. However, no bound is placed on the mass contributed by z. Part (5) says that if we restrict our attention to the nodes of a particular rank in H_x, their total mass is roughly bounded by the total mass of H_x.

The purpose of Properties 3(1)–(5) is to let us bound the running time of our SSSP algorithm, under the hierarchy $\mathcal{RH}$. Lemma 29 claims that $\phi(\mathcal{RH})$ is linear. Recall that according to Theorem 5 ϕ corresponds to the cost of bucketing nodes.

Lemma 29 $\phi(\mathcal{RH}) = O(n)$

Proof: By Observation 2(3) $\textsc{mass}(x)$ is an upper bound on the diameter of $V(x)$. We will freely substitute references to $\textsc{diam}(x)$ with $\textsc{mass}(x)$; for instance, ϕ could now be written as:

$$\phi(\mathcal{RH}) = \sum_{y\in\mathcal{RH}} \textsc{deg}(y)\cdot\log\frac{\textsc{mass}(y)}{\textsc{norm}(y)}$$

Suppose y were a non-leaf rank j node in $H_x \subseteq \mathcal{RH}$. Using the terms from Property 3(4), let $\alpha = (\textsc{mass}(Y_1)+\textsc{mass}(Y_2))/\textsc{norm}(x)$ and $\beta = \textsc{mass}(y)/\textsc{norm}(x) - \alpha$. Property 3(4) states that $\alpha < (2+o(1))\cdot\lambda_j$ and Property 3(3,4) implies that $\textsc{deg}(y) \le 2\alpha/\lambda_{j-1}+2$, where the $+2$ represents the stunted child and the child z exempted from Property 3(4). We now bound the term of $\phi(\mathcal{RH})$ associated with y:

$$\begin{aligned}
\textsc{deg}(y)\log\frac{\textsc{mass}(y)}{\textsc{norm}(y)} &\le \left(\frac{2\alpha}{\lambda_{j-1}}+2\right)\log(\alpha+\beta) \qquad \{\text{See explanations below}\}\\
&= O\left(\frac{\max\{\alpha\log(2\lambda_j),\beta\}}{\lambda_{j-1}}\right)\\
&= O\left(\frac{\alpha+\beta}{2^{j-1}}\right) = O\left(\frac{\textsc{mass}(y)}{\textsc{norm}(y)\cdot 2^{j-1}}\right)
\end{aligned}$$

The first line follows from our bound on $\textsc{deg}(y)$ and the definition of α and β. The second line follows since $\alpha < (2+o(1))\lambda_j$, and $\alpha\log(\alpha+\beta) = O(\max\{\alpha\log\alpha,\beta\})$. The last line follows since $\log\lambda_j = \lambda_{j-1}/2^{j-1} > 1$. By the above bound and Property 3(5), $\sum_{y\in H_x}\textsc{deg}(y)\log(\textsc{mass}(y)/\textsc{norm}(y)) = O(\textsc{mass}(x)/\textsc{norm}(x))$. Thus $\phi(\mathcal{RH}) = \sum_{x\in\mathcal{SH}} O(\textsc{mass}(x)/\textsc{norm}(x))$, which is $O(n)$ by Lemma 3(6). (Note that for undirected graphs, if $x \in \mathcal{SH}$ then $\textsc{mass}(x) = \Theta(\textsc{diam}(x))$, where $\textsc{diam}(x)$ is calculated using Equation 3.2 from Section 3.2.)

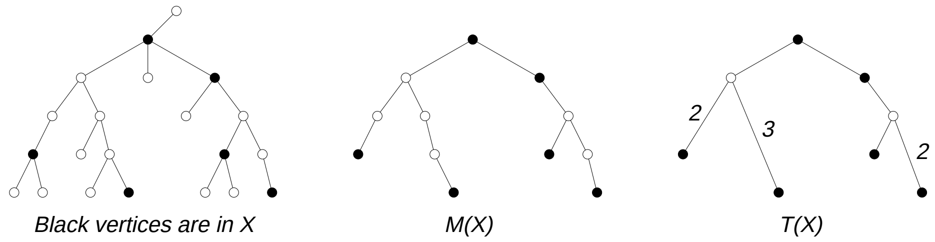


Figure 5.3: On the left is a subtree of M, the MST, where X is the set of blackened vertices. In the center is $M(X)$, the minimal subtree of M connecting X, and on the right is T_x, derived from $M(X)$ by splicing out unblackened degree 2 nodes in $M(X)$ and adjusting edge lengths appropriately. Unless otherwise marked, all edges are of length 1.

$\square$

Theorem 5 and Lemma 29 imply that our SSSP algorithm based on UNDIRECTED-VISIT runs in $O(\text{SPLIT-FINDMIN}(m, n))$ time, given the refined hierarchy $\mathcal{RH}$. In Sections 5.2.3 and 5.2.4 we address the problem of computing $\mathcal{RH}$.

5.2.3 Phase 2: Computing Shortcut Trees

It is entirely possible to compute an H_x satisfying Property 3, given only the subtree $M(x)$. However, the overall running time of such an algorithm would be unacceptable, since, in general, $|M(x)|$ can be much larger than $|H_x|$. Recall that the leaves of H_x are the children of x in $\mathcal{SH}$, thus, $|H_x| = \Theta(\text{DEG}(x))$. In Phase 2 we compute a tree called T_x that has roughly $\text{DEG}(x)$ vertices and preserves much of the character of the original, larger tree $M(x)$. In Phase 3 we cluster the T_x tree, rather than $M(x)$, and show that little is lost in the substitution.

Definitions

Let X be a set of vertices. We define the *shortcut tree* on X to be the tree derived from $M(X)$ by splicing out all one-child nodes in $M(X) - X$ — see Figure 5.3. Therefore, the shortcut tree consists of degree-1 vertices (leaves or root), which must be in X, degree-2 vertices (internal vertices or root), which also must be in X, and vertices of degree at least 3, which are not necessarily in X. We define T_x to be the shortcut tree on the set $\text{ROOT}(M(\text{CHILD}(x)))$, i.e., the roots of the MST-subgraphs corresponding to the children of x in $\mathcal{SH}$. Notice that edges in T_x correspond to paths in M. We define the length of a T_x-edge to be the length of the associated path.

Because every vertex in T_x is either in the set $\textsc{root}(M(\textsc{child}(x)))$ or has degree at least 3, we have:

$$|T_x| = [\textsc{deg}(x), 2\textsc{deg}(x))$$

Since T_x is thought of as a succinct representation of $M(x)$, it will be convenient to talk about the mass of a subtree in T_x, and have that mass roughly match up with the corresponding subtree in $M(x)$. To that end we attribute certain amounts of mass to the *vertices* of T_x. Let v be a T_x vertex. If $v = \textsc{root}(M(y))$, for a $y \in \textsc{child}(x)$, then $\textsc{mass}(v) = \textsc{mass}(y)$; otherwise, $\textsc{mass}(v) = 0$. We define the mass of a subtree T of T_x as:

$$\textsc{mass}(T) \;=\; \sum_{e \in E(T)} \ell(e) + \sum_{v \in V(T)} \textsc{mass}(v)$$

We will think of a subtree of T_x as corresponding to a subtree of $M(x)$. Each edge in T_x corresponds naturally to a path in $M(x)$ and each vertex in T_x with non-zero mass corresponds to a subtree of $M(x)$.

Lemma 30 *If T_1 is a subtree of T_x and M_1 the corresponding subtree of M, Then* $\textsc{mass}(T_1) \;=\; [\textsc{mass}(M_1), 2\textsc{mass}(M_1)]$.

Proof: Every edge in M_1 contributes, at the very least, to the mass of one edge or vertex in T_1, and at the most, to one edge *and* vertex in T_1.

□

Construction of T_x

Our algorithm for constructing the trees $\{T_x\}_{x \in \mathcal{SH}}$ can be understood in isolation. Its particulars have no bearing on the construction algorithm for $\mathcal{RH}$, nor on our overall SSSP algorithm.

Consider the vertex set of T_x. We could just as easily defined it as the *least common ancestor closure* of $\textsc{root}(M(\textsc{child}(x)))$. In general, the least common ancestor closure of X, or $\textsc{lca}(X)$, is the minimal set satisfying:

$$X \subseteq \textsc{lca}(X)$$
$$u, v \in \textsc{lca}(X) \implies \textsc{lca}(u, v) \in \textsc{lca}(X)$$

where $\textsc{lca}(u, v)$ is the least common ancestor in some rooted tree known by context, which in our case is M.

Let $\mathcal{U} = \{u_1, u_2, \ldots, u_{\textsc{deg}(x)}\}$ be the vertices of $\textsc{root}(M(\textsc{child}(x)))$ in order of their depth-first search numbers (DFS numbers), and $\mathcal{U}_i = \{u_1, \ldots, u_i\}$. It is trivial to show that

$$\textsc{lca}(\mathcal{U}_{i+1}) \;=\; \textsc{lca}(\mathcal{U}_i) \;\cup\; \{u_{i+1},\, \textsc{lca}(u_i, u_{i+1})\}$$

where we note that $\text{LCA}(u_i, u_{i+1})$ may or may not already be in $\text{LCA}(\mathcal{U}_i)$. Our algorithm for computing the set $\{T_x\}_{x\in\mathcal{SH}}$ is as follows.

1. Find a DFS numbering of the vertices w.r.t. M
2. For each $x \in \mathcal{SH}$, let Let $u_1^x, \ldots, u_{\text{DEG}(x)}^x$ be $\text{ROOT}(M(\text{CHILD}(x)))$ in DFS-order
3. The vertex set of T_x corresponds to
$$\{u_1^x, \ldots, u_{\text{DEG}(x)}^x, \text{LCA}(u_1^x, u_2^x), \ldots, \text{LCA}(u_{\text{DEG}(x)-1}^x, u_{\text{DEG}(x)}^x)\}$$
4. Construct T_x from the DFS-order of its vertices. For an edge $(u, p(u))$ in T_x, $\ell(u, p(u))$ can be computed as the distance from u to $\text{ROOT}(M)$ (in M) less the distance from $p(u)$ to $\text{ROOT}(M)$ — see Section 2.4 for a simulation of subtraction in the comparison-addition model.

Tarjan's least common ancestor algorithm (see [47]) is said to be an off-line algorithm though this is not entirely accurate. In the context of a depth-first search, Tarjan's algorithm can handle on-line LCA queries of the form $\text{LCA}(u, v)$ if (a) v is the last vertex scanned by DFS and (b) u precedes v in this DFS. Using Tarjan's algorithm, one can compute $\{T_x\}_{x\in\mathcal{SH}}$ with one DFS traversal of M; however, the overall running time is $O(n\alpha(n))$ since there are $\sum_{x\in\mathcal{SH}}(\text{DEG}(x) - 1) < n$ LCA queries and each one takes $O(\alpha(n))$ amortized time. Using the more complicated off-line LCA algorithms of Harel and Tarjan [103] (for a RAM) or Buchsbaum et al. [24] (for a pointer machine) we can compute $\{T_x\}_{x\in\mathcal{SH}}$ in $O(n)$ time.

Lemma 31 *Given $\mathcal{SH}$, the set $\{T_x\}_{x\in\mathcal{SH}}$ can be computed in $O(n)$ time.*

5.2.4 Phase 3: Computing a Refined Hierarchy

In this section we show how to construct an H_x from T_x that is consistent with Property 3. The running time of our method is linear in the size of T_x. Together with Lemma 31, this implies that $\mathcal{RH}$ can be computed in $O(n)$ time, given $\mathcal{SH}$ and M.

The REFINE-HIERARCHY procedure, given as pseudocode in Figure 5.4, constructs H_x in a bottom-up fashion by traversing the tree T_x. Although necessity may force it to do otherwise, REFINE-HIERARCHY seeks to create an *ideal* solution: rank j nodes in H_x should be the children of rank $j+1$ nodes and have mass $\lambda_j \cdot \text{NORM}(x)$.

A call to REFINE-HIERARCHY(v), where $v \in T_x$ will produce an array of sets $v[\cdot]$ whose elements are nodes in H_x that represent (collectively) the subtree of T_x rooted at v, which in turn represents the subtree of $M(x)$ rooted at v. The set $v[j]$ holds rank j nodes, which, taken together, are not yet massive enough to become a rank $j+1$ node.

We use the notation $\textsc{mass}(v[j])$ as short for $\textsc{mass}(v[0] \cup \cdots \cup v[j])$, in other words, the mass of the subgraph of T_x induced by elements in the sets $v[0] \cup \cdots \cup v[j]$.

The structure of a call to $\textsc{Refine-Hierarchy}(v)$ is fairly simple. To begin with, we initialize $v[\cdot]$ be an array of empty sets. If v itself has non-zero mass in T_x, i.e. if $v = \textsc{root}(M(y))$, where $y \in \textsc{child}(x)$, then we place a node called y somewhere in $v[\cdot]$. (Recall from the definition of H_x that its leaves correspond to the children of x in $\mathcal{SH}$.) We determine the j satisfying:

$$\lambda_j \leq \frac{\textsc{mass}(y)}{\textsc{norm}(x)} < \lambda_{j+1}$$

and place y in $v[j]$.

Next we process the children of v in T_x, one by one. Each pass through the loop we pick an as yet unprocessed child w of v, recurse on w, producing sets $w[\cdot]$ representing the subtree rooted at w, then merge the sets $w[\cdot]$ into their counterparts in $v[\cdot]$. At this point, the mass of some sets may be beyond a critical threshold; the threshold for $v[j]$ is $\lambda_{j+1} \cdot \textsc{norm}(x)$. In order to restore a quiescent state in the sets $v[\cdot]$ we perform *promotions* until no set's mass is above threshold.

Definition 4 *Promoting the set $v[j]$ involves removing the nodes from $v[j]$, making them the children of a new rank $j+1$ node, then placing this node in $v[j+1]$. There is one exception: if $|v[j]| = 1$ then we simply move the node from $v[j]$ to $v[j+1]$, without changing its rank. Promoting the sets $v[0], v[1], \ldots, v[j]$ means promoting $v[0]$, then $v[1]$, up to $v[j]$, in that order.*

Suppose that after merging $w[\cdot]$ into $v[\cdot]$, j is maximal such that $\textsc{mass}(v[j])$ is beyond its threshold of $\lambda_{j+1} \cdot \textsc{norm}(x)$ (there need not be such a j.) We promote the sets $v[0], \ldots, v[j]$, which has the effect of emptying the sets $v[0], \ldots, v[j]$ and adding a new node to $v[j+1]$ representing the nodes formerly in $v[0], \ldots, v[j]$. We compute H_x by calling $\textsc{Refine-Hierarchy}(\textsc{root}(T_x))$; the resulting node is the root of H_x and is therefore identified with $x \in \mathcal{SH}$.

Lemma 32, given below, shows that we can compute $\{H_x\}_{x \in \mathcal{SH}}$ in linear time, provided that $\{T_x\}_{x \in \mathcal{SH}}$ is given.

Lemma 32 *Given the trees $\{T_x\}_{x \in \mathcal{SH}}$, a set of trees $\{H_x\}_{x \in \mathcal{SH}}$ satisfying Property 3 can be constructed in $O(n)$ time.*

Proof: We first argue that $\textsc{Refine-Hierarchy}(\textsc{root}(T_x))$ produces an H_x satisfying Property 3. We then look at how to implement $\textsc{Refine-Hierarchy}$ in linear time.

Property 3(1) states that internal nodes in H_x must have $\textsc{norm}$-values equal to that of x, which we satisfy by simply assigning them the proper $\textsc{norm}$-values, and that

```
Refine-Hierarchy($v$)          where $v$ is a vertex in $T_x$.
1.    Initialize $v[j] := \emptyset$ for all $j$.
2.    If $v = \textsc{root}(M(y))$, where $y$ is a child of $x$ in $\mathcal{SH}$
3.        Let $j$ be maximal s.t. $\textsc{mass}(y)/\textsc{norm}(x) \geq \lambda_j$.
4.        $v[j] := \{y\}$;   (i.e., $y$ is implicitly designated a rank $j$ node)
5.    For each child $w$ of $v$ in $T_x$:
6.        Refine-Hierarchy($w$)
7.        For all $i$,  $v[i] := v[i] \cup w[i]$
8.        Let $j$ be maximal such that $\textsc{mass}(v[j])/\textsc{norm}(x) \geq \lambda_{j+1}$
9.        Promote $v[0], \ldots, v[j]$      (see Definition 4)
10.   If $v$ is the root of $T_x$, promote $v[0], v[1], \ldots$ until one node remains.
      (This final node is the root of $H_x$, which is returned.)
```

Figure 5.4: An algorithm for constructing H_x, given T_x.

no node of H_x have one child. By our treatment of one-element sets in the promotion procedure of Definition 4, it is simply impossible to create a one-child node in H_x. Property 3(5) follows from Lemma 30 and the observation that the mass (in T_x) represented by nodes of the same rank is disjoint. Now consider Property 3(3), regarding stunted nodes. We show that whenever a set $v[j]$ accepts a new node z, either $v[j]$ is immediately promoted, or z is not stunted, or the promotion of z into $v[j]$ represents the *last* promotion in the construction of H_x (Refine-Hierarchy, line 10). Consider the pattern of promotions in Line 9. We promote the sets $v[0], \ldots, v[j]$ in a cascading fashion: $v[0]$ to $v[1]$, $v[1]$ to $v[2]$ and so on. The only set accepting a new node which is not immediately promoted is $v[j+1]$, so in order to prove Property 3(3) we must show that the node derived from promoting $v[0], \ldots, v[j]$ is not stunted. By choice of j, $\textsc{mass}(v[j]) \geq \lambda_{j+1} \cdot \textsc{norm}(x)$, where mass is w.r.t. the tree T_x. By Lemma 30 the mass of the equivalent tree in $M(x)$ is at least $\lambda_{j+1} \cdot \textsc{norm}(x)/2$, which is exactly the threshold for this node being stunted. Finally, consider Property 3(4). Before the merging step in Line 7 none of the sets in $v[\cdot]$ or $w[\cdot]$ is massive enough to be promoted. Let $v[\cdot]$ and $w[\cdot]$ denote the sets associated with v and w before the merging in step 7, and let $v'[\cdot]$ denote the set associated with v after the step 7. By the definition of mass we have:

$$\textsc{mass}(v'[j]) = \textsc{mass}(v[j]) + \textsc{mass}(w[j]) + \ell(v,w) < 2 \cdot \lambda_{j+1} \cdot \textsc{norm}(x) + \ell(v,w)$$

Since (v,w) is an edge in T_x it can be arbitrarily large compared to $\textsc{norm}(x)$, meaning we cannot place any reasonable bound on $\textsc{mass}(v'[j])$ after the merging step.

Let us consider how Property 3(4) is maintained. Suppose that $v'[j]$ is promoted in Lines 9 or 10 and let y be the resulting rank $j+1$ node. Using the terminology from Property 3(4), let $Y_1 = v[j], Y_2 = w[j]$ and let z be the node derived by promoting $v'[0], \ldots, v'[j-1]$. Since neither $v[j]$ nor $w[j]$ were sufficiently massive to be promoted before they were merged, we have $(\textsc{mass}(Y_1) + \textsc{mass}(Y_2))/\textsc{norm}(x) < 2\lambda_{j+1}$. (Note that by Lemma 30 this inequality holds if mass is taken w.r.t. to either T_x or $M(x)$.) This is slightly stronger than what Property 3(4) calls for, which is the inequality $< (2+o(1))\lambda_{j+1}$. We'll see why the $(2+o(1))$ is needed below.

Suppose that we implemented Refine-Hierarchy in a straightforward manner. Let L be the maximum possible index of any non-empty set $v[\cdot]$ during the course of Refine-Hierarchy. One can easily see that the initialization in Lines 1–4 take $O(L+1)$ time and that exclusive of recursive calls, each time through the for loop in Line 5 takes $O(L+1)$ amortized time. (The bound on Line 5 is *amortized* since promoting a set $v[j]$ takes worst case $O(|v[j]|+1)$ time but only constant amortized time. The cost of examining a node in $v[j]$ can be charged to the promotion that created it.) The only hidden costs in this procedure are dynamically updating the $\textsc{mass}$ of $v[0], \ldots, v[L]$, as the contents of these sets change. After the merging step in Line 7 we *could* simply set $\textsc{mass}(v[j]) := \textsc{mass}(v[j]) + \ell(v,w) + \textsc{mass}(w[j])$ for each $j \le L$. Therefore the total cost of computing H_x from T_x is $O((L+1) \cdot |T_x|)$. We can prove $L \le 2\log^*(4n)$ as follows. The first node placed in any previously empty set is unstunted; therefore, by Lemma 28, the maximum non-empty set has rank at most $2\log^*(\textsc{mass}(T_x)/\textsc{norm}(x))$. By Lemma 30 and Lemma 3(5) $\textsc{mass}(T_x) \le 2 \cdot \textsc{mass}(M(x)) < 4(n-1) \cdot \textsc{norm}(x)$.

In order to reduce the cost to linear we make a couple adjustments to the Refine-Hierarchy procedure. First, $v[\cdot]$ is represented as a linked-list of non-empty sets. Second, we update the mass variables in a lazy fashion. The time for Steps 1–4 is dominated by the time to find the appropriate j in Step 3, which takes time t_1 – see below. The time for merging the $v[\cdot]$ and $w[\cdot]$ sets in Step 7 is only proportional to the shorter list; this time bound is given by expression t_2 below.

$$t_1 = O\left(1 + \log^* \frac{\textsc{mass}(v)}{\textsc{norm}(x)}\right)$$

$$t_2 = O\left(1 + \log^* \frac{\min\{\textsc{mass}(v[\cdot]), \textsc{mass}(w[\cdot])\}}{\textsc{norm}(x)}\right)$$

where $\textsc{mass}(v[\cdot]) = \textsc{mass}(v[0] \cup v[1] \cup \cdots)$. After Steps 1–4 we update the mass of the sets $v[0..\Theta(t_1)]$, and after Step 7 we update the mass of $v[0..\Theta(t_2)]$. Furthermore, as a rule we update $v[j+1]$ no less than half as often as $v[j]$. It is routine to show that Refine-Hierarchy will have a lower bound on the mass of $v[j]$ which is off by a 1+o(1) factor, where the $o(1)$ is a function of j.[3] This leads to the conspicuous $2+o(1)$

[3]The proof of this is somewhat tedious. Basically one shows that for $i < j$, the mass of $v[i]$ can

in Property 3(4). To bound the cost of REFINE-HIERARCHY we model its computation as a binary tree, the leaves of which are identified with the H_x-nodes created in Lines 1–4. Its internal nodes represent the merging events in Line 7. The *cost* of a leaf f is $\log^*(\text{MASS}(f)/\text{NORM}(x))$, and the cost of an internal node f with children f_1 and f_2 is $1 + \log^*(\min\{\text{MASS}(f_1)/\text{NORM}(x), \text{MASS}(f_2)/\text{NORM}(x)\})$, where the mass of a node is the sum of the mass of its descendant leaves. We can think of charging the cost of f collectively to the mass of the leaves of the subtree of f_1 or f_2, whichever is smaller. Therefore, no unit of mass can be charged for two nodes f and g if the mass of f is within twice the mass of g. The total cost is then:

$$\sum_f cost(f) \;=\; O(|T_x| + \frac{\text{MASS}(T_x)}{\text{NORM}(x)} \cdot \sum_{i=0}^{\infty} \frac{\log^*(2^i)}{2^i}) \;=\; O\left(\frac{\text{MASS}(x)}{\text{NORM}(x)}\right)$$

The last equality follows since both $|T_x|$ and $\frac{\text{MASS}(T_x)}{\text{NORM}(x)}$ are $O(\frac{\text{MASS}(M(x))}{\text{NORM}(x)}) = O(\frac{\text{MASS}(x)}{\text{NORM}(x)})$. Summing over all $x \in \mathcal{SH}$, the total cost of constructing $\{H_x\}_{x\in\mathcal{SH}}$ is $O(n)$ by Lemma 3(6).

□

Theorem 6 *The time required to compute a refinement of $\mathcal{SH}$ satisfying Property 3 is* $O(\text{MST}^*(m,n) + \min\{n\log\log r,\ n\log n\})$

Proof: Follows from Lemmas 27, 31, 32.

□

Remark. The running time of Theorem 6 will be unaltered even if we use the simpler implementation of the algorithm in Lemma 32, provided that either $\log^* n = O(m/n)$ or $\log^* n = O(\log\log r)$.

5.3 Variations on the Algorithm

5.3.1 Simpler and Slower

One objection to our algorithm is that it uses a non-standard data structure, namely our specialized Bucket-Heap. Is it possible to use only off-the-shelf data structures without increasing the overall running time? In this section we observe that if GENERALIZED-VISIT is run on $\mathcal{RH}$, rather than $\mathcal{SH}$, and if Fibonacci heaps [73] are used in lieu of a bucket array, then SSSP can be computed in $O(m + n\log^* n)$ time, which is not much worse than the $O(\text{SPLIT-FINDMIN}(m,n)) = O(m\log\alpha(m,n))$ bound of Theorem 5.

be updated at most $2^{j-i} - 1$ times before the mass of $v[j]$ is updated. Since $2^{j-i} - 1 \cdot \lambda_i << \lambda_j$, our neglecting to update the mass of $v[j]$ causes only a negligible error.

Claim 1 *Undirected SSSP can be computed in $O(m + n\log^* n)$ time with an implementation of* GENERALIZED-VISIT *that operates on $\mathcal{RH}$. The only non-trivial data structures used are split-findmin and Fibonacci heaps.*

Proof: For $x \in \mathcal{RH}$, we simulate the bucket array of x using a linked-list and a Fibonacci heap. The linked list holds the active children of x that belong in the current bucket. When a new bucket becomes active, any inactive nodes in x's Fibonacci heap that belong in the current bucket are moved to x's linked list. The upshot is that every node in $\mathcal{RH}$ is removed from a Fibonacci heap just once. Thus, the heap-cost of this implementation is:

$$O(m) + \sum_{x \in \mathcal{RH}} \textsc{deg}(x) \log \textsc{deg}(x)$$

where the $O(m)$ term represents the cost of decrease-keys. It is simple to show that $\sum_x \textsc{deg}(x) \log \textsc{deg}(x) = O(n)$ using a proof similar to that of Lemma 29. The cost per recursive call to GENERALIZED-VISITis constant, but unlike before, the total number of recursive calls may be super-linear. It is bounded by:

$$\begin{aligned}
\sum_{x \in \mathcal{RH}} \frac{\textsc{diam}(x)}{\textsc{norm}(x)} + 1 &= \sum_j \sum_{\substack{x \in \mathcal{RH} \\ \text{rank}(x) = j}} \frac{\textsc{diam}(x)}{\textsc{norm}(x)} + 1 \\
&< 2n + 2\log^* n \cdot \sum_{x \in \mathcal{SH}} \frac{\textsc{diam}(x)}{\textsc{norm}(x)} \qquad \{\text{Lemma 28}\} \\
&= O(n\log^* n) \quad \{\text{Lemma 3(6)}\}
\end{aligned}$$

The other costs of GENERALIZED-VISIT are covered by the split-findmin structure, whose complexity is $O(m \log \alpha(m, n)) = O(m + n\log^* n)$. Thus the total cost of computing undirected SSSP with GENERALIZED-VISIT is $O(m + n\log^* n)$, provided $\mathcal{RH}$ is given.

□

Remark. Claim 1 is probably a little pessimistic. We would not expect the $n\log^* n$ term to manifest itself, unless the input graph had a very carefully chosen structure.

5.3.2 Sort-of-Undirected Graphs

Consider the graph corresponding to the road map of your favorite city. In all likelihood this graph is almost planar and nearly undirected. Does this mean that the wealth of algorithms for planar and/or undirected graphs simply cannot be applied? Probably not. Such algorithms (or their analyses) can frequently be modified to tolerate minor deviations from an ideal graph type. For instance, the *crossing number* is a good measure of how *nonplanar* a graph is. We would expect the asymptotic performance of certain planar shortest path algorithms [66, 105] to be unaffected, provided that the crossing number is sufficiently low.

One generalization of weighted undirected graphs is *disparity-bounded* graphs. Let $\mathcal{G}[k]$ be the set of *directed* graphs of the form $G = (V, E, \ell)$ where if $(u, v) \in E$ then $\ell(v, u)/\ell(u, v) \leq k$, where $(v, u) \notin E$ implies $\ell(v, u) = \infty$. Then $\mathcal{G}[\infty]$ is the set of all weighted directed graphs, and $\mathcal{G}[1]$ is the set of weighted undirected graphs.

It is not difficult to show that our undirected SSSP algorithm also works well with disparity-bounded graphs; we do, however, need to make a few modifications to the algorithm. In the construction of our hierarchies we let the length of the (imagined) undirected edge $\{u, v\}$ be $\min\{\ell(u, v),\ \ell(v, u)\}$. If the input graph appears in $\mathcal{G}[k]$ then all our calculations of DIAM and MASS are off by at most a factor of k. One can then prove updated versions of Lemma 3(5) and (6), and Theorems 5 and 6. The result is summarized in Theorem 7.

Theorem 7 *Suppose G is a graph drawn from $\mathcal{G}[k]$. Then the $\mathcal{RH}$ hierarchy for G can be computed in time $O(\text{MST}^*(m, n) + \min\{n \log\log r,\ n \log n\})$, where m and n are the number of edges and vertices, r a bound on the ratio of any two non-zero edge-lengths, and MST* the decision-tree complexity of the minimum spanning tree problem. Furthermore, if given the $\mathcal{RH}$ hierarchy, SSSP can be computed in time $O(\text{SPLIT-FINDMIN}(m, n) + kn)$, where SPLIT-FINDMIN is the decision-tree complexity of the split-findmin problem.*

Thus, if $k = O(m/n)$, computing SSSP on a graph from $\mathcal{G}[k]$ is just as easy as from $\mathcal{G}[1]$.

The class $\mathcal{G}[k]$ does not really capture road networks, where any edge (street) is either purely directed (one-way) or purely undirected (two-way). One can extend Theorem 7 to the case when all edges have bounded disparity, except for, say, $n/\log n$ of them, which can be purely directed (disparity ∞). However, our SSSP algorithm would require some non-trivial modifications to deal effectively with this class of graphs.

5.4 Discussion

The running time of our SSSP algorithm has an unsightly dependence on r, the ratio of the maximum-to-minimum edge length. Although our algorithm is asymptotically faster than Dijkstra's for common values of m, n, and r (say, $m = O(n)$ and $r = \text{poly}(n)$), it provides no improvement in general. Theorem 2 gives an explanation for this dependence on r. Basically, the $\min\{n \log\log r,\ n \log n\}$ term is unavoidable under some weak assumptions about *how* a hierarchy-type algorithm computes SSSP. Thus, any faster undirected SSSP algorithm must do *something* fundamentally different than our algorithm. No tweaks or data structural improvements will suffice.

It is a little counterintuitive that increasing r should make the SSSP problem conceptually more difficult. Generally speaking, the single-source shortest path tree of a graph whose edge-lengths vary widely in magnitude should resemble the graph's minimum spanning tree, regardless of the source vertex. Since the MST problem is manifestly simpler than SSSP — see Chapter 7 — one would think that large values of r should make the SSSP problem easier. We suspect that the key to a faster undirected SSSP algorithm is some simple, but as-of-yet unobserved property of minimum spanning trees.

Chapter 6

Experimental Evaluation of a Shortest Path Algorithm

6.1 Introduction

Whenever one develops a theoretically faster algorithm, one immediately wonders how well it *actually* performs, running on a physical computer. In this chapter we attempt to gauge how well a simplified version of our *undirected* shortest path algorithm [Chapter 5] performs in practice. We are interested in its speed, both relative to Dijkstra's algorithm (the previous best for real-weighted graphs), and relative to a *hypothetical* practically optimal shortest path algorithm.[1]

Our experimental findings indicate that our algorithm outperforms Dijkstra's on a variety of sparse graphs, ranging in size from a few thousand to a few million vertices. More surprisingly, the ratio of our running time to that of breadth first search (BFS) is always less than 2.77, and usually less than 2.15. This shows that our algorithm is very nearly optimal, under the plausible assumption that BFS is a *practical* lower bound on the shortest path problem.

The timings mentioned above refer to the *marginal* cost of our algorithm, that is, the cost of computing SSSP after a suitable hierarchy has been constructed. A key question is: how many SSSP computations make this initial, one-time cost worthwhile? On the classes of graphs we tested, this critical threshold was always between 3 and 12, well within the range of acceptability for many practical applications.

We chose not to experiment with our *directed* shortest path algorithm [Chapter 4] for a number of reasons. First and foremost, it only dominates Dijkstra's algorithm when computing APSP, or more accurately, SSSP from at least $\omega(m/\log n)$ different

[1]The results of this chapter were published in: S. Pettie, V. Ramachandran, and S. Sridhar, An experimental evaluation of a new shortest path algorithm, Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 126–142, 2002.

sources. Therefore, we would not have time to test it on any really large graphs, where its asymptotic advantage would supposedly emerge. Second, on small graphs its running time would likely be dominated by some ungainly constant factors, which were conveniently masked by the asymptotic analysis. In contrast, our *undirected* shortest path algorithm [Chapter 5] does not suffer from these problems. After constructing a suitable hierarchy in $O(m + n \log n)$ time, it can begin computing SSSP in near-linear time; moreover, the algorithm is streamlined and relatively simple to code.

6.1.1 Previous Work

The performance of Dijkstra's algorithm is largely determined by the choice of priority queue or *heap*. We are unaware of any thorough study of the best heap for use in Dijkstra's shortest path algorithm. However, Moret and Shapiro [166] did an extensive study of the Dijkstra-Jarník-Prim[2] minimum spanning tree algorithm, which is nearly identical to Dijkstra's shortest path algorithm. They found that the *pairing heap* of Fredman et al. [72] was usually the best, and the only heap to consistently perform well over a variety of inputs.

Goldberg & Co. [90, 33, 92, 34, 88] experimented with several directed SSSP algorithms, both for positive and arbitrary edge lengths. In these experiments the edge-lengths were chosen to be relatively small, uniformly distributed integers. Moreover, most of their algorithms assumed, and were tailored to, integer edge-lengths. (For instance, [88] tests a number of algorithms based on multi-level bucketing structures, which are not appropriate in the context of real-weighted graphs.) The best algorithm in Goldberg's recent study [88] never performed worse than 2.5 times the speed of BFS, and usually no more than 2 times BFS.

6.1.2 Scope of the Experiment

In our experiments we focus on undirected SSSP algorithms for the comparison-addition model with provably good worst-case running times. Because the comparison-addition model abstracts away details of numerical representation and implementation, it can be viewed as a *programming interface*. Therefore, any comparison-addition based algorithm works – without modification – with any conceivable numerical data type.

We have a number of candidate algorithms to choose from. However, many of them have quadratic or cubic worst-case running times; for instance, Bellman-Ford, Floyd-Warshall, and min-plus matrix multiplication (refer to [47]) are all inappropriate as undirected SSSP algorithms. The real contenders are Dijkstra's algorithm [52] and the algorithm presented in Chapter 5. (We remark that Meyer [160] has an SSSP algorithm

[2]Sometimes called *Prim's algorithm*; see Chapter 8 for more on this.

running in expected linear time if the edge-lengths are uniform in $[0, 1]$. However, it is fairly complicated and has a quadratic worst-case running time.)

Since the undirected SSSP problem has essentially been solved for graph-density $m/n \geq \log n$ (Dijkstra's algorithm runs in linear time), we focus on sparse graphs. In real-world situations sparse graphs are the norm; for instance, planar graphs, d-dimensional grid graphs, and the web-graph are all sparse.

Single-Source vs. Multi-Source Shortest Paths

Our undirected shortest path algorithm [Chapter 5] has a one-time cost of $O(m+n\log n)$ to construct a suitable hierarchy, whereas Dijkstra's algorithm has no startup cost, aside from the negligible cost of allocating memory. In our timings we consider both the startup cost of our algorithm and the marginal cost of one SSSP computation. In other words, we are studying the more general multi-source shortest path problem (MSSP), which subsumes both SSSP and APSP.

In many practical situations it is MSSP that needs to be solved, not SSSP. For instance, in a recent algorithm for graphic facility location and k-median, Thorup [198] begins by performing a polylogarithmic number of of SSSP computations. In this situation one would be happy to trade a large startup cost in exchange for reducing the marginal cost of SSSP.

6.2 Design Choices

6.2.1 Dijkstra's Algorithm

Based on the experimental study of Moret and Shapiro [166], we chose to implement the priority queue in Dijkstra's algorithm with a pairing heap [72]. Among heaps supporting the decrease-key operation pairing heaps seem to be the fastest in practice; the theoretically optimal Fibonacci heap [73] is typically slower. It was proved recently [70] that the pairing heap is technically suboptimal: decrease-key operations can – in the worst case – take $\Omega(\log\log n)$ time.[3]

There has been a lot of work recently on data structures with good cache usage. However, none of the cache-sensitive heaps [177, 11, 23] supports the decrease-key operation. Indeed, it seems that even allowing decrease-keys destroys the possibility of optimal cache usage.

[3]However, based on [70] a plausible scenario is that pairing heaps support n insert/delete operations and m decrease-keys in $O(m\log\log n + n\log n)$ time, meaning they would still be optimal for density $m/n < \log n/\log\log n$. Thus, for applications on sparse classes of graphs, there may not even be a *theoretical* objection to pairing heaps.

The experimental studies by Goldberg et al. [33, 92, 88] have used multi-level bucket structures to implement the heap in Dijkstra's algorithm. Constant-time bucketing, however, is only available with integer edge-lengths. The bucketing strategies of [5, 160, 89] can, in principle, be applied to real edge-lengths, though the resulting data structure is much clumsier than a pairing heap.

6.2.2 Pettie-Ramachandran Algorithm

We call the algorithm presented in Chapter 5 the Pettie-Ramachandran or *PR* algorithm. In Chapter 5 our main concern was asymptotic running time, not efficient code. In this chapter we test a version of the PR algorithm simplified in many respects; we do not guarantee the same worst-case asymptotic running time.

Finding the MST: The PR algorithm constructs a hierarchy based on a categorization of the minimum spanning tree edges by length; in the worst case the MST edges are sorted. We choose Kruskal's MST algorithm because it is reasonable fast – $O(m \log n)$ time – and produces the MST edges in sorted order. Some of our data on larger and denser graphs suggests that it may be better to use the Dijkstra-Jarník-Prim MST algorithm, which is empirically faster than Kruskal's [166], followed by a step to sort only the MST edges.

Updating D-values: In Chapter 5 we recommend using a split-findmin data structure to handle the tentative distances (D-values) of hierarchy nodes. In our implementation of the PR algorithm we update D-values using the naïve method: whenever the D-value of a leaf is reduced we propagate the new D-value up the hierarchy, until a node is reaches with lesser D-value. The worst-case time of this method is proportional to the height of the hierarchy. However, if the naïve method is used in conjunction with *Dijkstra nodes* (see below) we found that updating a D-value generally requires examining only two hierarchy nodes.

Switching to Dijkstra: The streamlined nature of Dijkstra's algorithm makes it the preferred algorithm on small graphs. For this reason we revert to Dijkstra's algorithm whenever the problem size (encountered in a recursive call of PR) drops below a certain threshold, say ν vertices. In other words, if x is a hierarchy-node and $|V(x)| \leq \nu$, we designate x a *Dijkstra node* and eliminate all the internal hierarchy-nodes below x, making $V(x)$ the children of x. This guarantees that every leaf's grandparent has at least ν leaf-descendants. In all our experiments we set $\nu = 50$.

Heaps vs. Bucket-Heaps: In Chapter 5 we recommended implementing the PR algorithm with a Bucket-Heap which is a non-standard and not necessarily practical data structure. In our experiments we use the same pairing heap used in Dijkstra's algorithm.

Refined Hierarchies In Chapter 5 our shortest path algorithm operated on a hierarchy called $\mathcal{RH}$, which was a refinement of an earlier hierarchy $\mathcal{SH}$. In our experiments we use a hierarchy more akin to $\mathcal{SH}$ than $\mathcal{RH}$. Introducing Dijkstra nodes (see above) seems to provide the same kind of benefit as refining the hierarchy, but is both simpler and faster.

6.2.3 Breadth First Search

We compare the PR algorithm with breadth first search (BFS) as well as Dijkstra's algorithm. We found the PR/BFS ratio (running time of PR vs. running time of BFS) to be a very interesting statistic. It gives us a rough estimate of the potential for improvement, and lets us compare our results against a previous study [88], which dealt with different shortest path algorithms and different hardware.

6.3 Experimental Set-up

Our main experimental platform was a SunBlade with a 400 MHz clock and 2GB DRAM and a small cache (.5 MB). The large main memory allowed us to test graphs with millions of vertices. For comparison purposes we also ran our code on selected inputs on the following machines.

1. PC running Debian Linux with a 731 MHz Pentium III processor and 255 MB DRAM.
2. SUN Ultra 60 with a 400 MHz clock, 256 MB DRAM, and a 4 MB cache.
3. HP/UX J282 with 180 MHz clock, 128 MB ECC memory.

6.3.1 Graph Classes

We ran the algorithms on the following classes of graphs:

$\mathbf{G_{n,m}}$ The distribution $G_{n,m}$ assigns equal probability to all graphs with m edges on n labeled vertices (see [59, 20] for structural properties of $G_{n,m}$). We assign edge-lengths identically and independently, using either the uniform distribution over $[0, 1)$, or the *log-uniform* distribution, where edge lengths are given the value 2^q, q being uniformly distributed over $[0, C)$ for some constant C. We use $C = 100$.

Geometric graphs Here we generate n random points (the vertices) in the unit square and connect with edges those pairs within some specified distance. Edge-lengths correspond to the distance between points. We present results for distance $1.5/\sqrt{n}$, implying an average degree $\approx 9\pi/4$ which is about 7.

Very sparse graphs These graphs are generated in two stages: we first generate a random spanning tree, to ensure connectedness, then generate an additional $n/10$ random edges. All edges-lengths are uniformly distributed.

Grid graphs The graph is a $\sqrt{n} \times \sqrt{n}$ grid with uniformly distributed edge lengths.

Random graphs can have properties that might actually be improbable in real-world situations. For example, $G_{n,m}$ almost surely produces graphs with low diameter, nice expansion properties, and very few small, dense subgraphs [20]. On the other hand, it may be that graph structure is less crucial to the performance of shortest path algorithms than edge length distribution. In the PR algorithm for instance, the random graph classes described above give rise to very similar-looking hierarchies. A typical hierarchy would be short, have very high-degree nodes near the root and low-degree nodes near the leaf-level.

We introduce two classes of random graphs that are engineered to produce hierarchies with some peculiar structure. The purpose of these graphs is to understand how the PR algorithm's performance is affected by the underlying hierarchy; they may or may not be of independent interest.

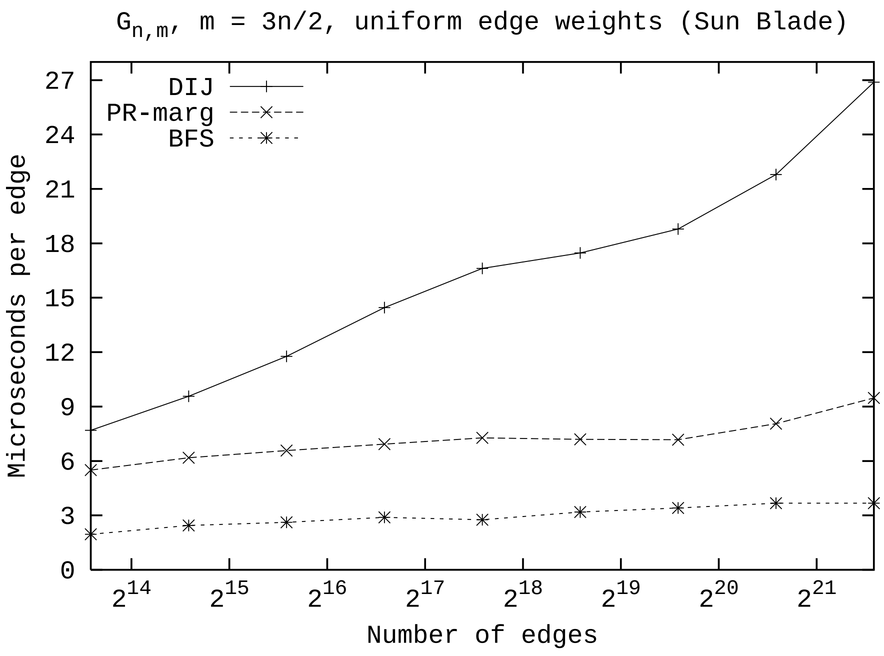
Hierarchical graphs A graph in this class almost surely produces a hierarchy that is a full b-ary tree, where $b \geq 2$. The number of edges is roughly $\frac{1}{2}n \log_b n$. We give timings for $b = 6$ and $b = 10$.

Bullseye graphs A bullseye graph is parameterized by o and d. It has $dn/2$ edges, and it produces a hierarchy that consists of a chain of o nodes, each of which is the parent of n/o leaves. We give timings for $d = 3$, $o = 25$ and $o = 100$.

In the hierarchical and bullseye classes the edge-lengths are chosen randomly, but with different constraints on each edge (because the hierarchy is predetermined). There are an equal number of edges with length in $[2^i, 2^{i+1})$, for $i < \log_b n$ (hierarchical graphs) and $i < o$ (bullseye graphs). Thus, their distributions are somewhat log-uniform.

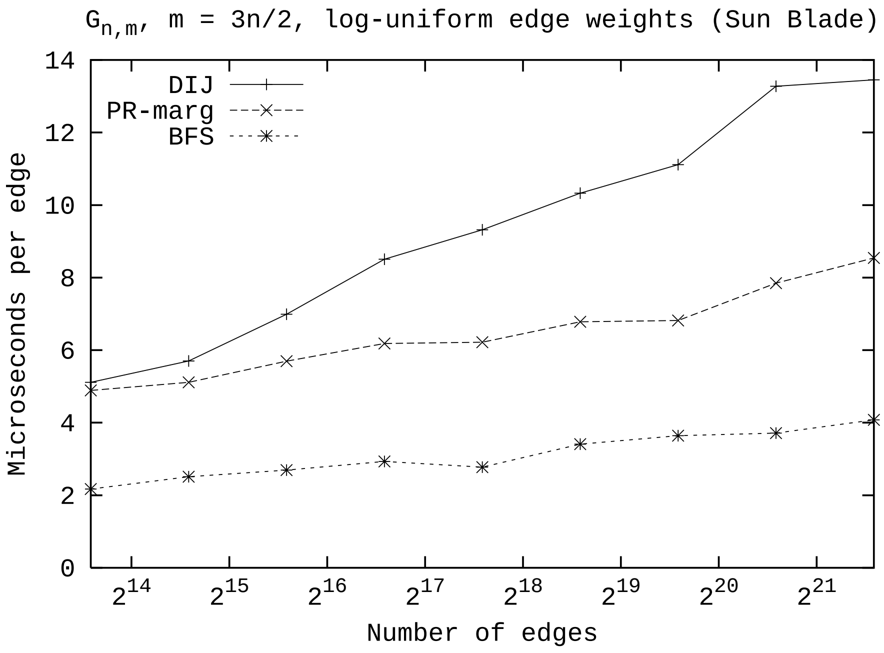
6.4 Results

The plots in Figures (a)-(i) give the running times of the two algorithms and BFS on the SunBlade for each of the graph classes we considered. Each point in the plots



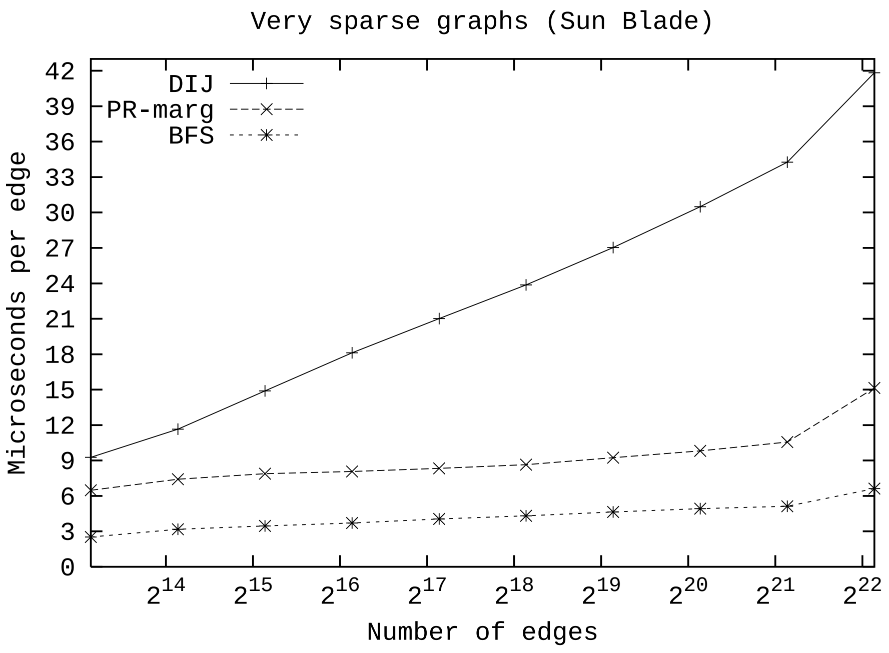
$G_{n,m}$, m = 3n/2, uniform edge weights (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
0
3
6
9
12
15
18
21
24
27
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
Number of edges

(a)



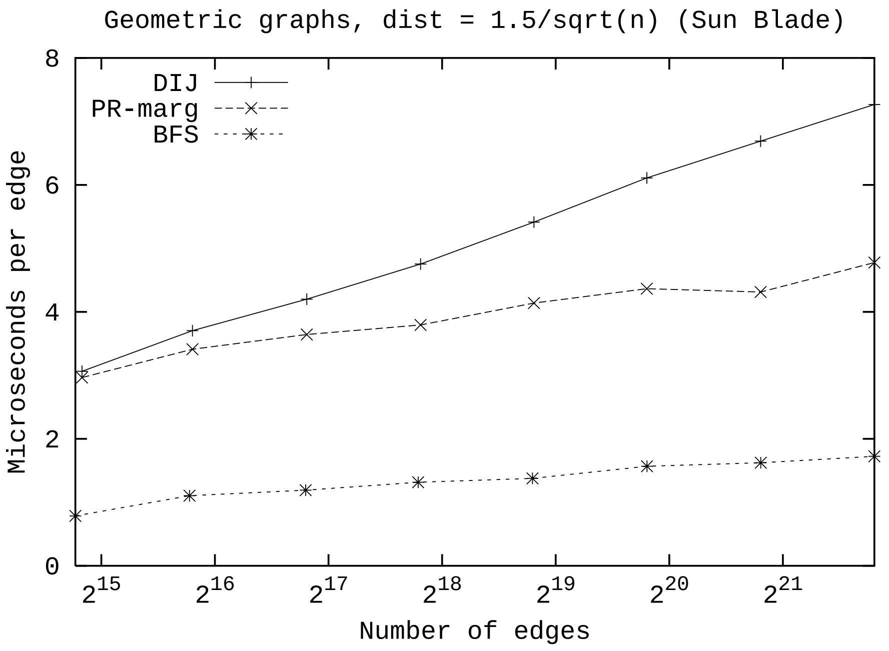
$G_{n,m}$, m = 3n/2, log-uniform edge weights (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
0
2
4
6
8
10
12
14
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
Number of edges

(b)



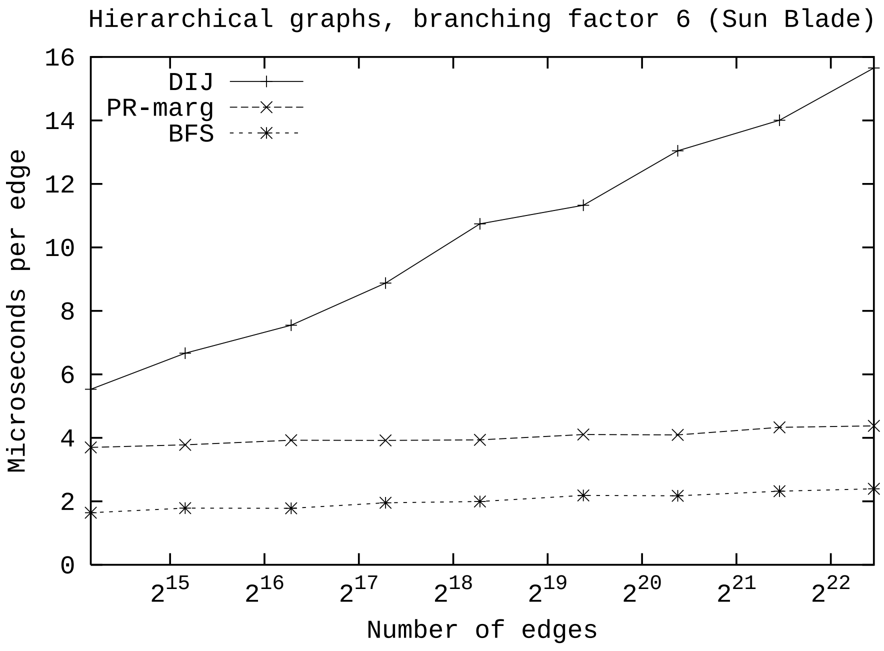
Very sparse graphs (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
0
3
6
9
12
15
18
21
24
27
30
33
36
39
42
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
2^{22}
Number of edges

(c)

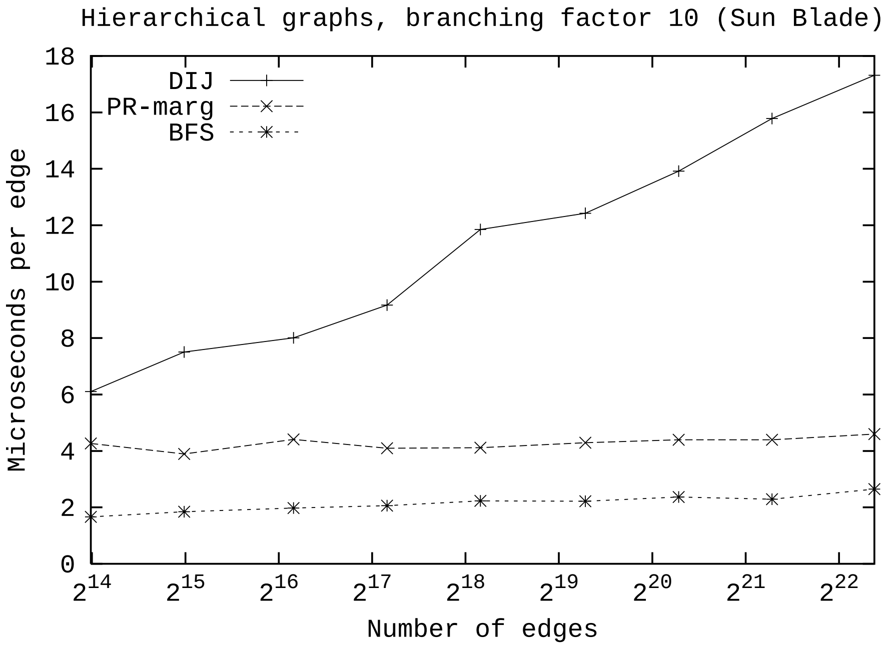


Geometric graphs, dist = 1.5/sqrt(n) (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
0
2
4
6
8
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
Number of edges

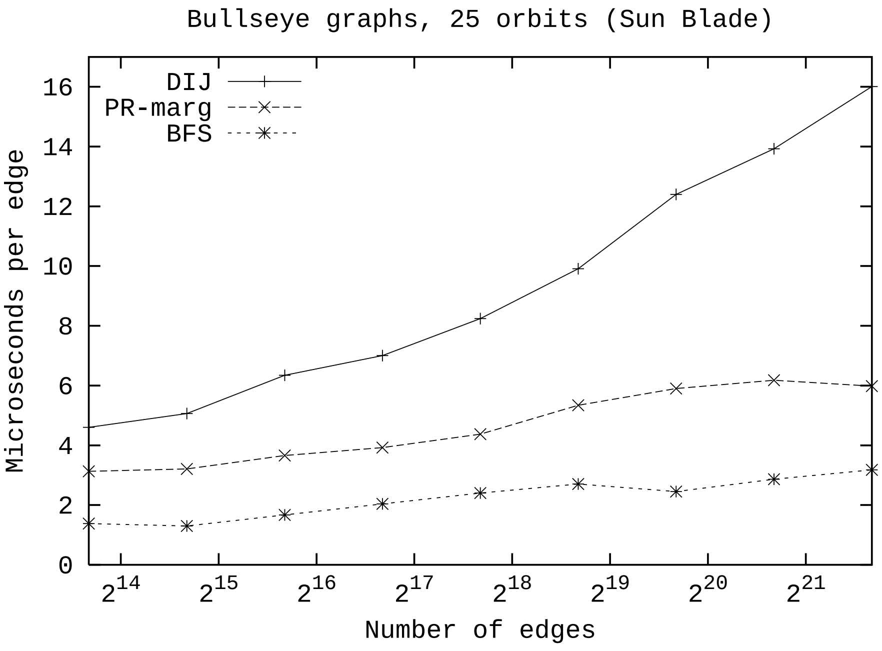
(d)


Hierarchical graphs, branching factor 6 (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges
0
2
4
6
8
10
12
14
16
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
2^{22}

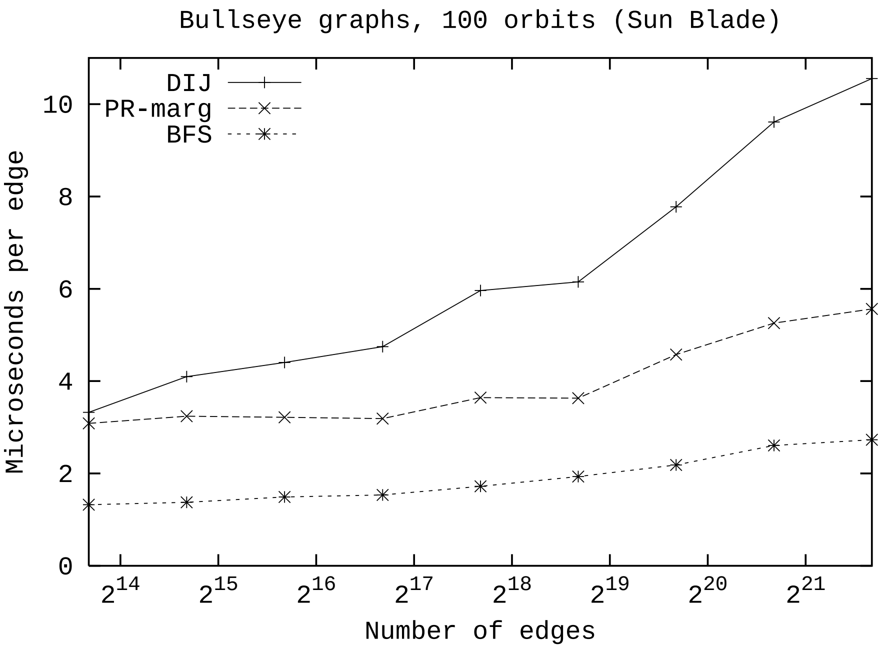
(e)


Hierarchical graphs, branching factor 10 (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
Number of edges
0
2
4
6
8
10
12
14
16
18
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
2^{22}

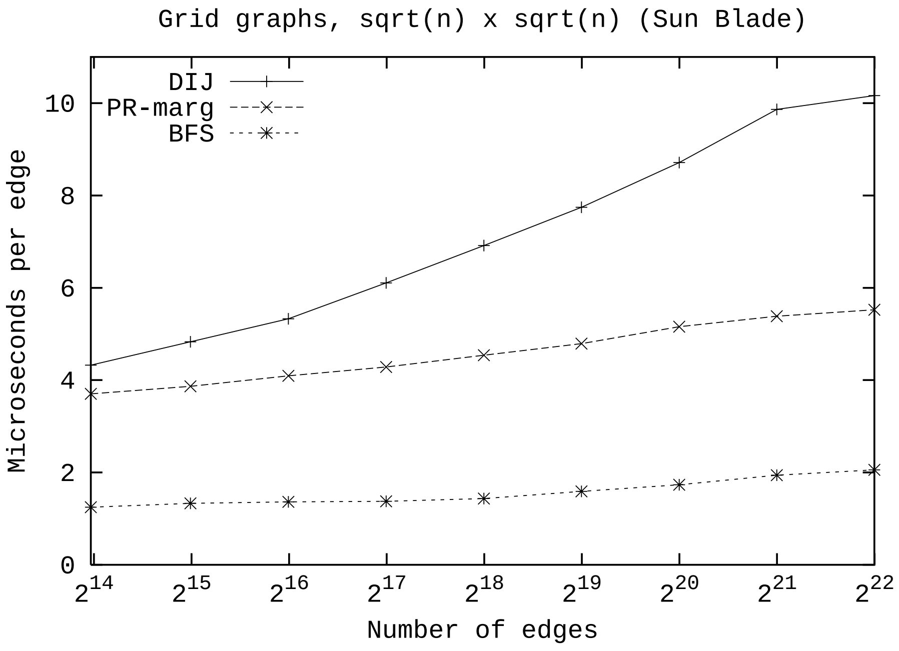
(f)


Bullseye graphs, 25 orbits (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
0
2
4
6
8
10
12
14
16
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
Number of edges

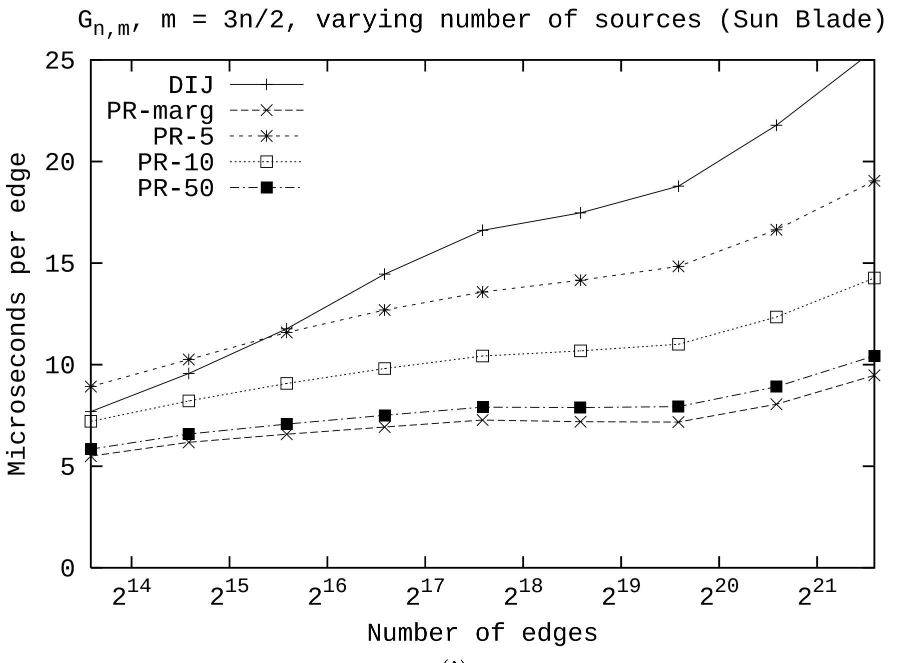
(g)


Bullseye graphs, 100 orbits (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
0
2
4
6
8
10
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
Number of edges

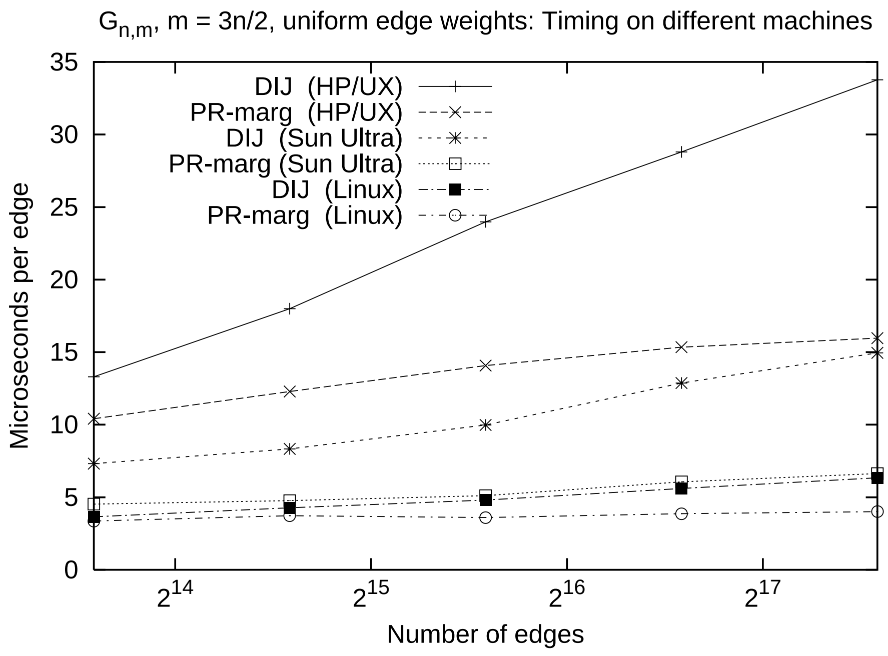
(h)


Grid graphs, sqrt(n) x sqrt(n) (Sun Blade)
DIJ
PR-marg
BFS
Microseconds per edge
0
2
4
6
8
10
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
2^{22}
Number of edges

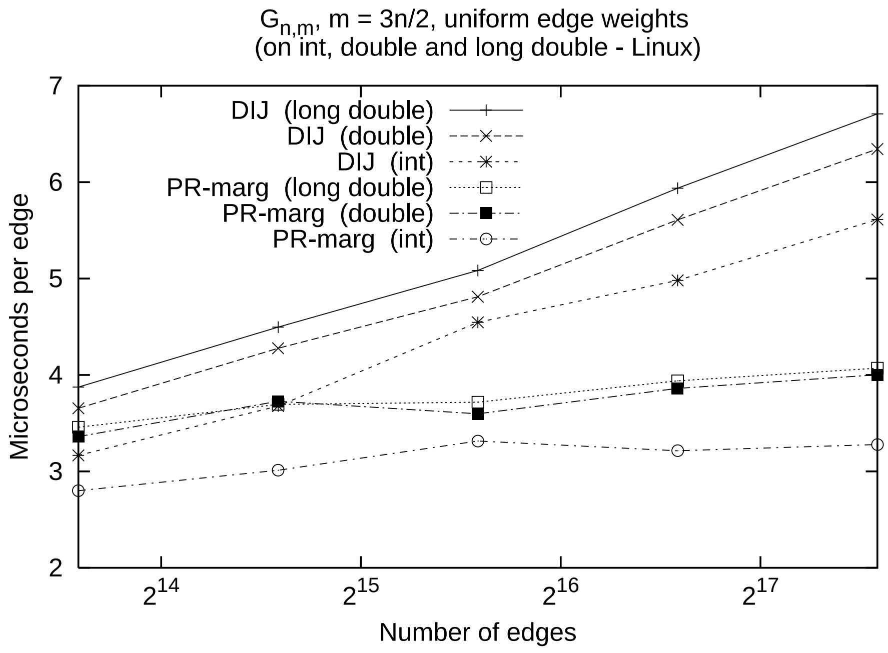
(i)


$G_{n,m}$, m = 3n/2, varying number of sources (Sun Blade)
DIJ
PR-marg
PR-5
PR-10
PR-50
Microseconds per edge
0
5
10
15
20
25
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
2^{21}
Number of edges

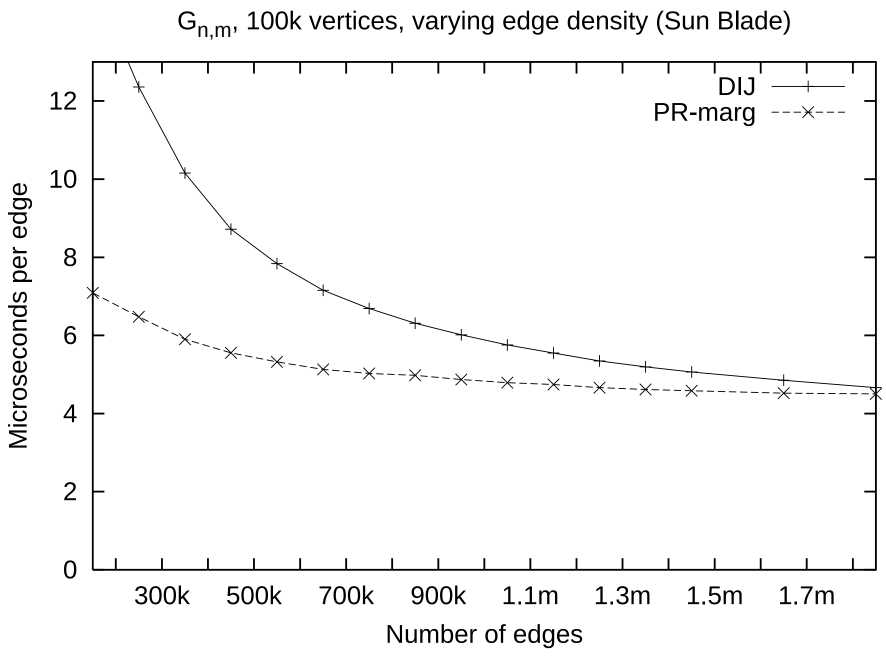
(j)


$G_{n,m}$, m = 3n/2, uniform edge weights: Timing on different machines
DIJ (HP/UX)
PR-marg (HP/UX)
DIJ (Sun Ultra)
PR-marg (Sun Ultra)
DIJ (Linux)
PR-marg (Linux)
Microseconds per edge
Number of edges
0
5
10
15
20
25
30
35
2^{14}
2^{15}
2^{16}
2^{17}

(k)

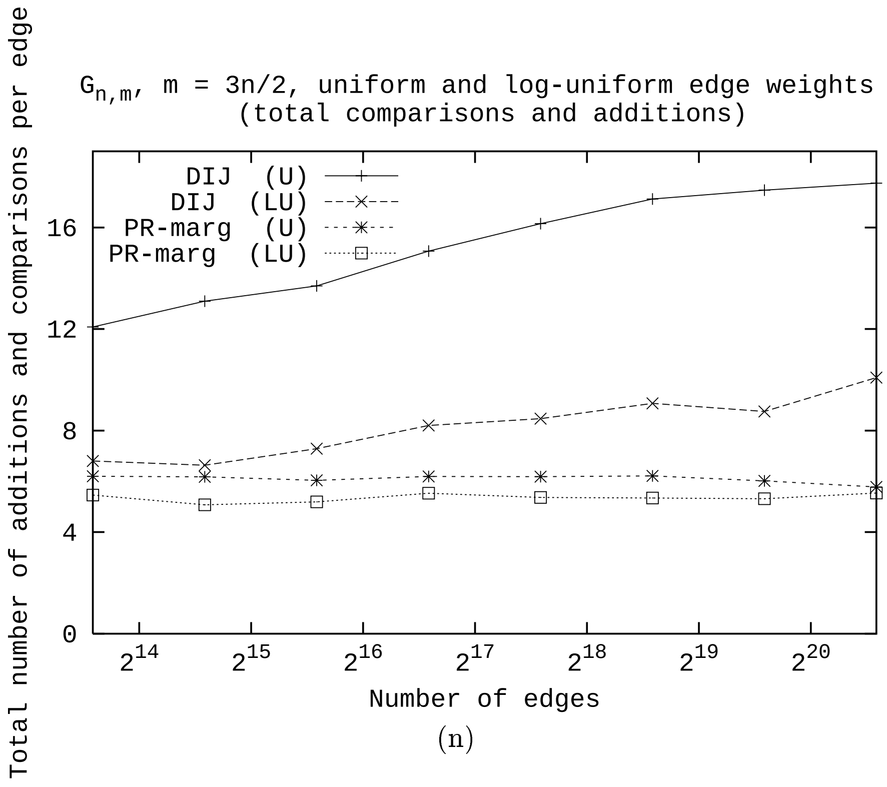

$G_{n,m}$, m = 3n/2, uniform edge weights
(on int, double and long double - Linux)
DIJ (long double)
DIJ (double)
DIJ (int)
PR-marg (long double)
PR-marg (double)
PR-marg (int)
Microseconds per edge
Number of edges
2
3
4
5
6
7
2^{14}
2^{15}
2^{16}
2^{17}

(l)



$G_{n,m}$, 100k vertices, varying edge density (Sun Blade)
DIJ
PR-marg
Microseconds per edge
0
2
4
6
8
10
12
300k
500k
700k
900k
1.1m
1.3m
1.5m
1.7m
Number of edges

(m)



$G_{n,m}$, m = 3n/2, uniform and log-uniform edge weights
(total comparisons and additions)
DIJ (U)
DIJ (LU)
PR-marg (U)
PR-marg (LU)
Total number of additions and comparisons per edge
0
4
8
12
16
2^{14}
2^{15}
2^{16}
2^{17}
2^{18}
2^{19}
2^{20}
Number of edges

(n)

represents the time to compute SSSP/BFS, averaged over thirty trials from randomly chosen sources, on three randomly chosen graphs from the class. The y-axis is a measure of 'microseconds per edge', that is, the time to perform one SSSP/BFS computation divided by the number of edges.

In the plots, DIJ represents Dijkstra's algorithm and PR-marg represents the marginal cost of computing SSSP with the Pettie-Ramachandran algorithm (excludes cost of computing the hierarchy).

It is unclear how close a comparison-addition based SSSP algorithm can get to the speed of BFS. Our results show that on a variety of graph types the marginal cost of the PR algorithm is *very* competitive with BFS, running between a factor of 1.87 and 2.77 times the BFS speed and usually less than 2.15 times BFS — see Table 1, third row. The PR algorithm clearly leaves little room for improvement.

The effect of precomputing a hierarchy is described in Figure (j) and Table 6.1. Table 6.1 (first row) lists, for each class of graphs, the critical threshold s_0 such that PR outperforms Dijkstra's algorithm, when computing SSSP at least s_0 times on a graph with 2^{20} vertices. Figure (j) shows the amortized time per SSSP computation, including the cost of precomputing the hierarchy, for varying numbers of sources in the $G_{n,m}$ graph class. Table 6.1 indicates that PR overtakes DIJ for a modest number of sources and Figure (j) indicates that the precomputation time quickly becomes negligible as the number of sources increases. In Figure (j), the line PR-i represents the amortized cost per SSSP computation (including precomputation), over i SSSP computations.

Figures (a) and (b) show the marginal performance of the PR algorithm to be stable over the uniform and log-uniform distributions. What is somewhat surprising is that Dijkstra's algorithm is significantly faster under the log-uniform distribution. (We hypothesize that this effect is due to the pairing heap. Under the log-uniform distribution, the next vertex visited, that is, the next extracted from the pairing heap, is typically one recently inserted into the heap. Iacono [112] has shown that the pairing heap naturally exploits these correlations.)

Figure (d), on geometric graphs, still shows the marginal cost of PR to be faster than Dijkstra on all graphs tested, though the separation in running times is not as dramatic as in $G_{n,m}$. We believe this is largely due to the density of the geometric graphs, which is about 7/2. Every edge is relaxed exactly once (in both PR and DIJ) but Dijkstra's algorithm is slightly faster in this regard. Also, the pairing heap in Dijkstra's algorithm usually contains $O(\sqrt{n})$ vertices, which should theoretically boost its speed.

Figures (e),(f),(g), and (h) suggest that the PR algorithm's performance is less affected by the shape of the hierarchy than Dijkstra's algorithm. Notice that Dijkstra's algorithm runs about 40 per cent faster on bullseye graphs with 2^{21} vertices and $o = 100$, as compared to bullseye with $o = 25$ or either of the hierarchical classes. We have no

	$G_{n,m}$ (uniform)	$G_{n,m}$ (log-uni.)	Sparse	Geometric
s_0	3	21	3	17
Hier/PR	5.05	11.75	4.38	8.48
PR/BFS	2.14	2.11	1.99	2.77

	Hier. $b = 6$	Hier. $b = 10$	Bull. $o = 25$	Bull. $o = 100$	Square Grid
s_0	4	4	4	7	10
Hier/PR	10.1	9.6	5.33	6.11	7.85
PR/BFS	1.87	1.92	2.15	2.01	2.77

Table 6.1: First line: number of SSSP computations (s_0) beyond which PR (including cost of computing a hierarchy) outperforms Dijkstra. Second line: ratio of time to construct the hierarchy to the marginal-cost of computing SSSP with PR algorithm. Third line: ratio of the marginal cost of computing SSSP with PR to time for one BFS computation. These statistics reflect graphs of 2^{20} vertices.

explanation for this.

The timings for square grids are given in Figure (i). Dijkstra's algorithm performs slightly better than average (likely due to the small average heap size) and the PR algorithm performed about as expected.

The results on the SUN Ultra 60, the Linux PC and the HP/UX machines are similar (see Figure (k) for a comparison of runs on $G_{n,m}$ with uniform distribution of edge lengths), except that the runs are proportionally faster on the Linux PC and slower on the HP machine. The Linux PC was also more sensitive to the representation of edge-lengths, whether integers, or double or quadruple precision floating points — see Figure (l). In contrast the results on the SUN machines were virtually the same for integers and double precision floating points. (Of course, because DIJ and PR were written for the comparison-addition model they needed no alterations; changing the numerical type of edge-lengths affected just one line of code.)

Figure (m) charts the performance of the PR and DIJ algorithms on graphs with a *fixed* number of vertices, 100, 000, as the graph density is increased from 1.5 to 19. The graph class here is $G_{n,m}$. As expected, Dijkstra's algorithm converges to a linear running time as the density increases. Still, the marginal cost of PR is superior for even relatively dense graphs.

Finally, in Figure (n) we plot the number of comparison/addition operations made by the algorithms, on $G_{n,m}$ under uniform and log-uniform edge-length distributions.

6.5 Discussion

The most surprising result of this chapter is that — in the undirected shortest path problem at least — very little *speed* needs to be sacrificed in order to adopt a *general* model of computation, namely the comparison-addition model. In our experiments the Pettie-Ramachandran algorithm always performed less than 2.77 times BFS speed, and usually less than 2.15 times BFS. Contrast this with a heuristic shortest path algorithm of Goldberg [88] (which we note was the result of much tinkering [90, 33, 92, 34, 89, 88]) that performed less than 2.5 times BFS and usually less than 2 times BFS. In this case, it seems the *price of generality* is roughly 7–10 per cent of the running time. We could not ask for much more.

Part II

Minimum Spanning Trees

Chapter 7

Introduction to Minimum Spanning Trees

7.1 History

The minimum spanning tree problem (MST) is rare among computer science problems in that it was studied long before the field of computer science itself. Chazelle [28] speculated that "it may be the oldest open problem in computer science" and Nešetřil [167] has called it "a cornerstone problem of combinatorial optimization and in a sense its cradle." The problem, as it is typically stated, is to find a minimum-weight tree spanning the vertices of a given weighted undirected graph.

Otakar Borůvka [21, 22] is widely regarded as the originator of the abstract MST problem and the inventor of the first MST algorithm, published in 1926. However, even this attribution is up for debate. As we will see, *attribution* is a tricky issue for the MST historian since most of the fundamental algorithms have been discovered and rediscovered, in a few cases within mere *months* of each other. For instance, Borůvka's algorithm was rediscovered by Choquet [38] in 1938 and Florek et al. [63] in 1951, and even as late as 1965 — long after Borůvka's paper was well known — the textbook of Berge and Ghouila-Houri [18] attributes the algorithm to Sollin [183]. Florek et al. [63] traced their algorithm back to a greedy clustering heuristic of Czekanowski [49] published in 1909. Czekanowski's method can be used to group related objects (humanoid skulls in his case), given a matrix of their pairwise similarities. Czekanowski apparently did not think of his heuristic as exactly solving the minimum spanning tree problem, but of approximately solving a phylogeny problem; see [49, 94].

In response to Borůvka's admittedly clumsy exposition [21, 22],[1] Jarník [115]

[1]One must bear in mind that the MST problem predates modern graph theory. Borůvka [21] managed to define and solve the problem using only notation for matrices, sets, and sequences; no mention is made of graphs, trees, paths, vertices, edges, or any other useful concepts.

proposed a simplified MST algorithm that was inexplicably ignored for years. Jarník's algorithm was rediscovered independently by Prim [173], in 1957 then Dijkstra [52] two years later.[2] Both Prim and Dijkstra published their algorithms as practical, space-efficient alternatives to Kruskal's MST algorithm [145], published in 1956.

Kruskal's algorithm was, itself, independently discovered a few months later by Loberman and Weinberger [153], and generalizations of Kruskal's algorithm were studied by Kalaba [123], Kotzig [143], Rosenstiehl [176], Dijkstra [53], and Guan [159]. The Kruskal-like algorithms are distinguished from all other minimum spanning tree algorithms in that they solve the more general *matroid optimization* problem — see [58, 149, 47].

Running Time

Early research on the MST problem focused on deriving *simple* and *space-efficient* algorithms. Rarely, if ever, is *running time* even mentioned as a statistic of interest. If the algorithms of Borůvka [22], Kruskal [145], Prim [173], and Dijkstra [52] were to be implemented as stated, Kruskal's algorithm would run in $O(n^3)$ time and the others in $O(n^2)$ time, where n is the number of vertices. One should bear in mind that Kruskal's algorithm (which identifies MST edges in sorted order) predated Hoare's quicksort [108] by 5 years, and that Williams's [205] binary heap was invented several years after the publications of Prim and Dijkstra.

In the 1970s there was a definite shift in the goals of minimum spanning tree research, from designing simple, provably correct algorithms to designing *asymptotically fast* ones, which were also correct and hopefully simple. Using straightforward techniques, such as sorting and binary heaps, the classical MST algorithms could all be implemented in $O(m \log n)$ time, where m is the number of graph edges. Many observed that Kruskal's algorithm could be made to run in near-linear time if the edge-weights were presented in sorted order — or sortable in linear time, say, with radix-sort. From a practical perspective this may sound like good news, since edge-weights can, indeed, frequently be sorted in linear time. However it is a thorn in the side of the designer of minimum spanning tree algorithms. Any result claiming a *faster* MST algorithm must be predicated explicitly on a model of computation *incapable* of sorting in linear time (or at least one for which linear-time sorting is non-trivial [75].) Since the 1970s a mainstay in the MST problem has been the assumption that edge-weights may be compared in constant time, but are otherwise uninspectable.[3]

[2]However, Dijkstra [55] claims to have coded his algorithm as early as 1956.

[3]This model is justified on several grounds: as aesthetically pleasing, abstract, minimal, conceptually simple, and widely applicable. It also makes the MST problem harder and keeps the algorithm designers in business.

In 1975 Yao [206] published the first improvement[4] over the classical algorithms. Yao showed how to implement Borůvka's algorithm in $O(m \log\log n)$ time using the recently developed linear-time selection algorithm of Blum et al. [19]. In the same year Johnson [118] gave an $O(m \log_d n)$ time implementation of the Jarník/Prim/Dijkstra algorithm, where $d = \max\{\frac{m}{n}, 2\}$ is the *density* of the graph. Johnson's algorithm was based on the d-heap, a generalization of Williams's binary heap [205] that gave a tradeoff between insert/delete operations and decrease-keys. Cheriton and Tarjan [32] gave a more sophisticated implementation of Borůvka's algorithm running in time $O(m \log\log_d n)$ time. Their algorithm was based on the mergeable leftist heaps of Crane [48] (see also Knuth [139]). In 1984 Fredman and Tarjan [73] invented the Fibonacci heap, which was the key component in their $O(m\log^*(m,n))$-time MST algorithm.[5] By making thriftier use of the Fibonacci heap, Gabow et al. [78] produced an MST algorithm running in time $O(m \log\log^*(m,n))$.

In retrospect the algorithm of Gabow et al. [78] was the last result of an era of research on the MST problem. It, like every MST algorithm that preceded it, could be viewed as merely instantiating a certain generalized *greedy* method. Unlike the greedy algorithms, which single-mindedly construct the MST one edge at a time, all recent MST algorithms take a *non-greedy* tack. Basically, the idea is to quickly construct an *approximately* minimum spanning tree, whose function is to rule out non-MST edges and thereby make the search for MST edges less costly.

In 1994 Karger, Klein, and Tarjan [138, 127] presented a *randomized* expected linear time MST algorithm, settling the randomized complexity of the problem. In short, their algorithm recursively finds the minimum spanning tree of a sampled graph (thus, an approximate MST), then uses Komlós's [141] linear-time MST verification routine to filter out certifiably non-MST edges. In 1997 Chazelle [27] developed a *deterministic* $O(m\alpha \log \alpha)$-time MST algorithm using new techniques that were fundamentally different than those of Karger et al. [127]. Here, $\alpha = \alpha(m, n)$ is the inverse-Ackermann function. Chazelle's algorithm had, as its basis, the Soft Heap [29], a data structure that circumvented the information-theoretic barriers of traditional heaps by allowing certain well-behaved *errors*. Chazelle [28] later reduced the complexity of his algorithm to $O(m\alpha(m, n))$; this improvement was also noted by Pettie [169] the same year.

Remark. Our historical account of the MST problem is based on a number of sources. Graham and Hell [94] survey the history of the problem up until the invention of Fibonacci heaps [73] in 1984, and take great care in considering the contributions of Czekenowski to the MST problem. A complementary paper by Nešetřil [167] surveys all developments on the problem through 1996. Nešetřil, Milková, and Nešetřilová [168]

[4]Yao cites an earlier, unpublished algorithm of Tarjan running in $O(m\sqrt{\log n})$ time.
[5]Here, $\log^*(m, n) = 1 + \log^* n - \log^* \frac{m}{n}$.

have done a great service in providing English translation of Borůvka's 1926 papers [21, 22]. Korte and Nešetřil [142] survey Jarník's contribution to graph theory and give partial translation of two papers, one on the MST problem [115] and another on what would be called today the generalized Steiner tree problem [116].[6] Our account of the MST problem also incorporates various anecdotes of Johnson [118], Tarjan [195], Kruskal [146], and Dijkstra [55].

Our Contributions

In Chapter 8 we present the first deterministic provably optimal minimum spanning tree algorithm, whose complexity happens to be unknown at this time. In particular we show that our algorithm's running time is asymptotically equivalent to the *decision-tree* complexity of the minimum spanning tree problem itself, which is enough to establish optimality. Our algorithm, like Chazelle's [28], is based on the Soft Heap [29].

The problem of producing an *explicit*, deterministic, and linear-time MST algorithm is still open. Chazelle [28] speculated that what is needed is a more flexible version of Komlós's MST verification algorithm (which is the key subroutine of Karger et al.'s randomized linear-time MST algorithm [127].) In Chapter 9 we prove an inverse-Ackermann type lower bound on any data structure for the *online* MST verification problem. Thus, such a data structure cannot be at the heart of a faster MST algorithm.

In Chapter 10 we present a *parallel* randomized MST algorithm that runs in logarithmic time and linear work[7] on the EREW PRAM [130]. It is provably time-work optimal and is the first such optimal parallel MST algorithm for any parallel model.

One major disadvantage of the randomized MST algorithms, both sequential [127] and parallel [40, 41, 172, 175] [Chapter 10], is that they use a linear number of random bits. In Section 8.6 we show that it is possible to compute an MST in expected linear time with only $o(\log^* n)$ random bits. The algorithm is a combination of our optimal MST algorithm [Chapter 8] and Karger et al.'s randomized MST algorithm [127]. Unlike Karger et al.'s algorithm, ours does not seem to be readily parallelizable. In Chapter 11 we develop a simple, parallelizable, randomness-efficient, expected linear-time MST algorithm. It is based on a different sampling strategy than the Karger et al. algorithm [127], and is designed to work well with a pair-wise independent sampler. It consumes a polylogarithmic number of random bits.

The remainder of this chapter is a primer on the minimum spanning tree problem. Before skipping it, we recommend that the reader get acquainted with our terminology and notation.

[6]As well known mathematicians go, Jarník is remarkably well ignored. He was apparently the first to formulate and analyze the generalized Steiner tree problem [115, 142].

[7]Work is defined as the time-processor product, i.e., the total number of operations.

7.2 Preliminaries

The input is a weighted graph $G = (V, E, w)$ where, as is usual, $|V| = n$, $|E| = m$, and $w : E \longrightarrow \mathbb{R}$ assigns a *weight* to each edge. We assume that the graph is *connected*, implying that $m \geq n - 1$, and that edge-weights are unique. Uniqueness can always be forced by ordering the vertices, then ordering identically weighted edges by their endpoints. Connectedness is also not a restrictive assumption.

A *minimum spanning tree* of G, denoted $MST(G)$, is a tree spanning the vertices of G whose total weight is minimum. (*Total weight* is a bit of a red herring since edge-weights need never be added to determine the MST.) The cut and cycle properties, given in Section 7.2.2, imply that the MST is unique if the edge-weights are unique.

7.2.1 The Model

Our core assumption is that edge-weights may be compared in constant time, and that no other operations are allowed on edge-weights. This assumption seems to dwarf all others in importance. For the sake of specificity, we assume that computation unrelated to edge-weights is handled by a pointer machine[8] [193]. We find the pointer machine appealing because it models the way actual computers work, without making *any* questionable assumptions. Thus as a model for proving *upper bounds* it is essentially beyond reproach. (However one cannot claim with conviction that a lower bound in the pointer machine model holds for all realistic machines.)

7.2.2 Cut and Cycle Properties

The *cut* and *cycle* properties of minimum spanning trees form the basis of every MST algorithm's proof of correctness. They are simple to state and to prove.

The Cut Property

The cut property states that for any set of vertices X, the lightest edge crossing the cut $(X, V - X)$ is in the MST. Suppose $e = (u, v)$ was this edge, but it was not in the spanning tree T. We prove the cut property by demonstrating that $T \neq MST$. Consider

[8]For the reader unfamiliar with a pointer machine, it works as follows. The memory consists of a collection of *records*, each containing a constant number of *data words* and a constant number of references to other records, or *pointers*. Records may only be modified when they reside in one of the machine's registers, of which there are a constant number. Data words are subject to the usual array of operations, numerical and logical. Pointers are subject to only three operations: assignment, dereference, and tests for equality. In particular a pointer (in a register) can be assigned the value of another pointer or to the null pointer. Given two pointers the program may branch depending on their (in)equality. Lastly, given a pointer (in a register), the record it references can be moved into any register in constant time. What is specifically disallowed is what is sometimes called *pointer arithmetic* (adding an integer to a pointer to yield another pointer).

the path P in T connecting u and v. An odd number of edges from P cross the cut $(X, V - X)$; let e' be any one of them. Since edge-weights are distinct, $w(e') > w(e)$. Thus $T - \{e'\} \cup \{e\}$ is a tree strictly lighter than T.

The Cycle Property

The cycle property states that any edge that is the heaviest on some cycle is not in the MST. To see this, consider any edge e that is the heaviest edge on the cycle $\{e\} \cup P$. If e appears in an arbitrary spanning tree T, we show that $T \neq MST$. $T - \{e\}$ has two connected components, defining a cut $(X, V - X)$ where X is some set of vertices. The path P has an odd number of edges crossing $(X, V - X)$; let e' be one of them. Since $w(e') < w(e)$, the tree $T - \{e\} \cup \{e'\}$ is strictly lighter than T.

It is not difficult to show that the cut and cycle properties are complete; that is, the only way to prove an edge is (or is not) in the MST is to show that it is the lightest crossing some cut, or the heaviest on some cycle.

Contraction and Contractibility

The cut and cycle properties can be used to prove more complicated properties of minimum spanning trees. A good example, and one we will use in Chapter 8, is the property of *contractibility* — see Definition 5. Let C be a set of edges. The notation $G \backslash C$ represents the graph derived from G by contracting C, i.e., replacing the connected components of C by single vertices and reassigning edge endpoints appropriately. We assume that self-loops (edges of the form (v, v)) introduced by contraction are removed, but that parallel edges (multiple edges with the same endpoints) are not.[9]

Definition 5 *A set of induced subgraphs C is* contractible *if every subgraph in C is contractible. An induced subgraph C is* contractible *if for any two edges (u, u') and (v, v'), with $u, v \in V(C)$ and $u', v' \notin V(C)$, there exists a path P from u to v in C such that*

$$\max\{w(u, u'), w(v, v')\} > \max_{e \in P} w(e)$$

Lemma 33 *If C is contractible with respect to G, then:*

$$MST(G) = MST(C) \cup MST(G \backslash C)$$

[9]At times it seems as though the chief operation performed by MST algorithms is contraction. We advise the reader to think of *edges* as simply being distinct objects with *original* edge-weights and *original* endpoints in the input graph. Thus, if a graph G' is derived by contraction, it does not contain *new* edges *associated* with those of G, but the original edges themselves, possibly under the guise of new endpoints and new edge-weights. This deliberate muddling of notation will cause no confusion. Trust us! We're algorithmicists!

Proof: One can easily see that if the Lemma holds when C is a single induced subgraph, it follows that the Lemma holds for multiple induced subgraphs. Therefore we assume C to be a single subgraph.

Clearly $MST(C) \cup MST(G \backslash C)$ is a spanning tree of G. Therefore, we must only prove two things: (a) edges in $E(C) - MST(C)$ are not in $MST(G)$, and (b) edges in $E(G \backslash C) - MST(G \backslash C)$ are also not in $MST(G)$. Since, by the definition of contractibility, C is an *induced* subgraph, (a) and (b) account for all non-MST edges. Consider Part (a). By the cycle property every edge in $E(C) - MST(C)$ was the heaviest on some cycle in C. Since those cycles exist in G as well, edges not in $MST(C)$ are also not in $MST(G)$. Turning to Part (b), let $e \in (E(G \backslash C) - MST(G \backslash C))$ and let Q be the cycle in $G \backslash C$ certifying this fact. If Q is also a cycle in G then by the cycle property $e \notin MST(G)$. The interesting case occurs when Q contains the vertex derived by contracting C. In this case Q corresponds to a *path* in G with end-edges (u, u') and (v, v') with $u, v \in V(C)$. By the contractibility of C there exists a path P in C connecting u to v such that all edges in P are lighter than either (u, u') or (v, v') — see Figure 7.1. Furthermore, e is the heaviest edge in Q, which includes (u, u') and (v, v'). Thus e is the heaviest edge on the path $P \cup Q$ and, by the cycle property, not in $MST(G)$.

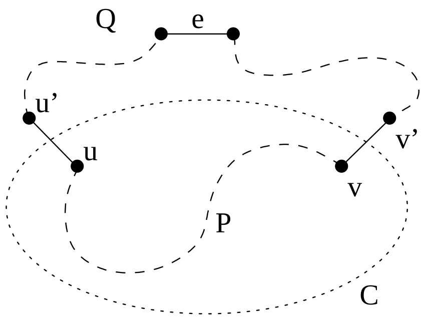


Figure 7.1: C is a contractible subgraph, Q is a cycle in $G \backslash C$ (that includes the vertex derived from C), e is the heaviest edge in Q, and P is the path in C guaranteed by its contractibility.

□

In Chapter 8 we will consider graphs that are weighted differently but are otherwise structurally identical. Thus a subgraph C may be contractible *w.r.t.* $G = (V, E, w)$ though not *w.r.t.* $G' = (V, E, w')$ if the weight functions w and w' differ.

7.2.3 Basic Algorithms

The classic MST algorithms proceed in simple and discrete steps, making, in each step, tangible progress towards the minimum spanning tree. For this reason they are called *greedy* algorithms.

Borůvka's Algorithm

Borůvka's algorithm [21] consists of a sequence of *Borůvka steps.* In each step, the lightest edge incident on each vertex is identified. Because edge weights are unique, these edges form a set of at most $\lfloor n/2 \rfloor$ trees. Let T be one such tree. It follows from the cut property that $T \subseteq MST(G)$ since the edges incident on any vertex form a cut set. Furthermore one can easily show the subgraph induced by T is contractible. Borůvka's algorithm is then straightforward. First, identify the lightest edge incident on each vertex in G. Call the set of such edges F. Recursively apply Borůvka's algorithm to $G \backslash F$ and return $F \cup MST(G \backslash F)$, which is, by Lemma 33, equal to $MST(G)$.

The important properties of Borůvka's algorithm are (1) each Borůvka step can be implemented in linear time, and (2) after i Borůvka steps the resulting graph has at most $n/2^i$ vertices. Thus Borůvka's *algorithm* can take as much as $\Omega(m \log n)$ time to run, but executing a few Borůvka *steps* is inexpensive and potentially useful.

The Dijkstra-Jarník-Prim Algorithm

We will refer to the algorithm discovered by Jarník[115], Prim [173], and Dijkstra [52] as the *DJP algorithm.* Our initial aim with the term DJP was to somehow straddle the borders between orthodoxy, inclusiveness, and fairness, but such a compromise is clearly impossible. 'DJP' is not orthodox (it is universally known as Prim's algorithm), nor is it particularly fair to Jarník, who had the algorithm decades before the others. However, it is inclusive!

The state of the DJP algorithm is represented by a cut $(X, V - X)$, which is initially $(\{v\}, V - \{v\})$ for some arbitrary vertex v. In each step DJP identifies the lightest edge (u, u') crossing the cut $(X, V - X)$, where $u \in X$, then sets $X := X \cup \{u'\}$. By the cut property $(u, u') \in MST(G)$. Thus, after exactly $n - 1$ steps of DJP the entire MST has been found. Using a Fibonacci heap [73] the DJP algorithm can be implemented to run in $O(m + n \log n)$ time.

Lemma 34 is easily proved by induction. We leave the proof to the reader.

Lemma 34 *Let $(X, V - X)$ be the current cut after any number of steps of the DJP algorithm. Then the subgraph induced by X is contractible.*

Chapter 8

An Optimal Minimum Spanning Tree Algorithm

Disclaimer

The title of this chapter has promised you, the reader, a minimum spanning tree algorithm with provably optimal performance, and with it, a certain amount of understanding about a fundamental mathematical problem. However, you will likely have no feeling of serene resolution as this brand of optimality can cause restlessness and frustration.[1]

8.1 Introduction

We give, as promised, a deterministic MST algorithm that is provably optimal. The catch is that its running time is unknown, and moreover, the algorithm itself betrays nothing interesting about its running time *except that it is optimal.* To state our result more formally, let $\textsc{mst}(m, n)$ be the complexity (worst case running time) of the optimal MST algorithm on m-edge, n-node graphs.[2] and let $\textsc{mst}^*(m, n)$ be the decision-tree complexity (number of edge-weight comparisons) of the MST problem. Because edge-weight comparisons are but one of many types of operations counted by the MST measure, we have:

$$\textsc{mst}^*(m, n) \leq \textsc{mst}(m, n)$$

In this chapter we give a constructive proof that:

[1]The results of this chapter are taken largely from: S. Pettie and V. Ramachandran, An optimal minimum spanning tree algorithm, J. ACM 49, pp. 16–34, 2002. The algorithm from Section 8.6 is from: S. Pettie and V. Ramachandran, Minimizing randomness in minimum spanning tree, parallel connectivity, and set maxima algorithms, Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 713–722, 2002.

[2]For the sake of specificity, assume MST is w.r.t. the pointer machine model of computation [193].

$$\textsc{mst}(m,n) = O(\textsc{mst}^*(m,n))$$

Moreover, our optimal MST algorithm is quite simple and does not sweep any large constants under *big-Oh's* carpet.

At the moment nothing is known about the non-uniform complexity of MST ($\textsc{mst}^*(m,n)$) that is not also true of its uniform complexity ($\textsc{mst}(m,n)$). Chazelle's algorithm [28] demonstrates that $\textsc{mst}^*(m,n) = O(m\alpha(m,n))$ and $\textsc{mst}^*(m,n)$ is trivially at least linear in m.

To anyone not immersed in certain areas of computer science, a statement like $\textsc{mst}(m,n) = \Theta(\textsc{mst}^*(m,n))$ is uninformative. Is uniform complexity not expected to be equal to non-uniform complexity? Are there historical precedents of natural problems with a uniform/non-uniform gap? And concerning the MST problem, is there reason to believe $\textsc{mst}^*(m,n)$ is linear or superlinear? If it is superlinear, is $\Theta(m\alpha(m,n))$ the only *natural* bound, or are there other plausible candidates that lie in $o(m\alpha(m,n))$? These questions are mostly a matter of opinion; therefore they can only be discussed, not answered.

8.1.1 Uniform vs. Non-Uniform Complexity

For now, let us just consider those problems whose input consists of real numbers. The non-uniform (or algebraic) complexity of such a problem is the number of real-number operations[3] required to deduce a solution, i.e., the cost of deciding which operations to perform is not counted. The uniform complexity is the running time of the fastest program solving the problem, where the program must be the same for all input sizes.

We are unaware of any really natural problems with interesting uniform/non-uniform complexity gaps, though there are both unnatural and uninteresting ones. An example of an unnatural one is the *weighted Hamiltonian cycle* problem: report the weight of a Hamiltonian cycle if one exists, and ∞ otherwise. The problem is NP-complete yet the solution can be found with $O(n)$ real-number operations. A natural problem with an uninteresting complexity gap is *set maxima*, because in general, the decision-tree complexity of set maxima [86] is sub-linear in the input size, which is a lower bound on any uniform algorithm.

We can name two fundamental problems for which complexity gaps are unproven but very plausible: *Erdős-Szekeres partitioning* and *all-pairs shortest paths.*

[3]Which real-number operations are available depends on the problem; it could be comparisons, comparisons and additions, or some other set.

Erdős-Szekeres Partitioning

A well known theorem of Erdős and Szekeres [60] implies that in any sequence of N weighted elements, there is a monotonically increasing or decreasing sub-sequence of $\sqrt{N}$ elements. The algorithmic problem here is to partition any sequence into $O(\sqrt{N})$ disjoint monotonic sub-sequences. Fredman [68] and Dijkstra [54] gave simple algorithms for finding the longest monotonic sub-sequence in $O(N \log N)$ time, which gives an $O(N^{1.5} \log N)$-time partitioning algorithm. Bar-Yehuda and Fogel [15] recently gave an $O(N^{1.5})$-time partitioning algorithm, which is the best to date. Note that the decision-tree complexity of partitioning is trivially $O(N \log N)$: after sorting the elements by weight, no more real-number operations are necessary. Therefore, the current gap stands at $\sqrt{N}/\log N$.

All-Pairs Shortest Paths

It is well known that when the weighted input graph is represented as matrix (and thus has size $\Theta(n^2)$) APSP has the same complexity, asymptotically, as min-plus matrix multiplication [4]. In 1976 Fredman [69] gave a min-plus multiplication algorithm performing $O(n^{2.5})$ real-number operations. However, the best implementations of his algorithm are better than $O(n^3)$ by only sub-logarithmic factors. Thus, the current gap for APSP is $\omega(\sqrt{n}/\log(n))$.

The lesson to be learned from these examples is that there are techniques suited for efficient non-uniform algorithms which do not seem to have any efficient realization. Unless there is a proof to the contrary, we would never assume that a problem's algebraic complexity is equal to — or even close to — its algorithmic complexity. Unfortunately, it does not seem possible to prove a significant algorithmic/algebraic complexity gap with the existing machinery.

8.1.2 Speculation about MST*

There are three possible scenarios:

(a) $\text{MST}^*(m,n) = \Theta(m)$

(b) $\text{MST}^*(m,n) = \omega(m)$ and $o(m\alpha(m,n))$

(c) $\text{MST}^*(m,n) = \Theta(m\alpha(m,n))$

All the smart money seems to favor (a), for reasons that are historical and aesthetic. Karger, Klein, and Tarjan's expected linear-time MST algorithm [127] suggests that $\text{MST}^*(m,n) = O(m)$, if only for the non-existence of probabilistic/deterministic complexity gaps in similar problems. One could support scenario (c) because the $\alpha(m,n)$

function appears naturally in other problems. However, inverse-Ackermann type lower bounds are typically only found in online problems such as Union-Find [193, 71] (see also Chapter 9), geometry-related problems [104, 204, 135] or certain restrictive algebraic problems [31, 191]. Thus, scenario (c) would set a new precedent, as would scenario (b) since, to our knowledge, there is no known problem with superlinear complexity that is $o(m\alpha(m,n))$. (Our split-findmin data structure is an interesting candidate; it has complexity $O(m \log \alpha(m,n))$ but this is not known to be tight.)

8.2 The Soft Heap

The primary data structure of our algorithm is Chazelle's *Soft Heap* [29]. The Soft Heap is a kind of priority queue that gives us an optimal tradeoff between accuracy and speed. It is parameterized by an error tolerance ϵ, and supports the standard priority queue operations:

MakeHeap(): returns an empty soft heap.

Insert(H, x, κ)**:** insert item x with key κ into H.

Findmin(H)**:** returns item with minimum key in heap H.

Delete(H, x)**:** delete x from heap H.

Meld(H_1, H_2)**:** create new heap containing the union of items stored in H_1 and H_2, destroying H_1 and H_2 in the process.

Although this description is identical to a normal priority queue, we no longer assume the immutability of keys. An item whose key has been increased is said to be *corrupted*, and once corrupted, a key never decreases in value. Thus, a Findmin operation only returns an item whose *current* key value is minimum, or, in other words, an item whose original key is less than the keys of all uncorrupted items. The proof of Lemma 35 is due to Chazelle [29].

Lemma 35 *Fix a parameter $0 < \epsilon < 1/2$, and consider a mixed sequence of operations that includes N inserts. On a Soft Heap, the amortized complexity of each operation is constant, except for insert, which takes amortized time $O(\log \epsilon^{-1})$. At most ϵN items are corrupted at any time.*

Note that Lemma 35 does not imply that ϵN items are corrupted in total. By its definition, the Soft Heap is free to corrupt any one item upon the deletion of a corrupted item. Indeed, we would expect nearly all items to become corrupted eventually.

8.3 Corruption and Contractibility

8.3.1 A Robust Contraction Lemma

In our algorithm we will compute portions of the minimum spanning tree of a *corrupted* graph, where some of the edge weights have been increased due to the use of a Soft Heap. Although a corrupted MST is of little value by itself (it may share no edges with the actual MST), we show, in Lemma 36, that it may still be possible to obtain useful information from a corrupted graph.

Let $G \Uparrow M$ be a graph derived from G by corrupting (increasing) the weights of all edges in $M \subseteq E(G)$. We use the term *H-weight* to refer to an edge's weight in H. If H is omitted, it is assumed we are talking about the original, uncorrupted graph G.

Lemma 36 *Let C be a subgraph of G, $M \subseteq E(G)$, and $M_C \subseteq M$ be those edges of M with one endpoint in C. If C is contractible w.r.t. $G \Uparrow M$, then:*

$$MST(G) \subseteq MST(C) \cup MST(G \backslash C - M_C) \cup M_C$$

Proof: The only edges claimed not to be in $MST(G)$ are those in $E(C) - MST(C)$ and those in $E(G\backslash C) - MST(G\backslash C - M_C)$. By the cycle property, any edge e not in $MST(C)$ or $MST(G\backslash C - M_C)$ is the heaviest on some cycle Q (in either C or $G\backslash C - M_C$). If Q corresponds to a cycle in G then $e \notin MST(G)$. Therefore, the only interesting case is when Q does not correspond to a cycle in G, i.e., when it contains the vertex derived by contracting C in $G\backslash C - M_C$. In this case Q corresponds to a path in G with end-edges (u, u') and (v, v') where $u, v \in V(C)$. By the contractibility of C, there exists a path P in C connecting u to v such that the $(G \Uparrow M)$-weights of all edges in P are lighter than the $(G \Uparrow M)$-weight of either (u, u') or (v, v'). Notice that both (u, u') and (v, v') are uncorrupted — if they were in M they'd be in M_C as well.

Let us summarize the facts. In terms of G-weight, e is the heaviest edge in Q. The G-weight of either (u, u') or (v, v') is larger than the $(G \Uparrow M)$-weight of all edges in P. Furthermore, $(G \Uparrow M)$-weight is an upper bound on G-weight. Thus, in terms of G-weight, e is the heaviest edge on the cycle $P \cup Q$, and therefore cannot be in $MST(G)$.

$\square$

A more complicated version of Lemma 36 is used (implicitly) in the other Soft Heap-based MST algorithms. For technical reasons, the [27, 28, 169] algorithms must artificially retain certain corrupted edges in their corrupted state, rather than discarding them altogether.

8.3.2 The Partition Procedure

Lemma 34 claims that if the DJP algorithm is run for any number of steps, the resulting subgraph is contractible. If, instead of a normal priority queue, we implement the DJP algorithm with a Soft Heap, it will produce a subgraph contractible with respect to a certain corrupted graph. Lemma 36 then allows us to claim certain things about the real minimum spanning tree of the original, uncorrupted graph.

Our Partition procedure finds a sequence of contractible subgraphs $C_1, \ldots, C_k$ in a graph whose edges are being corrupted by the Soft Heap. Let M be the set of corrupted edges, and $M_{C_i} \subseteq M$ be just those incident on C_i. Partition guarantees that C_i is contractible w.r.t.

$$(G \Uparrow M) \backslash (\bigcup_{j=1}^{i-1} C_j) \; - \; \bigcup_{j=1}^{i-1} M_{C_j}$$

By repeated application of Lemma 36, it follows that when C_i is complete:

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

The Partition procedure is given in Figure 8.1. The arguments appearing before the semicolon are inputs; the others are outputs. $\mathcal{C}=\{C_1, \ldots, C_k\}$ is a set of subgraphs of G, and $M_{\mathcal{C}}$ is a set of corrupted edges with endpoints in different C_i's. No edge will appear in more than one of $M_{\mathcal{C}}, C_1, \ldots, C_k$.

Initially, Partition sets every vertex to be *live*. The objective is to convert each vertex to *dead*, signifying that it is part of a component C_i with $\leq \tau$ vertices and part of a *conglomerate* of $\geq \tau$ vertices, where a conglomerate is a connected component of the graph $\bigcup E(C_i)$. Intuitively a conglomerate is a collection of C_i's linked by common vertices. This scheme for growing components is very similar to that of Fredman and Tarjan [73] who used a Fibonacci heap rather than a Soft Heap.

We grow the C_i's one at a time using the DJP algorithm, where the weight of an edge may be increased at the whim of the Soft Heap. A component is done growing if it spans τ vertices or if it attaches itself to an existing component. Clearly, if a component does not reach τ vertices it must be part of a conglomerate of at least τ vertices. Hence, all its vertices may be designated *dead*. Upon completion of a component C_i, we discard M_{C_i}, the set of corrupted edges incident to C_i.

The running time of Partition is dominated by the heap operations, and hence depends on ϵ. Each edge is inserted into a Soft Heap no more than twice (once for each endpoint), and extracted no more than once. We can charge the cost of dismantling the heap to the insert operations that created it, hence the total running time is $O(m \log(\frac{1}{\epsilon}))$. By Lemma 35, the number of discarded edges is bounded by the number of insertions

```
Partition($G, \tau, \epsilon$ ; $M_{\mathcal{C}}, \mathcal{C}$)
    All vertices are initially live
    $M_{\mathcal{C}}$ := $\emptyset$
    $i := 0$
    While there is a live vertex
        Increment $i$
        Let $V_i := \{v\}$, where $v$ is any live vertex
        Create a Soft Heap consisting of $v$'s edges (uses $\epsilon$)
        While all vertices in $V_i$ are live and $|V_i| < \tau$
            Repeat
                Find and delete min-weight edge $(x, y)$ from Soft Heap
                W.l.o.g., assume $x \in V_i$
            Until $y \notin V_i$
            $V_i := V_i \cup \{y\}$
            If $y$ is live then insert each of $y$'s edges into the Soft Heap
        Set all vertices in $V_i$ to be dead
        Let $M_{V_i}$ be the corrupted edges with one endpoint in $V_i$
        $M_{\mathcal{C}} := M_{\mathcal{C}} \cup M_{V_i}$
        $G := G - M_{V_i}$
        Dismantle the Soft Heap
    Let $\mathcal{C}$ := $\{C_1, \ldots, C_i\}$ where $C_z$ is the subgraph of $G$ induced by $V_z$
    Return $M_{\mathcal{C}}$ and $\mathcal{C}$.
```

Figure 8.1: The Partition Procedure.

scaled by ϵ, thus $|M_{\mathcal{C}}| \leq 2\epsilon m$. The relevant properties of Partition are summarized in Lemma 37.

Lemma 37 *Given a graph G, and parameters $0 < \epsilon < \frac{1}{2}$, and $\tau \geq 1$, Partition finds edge-disjoint subgraphs $M_{\mathcal{C}}, C_1, \ldots, C_k$ in time $O(|E(G)| \cdot \log(\frac{1}{\epsilon}))$ while satisfying several conditions:*
a) For all $v \in V(G)$ there is some i s.t. $v \in V(C_i)$.
b) For all i, $|V(C_i)| \leq \tau$.
c) Each connected component (conglomerate) of $\bigcup_i E(C_i)$ spans at least τ vertices.
d) $|E(M_{\mathcal{C}})| \leq 2\epsilon \cdot |E(G)|$.
e) $MST(G) \subseteq \bigcup_i MSF(C_i) \cup MSF(G \backslash (\bigcup_i C_i) - M_{\mathcal{C}}) \cup M_{\mathcal{C}}$.

We observe that Partition never *melds* two Soft Heaps, and in fact, only operates on one Soft Heap at any given time (in contrast to [28]). It is possible to implement Partition with a simplified, space-efficient version of the Soft Heap where the links between heap-nodes are represented implicitly, as in a binary heap.

8.4 The Algorithm

8.4.1 Overview

Here is a brief overview of our minimum spanning tree algorithm.

- In the first stage we find subgraphs $C_1, C_2, \ldots, C_k$ contractible w.r.t. a corrupted graph $G \Uparrow M$. Let $M_\mathcal{C}$ be those corrupted edges connecting distinct C_i's.
- In the second stage we find the MSTs of each C_i and the graph derived by contracting the C_i's. By Lemma 36, $MST(G)$ is contained in $\bigcup_i MST(C_i) \cup MST(G \backslash (\bigcup_i C_i) - M_\mathcal{C}) \cup M_\mathcal{C}$. Note that at this point, no MST edges have been found.
- In the third stage we find some MST edges, using a constant number of Borůvka steps, and recurse on the graph derived by contracting these edges.

The first stage is implemented using the Partition procedure from Section 8.3.2. We implement the second stage using a combination of techniques. We find $\bigcup_i MST(C_i)$ using a set of precomputed, optimal *decision trees*. These are hardwired algorithms designed to compute the MST of a specific graph topology using an optimal number of edge-weight comparisons. In general, decision trees are much larger than the size of the problem that they solve and finding optimal ones is very time consuming. We can afford the cost of precomputing decision trees by guaranteeing that each one is extremely small. At the same time, we guarantee that conglomerates (connected components of $\bigcup_i E(C_i)$) are relatively large. This allows us to determine the MST of $G \backslash (\bigcup_i C_i) - M_\mathcal{C}$ in linear time.

The effect of the first two stages is to reduce the number of edges by a constant factor. The Borůvka steps in stage three give a similar reduction in the number of vertices. Thus, the sequence of recursive calls to our algorithm operate on graphs of geometrically decreasing size.

8.4.2 MST Decision Trees

Any deterministic MST algorithm ultimately makes decisions based solely on the structure of the input graph and what is known about the permutation of edge-weights. If the graph structure is fixed then its decisions are based only on knowledge of the edge-weights. One can easily see that for a fixed graph H, any algorithm computing $MST(H)$ can be modeled as a *decision tree.*

For our purposes, a decision tree is a rooted tree where each node has zero or two children. Each internal node x is associated with an inequality of the form $w(e) < w(f)$, where e and f are edges of H. The left child of x represents that x's inequality is true, and the right that it is false. Each leaf is associated with a spanning tree of H. A decision tree for H is *correct* if the edge-weight inequalities encountered on any path from the root to a leaf uniquely identify the spanning tree at that leaf as the MST. A decision tree for H is *optimal* if it is correct and there exists no correct decision tree for H with lesser depth.

Let us bound the time needed to find optimal decision trees for all graphs on τ vertices by brute force search. There are $2^{\binom{\tau}{2}}$ such graphs and for each graph we must check all possible decision trees bounded by a sufficient depth. Since the DJP algorithm uses no more than $\tau(\tau-1)$ comparisons on any graph on τ vertices, a depth of τ^2 is sufficient. Hence the tree has fewer than 2^{τ^2} internal nodes. There are no more than τ^4 possibilities for each internal node (its inequality names two edges, each naming two vertices) and therefore there are at most $\tau^{2^{\tau^2+2}}$ distinct decision trees to check. To determine if a decision tree is correct we simply test it against an MST algorithm known to be correct, on each of the $|E(H)|!$ possible edge-weight permutations. This process takes no longer than $\tau^2!$ per decision tree. Therefore, the total time required to find an optimal decision tree for all graphs on at most τ vertices is bounded by $2^{\tau^2} \cdot \tau^{2^{\tau^2+2}} \cdot \tau^2!$, which is less than $2^{2^{\tau^2+o(\tau)}}$. Setting $\tau = \log^{(3)} n$ allows us to precompute all optimal decision trees in $o(n)$ time.

In our optimal algorithm we will use a procedure DecisionTree($\mathcal{G}; \mathcal{F}$), which takes as input a set of graphs $\mathcal{G}$, each with at most τ vertices, and returns their minimum spanning trees in $\mathcal{F}$ using the precomputed decision trees.

8.4.3 The Dense Case Algorithm

With the exception of Kruskal's algorithm [145], all existing MST algorithms run in linear time *for sufficiently dense graphs.* We will assume an algorithm called DenseCase that runs in $O(m + n \log^{(3)} n)$ time, that is, linear time for edge density $m/n \geq \log^{(3)} n$. A call to DenseCase(G; F) returns the MST of G in F. Any of the algorithms presented in [73, 78, 27, 29, 169] are suitable for DenseCase; the simplest is easily that of Fredman and Tarjan [73].

We use DenseCase on graphs of $m' \leq m$ edges and $n' \leq n/\log^{(3)} n$ vertices. Hence it runs in time $O(m+n)$: linear in the size of the original graph but not necessarily its input graph.

8.4.4 An Optimal Algorithm

Our minimum spanning tree algorithm is given in Figure 8.2. Before the algorithm proper begins we compute optimal MST decision trees for all graphs on at most $\log^{(3)} n$ vertices. We then call the recursive procedure OptimalMST(G). Using the Partition procedure, OptimalMST finds a set of subgraphs $\mathcal{C} = C_1, \ldots, C_k$ and an edge-set $M_{\mathcal{C}}$

such that every MST edge is contained in one of $\bigcup_i MST(C_i)$, $M_{\mathcal{C}}$, or $MST(G\backslash \bigcup_i C_i - M_{\mathcal{C}})$. The $\bigcup_i MST(C_i)$ are determined using the precomputed optimal decision trees, and $MST(G\backslash \bigcup_i C_i - M_{\mathcal{C}})$ is determined using the DenseCase algorithm. Finally, we apply two Borůvka steps and recurse on the resulting contracted graph. OptimalMST makes calls to a procedure Boruvka2(G; F, G'). It performs two Borůvka steps on G, returns the MST edges found in F, as well as the contracted graph $G' = G\backslash F$.

Constants: $\epsilon = 1/8$ and $\tau = \log^{(3)} n$.

```
OptimalMST($G$)
    If $E(G) = \emptyset$ then Return($\emptyset$)
    Partition($G, \tau, \epsilon$;  $M_{\mathcal{C}}, \mathcal{C}$)
    DecisionTree($\mathcal{C}$; $\mathcal{F}$)
    Let $k := |\mathcal{C}|$ and let $\mathcal{C} = \{C_1, \ldots, C_k\}$, $\mathcal{F} = \{F_1, \ldots, F_k\}$
    $G_a := G\backslash(F_1 \cup \ldots \cup F_k) - M_{\mathcal{C}}$
    DenseCase($G_a$; $F_0$)
    $G_b := F_0 \cup F_1 \cup \ldots \cup F_k \cup M_{\mathcal{C}}$
    Boruvka2($G_b$; $F', G_c$)
    $F$ := OptimalMST($G_c$)
    Return($F \cup F'$)
```

Figure 8.2: An optimal minimum spanning tree algorithm.

Apart from recursive calls and using the decision trees, the computation performed by OptimalMST is clearly linear. Computing decision trees takes $o(n)$ time, Partition takes $O(m \log \epsilon^{-1})$ time, which is $O(m)$, and each Borůvka step takes $O(m)$ time. The number of edges passed to the recursive call is fewer than $2\epsilon m + n/2^b$, where $\epsilon = 1/8$ and $b = 2$ is the number of Borůvka steps performed. Thus, the graph passed to the recursive call has at most $m/2$ edges and $n/4$ vertices. Since no MST algorithm can do better than linear time, the bottleneck, if any, must lie in the use of decision trees, which are optimal by construction.

We will now establish a formal recurrence relation for the running time of OptimalMST. Let $T(m, n)$ be the worst case running time of OptimalMST on any m-edge, n-vertex graph. Let $\textsc{mst}^*(H)$ be the decision-tree complexity of MST, on the *specific* graph H. Thus $\textsc{mst}^*(m, n)$ can be defined as:

$$\textsc{mst}^*(m,n) \;=\; \max\{\textsc{mst}^*(H) \;:\; |E(H)| = m, |V(H)| = n\}$$

The recurrence relation for T is given below. Recall from Chapter 7 that we assumed the graph G is connected. Furthermore, if the input to OptimalMST is connected, so is the graph passed to the recursive call. Therefore, $T(0, 1) = O(1)$ is the only base case we need to consider.

$$T(m,n) \;\le\; \sum_i c_1 \cdot \textsc{mst}^*(C_i) + T(m/2, n/4) + c_2 \cdot m \tag{8.1}$$

It is straightforward to see that if $\textsc{mst}^*(m,n) = O(m)$ then the above recurrence gives $T(m,n) = O(m)$. One can also show that $T(m,n) = O(\textsc{mst}^*(m,n))$ if $\textsc{mst}^*$ is any number of natural functions, including $O(m\alpha(m,n))$. However, Line 8.1 does not imply $T(m,n) = O(\textsc{mst}^*(m,n))$ if $\textsc{mst}^*(m,n)$ is only assumed to be non-decreasing.

In Section 8.4.5 we prove some simple properties of MST decision trees, then analyze the asymptotic behavior of $T(m,n)$.

8.4.5 Analysis

The results of this section will lead to a proof that $T(m,n) = O(\textsc{mst}^*(m,n))$. We begin by noting some simple facts about $\textsc{mst}^*$.

Fact 1 *Let $m > m' > 2$ and $n > n' > 2$.*

1. $\textsc{mst}^*(m,n) \ge m/2$.

2. $\textsc{mst}^*(m,n) \ge \textsc{mst}^*(m',n)$

3. $\textsc{mst}^*(m,n) \ge \textsc{mst}^*(m,n')$

Fact 1(1) holds since for $m,n > 2$ every edge can be placed on some cycle, and must therefore participate in at least one comparison. Facts 1(2,3) clearly hold since adding edges or isolated vertices cannot make the problem easier.

Lemma 38 *Suppose that H is a graph satisfying the identity $MST(H) = \bigcup_i MST(C_i)$, where the $\{C_i\}_i$ are subgraphs and the identity holds regardless of the permutation of H's edge-weights. Then $\textsc{mst}^*(H) = \sum_i \textsc{mst}^*(C_i)$.*

Proof: Notice that $E(H) = \bigcup_i E(C_i)$ since *any* edge in $E(H)$ can be in $MST(H)$, for some permutation of the edge-weights. It follows from the identity $MST(H) = \bigcup_i MST(C_i)$ that $\textsc{mst}^*(H) \le \sum_i \textsc{mst}^*(C_i)$. We show that any MST algorithm on H can be forced to make *at least* $\sum_i \textsc{mst}^*(C_i)$ comparisons, which gives the lemma.

Consider a worst-case adversary for C_i. It decides the outcomes of edge-weight comparisons (based on known inequalities between edge-weights in C_i) so that any MST algorithm on C_i makes at least $\textsc{mst}^*(C_i)$ comparisons. Our worst-case adversary for H decides the outcomes of comparisons as follows. Suppose $w(e)$ is compared against $w(f)$. (1) If $e \in C_i$, $f \in C_j$, and $i < j$, it proclaims that $w(e) < w(f)$. (2) If, on the other hand, $e, f \in C_i$, it lets the worst-case adversary for C_i decide the outcome. Comparisons of type (1) shed no light on the relative weight of two edges in the same component, and comparisons of type (2) clearly do not betray information about edge-weights in C_j, $j \neq i$. Therefore, the total number of comparisons of type (2) is, by definition of worst-case adversary, at least $\sum_i \text{MST}^*(C_i)$.

□

Notice that if $\{C_i\}_i$ is the set of components returned by the Partition procedure, then $H = \bigcup_i C_i$ satisfies the equality $MST(H) = \bigcup_i MST(C_i)$. This follows from the observation that C_j shares at most one vertex with $\bigcup_{i=1}^{j-1}$, implying that any simple cycle in H is contained in precisely one subgraph C_j. A simple corollary of Lemma 38 is that MST^* possesses a certain super-additive property:

Corollary 1 *For any m and n, $\text{MST}^*(2m, 2n) \geq 2 \cdot \text{MST}^*(m, n)$*

We can now solve the recurrence relation for the running time of OptimalMST given in the previous section.

$$\begin{aligned}
T(m,n) &\leq \sum_i c_1 \cdot \text{MST}^*(C_i) + T(m/2, n/4) + c_2 \cdot m \\
&= c_1 \cdot \text{MST}^*(\bigcup_i C_i) + T(m/2, n/4) + c_2 \cdot m \qquad \{\text{Lemma 38}\} \\
&\leq c_1 \cdot \text{MST}^*(m,n) + T(m/2, n/4) + c_2 \cdot m \qquad \{\text{Fact 1(2)}\} \\
&\leq c_1 \cdot \text{MST}^*(m,n) + c \cdot \text{MST}^*(m/2, n/4) + c_2 \cdot m \quad \{\text{assume inductively}\} \\
&\leq \text{MST}^*(m,n)(c_1 + c/2 + 2c_2) \qquad \{\text{Facts 1(1,3), Corollary 1}\} \\
&\leq c \cdot \text{MST}^*(m,n) \qquad \{\text{for } c = 2c_1 + 4c_2;\ \text{this completes the induction}\}
\end{aligned}$$

Theorem 8 follows:

Theorem 8 *The algorithm* `OptimalMST` *solves the minimum spanning tree problem deterministically in time $O(\text{MST}^*(m,n))$, where m and n are the number of edges and vertices, respectively, and MST^* is the worst-case decision-tree complexity of the problem on any m-edge, n-vertex graph.*

8.5 Avoiding Pointer Arithmetic

Consider the procedure DecisionTree used in our OptimalMST algorithm. It is given a set of graphs, each on at most $\tau = \log^{(3)} n$ vertices, and must match the input graphs with their associated optimal MST decision trees. The most obvious way to implement DecisionTree is with a table. Each of the possible graph topologies is assigned a distinct identification number less than $2^{\tau^2} = o(\log n)$. We could then use the id numbers to find the associated decision trees in constant time using a table lookup. In the pointer machine model, however, there is no way to implement constant-time array access.

We use the pointer machine-based radix-sort of Buchsbaum et al. [24] to implement DecisionTree. The input graphs $\{C_1, \ldots, C_k\}$ are assumed to be represented as adjacency lists. We first rewrite each C_i in canonical form. We create *buckets* associated with the numbers $1..\tau$, then associate the ith vertex of C_i with the ith bucket. An edge is then written as a pair of "numbers" (pointers to two numbered buckets), and a graph C_i on m_i edges is then a list of $4m_i$ "numbers" between 1 and τ. Using this canonical numbering, we radix-sort[4] the graphs in time $O(\tau^3 + \sum_i m_i)$: each of the $4\sum_i m_i$ elements is examined once, and each of the at most τ^2 passes takes at least $O(\tau)$ time. Note that $\tau^3 = o(\log\log n)$ is not the dominant term.

Once $C_1, \ldots, C_k$ are in sorted order, it is easy to divide them into at most $2^{\tau^2} = o(\log n)$ sub-lists of topologically identical graphs. For each sub-list (graph topology) we find the MST decision tree appropriate to that topology. Apart from the cost of actually *using* the decision trees, which is unknown, the overhead of DecisionTree is clearly linear in the size of the graphs $C_1, \ldots, C_k$.

8.6 Introducing A Little Randomness

In this section we show that with some minor modifications to our algorithm, we can compute minimum spanning trees in expected linear time using hardly any random bits.

Theorem 9 *Let r be any function such that $\alpha(n \cdot r(n), n) = O(1)$. Using $r(n)$ random bits, the minimum spanning tree of a graph can be computed in linear time with probability $1 - e^{-\Omega(r(n))}$.*

Proof: (Sketch) The theorem is proved using a minor modification to our OptimalMST algorithm. We use the same Partition procedure, but this time with threshold $\tau = \sqrt{r(n)}$. Partition returns a set of components $\{C_i\}_i$. We may further group these components, giving $\{C_i'\}_i$, so that every C_i' has $\Theta(r(n))$ edges. Rather than compute $\bigcup_i MST(C_i')$ with a set of optimal decision trees, we use Karger et al.'s [127] randomized MST algorithm, *reusing* the same $r(n)$ random bits on each C_i'. For our DenseCase algorithm we use Chazelle's $O(m\alpha(m,n))$-time algorithm. Since $\tau = \sqrt{r(n)}$, DenseCase runs in linear $O(m+n)$ time on graphs with $m' < m$ edges and $n' < n/\tau$ vertices. (The constant in $O(m+n)$ depends on our choice of r).

In [127] Karger et al. show that with probability $1 - e^{-\Omega(m)}$, their algorithm runs in $O(m)$ time on a graph with m edges. Thus, for any particular C_i', the probability

[4]Note that radix-sort is generally not implementable on a pointer machine because there is no way to map the natural number i to the ith bucket in constant time. We are, as in [24], circumventing this issue by representing the natural i as a pointer to the ith bucket.

that Karger et al.'s algorithm runs out of random bits is $e^{-\Omega(r(n))}$. If this unlikely event occurs we switch to a textbook algorithm and compute $MST(C_i')$ in $O(r(n)\log(r(n)))$ time. Let a choice of the $r(n)$ random bits be *bad* if it causes Karger et al.'s algorithm to run out of random bits, on more than a $1/\log(r(n))$ fraction of the $\{C_i'\}$. The fraction of random-bit sequences that are bad is at most $\log(r(n))\cdot e^{-\Omega(r(n))} = e^{-\Omega(r(n))}$. Clearly, if the random-bit sequence is *not* bad then this recursive call of the algorithm takes linear time. This analysis can be extended to the full algorithm just by letting $\{C_i'\}$ be the components found in all recursive calls.

$\square$

To pick a concrete example, if $r(n) = \log^* n$ then we can compute the MST of any n-node graph in expected linear time using $\log^* n$ random bits. The function $\log^* n$ is not special; in general, r can be any function such that $r(n) = \Omega(r'(n))$ and r' has the following form:

$$r'(n) = \log^{\overbrace{*\ *\ *\ \cdots\ *}^{O(1)\text{ stars}}} n$$

where $g^*(n) = \min\{i \ : \ g^{(i)}(n) \le 1\}$, and $g^{(i)}n$ is the i-fold application of g on n. Here, g is assumed to be decreasing.

The algorithm sketched above is, like OptimalMST, not easily parallelizable. In Chapter 11 we design a parallel, expected linear-work MST algorithm that uses a polylogarithmic number of random bits.

8.7 Performance on Random Graphs

Even if we assume that $\text{MST}^*(m,n)$ is superlinear in m, we show, in this section, that $\text{MST}^*(G) = O(|E(G)|)$ for nearly all graphs G. Recall that $\text{MST}^*(G)$ is the *worst case* decision tree complexity for the graph topology G. In other words, we are still assuming a nasty adversary that can choose the edge-weight permutation.

Our result is to be contrasted with the weaker result of Karp and Tarjan [131], who showed that the expected complexity of MST is linear, if *both* the graph topology and edge-weight permutation were selected at random. Our result may also be compared with the randomized algorithm of Karger et al. [127], which is shown to run in $O(m)$ time with high probability. However, for any graph and any edge-weight permutation, the Karger at al. algorithm can be just as slow as Borůvka's — $O(m\log n)$ time.

None of the earlier published MST algorithms appear to have this property of running in linear time w.h.p. on random graphs for all edge-weight permutations. However, one can prove a similar result using suitably souped-up versions of any of the algorithms in [73, 78, 29].

Our analysis hinges on the observation that for sparse random graphs, w.h.p. any subgraph constructed by the Partition routine has only a miniscule number in edges in excess of the number of spanning tree edges in that subgraph. The MST of such graphs can be computed in linear time, and hence the computation on optimal decision trees takes linear time on these graphs.

Throughout this section α will denote $\alpha(m, n)$.

The random graph model $G_{n,m}$ (see [59]) assigns all $\binom{\binom{n}{2}}{m}$ graphs with m edges equal probability. In $G_{n,p}$ any graph on m edges is assigned probability $p^m(1-p)^{\binom{n}{2}-m}$. In other words, each possible edge is included independently with probability p.

Theorem 10 *Let* MST$^*(G)$ *be the worst case decision tree complexity of the MST problem on graph G, over all permutations of G's edge-weights.*

1. *Let G be selected from $G_{n,m}$. Then* MST$^*(G) = O(m)$ *with probability* $1-e^{-\Omega(m/\alpha^2)}$.
2. *Let G be selected from $G_{n,p}$. Then* MST$^*(G) = O(pn^2)$ *with probability* $1 - e^{-\Omega(pn^2/\alpha^2)}$.

In the next section we describe the *edge-addition martingale* for the $G_{n,m}$ model. In Section 8.7.2 we use this martingale and Azuma's inequality to prove Part (1) of Theorem 10. Part (2) is shown to follow from part (1).

8.7.1 The Edge-Addition Martingale

It was observed by Erdős and Rényi [59] that a random graph from the $G_{n,m}$ model can be generated in an incremental fashion as follows. We begin with n labeled vertices, adding one random edge at a time which was not previously selected. Let X_i be a random edge s.t. $X_i \neq X_j$ for $j < i$, and $G_i = \{X_1, \ldots, X_i\}$ be the graph made up of the first i edges, with G_0 being the graph on n vertices having no edges.

A *martingale* is a sequence of random variables $Y_0, \ldots, Y_m$ s.t. $\mathrm{E}[Y_i \,|\, Y_{i-1}] = Y_{i-1}$ for $0 < i \leq m$. We now prove that if g is any graph-theoretic function and $\hat{g}(G_i) = \mathrm{E}[g(G_m) \,|\, G_i]$, then $\hat{g}(G_i)$, for $0 \leq i \leq m$ is a martingale.

Lemma 39 *The sequence $\hat{g}(G_i) = \mathrm{E}[g(G_m) \,|\, G_i]$, for $0 \leq i \leq m$, is a martingale, where g is any graph theoretic function, G_0 is the edge-free graph on n vertices, and G_i is derived from G_{i-1} by including a random edge not already in G_{i-1}.*

Proof: Let $X_i^j = \{X_i, \ldots, X_j\}$. Given that G_{i-1} has been fixed,

$$\mathrm{E}[\hat{g}(G_i)] \quad = \quad \sum_{X_i = x_i} \Pr[X_i = x_i \,|\, G_{i-1}]$$

$$\begin{aligned}
&\cdot \sum_{X_{i+1}^m = x_{i+1}^m} \Pr[X_{i+1}^m = x_{i+1}^m \mid G_{i-1}, X_i = x_i] \cdot g(G_{i-1} \cup x_i^m) \\
= &\sum_{X_i^m = x_i^m} \Pr[X_i^m = x_i^m \mid G_{i-1}] \cdot g(G_{i-1} \cup x_i^m) \\
= &\ \mathrm{E}[g(G_m) \mid G_{i-1}] = \hat{g}(G_{i-1})
\end{aligned}$$

□

We now recall the well known inequality of Azuma (see, e.g., [8]).

Theorem 11 *(Azuma'a Inequality.) Let* $Y_0, \cdots, Y_m$ *be a martingale with* $|Y_i - Y_{i-1}| \leq 1$ *for* $0 < i \leq m$. *Let* $\lambda > 0$ *be arbitrary. Then* $\Pr[|Y_m - Y_0| > \lambda\sqrt{m}] < e^{-\lambda^2/2}$.

To facilitate the application of Azuma's inequality to our edge-addition martingale we establish the following lemma.

Lemma 40 *Consider the sequence proved to be a martingale in Lemma 39. Let* g *be any graph-theoretic function such that* $|g(G) - g(G')| \leq 1$ *for any pair of graphs* G *and* G' *of the form* $G = H \cup \{e\}$ *and* $G' = H \cup \{e'\}$, *for some graph* H *and edges* e, e' *not in* H. *Then* $|\hat{g}(G_i) - \hat{g}(G_{i-1})| \leq 1$, *for* $0 < i \leq m$.

Proof: Let y_i^m and z_{i+1}^m denote possible outcomes of X_i^m and X_{i+1}^m consistent with G_{i-1} and G_i, respectively. Thus, $\hat{g}(G_i)$ is the average of the $g(G_i \cup z_{i+1}^m)$, taken over all possibilities of z_{i+1}^m. We associate each z_{i+1}^m with a set $S(z_{i+1}^m)$ of outcomes of X_i^m as follows. Suppose that $G_i = G_{i-1} \cup \{a\}$, i.e. $X_i = a$ in G_i. Then $y_i^m \in S(z_{i+1}^m)$ iff $y_{i+1}^m = z_{i+1}^m$ (i.e., only X_i may differ) OR $y_i^m - \{a\} = z_{i+1}^m$ (i.e., y_i^m is identical to $a \cdot z_{i+1}^m$ except that a may appear in a different position). Observe that $a \cdot z_{i+1}^m$ and $y_i^m \in S(z_{i+1}^m)$ differ in at most one edge, implying $|g(G_i \cup z_{i+1}^m) - g(G_{i-1} \cup y_i^m)| \leq 1$, and that $|S(z_{i+1}^m)|$ is the same for all z_{i+1}^m. This implies that $|\hat{g}(G_i) - \hat{g}(G_{i-1})| \leq 1$.

□

8.7.2 Analysis

We define the *excess* of a subgraph H to be $|E(H)| - |V(H)| + cc(H)$, where $cc(H)$ is the number of connected components in H. In other words, the excess is the maximum number of edges that can be removed without introducing more connected components. Let $\{C_i\}$ be the subgraphs returned by the Partition procedure. We define $f(G)$ to be the maximum excess of $H = \bigcup_i C_i$, over all possible choices of H. (Recall that $|V(C_i)| \leq \tau = \log^{(3)} n$.)

The key observation is that each pass of our optimal algorithm *definitely* runs in linear time if $f(G) \leq m/\alpha(m, n)$. To see this, note that if this bound on $f(G)$ holds, we can reduce the *total* number of intra-component edges to $\leq 2m/\alpha$ in time $O(n +$

$m \log \alpha/\alpha) = O(m)$ using $\log \alpha$ Borůvkasteps. The existence of Chazelle's algorithm shows that we can then find the MST of the resulting graph in time $O(m\alpha(m/\alpha, n)/\alpha(m,n)) = O(m)$. We show below that if a graph is randomly chosen from $G_{n,m}$, $f(G) \leq m/\alpha(m,n)$ with high probability.

We now show that Lemma 40 applies to the graph-theoretic function f, and then apply Azuma's inequality to obtain our desired result.

Lemma 41 *Let $G = H \cup \{e\}$ and $G' = H \cup \{e'\}$ be two graphs on a set of labeled vertices which differ by no more than one edge. Then $|f(G) - f(G')| \leq 1$.*

Proof: Suppose that $f(G) - f(G') > 1$, then we could apply the optimal set of components of G to G'. Every intra-component edge of G remains an intra-component edge, except possibly e. This can reduce the excess by no more than one, a contradiction. The possibility that e' may become an intra-component edge can only help the argument.
□

Lemma 42 *$\hat{f}(G_0) = o(m/\alpha)$.*

Proof: Notice that if $\frac{m}{n} \geq \alpha\tau$, it is simply impossible to have m/α intra-component edges, so we assume $\frac{m}{n} < \alpha\tau$.

An upper bound on $\hat{f}(G_0)$ is the expected number of indices i such that edge X_i completes a cycle of length $\leq \tau$ in G_{i-1}, since all edges which caused f to increase must have satisfied this criterion. Let p_i be the probability that X_i completed a cycle of length $\leq \tau$. By bounding the number of such cycles, and the probability they exist in the graph, we have

$$\begin{aligned}
p_i &< \sum_{j=3}^{\tau} n^{j-2} \left(\prod_{\ell=1}^{j-1} \frac{i-\ell}{\binom{n}{2} - (\ell-1)} \right) \\
&< \frac{1}{n} \sum_{j=3}^{\tau} \left(\frac{nm}{\binom{n}{2}} \right)^{j-1} \qquad \text{(recall that } i \leq m) \\
&= O\left(\frac{(2m)^{\tau-1}}{n^{\tau}} \right)
\end{aligned}$$

Thus, $\hat{f}(G_0) \leq \sum_i p_i = O(2m/n)^{\tau} = O(2\alpha\tau)^{\tau} = o(\log n)$
□

Lemma 43 *Let G be chosen from $G_{n,m}$. Then* $\Pr[f(G) > m/\alpha] < e^{-\Omega(m/\alpha^2)}$.

Proof: By applying Azuma's inequality, we have that $\Pr[|\hat{f}(G_m) - \hat{f}(G_0)| > \lambda\sqrt{m}] < e^{-\lambda^2/2}$. Setting $\lambda = \sqrt{m}/\alpha - \hat{f}(G_0)/\sqrt{m}$ gives the Lemma. Note that by Lemma 42 $\hat{f}(G_0)$ is quite insignificant.

□

We are now ready to prove Theorem 10.

Proof: We examine only the first $\log \tau$ passes of our optimal algorithm, since all remaining passes certainly take $o(m)$ time. Lemma 43 assures us that the first pass runs in linear time w.h.p. However, the topology of the graph examined in later passes *does* depend on the edge weights. Assuming the Borůvka steps contract all parts of the graph at a constant rate, which can easily be enforced, a partition of the graph in one pass of the algorithm corresponds to a partition of the original graph into components of size less than τ^c, for some fixed c. Using τ^c in place of τ does not affect Lemma 42, which gives the Theorem for $G_{n,m}$, that is, part (1). For $G_{n,p}$ note that the probability that there are not $\Theta(pn^2)$ edges is exponential in $-\Omega(pn^2)$, hence the probability that the algorithm fails to run in linear time is dominated by the bound in part (1).

□

For the sparse case where $m < n/\alpha$, Theorem 10 part (1) holds with probability 1, and for $p < 1/n\alpha$, by a Chernoff bound, part (2) holds with probability $1 - e^{-\Omega(n/\alpha)}$.

8.8 Discussion

We have established a minimum spanning tree algorithm that is provably optimal, but whose running time is, at present, unknown. Results of this nature have been proved for other problems, such as searching monotone matrices [148] and performing sensitivity analysis of minimum spanning trees [57]. In [57] it was also noted that the technique could be used to obtain an alternate linear-time algorithm for polygon triangulation. However, the application of optimal decision trees to these problems is relatively straightforward, compared to our algorithm.

Our result should be contrasted against a more general complexity theory result of Levin [122], who showed how to construct an optimal algorithm for any problem, given an optimal verification routine. The idea is to enumerate all possible programs $P_1, P_2, \ldots$ and execute them in parallel, P_{i+1} at half the rate of P_i. Whenever a program stops, its answer is verified for correctness, so if P_C happens to be the first optimal program in the enumeration, the overall algorithm runs in at most $2^{C+O(1)} \cdot Opt$ steps. Levin's result is weak for several reasons. First, the mere *existence* of an optimal algorithm is never in question. Second, the algorithm constructed is optimal only w.r.t. a particular model of computation (e.g., Turing machine, RAM, pointer machine) and says nothing about the relationship between algorithmic complexity and decision-tree/algebraic/non-uniform complexity. There are also two unknowns in Levin's construction: the optimal running time *Opt* and the constant of proportionality $2^{C+O(1)}$, which is doubly exponential in the length of the optimal program, assuming programs are enumerated in the usual way. Thus, an optimal minimum spanning tree algorithm *could* have been constructed using Levin's technique and Komlós's MST verification algorithm [141]. However, this would not prove nor suggest our main result, that the algorithmic complexity of MST is equal to its decision-tree complexity.[5]

So, what *is* the decision-tree complexity of MST? Here we will venture a modest conjecture.

Conjecture 1 $\text{MST}^*(m,n) = O(m)$

This conjecture does not refer to our algorithm, and clearly could have been stated earlier. The real question, in our opinion, is whether the technique of *precomputation* (as in precomputing optimal decision trees) is absolutely necessary. Why might precomputation be so crucial? One plausible scenario is that the optimal MST decision tree is extremely sensitive to the *structure* of the input graph, in ways that cannot easily be described with simple rules. Furthermore, to achieve optimality it may be necessary to perform comparisons in a *chaotic* fashion, in contrast to the ordered procession of comparisons made by existing MST algorithms. To make discussion of this point less vague, consider the following properties shared by nearly all MST algorithms: *obliviousness* and *justifiability.*

justifiability Whenever edges e and f are compared, there must be a certificate path P connecting e to f such that every edge on P is known to be lighter than e or f. (For the greedy MST algorithms, P is always known to be one of the MST paths between e and f.)

obliviousness Comparisons made inside data structures/subroutines are oblivious of the graph structure. Other comparisons form a small minority.

Kruskal's algorithm [145] does not make justifiable comparisons, simply because it ignores graph structure altogether. With the possible exception of our optimal MST algorithm, Karger et al.'s randomized algorithm [127] is the only non-oblivious algorithm. (Its MST verification subroutine [141] makes heavy use of the graph's structure.) One can easily check that every other MST algorithm is both oblivious and justifiable, even

[5]To complicate matters, one could derive *two* optimal MST algorithms with Levin's technique: one optimal MST algorithm for the RAM and one optimal MST algorithm for the pointer machine. There would then be no reason to believe the two optimal algorithms had the same complexity, asymptotically. Here *complexity* refers to the total number of operations in the appropriate machine model, either RAM or pointer machine.

the very sophisticated algorithm of Chazelle [28]. If there is an explicitly linear-time, deterministic MST algorithm, we are virtually certain that it is non-oblivious and suspect that it might need to be un-justifiable as well.

The results in Chapter 9 bolster our belief that any linear-time MST algorithm must be non-oblivious. There we show that a subtle generalization of MST verification (in which the graph is revealed one edge at a time) cannot be solved in linear time. Thus, it seems as if the pursuit of an explicit linear-time MST algorithm should go hand-in-hand with the development of non-oblivious techniques, for the MST problem and in general. Here we would like to highlight a few problems that might be worthy of study in conjunction with the MST problem.

Graphic Union-Find

Consider the following union-find type problem. There are initially n elements in singleton sets. The operation $\mathit{Union}(u,v)$ indicates whether u and v are in the same set, and if not, it merges the (disjoint) sets containing u and v into one. The existing lower bounds on the Union-Find problem [193, 71] show that answering m *Union* operations on-line takes $\Omega(m\alpha(m,n))$ time. Now suppose that all m *Union* operations were given ahead of time in the form of an undirected graph G, where $(u,v) \in E(G)$ if the operation $\mathit{Union}(u,v)$ will be issued at some time. This is a generalization of several related problems proposed in [32, 79, 97]. One can easily show, using essentially the same proof from Section 8.7, that the standard Union-Find algorithm takes linear time for nearly all graph topologies. Even on "hard" topologies, the inverse-Ackermann-type behavior of Union-Find [190] will usually not arise, if the order of the edges (i.e., *Unions*) is permuted randomly. So the problem here is to develop techniques for the very few hard topologies under the very few hard edge-permutations.

Minimum Spanning Tree Sensitivity

The MST sensitivity problem [192, 194, 57] is equivalent to the following: Given a weighted graph G and spanning tree T, decide for each edge $e \in T$, the best replacement edge $e' \in G-T$, that is, the one minimizing the weight of the spanning tree $T-\{e\}\cup\{e'\}$. Treated as a partial-sums problem (see Chapter 9) MST sensitivity analysis is easily shown to be the dual of MST verification. In fact, MST verification[6] is linear-time reducible to MST sensitivity analysis, which is, in turn, linear-time reducible to the MST problem itself — these are easy exercises left to the reader. Tarjan's algorithm [192, 194] solves the problem in $O(m\alpha(m,n))$ time, and the [57] algorithm solves it in randomized linear time. Our split-findmin data structure yields an $O(m\log\alpha(m,n))$-time MST sensitivity algorithm, which is asymptotically better but obviously does not

[6]Here we mean a stricter form of MST verification: the algorithm need only answer *yes* or *no*.

exorcize the spirit of (inverse-)Ackermann. Whether MST is reducible to split-findmin or MST sensitivity is unknown.

We think the value of MST sensitivity is as a testing ground. It is simpler than the MST problem, and is a wonderful setting to toy around with new algorithmic techniques.

Set Maxima

Given a set system $(\chi, \mathcal{Q})$, where χ is a set of weighted elements, and $\mathcal{Q}$ a collection of subsets of χ, the set maxima problem is to determine $\max Q$ for each $Q \in \mathcal{Q}$. It subsumes MST verification, MST sensitivity analysis, and a number of other problems [95, 57, 86, 150], and it can be rephrased as the general matroid verification problem [150]. Goddard et al. [86] showed that the randomized complexity of the problem is $\Theta(n \log \frac{m+2n}{n})$, where $n = |\chi|$ and $m = |\mathcal{Q}|$.

We speculate that the minimum spanning tree problem is reducible to set maxima. That is, given an optimal black-box set maxima solver, it should be possible to construct from it a linear-time MST algorithm.

Chapter 9

A Lower Bound on MST Verification

9.1 Introduction

Anyone wishing to solve the minimum spanning tree problem now has at his disposal a toolbox full of interesting data structures and algorithmic techniques, including approximate priority queues [29], Fibonacci heaps [73], Borůvka steps, sampled graphs [127], and MST verification [141]. Is it possible to assemble an explicit, linear-time MST algorithm from this bag of tricks, or is it the case that our toolbox is missing an essential tool? In [28, p. 1029] Chazelle speculates on the subject of the missing tool:[1]

> Given a spanning tree T, to verify that it is minimum can be done in linear time [Dixon et al. 1992; King 1997; Komlós 1985]. The problem is to check that any edge outside T is the most expensive along the cycle it forms with T. With real costs this can be viewed as a problem of computing over the semigroup $(\mathbb{R}, \max)$ along paths of a tree. Interestingly, this problem requires $\Omega(m\alpha(m,n))$ time over an arbitrary semigroup [Chazelle and Rosenberg 1991; Tarjan 1978]. This lower bound suggests that in order to improve upon our algorithm specific properties of $(\mathbb{R}, \max)$ will have to be exploited. This is done statically in [Dixon et al. 1992; King 1997; Komlós 1985]. We speculate that an answer might come from a dynamic equivalent.

In this chapter we consider one such "dynamic equivalent", namely the *online MST verification problem.* We contrast the online problem with the *offline* MST verification

[1]The results of this chapter appeared in: S. Pettie, An inverse-Ackermann style lower bound for the online minimum spanning tree verification problem, Proc. 43rd IEEE Symp. on Found. of Computer Science, (FOCS), pp. 155–163, 2002.

problem considered in [192, 141, 57, 133, 16, 24]. Given G, T, where G is a graph and T a spanning tree of G, the offline problem is to decide, for each edge e, whether $e \in MST(T \cup \{e\})$. It follows from the cycle property that $T = MST(G)$ precisely when $e \notin MST(T \cup \{e\})$ for all $e \in G - T$. Komlós [141] demonstrated that the decision-tree complexity of MST verification is linear. The problem of developing a linear-*time* MST verifier was solved in [57], and in [133, 16, 24] other solutions are presented that are supposedly simpler or in a simpler model of computation.

In the *online* version of the problem T is given in its entirety but $G-T$ is presented one edge at a time. For each *query edge* e we must decide whether $e \in MST(T \cup \{e\})$ before receiving the next query edge. Our main result is that there is no linear-time online MST verifier, ruling out this sort of data structure as the basis of a faster explicit MST algorithm. In particular:

Theorem 12 *Any online minimum spanning tree verification algorithm performs $\Omega(m\alpha(m,n)+n)$ edge-weight comparisons, where m is the number of queries, n is the size of the fixed tree, and α is the inverse-Ackermann function.*

Perhaps restricting the MST verifier to answer one query at a time is too unrealistic. A simple corollary of Theorem 12 is that giving the MST verifier large *batches* of queries does not significantly affect the complexity of the problem. For instance, if queries are grouped into batches of size $m/\log^* n$, Corollary 2 implies that the complexity of the problem remains $\Omega(m\alpha(m,n) + n)$.

Corollary 2 *Consider the problem of answering m MST verification queries, presented in k batches, where each batch of queries must be answered before receiving the next. Any algorithm for this problem performs $\Omega(m\alpha(\frac{m}{k}, \frac{n}{k}) + n)$ edge-weight comparisons.*

We note that the MST problem is just one in a larger class of matroid optimization problems. Thus, given a basis $\mathcal{B}$ of an arbitrary weighted matroid, the problem of answering m online verification queries requires $\Omega(m\alpha(m, |\mathcal{B}|) + |\mathcal{B}|)$ comparisons. Of course, for any *specific* matroid this lower bound may not hold.

9.1.1 Related Work

The minimum spanning tree verification problem is ostensibly about minimum spanning trees; however, as Chazelle noted, it can also be viewed as a partial sums (or range searching) problem over the semigroup $(\mathbb{R}, \max)$. The range searching problem has been the subject of intense study for quite some time (see, e.g., the survey of Agarwal & Erikson [3]). In a typical formulation of the problem we must preprocess a set system (or *range space*) $(X, \mathcal{Q})$, where X is a set of weighted elements and $\mathcal{Q}$ is a collection of subsets of X, so that given any *query set* $Q \in \mathcal{Q}$ we can quickly determine the sum of the weights of the elements in Q. In a commonly studied scenario X corresponds to points in $\mathbb{R}^d$ and $\mathcal{Q}$ corresponds to the set of all regions with some nice structure, such as d-rectangles or simplices.

The *weights* of elements in X may be real numbers and *sum* may be the usual notion of sum. However, the standard assumption [3] is that weights are drawn from some (possibly commutative) semigroup $(S, \circ)$, where the only operation allowed on semigroup variables is $\circ$.[2] As a consequence, any algorithm in the semigroup model can be written as a *straight-line program*.[3] However, for the specific semigroups $(\mathbb{R}, \max)$ and $(\mathbb{R}, \min)$, it is most natural to assume the decision-tree model, where the program can branch based on the outcome of previous comparisons.

If we view the MST verification problem as a range searching problem, the set of weighted elements X corresponds to the weighted tree edges, $\mathcal{Q}$ corresponds to the set of tree paths, and edge weights are drawn from $(\mathbb{R}, \max)$. This is not a geometric problem in general, although it subsumes one as a special case. If the input tree is a single path, MST verification becomes isomorphic to a 1-dimensional range searching problem where the query sets correspond to intervals. We will call this problem *interval maximum* if the semigroup is $(\mathbb{R}, \max)$ and *interval sum* under arbitrary semigroups. Similarly, *tree sum* refers to the MST verification problem when generalized to arbitrary semigroups.

Tarjan [192] gave an algorithm for the *offline* tree sum problem and other algorithms for certain online variants of tree sum. All the algorithms in [192] are based on the same path-compression technique used in the standard union-find algorithm [190] and therefore run in time $\Theta(m\alpha(m, n))$ where n is the size of the tree and $m \geq n$ is the number of queries. Yao [207] proved that the query time of any online interval sum algorithm is $\Theta(\alpha(m, n))$ if it uses $m \geq n$ units of storage. See also [191, 7] for related results. Chazelle [26] showed that algorithms on intervals (such as one solving interval sum) can be systematically translated into algorithms on trees. This yielded an online tree sum algorithm which answers queries in $\Theta(\alpha(m, n))$ time using $m \geq n$ units of storage. Chazelle and Rosenberg [31] strengthened the results of Yao [207] and Alon and Schieber [7] by showing that *offline* interval sum is just as hard as online interval sum. In other words, there exist m queries that can only be answered using $\Theta(m\alpha(m, n))$ operations. Tarjan [194] showed that the complexity of any offline subset sum problem over an arbitrary semigroup is precisely the same as its dual.[4] Tarjan's result was used

[2]The main advantage of the semigroup model is generality. It is still largely an open question how much *group* complexity can differ from semigroup complexity — in short, how useful is subtraction? See [30] for more discussion of this issue.

[3]Here a straight-line program would be a sequence of commands of the form $x := y \circ z$, where x, y, z are variables of type S.

[4]A subset sum problem can be represented as a zero-one matrix M where $M[i, j] = 1$ indicates element i is in set j. The dual of this subset sum problem is M^T, i.e. elements become sets and sets become weighted elements.

to prove an $O(m\alpha(m,n))$ upper bound on the MST sensitivity analysis problem (see Section 8.8), which is the dual to MST verification. Together with [31] it also proves an $\Omega(m\alpha(m,n))$ lower bound on any straight-line MST sensitivity analysis algorithm.

The complexity of all these problems seems to change when we substitute the specific semigroup $(\mathbb{R}, \max)$ in place of an arbitrary semigroup. Komlós [141] gave a simple reduction from the (online) interval-maximum problem to the (online) least common ancestors problem, which, together with Harel and Tarjan's online LCA algorithm [103], implies that interval-maximum queries can be answered in $O(1)$ time after an $O(n)$-time preprocessing phase. Compare this with the inverse-Ackermann type lower bounds of [207, 7, 31]. Komlós [141], making use of his interval-maximum algorithm, showed that *offline* MST verification can be solved with $2m + O(n \log \frac{m+n}{n}) = O(m+n)$ comparisons, where n and m are the number of tree and non-tree edges, respectively. The MST sensitivity analysis problem was investigated in [57]; they gave two algorithms: a randomized, expected linear-time algorithm and a provably optimal algorithm with unknown running time. One application of our split-findmin data structure — see Chapter A — is a deterministic MST sensitivity analysis algorithm running in time $O(m \log \alpha(m,n))$.

Given this history one might be tempted to conjecture that *all* partial sums/range searching problems are significantly easier under $(\mathbb{R}, \max)$, as opposed to an arbitrary semigroup.[5] Our lower bound disproves this conjecture — decision-tree complexity doesn't buy you anything in the online MST verification problem — and sets another precedent. It is the first inverse-Ackermann type lower bound on a *purely comparison-based* problem.

Because the elementary operation in our model is the *comparison*, as opposed to, e.g., the cell probe [71], the matrix query [135], or the pointer manipulation [193], our proof techniques are information-theoretic in nature. This is in contrast to traditional inverse-Ackermann style lower bounds [193, 207, 7, 31, 135, 104, 71], which are based on mostly structural properties of the problem or problem instance. We suspect that our techniques will be useful in lower-bounding other problems in a comparison-based model of computation.

9.1.2 Organization

Section 9.2 defines our notation and a class of "hard" problem instances. The lower bound proper appears in Section 9.3. In Section 9.4 we provide almost matching upper bounds for online MST verification, and show that the problem becomes significantly easier when the input edge-weights are permuted randomly.

[5]This conjecture basically holds for *random* offline partial sums problems: nearly all such problems on n elements and m sets require $\Theta(mn/\log m)$ semigroup operations to solve (see [191]), whereas they require an expected $\Theta(n \log \min\{n, \frac{m+n}{n}\})$ comparisons to solve [86] when the semigroup is $(\mathbb{R}, \max)$.

9.2 Preliminaries

The problem is to answer, in an online fashion, a sequence of m minimum spanning tree verification queries. Each is of the form: given an edge e, decide whether $e \in MST(T \cup \{e\})$, where T is a *fixed*, edge-weighted tree. By the *cycle property* of minimum spanning trees this is equivalent to asking whether e is not the heaviest edge on the unique cycle in $T \cup \{e\}$. For the sake of simpler notation we consider input trees that are *vertex*-weighted rather than edge-weighted. (Hence we decide if e is heavier than all *vertices* in the unique cycle of $T \cup \{e\}$.) To further simplify matters we restrict the types of inputs and queries, as described below.

- The fixed tree T is a full, rooted binary tree.
- Every query edge will connect a leaf to one of its ancestors.

It is clear that the query edge must participate in at least one comparison. The parameter $t \geq 1$ used throughout the paper represents the desired number of comparisons per query. We will prove that for each t there is an input distribution *Distr*(t) such that for some query, either (a) Answering the query requires at least $t+1$ comparisons (worst case or amortized) or (b) The verification algorithm already performed $cn \log \lambda_t(n)$ comparisons preceding the query, where c is an absolute constant and λ_t is the t^{th}-row inverse of a function similar to Ackermann's function. By judiciously setting $t(m,n) = \max\{t : cn \log \lambda_t(n) \geq tm\}$, this implies that answering m verification queries requires $m \cdot t(m,n)$ comparisons. It is then straightforward to show that $t(m,n)$ is always within an absolute constant of the traditional inverse-Ackermann function defined by Tarjan [190].

9.2.1 A Variation on Ackermann's Function

In the field of algorithms & complexity, Ackermann's function [1] is rarely defined the same way twice (see e.g., [1, 193, 71, 31, 46, 28, 47]). We would not presume to buck such a well-established precedent. Here is a slight variant:

$$\begin{aligned} A(1,j) &= 2^j \\ A(i+1,0) &= A(i,1) \\ A(i+1,j+1) &= A(i, 2^{2^{A(i+1,j)}}) \end{aligned}$$

Let $\lambda_i(n)$ be the inverse of the i^{th} row of A, defined as:

$$\lambda_i(n) = \min\{j : A(i,j) \geq n\}$$

9.2.2 The Input Distribution *Distr*(t)

We denote nodes of the fixed tree T with lower case letters. If q is a tree node we let $w(q)$ be the weight of q and size(q) be the number of leaf-descendants of q. Since T is a full binary tree, size(q) is always a power of two. Note also that $A(i,j)$ is always a power of two. The input distribution *Distr*(t) is fully characterized by Definitions 6 and 7 and Property 4.

Definition 6 *In Distr(t), a non-leaf node p is an i-node if size$(p) = A(i,j)$ for $1 \leq i \leq t$ and some j, and an $\overline{i}$-node if it is an i-node but not an $(i+1)$-node. In Distr(t), leaf-nodes are $\overline{(t+1)}$-nodes by definition.*

Definition 7 *Let p be an arbitrary $\overline{(i+1)}$-node and let q be the nearest $(i+1)$-node ancestor of p. We define the set C_p to be those $\overline{i}$-nodes that lie on the path between p and q.*

Property 4 *Let p be a tree node. If p is a $\overline{1}$-node then $w(p)$, the weight of p, equals the height of p in T, i.e. $\log size(p)$. If p is an $\overline{i}$-node, for $i > 1$, then $w(p) = w(X_p)$, where X_p is a tree node selected uniformly at random from C_p, and independent of $\{X_q\}_{q \neq p}$ — see Figure 9.1.*

In our analysis we will assume that the online MST verification algorithm knows that the permutation of tree weights was drawn from *Distr*(t). Therefore, a statement of the form "it is known that $w(p) < w(q)$" means the inequality $w(p) < w(q)$ follows from the inequalities discovered by the algorithm and the inequalities implicit in Property 4. For instance, Property 4 implies the following Lemma.

Lemma 44 *Suppose p_1 is an $\overline{i_1}$-node, p_2 is an $\overline{i_2}$-node, $i_1 \leq i_2$, and p_1 appears at a lower level of T than p_2. Then $w(p_1) < w(p_2)$.*

Proof: Notice that all $\overline{1}$-nodes at the same level are assigned the same weight. We define span(q) to be the set of $\overline{1}$-nodes r for which the equality $w(q) = w(r)$ could possibly hold, given Property 4. So, if q is a $\overline{1}$-node then span(q) consists of all nodes on q's level in T. We prove the following claim: that if q is an $\overline{i}$ node, span(q) lies strictly below[6] any i-nodes above q. This will imply the Lemma since, by Property 4, any $\overline{1}$-node is lighter than any $\overline{1}$-node ancestor. The claim clearly holds for $\overline{1}$-nodes; assume inductively that it holds for all $\overline{j}$-nodes, where $j < i$. Let q be an arbitrary $\overline{i}$-node and q' be the next i-node above q. By Property 4, span$(q) = \bigcup_{r \in C_q}$ span(r) and by Definition 7 C_q lies strictly below q'. Therefore, by our inductive assumption,

[6]We think of the root being at the "top" of the tree. Therefore statements like "x lies (strictly) below y" should be interpreted as "y is (strictly) closer to the root than x".

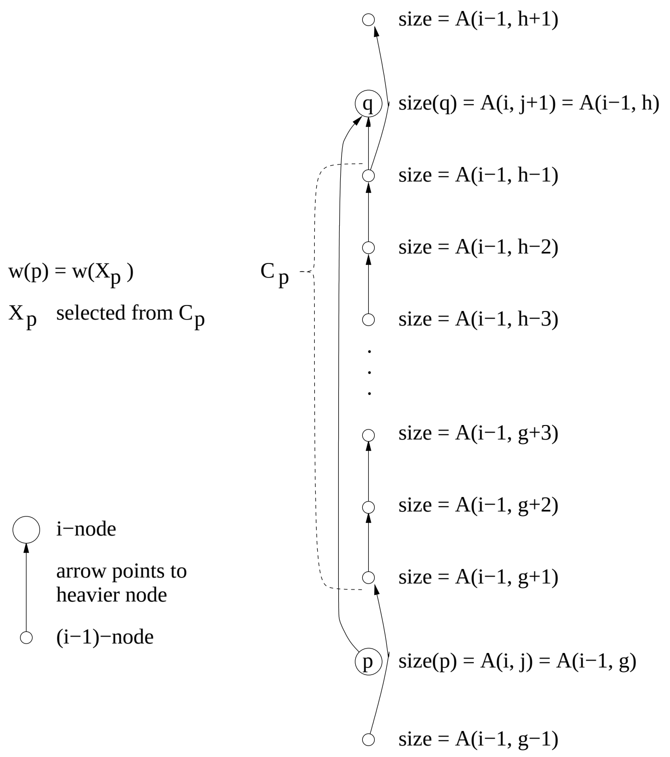


Figure 9.1: An $\bar{i}$-node p, where $i > 1$, and its associated set C_p. C_p consists of the $(i-1)$-nodes between p and q, its nearest i-node ancestor. The weight of p is equal to that of some node in C_p selected uniformly at random.

$\bigcup_{r \in C_q} \text{span}(r)$ lies strictly below q', since C_q consists of $(i-1)$-nodes and q' is, by virtue of being an i-node, an $(i-1)$-node as well. This proves the claim and the lemma.

□

A consequence of Lemma 44 is that, on any leaf-to-root path, the weights of the $\bar{i}$-nodes are monotonically increasing, for any i. This implies that any query can be answered in at most $t+1$ comparisons. If a query edge e connects a leaf to one of its ancestors there are at most $t+2$ candidate maxima on the cycle in $T \cup \{e\}$: the most ancestral $\bar{i}$-node, for $1 \leq i \leq t+1$, plus the edge e itself.

9.2.3 A Measure of Information

Before the verification algorithm performs any comparisons it knows, by Property 4, that $w(q) = w(X_q)$, where q is any $\bar{i}$-node, $i > 1$, and X_q is uniformly distributed over C_q. We define, with respect to the current moment, the set $D_q \subseteq C_q$ as:

$$D_q = \{p \in C_q \;:\; w(q) = w(p) \text{ is consistent}\}$$

That is, $C_q - D_q$ consists of those $p \in C_q$ for which the equality $w(q) = w(p)$ is *impossible*, given Property 4 and all the inequalities discovered by the verification algorithm up until the current moment. It follows from Property 4 that D_q is non-empty. We will use the sizes of the C_q and D_q sets to roughly measure how much information is known about X_q. Define ϕ as:

$$\phi(q) = \begin{cases} \log|C_q| + 2(t+1)^2 & \text{if } |D_q| = 1 \text{ and } \log|C_q| \geq 2(t+1)^2 \\ \log\frac{|C_q|}{|D_q|} & \text{otherwise} \end{cases}$$

Define Φ as:

$$\Phi = \sum_{q \in T} \phi(q).$$

It would be preferable to define $\phi(q)$ as simply $\log(|C_q|/|D_q|)$. However, we need to differentiate between the case when X_q is completely known and when there is a little uncertainty left in X_q (corresponding to $|D_q| = 1$ and $|D_q| > 1$, respectively.) Therefore, we add to $\phi(q)$ a little "bonus" of $2(t+1)^2$ whenever $|D_q|$ reaches 1. Lemma 45 relates the Φ measure to the number of bits of information learned about the verification algorithm.

Lemma 45 *$\Phi/2$ is a lower bound on the information learned about the tree-weights, as drawn from Distr(t).*

Proof: The number $\log(|C_q|/|D_q|)$ measures the bits of information learned about the variable X_q, in the special case when X_q is uniformly distributed over D_q and independent of $\{X_p\}_{p \neq q}$. This is certainly a lower bound on the *actual* number of bits of information learned about X_q. By definition, $\phi(q) \leq 2\log(|C_q|/|D_q|)$. Therefore $\Phi = \sum_q \phi(q)$ is at most twice the number of bits of information learned about the tree-weights.

□

Our lower bound proof uses the Φ and ϕ values to argue about the existence of "hard" queries and to quantify their exact hardness. We show that if Φ is sufficiently small, where small is a function of n and t, then there exists a query that requires $t+1$ comparisons to be answered in the *worst case.* However, there is a danger in continually forcing worst-case behavior. In so doing we may inadvertently expose significantly more than one bit of information per comparison. Therefore, we might no longer be able to claim that, on the average, the number of bits of information learned about the input distribution is at most the number of comparisons performed by the algorithm. We prove amortized bounds by showing that either (a) the verification algorithm answers the query in at least $t+1$ comparisons or (b) after the query is answered, $\Phi/2$ increases by some amount much larger than $t+1$. Theorem 13 shows that if such a dichotomy exists then with high probability the amortized cost of the query is lower bounded by $t+1$ minus some small quantity.

Theorem 13 *Let π be an unknown permutation drawn from some known probability distribution. An algorithm performs a sequence of m operations by comparing elements of π. Let $\Psi_i \geq \Psi_{i-1}$ be a* lower bound *on the amount of information learned about π just before the ith operation. Suppose that it is known that the ith operation either performs at least ν comparisons or $\Psi_{i+1} - \Psi_i \geq \mu$. Let $\text{amort}(\nu,\mu,\epsilon) = \nu(1+(1+\epsilon)\nu/\mu)^{-1}$. Let C be the total number of comparisons made in m operations and $M = m \cdot \text{amort}(\nu,\mu,\epsilon)$. Then:*

$$\Pr[C \ \geq \ M] \geq 1 - 2^{-\epsilon M}$$

Proof: Let I denote the number of bits of information learned about π after C comparisons. We have $\Psi_{m+1} \leq I$. It is simple to prove by induction that $\Pr[I > (1+\delta)C] < 2^{-\delta C}$. Therefore, if $C < M$ then $\Pr[I > (1+\epsilon)M] < 2^{-\epsilon M}$. Let m_ν be the number of operations that perform ν comparisons, and let $m_\mu = m - m_\nu$ be the number of operations that increase Ψ by at μ. We have the following inequalities:

$$\begin{aligned} m &= m_\nu + m_\mu \\ &\leq C/\nu + I/\mu \\ &< M(1/\nu + (1+\epsilon)/\mu) \quad \text{(with prob. greater than } 1-2^{-\epsilon M}) \\ &= M(\text{amort}(\nu,\mu,\epsilon))^{-1} \ = \ m \end{aligned}$$

Therefore, $M > C$ only with probability less than $2^{-\epsilon M}$.

□

In our lower bound proof $\Phi/2$ will take the role of Ψ in Theorem 13 and if we select the input from *Distr*(t) then ν will be $t+1$. We are basically free to set the other parameters. For $\epsilon = 1/t$ and $\mu = (t+1)^2$, Theorem 13 says that the actual amortized cost (C/m) is at least $\text{amort}(t+1,(t+1)^2,1/t) = t$ with probability $1-2^{-m}$. By increasing μ or decreasing ϵ the lower bound on the amortized cost can be driven arbitrarily close to $t+1$, with probability at least $1-2^{-\Omega(m)}$. For the sake of specificity, let the term *amortized cost* be w.r.t. $\epsilon = 1/t$.

9.3 The Lower Bound

Theorem 12 will follow very easily from Lemma 46, given below. The remainder of this section will constitute a proof of Lemma 46.

Lemma 46 *Suppose that $\Phi < \frac{1}{8}n \log \lambda_t(n)$, where it is assumed that $\lambda_t(n) > 2^{3(t+1)^2}$. Then there exists a verification query whose amortized cost is at least t.*

The proof of Lemma 46 is structured as follows. We define a measure $\text{cost}(q)$ over the leaves q of the input tree and show that if any leaf's cost is sufficiently small, then there exists a "hard" query edge whose endpoints connect that leaf to an ancestor. In particular, we generate a sequence of nodes $q_{t+1}, q_t, \ldots, q_1$ where q_{t+1} is a low-cost leaf (recall, leaves are $(t+1)$-nodes), and q_i is an $\bar{i}$-node ancestor of q_{i+1}. The query edge is then $e = (q_{t+1}, q_1)$. We show that $q_{t+1}, q_t, \ldots, q_1$ are all candidate maxima on the unique cycle in $T \cup \{e\}$, which immediately implies that in the *worst case*, the verification algorithm must make $t+1$ comparisons to certify that e is heavier than $q_{t+1}, q_t, \ldots, q_1$. We then show that if, somehow, the verification algorithm got by with fewer than $t+1$ comparisons, then it must have caused Φ to increase by at least $2(t+1)^2$. By appealing to Theorem 13 we can then show that the amortized cost of the query is at least t.

Define $\text{cost}(q)$, where q is a leaf, as

$$\text{cost}(q) = \sum_{\substack{p \text{ ancestral to } q \\ (\text{including } q)}} \frac{\phi(p)}{\text{size}(p)}$$

That is, we can think of q contributing $\phi(q)/\text{size}(q)$ to the cost of each of the $\text{size}(q)$ leaf-descendants of q. Clearly $\sum_q \text{cost}(q) = \sum_q \phi(q) = \Phi$. We choose a hard query as follows:

1. Let q_{t+1} be a leaf such that $\text{cost}(q_{t+1}) < \frac{1}{8} \log \lambda_t(n)$
2. For i from t down to 1: let q_i be the second most ancestral node in $D_{q_{i+1}}$
3. The query edge is $e = (q_{t+1}, q_1)$. We fix $w(e)$ to be more than $w(q_1)$, but less than any other weight more than $w(q_1)$.

In (1), such a q_{t+1} can always be found because the average leaf cost is bounded by $\log \lambda_t(n)/8$. For (2) we clearly require $|D_{q_{i+1}}| \geq 2$. For our analysis to go through we will actually require $D_{q_{i+1}}$ to have at least $2^{2(t+1)^2}$ elements. We first prove bounds on $|C_{q_i}|, |D_{q_i}|$ and $\text{size}(q_i)$. We then bound the cost of answering the query (q_{t+1}, q_1).

Lemma 47 *If q is a leaf then $|C_q| = \lambda_t(n)$. If q is a non-leaf $\bar{i}$ node and there exists some $\bar{i}$ node above q, then $|C_q| \geq 2^{2^{size(q)}-1}$.*

Proof: When q is a leaf C_q consists of all t-node ancestors of q, of which there are $\lambda_t(n)$, by the definition of λ_t. In the general case q is an $\bar{i}$-node, where $i > 1$. (If $i = 1$ then there is no set C_q.) By the definition of C_q this means that for some j, $\text{size}(q) = A(i,j) = A(i-1,j_1)$. The first i-node ancestor of q, say p, satisfies $\text{size}(p) = A(i,j+1) = A(i-1,j_2)$. The set $|C_q|$ consists of the $j_2 - j_1 - 1$ $\overline{(i-1)}$-nodes between q and p. If $j = 0$ then by the definition of A, $|C_q| = 2^{2^{A(i,0)}} - 2$. If $j > 0$ then $|C_q| = 2^{2^{A(i,j)}} - 2^{2^{A(i,j-1)}} - 1$. In either case $|C_q| \geq 2^{2^{\text{size}(q)-1}}$ since $A(i,j-1) < \frac{1}{2}A(i,j)$ when $i > 1$.

□

Lemma 48 *Let p be an arbitrary leaf descendant of an $\bar{i}$ node q, where $i > 1$. Then*

$$|D_q| \geq \frac{|C_q|}{2^{cost(p)\,size(q)}}$$

Proof: By definition of ϕ, $|D_q| \geq |C_q|/2^{\phi(q)}$. By the definition of $\text{cost}(p)$ we have $\text{cost}(p) \geq \phi(q)/\text{size}(q)$. The lemma follows.

□

Lemma 49 *For $1 \leq i \leq t$, size$(q_i) \;\geq\; \lambda_t(n) \;\geq\; 2^{3(t+1)^2}$.*

Proof: Since q_i is an ancestor of q_{i+1}, implying $\text{size}(q_i) > \text{size}(q_{i+1})$, we need only prove the lemma for $i = t$. We selected q_{t+1} for satisfying $\phi(q_{t+1}) < \log \lambda_t(n)/8$. Therefore, it follows from Lemmas 47 and 48 that $|D_{q_{t+1}}| \geq (\lambda_t(n))^{7/8}$. We chose q_t to be the second most ancestral node in $D_{q_{t+1}}$. Therefore, $\text{size}(q_t) \geq A(t, (\lambda_t(n))^{7/8} - 2)$, which is at least $\lambda_t(n)$ since $A(t,j) \geq 2^j$ and $(\lambda_t(n))^{7/8} - 2 \geq \log \lambda_t(n)$ for all but small constant values of $\lambda_t(n)$. (Recall that we assumed $\lambda_t(n) \geq 2^{3(t+1)^2} \geq 2^{12}$.

□

Lemma 50 *For $1 < i \leq t+1$, $|D_{q_i}| \geq 2^{2(t+1)^2}$.*

Proof: It was already shown in the proof of Lemma 49 that $D_{q_{t+1}} \geq (\lambda_t(n))^{7/8} \geq 2^{2(t+1)^2}$. Consider q_i, for $i \leq t$. We have the inequalities:

$$|D_{q_i}| \geq \frac{|C_{q_i}|}{2^{\text{size}(q_i)\,\text{cost}(q_{t+1})}} \geq 2^{\left(2^{\text{size}(q_i)-1-\text{size}(q_i)\log\lambda_t(n)}\right)} \geq \text{size}(q_i) \geq 2^{2(t+1)^2}$$

The first inequality follows from Lemma 48. The second follows the inequality $\text{cost}(q_{t+1}) < \log \lambda_t(n)/8$ and Lemma 47. The third and fourth follow from Lemma 49.

□

Lemma 51 *Answering the query $e = (q_{t+1}, q_1)$ requires $t+1$ comparisons to answer in the worst case.*

Proof: Consider the unique cycle in $T \cup \{e\}$. We will show that among the weighted elements on this cycle (weighted vertices and the edge e), the set of possible maxima is precisely $\{e, q_{t+1}, q_t, \ldots, q_1\}$. Therefore, confirming that e is indeed the heaviest element will require $t+1$ comparisons in the worst case since for any comparison made by the verification algorithm, at least one outcome eliminates at most one possible maximum. Suppose that before the query algorithm began it was already known that $w(q_i) \leq w(q_j)$, ruling out q_i as a potential maximum. We consider two cases, depending on the ordering of i and j. Assume first that $j < i$, that is, q_j is an ancestor of q_i. Let p be the most ancestral element in D_{q_i}. By our selection of q_{i-1} from D_{q_i}, q_{i-1} lies strictly below p, which implies $q_{i-1}, q_{i-2}, \ldots, q_j, \ldots, q_1$ lie strictly below p — see Figure 9.2 for a diagram. This leads to a contradiction since $p \in D_{q_i}$ implies $w(p) = w(q_i)$ is consistent

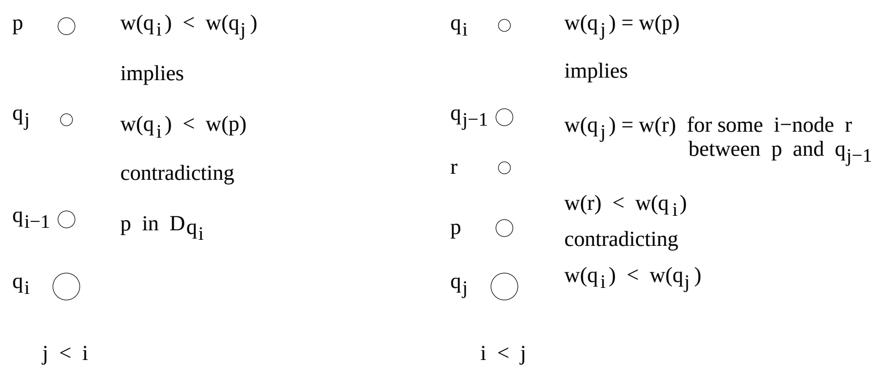


Figure 9.2: Before answering the query the verification algorithm cannot know that $w(q_i) \leq w(q_j)$. The two situations, $i < j$ and $j < i$, are diagrammed. Some of the nodes depicted may actually represent the same node, as in the case when $j = i - 1$ (left) or $i = j - 1$ (right).

with any known inequalities. However, we know that $w(q_i) \leq w(q_j) < w(p)$, where the first inequality is by assumption and the second by Lemma 44. The other case, when $i < j$, is similar but not exactly symmetrical. Let p be the least ancestral element in D_{q_j}. By our choice of q_{j-1} and Lemma 50, we know that p lies strictly below q_{j-1}. If $w(q_j) = w(p)$, which is possible since $p \in D_{q_j}$, then by Property 4 $w(q_j) = w(r)$ where r is some $\bar{i}$-node between p and q_{j-1}. By Lemma 44 $w(r) < w(q_i)$, giving us the inequalities $w(r) < w(q_i) \leq w(q_j) = w(r)$, a contradiction.

□

In Lemmas 52 and 53 we show that if the query (q_{t+1}, q_1) is answered in fewer than $t+1$ comparisons, then Φ must increase by at least $2(t+1)^2$. Let Δ_Φ denote the change in Φ, measured before and after the query.

Lemma 52 *Either* $\Delta_\Phi \geq 2(t+1)^2$ *or* $w(q_{t+1}) \leq w(q_t) \leq \cdots \leq w(q_1) < w(e)$.

Proof: Let high(i) denote the event that $w(q_i) = w(p)$ where p is the most ancestral node in D_{q_i}. Recall that q_{i-1} was chosen to be the second most ancestral node in D_{q_i} and that $w(e) > w(q_1)$. Therefore, it must be the case that $w(q_{t+1}) \leq \ldots \leq w(q_1) < w(e)$ or high$(t+1) \vee$ high$(t) \vee \cdots \vee$ high(2). The event high(j) occurs iff $w(q_j) > w(q_1)$, which implies $w(q_j) > w(e)$. Since, by Lemma 51 the only possible maxima on the cycle in $T \cup \{e\}$ are $q_{t+1}, q_t, \ldots, q_1, e$, if e is not maximal then the verification algorithm must have discovered that $w(q_j) > w(e)$ for some j, i.e. it must have discovered that high(j) holds. According to Lemma 50 $|D(q_j)| \geq 2^{2(t+1)^2}$ before the query e is issued. However, after the query high(j) holds, which implies $|D_{q_j}| = 1$. According to the definition of ϕ, $\phi(q_j)$ must increase by at least $2(t+1)^2$, implying Δ_Φ is at least that much.
$\square$

Lemma 53 *Either* $\Delta_\Phi \geq 2(t+1)^2$ *or each comparison made by the verification algorithm eliminates at most one candidate maximum from the set* $\{q_{t+1}, q_t, \ldots, q_1, e\}$.

Proof: Suppose that there exists a comparison that eliminates two or more candidate maxima from the unique cycle in $T \cup \{e\}$. By Lemma 51 the set of candidate maxima initially includes $\{q_{t+1}, q_t, \ldots, q_1, e\}$. Suppose that the comparison reveals the inequality $w(q_i) \geq w(x)$ and suppose that it is already known that $w(x) \geq w(q_j), w(q_k)$, where q_j and q_k are candidate maxima at the time of the query. Let x be an $\overline{\ell}$-node. There are several cases, depending on the ordering of i, j, k and ℓ. Lemma 52 lets us restrict our attention to the case when $w(q_{t+1}) \leq w(q_t) \leq \ldots \leq w(q_1) < w(e)$, which immediately implies $i < j < k$. The proof is structured as follows. Let p be the next $\overline{(k-1)}$-node above q_{k-1}, i.e. at the time the query e was issued, p was the most ancestral node in D_{q_k}. We will show that either $p \notin D_{q_k}$ just before the comparison, or the comparison causes Φ to increase by at least $2(t+1)^2$, which implies Δ_Φ is at least that much. The case $p \notin D_{q_k}$ leads to a contradiction because it implies, by the definition of D_{q_k}, that $w(q_k) \leq w(q_{k-1})$, meaning q_k was not a candidate maximum just before the comparison.

We will first show that $\ell \geq k$ and x is at the same level in T as q_ℓ. Suppose that $\ell < k$. If the inequalities $w(q_k) \leq w(x) \leq w(q_i)$ were a possibility before the comparison was made then x must lie in the band of T between the levels of q_k and p: any nodes outside that band are, by Lemma 44, definitely lighter than q_k or heavier than q_i. However, the inequality $w(x) \geq w(q_k)$ then implies, by Lemma 44, that $w(p) > w(q_k)$, that is, $p \notin D_{q_k}$, a contradiction. This shows that $\ell \geq k$. We now consider the height of x in T. We cannot have x lower in T than q_ℓ, for this would imply by Lemma 44 that $w(x) < w(q_\ell) \leq w(q_k)$. It also cannot lie above q_ℓ, as this implies $w(q_i) < w(x)$, again by Lemma 44. We define $x_\ell, x_{\ell-1}, \cdots, x_1$ to be the ancestors of x at the same levels as, respectively, $q_\ell, q_{\ell-1}, \ldots, q_1$, where $x = x_\ell$ — see Figure

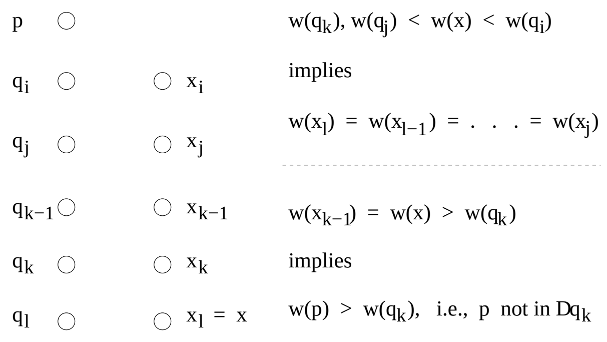

p
q_i
q_j
q_{k-1}
q_k
q_l
x_i
x_j
x_{k-1}
x_k
$x_l = x$
$w(q_k), w(q_j) < w(x) < w(q_i)$
implies
$w(x_l) = w(x_{l-1}) = . . . = w(x_j)$
$w(x_{k-1}) = w(x) > w(q_k)$
implies
$w(p) > w(q_k)$, i.e., p not in Dq_k

Figure 9.3: It was discovered that $w(q_j), w(q_k) \leq w(x) \leq w(q_i)$, ruling out q_j and q_k as potential maxima. The node $x = x_\ell$ is proved to be at the same level as q_ℓ, for some $\ell \geq k$. The nodes $x_{\ell-1}, \ldots, x_1$, are by definition the ancestors or x at the same levels as $q_{\ell-1}, \ldots, q_1$.

9.3. Notice that if $w(q_j), w(q_k) \leq w(x) = w(x_\ell) \leq w(q_i)$, it must be the case that $w(x_\ell) = w(x_{\ell-1}) = \cdots = w(x_j)$, implying that $|D_{x_\ell}| = |D_{x_{\ell-1}}| = \cdots = |D_{x_{j+1}}| = 1$. If any one of these D-sets, say D_{x_r}, had strictly more than 1 element before the query e was issued then we argue that $\Delta_\Phi \geq 2(t+1)^2$. Since x_r is at the same level as q_r, it follows from Lemma 50 that $|C_{x_r}| \geq 2^{2(t+1)^2}$. Given this lower bound on $|C_{x_r}|$, the transition from $|D_{x_r}| > 1$ to $|D_{x_r}| = 1$ causes $\phi(x_r)$ to increase by at least $2(t+1)^2$, which causes Φ to increase by at least that much. Therefore, if $|D_{x_r}| > 1$ before the query was issued then $\Delta_\Phi \geq 2(t+1)^2$. The last situation to consider is when $|D_{x_\ell}| = |D_{x_{\ell-1}}| = \cdots = |D_{x_{j+1}}| = 1$ *before* the query e was issued. In this situation the known inequality $w(x) = w(x_\ell) \geq w(q_k)$ implies $w(x_{k-1}) \geq w(q_k)$ since $j \leq k-1$. However, Lemma 44 implies that $w(p) > w(x_{k-1})$, and consequently, that $p \notin D_{q_k}$ just before the comparison was made. This contradicts the claim that q_k was a candidate maximum.

□

Lemma 53 implies that the query e is answered in $t+1$ comparisons or else increases Φ by at least $2(t+1)^2$. By invoking Theorem 13 with $\Psi = \Phi/2$, $\epsilon = 1/t$, $\nu = t+1$, $\mu = (t+1)^2$, the amortized cost of the query e is at least t. In the next section we prove that Lemma 46 implies Theorem 12.

9.3.1 Proof of Main Theorem

In this section we define a function $t(m, n)$ and prove a lower bound on the online MST verification problem in terms of m, n, and $t(m, n)$. We then show that $t(m, n)$ is within an absolute constant of the inverse-Ackermann function α as defined by Tarjan [190].

Define $t(m,n)$ as:

$$t(m,n) \;=\; \max\left\{t \;:\; \lambda_t(n) \;\geq\; 2^{16t^2\lceil m/n\rceil}\right\}$$

It is true that $t(m,n)$ is undefined when $\lambda_1(n) \;\geq\; 2^{16\lceil m/n\rceil}$. For the sake of completeness, assume $t(m,n) = 1$ in this situation.

Lemma 54 *Any algorithm answering m MST verification queries on an n-node tree makes, on a worst case input,* $\max\{m \cdot t(m,n),\; n\}$ *comparisons with probability at least* $1 - 2^{-\Omega(m)}$.

Proof: If $t(m,n) = 1$ or $m \cdot t(m,n) < n$ then the lemma is trivial. Any MST verification algorithm must perform 1 comparison per query and in the case where $T \cup \{e\}$ consists of a single cycle, confirming that e is the heaviest edge requires n comparisons. We assume that $t(m,n) > 1$. Suppose that the lemma is false, that the verification algorithm makes fewer than $m \cdot t(m,n)$ comparisons with probability greater than $2^{-\Omega(m)}$. By the definition of $t(m,n)$ it follows that $\frac{1}{16}n \log \lambda_t(n) \;\geq\; 2 \cdot m \cdot t(m,n)$. If the MST verification algorithm made fewer than $m \cdot t(m,n)$ comparisons it follows from Lemma 45 that $\Phi < \frac{1}{8}n \log \lambda_t(n)$ with probability at least $1 - 2^{-m \cdot t(m,n)}$. It also follows from the definition of $t(m,n)$ that $\lambda_t(n) \geq 2^{3(t+1)^2}$. Therefore, the preconditions of Lemma 46 are met with high probability. By Lemma 46 the amortized cost of each query is at least t, and by Theorem 13 the probability that the amortized costs exceed the actual costs is no more than $2^{-\epsilon \cdot m \cdot t} = 2^{-m}$.

□

In [190] Tarjan defined a variant of Ackermann's function and a certain inverse which we denote by B and α, respectively. They are defined as:

$$\begin{array}{rcll} B(0,j) & = & 2j & \text{for } j \geq 0 \\ B(i,0) & = & 0 & \text{for } i \geq 1 \\ B(i,1) & = & 2 & \text{for } i \geq 1 \\ B(i,j) & = & B(i-1, B(i,j-1)) & \text{for } i \geq 1, j \geq 2 \end{array}$$

$$\alpha(m,n) = \min\{i \;:\; B(i, 4\lceil \tfrac{m}{n} \rceil) > \log n\}$$

Lemma 55 $\alpha(m,n) - 3 \;\leq\; t(m,n) \;\leq\; \alpha(m,n) + 1.$

Proof: We will use several properties of A and B, given in Lines (9.1) through (9.5). They are all simple to prove by induction. The proofs, which are somewhat tedious, are left to the reader.

$$\begin{aligned}
A(i,j) &\geq B(i,j) \quad \text{for } i \geq 1 \text{ and } j \geq 0 && (9.1)\\
B(i,3j) &\geq A(i,j) \quad \text{for } i \geq 1, j \geq 1 && (9.2)\\
B(i+1,j) &\geq 2^{B(i,j)} \quad \text{for } i \geq 2, j \geq 3 && (9.3)\\
B(i+1,2j) &\geq 2^{B(i,j)} \quad \text{for } i \geq 0, j \geq 1 && (9.4)\\
B(i+1,j) &\geq 2^{B(i,j)/2} \quad \text{for } i \geq 1, j \geq 3 && (9.5)
\end{aligned}$$

We had defined $t(m,n) = \max\{t \ : \ \lambda_t(n) \geq 2^{16t^2\lceil m/n\rceil}\}$. One can see that this is equivalent to the definition:

$$t(m,n) \ = \ \min\left\{t \geq 1 \ : \ A\left(t+1, \ 2^{16(t+1)^2\lceil m/n\rceil}\right) > n\right\}$$

which is more in line with the definition of α. We will first show that $\alpha(m,n) \leq t(m,n)+3$. Consider the inequalities below, where $t = t(m,n)$.

$$\begin{aligned}
A\left(t+1, 2^{16(t+1)^2\lceil m/n\rceil}\right) &> n && (9.6)\\
B\left(t+1, 2^{17(t+1)^2\lceil m/n\rceil}\right) &> \log n && (9.7)\\
B\left(t+3, 4\lceil\frac{m}{n}\rceil\right) &> \log n && (9.8)
\end{aligned}$$

Line (9.6) follows from the definition of $t(m,n)$. Line (9.7) follows from Line (9.2). Line (9.8) follows from Line (9.3), though not directly. Let $x \uparrow y$ denote an exponential stack of x 2's with a y on top; e.g. $2 \uparrow 3 = 2^{2^3} = 256$. One can easily show that $B(1,j) = 2^j = 1 \uparrow j$. Lines (9.3) and (9.5) together imply that $B(i,j) \geq i \uparrow (j-1)$ for all $i \geq 1, j \geq 3$. Therefore, $B(t{+}3, 4\lceil\frac{m}{n}\rceil) = B(t{+}2, B(t{+}3, 4\lceil\frac{m}{n}\rceil{-}1)) \geq B(t{+}2, (t{+}3) \uparrow (4\lceil\frac{m}{n}\rceil{-}2))$, which is greater than $B(t{+}2, 2^{17(t+1)^2\lceil m/n\rceil})$ since $(t{+}2) \uparrow (4\lceil\frac{m}{n}\rceil{-}2)$ always dominates $17(t+1)^2\lceil\frac{m}{n}\rceil$. Therefore Line (9.8) follows from Line (9.7), and implies, by the definition of $\alpha(m,n)$, that $\alpha(m,n) \leq t+3 = t(m,n)+3$.

The bound $t(m,n) \leq \alpha(m,n)+1$ is proved in a similar fashion. Consider the following inequalities, where α is short for $\alpha(m,n)$.

$$\begin{aligned}
B\left(\alpha, 4\lceil\frac{m}{n}\rceil\right) &> \log n && (9.9)\\
B\left(\alpha, 2^{8\lceil m/n\rceil}\right) &> n && (9.10)\\
A\left(\alpha+1, 2^{16\alpha^2\lceil m/n\rceil}\right) &> n && (9.11)
\end{aligned}$$

Line (9.9) follows from the definition of $\alpha(m,n)$. Line (9.10) follows from the monotonicity of B and $B(i,j) \geq 2j$. Line (9.11) follows from Line (9.1) and the monotonicity of A and B. Line (9.11) is not given as $A(\alpha,\ldots) > n$ because α may be zero while $A(i,j)$ is only defined for $i \geq 1$. Therefore, $t(m,n) \leq \max\{\alpha,1\} \leq \alpha+1$.

□

9.4 Upper Bounds

In this section we show that the actual complexity of online minimum spanning tree verification is not too far from the lower bound presented in Section 9.3. We also provide nearly tight bounds on the average case complexity of the problem, where the average is over all permutations of the tree weights. We will think of the MST verification algorithm as divided into two parts: a *preprocessing algorithm* that, given T, produces some fixed data structure, and a *query algorithm* that answers MST verification queries by referring only to the fixed structure. The complexities of interest are the number of preprocessing comparisons and the worst case number of comparisons per query.

Theorem 14 *Suppose the tree-weights are permuted randomly. With no more than* $2n$ *preprocessing comparisons (expected), MST verification queries can be answered with no more than* 2 *comparisons. If queries must be answered with* 1 *comparison then the expected preprocessing time is* $\Theta(n \log n)$.

Proof: First consider the 1-comparison case with randomly permuted weights. Let T be a star with center vertex c. For every two leaves u and v such that $w(u), w(v) > w(c)$ the preprocessing algorithm must determine the heavier of u and v (otherwise the query $e = (u, v)$ would require 2 comparisons to answer.) Therefore, the query algorithm must sort n' elements, where n' is the number of leaves heavier than c. This takes expected $\Omega(n \log n)$ time since n' is uniform over $[0..n-1]$. $O(n \log n)$ preprocessing time is also clearly sufficient for any tree T. (Remark: If all tree nodes have degree bounded by a constant, it seems likely that $o(n \log n)$ preprocessing would be required on average.)

For the 2-comparison case the preprocessing algorithm must reduce the number of candidate maxima on any query to at most 2 (not counting the query edge). We root the tree arbitrarily and divide any query (u, v) into two queries $(u, LCA(u, v))$ and $(v, LCA(u, v))$, where $LCA(u, v)$ is the least common ancestor of u and v. Therefore it will be sufficient to reduce the number of candidate maxima on a query (z, a) to one, where a is an ancestor of z. For any node $z \in T$ let $z = z_0, z_1, z_2, z_3, \ldots$ be the sequence of nodes from z up to the root. We must find the prefix-maxima of the sequences $\{(w(z_i))\}_{z \in T, i \geq 0}$, which is tantamount to finding the subsequence $L(z) = (z_{i_1}, z_{i_2}, z_{i_3}, \ldots)$ where z_{i_p} has maximum weight among $(z_0, \ldots, z_{i_{p+1}-1})$. We compute $L(z)$ from $L(z_1) = (z_{j_1}, z_{j_2}, z_{j_3}, \ldots)$. One can see that $L(z)$ is derived from $L(z_1)$ by substituting a (possibly empty) prefix of $L(z_1)$ with $z = z_0$. We find such a prefix in the obvious manner, by comparing $w(z_0)$ with $w(z_{j_1}), w(z_{j_2}), \ldots$ until j_q is found such that $w(z_0) < w(z_{j_q})$. (If there is no such z_{j_q} then for the sake of consistent notation we let it be the non-existent parent of the root.) The cost of this procedure, which is performed for every node in the input tree, is no more than q.[7] We analyze the behavior of q and j_q under the assumption that the tree edge-weights are randomly permuted. We have

$$\begin{aligned}
\Pr[j_q = r] &\leq 1/r(r+1) && (9.12)\\
\mathrm{E}[q \mid j_q = r] &\leq 1 + \sum_{i=1}^{r-1} \Pr[w(z_i) = \max_{1 \leq k \leq i}\{w(z_k)\}] \\
&= 1 + H_{r-1} && (9.13)\\
\mathrm{E}[q] &= \sum_{r=1}^{\infty} \Pr[j_q = r] \cdot \mathrm{E}[q \mid j_q = r] \\
&\leq 1 + \sum_{r=2}^{\infty} \frac{H_{r-1}}{r(r+1)} \\
&= 1 + \sum_{i=1}^{\infty} \left(\frac{1}{i} \cdot \sum_{r=i+1}^{\infty} \frac{1}{r(r+1)} \right) \\
&= 1 + \sum_{i=1}^{\infty} \frac{1}{i(i+1)} \quad = \quad 2 && (9.14)
\end{aligned}$$

Lines 9.12 and 9.13 are inequalities, rather than equalities, due to the finiteness of the $(z_i)_i$ sequence. Line 9.14 follows from Lines 9.12 and 9.13 and the identity $\sum_{i=1}^{k} 1/i(i+1) = 1 - 1/(k+1)$, which is easily proved by induction on k.

□

Any (online) tree-sum algorithm for arbitrary semigroups can be used as an (online) MST verification algorithm. The constructions from [7, 26] show that a tree T can be preprocessed in $O(n\lambda_t(n))$ time so that the sum of the weights on any path is computable with $2t-1$ semigroup operations. We sketch below how the preprocessing time can be reduced to $O(n \log \lambda_t(n))$ for the online MST verification problem, without affecting the query time. The [7, 26] preprocessing algorithms implicitly generate a set of forests of rooted trees $F_1, F_2, \ldots, F_t$ with the property that the sum of the weights of any path in T is equal to the sum of the weights of $2t$ paths: 2 paths in each of $T_1 \in F_1, T_2 \in F_2, \ldots, T_t \in F_t$, where $T_1, \ldots, T_t$ are trees in their respective forests. The paths in $T_1, \ldots, T_t$ run from a leaf to one of its ancestors. Preprocessing the trees in $F_1, \ldots, F_t$ to answer leaf-to-ancestor queries is done in the obvious fashion: for a tree with size s and height h the preprocessing time is $O(sh)$. $F_1, \ldots, F_{t-1}$ are constructed so that their preprocessing time is $O(n)$, and F_t is guaranteed to be a single tree with size at most n and height at most $\lambda_t(n)$; therefore the total preprocessing time is $O(n\lambda_t(n))$. This algorithm can be improved, slightly, when the semigroup is

[7]It is usually equal to q, unless z_{j_q} happens to be the parent of the root, in which case the comparison $w(e_0) < w(e_{j_q})$ never takes place.

$(\mathbb{R}, \max)$ and $t > 1$. Komlós's MST verification algorithm [141] can be thought of as an $O(s \log h)$-time preprocessing scheme for answering leaf-to-ancestor queries. Using it to preprocess F_t instead of the [7, 26] algorithms yields an $O(n \log \lambda_t(n))$ preprocessing algorithm with query complexity $2t$ (not $2t-1$, as in [7, 26], because the query edge must be compared against the tree-weights too!) We can reduce the query complexity to $2t-1$ by preprocessing the the trees in F_1 more effectively. Any tree $T_1 \in F_1$ with size s and height h has the property that $s \log s = O(sh)$. Therefore, we shall preprocess the F_1 trees by sorting their weights rather than use the [7, 26] algorithm. The preprocessing time is unaffected but the query algorithm may reduce the number of comparisons by 1.

The above construction assumed $t > 1$, specifically that $F_1 \neq F_t$. What is the best preprocessing time we can achieve for query complexity 1, 2 and $2t$? For 1-comparison queries the optimal preprocessing time is clearly $\Theta(n \log n)$. For 2-comparison queries there is a simple algorithm with $O(n \log\log n)$ preprocessing time. King [133] showed that any tree T could be easily transformed into a new tree T' with several nice properties. First, any MST verification query on T can be mapped to an equivalent query in T'. Second, T' has size linear in $|T|$ and height logarithmic in $|T|$. Applying Komlós's $O(n \log\log n)$-time preprocessing algorithm to T' lets us answer queries in 2 comparisons. It can be easily shown that this bound is optimal for 2-comparison queries, using the input distribution *Distr*(1) and an argument similar to that of Lemma 46. If the query complexity is fixed at $2t$, where $t \geq 2$, we do not know of any faster preprocessing algorithm than the one for query complexity $2t-1$. Our results on the worst case complexity of online MST verification are summarized in Theorem 15.

Theorem 15 *For query complexity* 1 *the optimal online MST verification algorithm preprocesses the tree in* $\Theta(n \log n)$ *time. For query complexity* 2 *the optimal preprocessing time is* $\Theta(n \log\log n)$*. For query complexity* $2t - 1 \geq 3$ *the optimal preprocessing time is* $O(n \log \lambda_t(n))$ *and* $\Omega(n \log \lambda_{2t-1}(n))$*.*

For query complexity $2t-1$ we suspect that the upper bound in Theorem 15 is tight. However, for the special case of leaf-to-ancestor queries the lower bound from Section 9.3 is tight. Therefore, to improve our lower bounds one must necessarily consider more general types of queries.

Chapter 10

A Time-Work Optimal Parallel MST Algorithm

In this chapter we present a randomized parallel minimum spanning tree algorithm that is simultaneously optimal with respect to both work and time, where *work* is defined as the time-processor product. Our algorithm runs in $O(\log n)$ time using $m/\log n$ processors (thus $O(m)$ work) on the EREW PRAM, the weakest of the PRAM models [130]. This is the first provably time-work optimal MST algorithm for any parallel model. In Section 10.8 we give a simple processor allocation scheme for "tree-like" computations, which may be of separate interest.[1]

10.1 The PRAM Model

The PRAM [130], or Parallel-RAM, is an abstract machine consisting of P processors with random access to a shared array of memory cells. Computation proceeds in synchronized, constant-time phases consisting of a read step, a computation step, and a write step.[2] The PRAM comes in several flavors: EREW, CREW, ERCW, and CRCW, corresponding to the general restrictions placed on memory access. The exclusive read (ER) and exclusive write (EW) models disallow two or more processors reading from and writing to the same memory cell in the same step, whereas the concurrent read and write models (CR and CW, resp.) tolerate this type of memory access.[3] *Time* is defined

[1]The results of this chapter are taken from: S. Pettie and V. Ramachandran, A randomized time-work optimal parallel algorithm for finding a minimum spanning forest, SIAM J. on Computing 31(6), pp. 1879–1895, 2002.

[2]Reading and writing are separated simply to avoid the issue of what happens during simultaneous reading and writing.

[3]There are actually several concurrent write models, depending on whether simultaneous memory writes need to be consistent, and if not, how inconsistencies are resolved.

as the number of phases in a computation and *work* is defined as the total number of operations, which is linear in the time-processor product.

In this chapter we consider primarily the EREW model, which is the weakest of the PRAMs. In Section 10.9 we discuss adaptations of our algorithm to other parallel models such as the BSP [202], QSM [85], and QRQW [84].

10.2 History

In the parallel context, the MST problem is intimately related to several other problems, such as computing an arbitrary spanning forest of an undirected graph, and transitive closure. The earliest MST and connected components algorithms [106, 107, 178, 35] run in $O(\log^2 n)$ time on the exclusive write (EW) PRAMs, but were work-inefficient on sparse graphs. Since computing anything non-trivial takes $\Omega(\log n)$ time on the EREW or CREW PRAMs [130], these algorithms were not *that* far from optimal.

If a CRCW PRAM is assumed then the MST and connectivity problems become significantly simpler. Shiloach and Vishkin [181] showed how to compute connected components in $O(\log n)$ time on the CRCW model. Awerbuch and Shiloach [12] then gave an $O(\log n)$-time, $O(m \log n)$-work MST algorithm; however, they assumed a particularly strong form of the CRCW model. Using the same model, Cole and Vishkin [43] developed $O(\log n)$-time algorithms for connectivity and MST, the connectivity algorithm using $O(m\alpha(m, n))$ work and the MST algorithm using $O(m \log\log\log n)$ work. Gazit [83] gave a *randomized* CRCW connectivity algorithm running in logarithmic time and linear work. This is clearly work-optimal. However, on the CRCW PRAM the best known lower bounds on the time-complexity of connectivity and MST are $\Omega(\log n/ \log\log n)$ [130]. Complementing Gazit's CRCW algorithm, Halperin and Zwick [99, 100] developed an optimal randomized connectivity algorithm for the EREW PRAM.

Johnson and Metaxes, working within the CREW model, established a number of useful techniques for solving the connectivity [121] and MST [120] problems. Both their algorithms run in $O(\log^{1.5} n)$ time, a $\sqrt{\log n}$ improvement over the straightforward methods of [106, 107, 178, 35]. Chong and Lam [37] elaborated on the Johnson-Metaxes approach, obtaining an $O(\log n \log\log n)$-time connectivity algorithm in the EREW model. Just a couple years ago Chong, Han, and Lam [36] closed the book on the *deterministic* time complexity of MST and connectivity in the EREW PRAM model. They gave an elegant MST algorithm running in $O(\log n)$ time and $O(m \log n)$ work.

After the randomized *sequential* complexity of MST was shown to be linear [138, 127], a number of parallel linear-work algorithms were developed based on the same random sampling technique developed by Karger, Klein, and Tarjan [124, 126, 138, 127].

Karger [125] gave a randomized $O(\log n)$-time $O(m + n^{1+\epsilon})$-work MST algorithm in the EREW model, where $\epsilon > 0$ is any constant. Thus, it is work-optimal only for sufficiently dense graphs. Cole, Klein, and Tarjan [41], working within the CRCW model, gave a randomized $O(\log n)$-time, linear-work MST algorithm. This was an improvement over their previous algorithm [40], which ran in linear-work but $O(\log n \cdot 2^{\log^* n})$ time. Poon and Ramachandran [172] adapted the algorithm from [40] to the EREW model, obtaining a randomized $O(\log n \log\log n\, 2^{\log^* n})$-time linear-work MST algorithm. Poon and Ramachandran [175] later observed that their algorithm could be made to run in $O(\log n\, 2^{\log^* n})$ time using the Chong-Han-Lam algorithm [36] as a subroutine.

In this chapter we improve on the Poon-Ramachandran algorithm [172, 175] by reducing the time-complexity to $O(\log n)$, which is optimal, and improve on the Cole-Klein-Tarjan algorithm [40, 41] by obtaining the same bounds — logarithmic-time and linear-work — in the weaker EREW model. Our MST algorithm is the first such algorithm provably optimal in both work and time. Our algorithm can also be used to find an *arbitrary* spanning forest or compute the connected components of an undirected graph. It is arguably much simpler than the EREW algorithms of Halperin and Zwick for these problems [99, 100] and more desirable than Gazit's connectivity algorithm [83] for the stronger CRCW model.

In Section 10.3 we review some basic techniques used in parallel MST and connectivity algorithms. In Section 10.4 we give a high-level description of our algorithm; its details are addressed in Sections 10.5 and 10.6. In Section 10.7 we prove that our bounds — logarithmic time and linear work — hold with high probability, and in Section 10.8 we describe a simple processor allocation scheme used by our algorithm. It is general enough to capture many types of recursively defined programs. In Section 10.9 we discuss other parallel models. We conclude with some remarks in Section 10.10.

The algorithm of this chapter uses a *linear* number of random bits. In Chapter 11 we give a parallel, expected linear work MST algorithm using a polylogarithmic number of random bits. However, its time is suboptimal.

10.3 Techniques

Borůvka Steps

The primary operation used in all parallel MST algorithms is the Borůvka step[4] – see Section 7.2.3. Parallel connectivity algorithms use a similar step, usually called *hook-and-contract* or something equally pedestrian. The idea is very simple. In a connectivity algorithm (resp., MST algorithm) every vertex simultaneously identifies any

[4]However, Borůvka seems to have been discovered late in the parallel algorithms community. JáJá's 1992 textbook [114] attributes the algorithm to Sollin [183, 18].

incident edge (the lightest[5] incident edge) and contracts it, yielding a graph with at most half the number of non-isolated vertices. (Thus, after $\log n$ such steps all vertices are isolated.) Clearly this operation preserves the connected components of the graph. It also preserves the minimum spanning tree in a certain sense. By the *cut property* of MSTs (Section 7.2.2) and the definition of *contractibility* (Definition 5) the MST consists of the contracted edges plus the MST of the contracted graph.

Growth Control Schedule

A lasting contribution of Johnson and Metaxes's algorithm [120] was the idea of a *growth control schedule.* It is very simple to implement Borůvka steps in time that is logarithmic in the length of the *maximum* degree of the resulting graph, provided this degree is known a priori. To avoid overall running times of $\Theta(\log^2 n)$ it is necessary to restrict the nominal degree of the graph, either by ignoring high-degree vertices or throwing away edges, if only temporarily. The growth control schedule from [120] used both these techniques in concert; variations on it were used in all subsequent parallel MST/connectivity algorithms.

Random Sampling, MST Verification

Karger, Klein, and Tarjan's randomized MST algorithm [127] is based on a simple sampling lemma. Let an edge e be called *H-light* if $e \in MST(H \cup \{e\})$, and *H-heavy* otherwise.

Lemma 56 *[127] Let G be a graph on n vertices, and H be derived from G by sampling each edge independently with probability p. Then the expected number of H-light edges in G is less than n/p.*

By the cycle property of MSTs, if e is H-heavy and H is a subgraph of G, then $e \notin MST(G)$. Similarly, if e is T-light and $T = MST(H)$ then e is H-light as well. These simple observations are enough to establish the correctness of Karger et al.'s algorithm [127]; Lemma 56 is only used to bound its expected running time. We give below a parameterized version of Karger et al.'s algorithm.

```
KKT(G)      :  returns the MST of G
1.   Perform $c$ Borůvka steps; let $F$ be the MST edges
     identified and let $G_c$ be the resulting contracted graph.
2.   Obtain $G_s$ from $G_c$ by sampling each edge with prob. $p$.
3.   $T_s :=$ KKT($G_s$)
4.   Let $G_f$ be the $T_s$-light edges of $G_c$
```

[5]Recall from Chapter 7 that edge-weights are assumed to be distinct.

```
5.    $T := \mathbf{KKT}(G_f)$
6.    Return($F \cup T$)
```

Each Borůvka step can be easily implemented in linear time, as can the sampling step in Line 2. Karger et al. find the T_s-light edges in Line 4 with a linear-time MST verification routine [141, 57, 133, 16, 24]. Thus, for *fixed* c and p, the running time of **KKT** basically conforms to the following recurrence relation. (This is a sketch; the unpredictableness of random sampling must be taken into account.)

$$R(m,n) = R(pm,\, n/2^c) + R(n/(p2^c), n/2^c) + O(cm)$$

Both recursive calls have at most $n/2^c$ vertices due to the Borůvka steps in Line 1. The first recursive call has an expected pm edges, and by Lemma 56, the expected number of edges in the second recursive call is $n/(p2^c)$. Karger et al. [127] show that for $c = p^{-1} = 2$, the expected running time of their algorithm is $O(m)$. However, c and p need to be chosen very carefully in order to parallelize the **KKT** algorithm. For instance, for $c = 2$ the depth of recursion is $\frac{1}{2}\log n$ (Line 1 reduces the number of vertices by a factor of four); thus there are $2^{\log n/2} = \sqrt{n}$ recursive calls, none of which can be executed concurrently. This can be reduced to n^ϵ by setting $c = 1/\epsilon$ but that is not an interesting improvement. The Cole et al. [40] and Poon-Ramachandran [172, 175] parallel MST algorithms choose c and p based on the depth of recursion, where c and p^{-1} both increase roughly exponentially as the depth increases. As the sampling probability decreases with the depth, we expect to see fewer edges passed to Line 3 recursive calls, which allows us to perform more Borůvka steps in Line 1. More Borůvka steps in Line 1 reduce the number of vertices in the graph, which, by Lemma 56 reduces the number of edges in Line 5 recursive calls, letting us perform even more Borůvka steps, and so on. There are $2^{\log^* n}$ recursive calls, each requiring $O(\log n)$-time, which leads to the unsightly $O(\log n \cdot 2^{\log^* n})$ running times of [40, 172, 175]. The revised Cole et al. [41] algorithm used the same method – $2^{\log^* n}$ recursive calls – but with the weaker objective of contracting only *part* of the MST; they were then able to reduce the time of each recursive call to $o(\log n/2^{\log^* n})$. In a second phase, Cole et al. give an $O(\log n)$-time method to find the rest of the MST, provided the number of vertices is sufficiently small, in their case $n/\log\log\log n$.

10.4 The High-Level Algorithm

Our algorithm is divided into two phases along the lines of the CRCW PRAM algorithm in [41]. The objective of Phase 1 is to reduce the number of vertices in the graph by a factor of $k_0 = (\log^{(2)} n)^2$, by contracting a large set of MST edges, in particular a k_0-*min* *forest* — see Section 10.5 for definitions. Phase 1 is structured much like Cole et al.'s [41] parallelization of Karger et al.'s [127] randomized sequential MST algorithm. We use the now familiar recursion-tree of $2^{\log^* n}$ recursive calls introduced in [40]. The primary challenge in Phase 1 is reducing the expected time per recursive call to something $O(\log n/2^{\log^* n})$. In Phase 2 we use the same sampling approach of Karger et al. to find the rest of the MST (any MST edges not contracted in Phase 1). With high probability, Phase 2 runs in $O(\log n)$ time and linear work, provided that the number of vertices is at most n/k_0. The high-level algorithm is given in Figure 10.1.

```
High-Level($G$)
    (Phase 1)  $G_t$ := For all $v \in G$, retain the lightest $k_0$ edges in edge-list($v$)
               $F$ := Find-$k$-min($G_t, \log^* n$)
               Let $G_c$ be $G$ after contracting edges in $F$
    (Phase 2)  $G_s$:=Sample edges of $G_c$ with prob. $1/\sqrt{k_0} = 1/\log^{(2)} n$
               $T_s$ :=Find-MST($G_s$)
               $G_f$ := Filter($G_s, F_s$)
               $T$ :=Find-MST($G_f$)
               Return($F \cup T$)
```

Figure 10.1: A randomized time-work optimal parallel MST algorithm, at a high-level.

Theorem 16 *Using $m/\log n$ EREW processors, High-Level(G) returns the MST of G in $O(\log n)$ time, with probability $1 - n^{-\omega(1)}$.*

In Section 10.5 we describe our Phase 1 algorithm, which produces a k-min forest. In Section 10.6 we give the Phase 2 algorithm and a proof of Theorem 16. In analyzing our algorithm we assume perfect processor allocation: that N jobs can be fairly divided among P processors with zero overhead. In Section 10.8 we consider our processor allocation problem in a more abstract setting. Out scheme is an improvement over [100], which is terribly complicated, and [99], which uses superlinear space.

10.5 Phase 1

Phase 1 reduces the number of graph vertices by finding and contracting a *k-min forest*, for $k = k_0$, defined below. Our Find-k-min algorithm is analagous to one of Cole et al. [40] but is considerably simpler. It is based on two procedures, *Borůvka-A* and *k-Min-Filter*. Borůvka-A executes batches of Borůvka steps very efficiently, while guaranteeing that the length of edge-lists remain relatively small. The k-Min-Filter procedure is a modified MST verification routine [134] designed to identify *k-min-heavy* edges (rather than H-heavy edges as in Karger et al.'s algorithm [127]) which cannot be in any k-min forest.

10.5.1 The k-Min Forest

We define the k-min tree of a, or k-min(a), to be the first k edges selected by the Dijkstra-Jarník-Prim (DJP) algorithm, when starting at vertex a. Refer to Section 7.2.3 for a description of the DJP algorithm. Although we assumed the original graph is connected, graphs derived by sampling may not be. Therefore, if a is in a connected component with k or fewer vertices, k-min(v) consists of the MST of a's component. We call F a *k-min forest* of G if $F \subseteq MST(G)$ and k-min$(a) \subseteq F$ for all a. In a connected graph, contracting a k-min forest clearly reduces the number of vertices by a factor of at least $k+1$.

Let F be an arbitrary forest. We define $P_F(a,b)$ to be the path from a to b in F (if it exists) and define maxweight$\{A\}$ to be the maximum edge-weight among edges in A, and ∞ if $A = \emptyset$. Using this notation, an edge (a,b) would be F-light if if $w(a,b) \leq \text{maxweight}(P_F(a,b))$. We generalize the notion of F-lightness to k-min forests. Call an edge e *k-min-light* w.r.t. F if F is not a k-min forest of $F \cup \{e\}$. We will use an equivalent definition of k-min-lightness more suited to proofs. Below, the k-min tree notation is with respect to some graph F known from context. Define $w_a^k(b)$ to be:

$$w_a^k(b) \overset{\textbf{def}}{=} \begin{cases} w_M(a,b) & \text{if } b \in M \overset{\textbf{def}}{=} k\text{-min}(a) \\ \text{maxweight}(M) & \text{otherwise} \end{cases}$$

An edge (a,b) is k-min-light if either of the following hold:

$$\begin{aligned} w(a,b) &\leq \max\{w_a^k(b),\, w_b^k(a)\} \\ k &\geq \min\{|k\text{-min}(a)|,\, |k\text{-min}(b)|\} \end{aligned}$$

In either case, (a,b) would be picked by the DJP algorithm in the first k steps, when starting from a or b. If an edge is not k-min-light then it is *k-min-heavy.*

Claim 2 *If an edge (a,b) is k_1-Min-heavy w.r.t. G_1, it is also k_2-Min-heavy w.r.t. G_2 where $k_2 \leq k_1$, $V(G_1) = V(G_2)$ and $E(G_1) \subseteq E(G_2)$.*

Proof: This follows from two observations: $w_a^k(b)$ is non-decreasing in k, and $w_a^k(b)$ is non-increasing as new edges are added to the underlying graph.

□

In other words, whenever an edge is found to be k-Min-heavy for $k \geq k_0$ and w.r.t. some subset of the original graph, this is a certificate that the edge is k_0-Min-heavy in the original graph.

Claim 3 *Let 'k-min tree' be defined w.r.t. H, and let $T = MST(H)$. Then for any k, $w_a^k(b) \leq \text{maxweight}\{P_T(a,b)\}$.*

Proof: There are two cases, when b falls inside the k-Min tree of a, and when it falls outside. If b lies inside k-Min(a), then $w_a^k(b) = \text{maxweight}\{P_T(a,b)\}$ since k-min$(a) \subseteq MST(H)$. Now suppose, for the purpose of obtaining a contradiction, that b falls outside k-Min(a) and $w_a^k(b) > \text{maxweight}\{P_T(a,b)\}$. In other words, there is a path from a to b in T consisting of edges lighter than maxweight(k-Min(a)). However, at each step in the DJP algorithm, at least one edge in $P_T(a,b)$ is eligible to be chosen in that step. Since $b \notin k$-Min(a), the edge with weight maxweight$\{k$-Min$(a)\}$ is never chosen, a contradiction.

□

Lemma 57 *Let H be a graph formed by sampling each edge in graph G with probability p. The expected number of edges in G that are k-Min-light w.r.t. H, for any k, is less than n/p.*

Proof: We show that any edge in G that is k-Min-light w.r.t. H is also H-light. The lemma then follows directly from Lemma 56, the sampling lemma of Karger et al.

If (a,b) is k-min-light because $k \geq \min\{|k\text{-min}(a)|, |k\text{-min}(b)|\}$ then it connects two connected components of H and is therefore H-light as well. Now suppose (a,b) is k-min-light because $w(a,b) \leq \max\{w_a^k(b),\, w_b^k(a)\}$. By Claim 3, $w_a^k(b) \leq \text{maxweight}(P_T(a,b))$, and therefore $w(a,b) \leq \text{maxweight}(P_T(a,b))$, which is the definition of (a,b) being H-light.

□

The procedure Find-k-min(G,d) takes as input a graph G and a suitable positive integer d, and returns a k_0-Min forest of G. The d argument relates to the depth of recursion, and is used to adjust certain parameters based on this depth. If $d = \log^* n$ on the initial call to Find-k-min, it runs in logarithmic time and linear work with high probability.

Since Phase 1 is concerned only with the k_0-Min tree of each vertex, it suffices to retain only the lightest k_0 edges incident on each vertex. Hence as stated in the first step of Phase 1 in algorithm High-Level from Section 10.4, we will discard all but the lightest k_0 edges incident on each vertex. They will be reconsidered in Phase 2. This step can be performed in logarithmic time and linear work by a simple randomized selection algorithm.

10.5.2 Borůvka-A Steps

In a basic Borůvka step [21], each vertex chooses its minimum weight incident edge, inducing a number of trees. We contract all such trees into single vertices and discard any self-loops.

Our algorithm for Phase 1 uses a *modified Borůvka step* in order to reduce the time bound to $o(\log n)$ per step. All vertices are classified as being either *live* or *dead.* Only live vertices participate in modified Borůvka steps, and a vertex is designated dead only when we are sure its k_0-min tree has been found. After a modified Borůvka step, if (a, b) is the least weight edge incident on a then the *parent pointer* of a, $p(a)$, is set to b. This induces a number of rooted trees. (A Borůvka step actually induces trees with two roots, each the other's parent. We assume that only one root is retained.) In addition, each vertex has a *threshold* which keeps the weight of the lightest discarded edge incident to a. The algorithm discards edges known not to be in the k_0-Min tree of any vertex. The threshold variable guards against vertices choosing edges which may not be in the MST.

The following three claims refer to any tree resulting from a (modified) Borůvka step. Their proofs are straightforward and are omitted.

Claim 4 *The sequence of edge weights encountered on a path from a to root(a) is monotonically decreasing.*

Claim 5 *If depth(a) $=$ d then d-Min(a) consists of the edges in the path from a to root(a). Furthermore, the weight of $(a, p(a))$ is greater than any other edge in d-Min(v).*

Claim 6 *If the minimum-weight incident edge of a is (a, b), k-Min$(a) \subseteq k$-Min$(b) \cup \{(a, b)\}$.*

Claim 7 may not be as obvious. A similar claim was proved in [36].

Claim 7 *Let T be a tree induced by a Borůvka step, and let T' be a subtree of T. If e is the minimum weight incident edge on T, then the minimum weight incident edge on T' is either e or an edge of T.*

Proof: Suppose, on the contrary that the minimum weight incident edge on T' is $e' \notin T$, and let v and v' be the end points of e and e' which are inside T. Consider the paths P (P') from v (v') to the root of T. By Claim 4, the edge weights encountered on P and P' are monotonically decreasing. There are two cases. If T' contains some, but not all of P', then e' must lie along P', a contradiction. If T' contains all of P', but only some of P, then some edge $e'' \in P$ is adjacent to T'. Then $w(e') < w(e'') < w(e)$, also a contradiction.

□

The procedure Borůvka-A(H, l, F), given in Figure 10.2, returns a contracted version of H with the number of live vertices reduced by a factor of l. All contracted edges, which are guaranteed to be in $MST(H)$, are returned in the set F.

```
Borůvka-A($H$, $l$; $F$)
        $F := \emptyset$
        Repeat $\log l$ times: ($\log l$ modified Borůvka steps)
                $F' := \emptyset$
                For each live vertex $a$
                        Choose min. weight edge $(a, b)$
                        If $w(a, b) >$ threshold$(a)$
(1)                             mark $a$ as dead; else
                        Else
                                p$(a) := b$
                                $F' := F' + (a, \mathrm{p}(a))$
                Each tree $T$ induced by parent pointers is one of two types:
                        If root of $T$ is dead, then
(2)                             Mark all vertices in $T$ as dead (Claim 6)
                        If $T$ contains only live vertices,
(3)                             For each $a \in T$ at depth $\geq k_0$, mark $a$ dead (Claim 5)
                                Contract the subtree of $T$ made up of live vertices.
                                The resulting vertex is live, has no parent pointer, and
                                keeps the smallest threshold of its constituent vertices.
                $F := F + F'$
```

Figure 10.2: The Borůvka-A procedure.

Lemma 58 *If Borůvka-A designates a vertex as dead, its k_0-Min tree has already been found.*

Proof: Vertices make the transition from live to dead only at the lines indicated by a number. By our assumption that we only discard edges that cannot be in the k_0-Min tree of any vertex, if the lightest edge adjacent to any vertex has been discarded, we know its k_0-Min tree has already been found. This covers line (1). The correctness of line (2) follows from Claim 6. Since $(a, p(a))$ is the lightest incident edge on a, k_0-Min$(a) \subseteq k_0$-Min$(p(a)) \cup \{(a, p(a))\}$. Thus, if the root of T is dead, all vertices at depth one are dead, implying those at depth two are dead, and so on. The validity of line (3) follows directly from Claim 5. If a vertex finds itself at depth $\geq k_0$, its k_0-Min tree lies along the path from the vertex to its root.

□

Lemma 59 *After a call to Borůvka-A$(H, k_0 + 1, F)$, the k_0-Min tree of each vertex is a subset of F.*

Proof: By Lemma 58, dead vertices already satisfy the lemma. After a single modified Borůvka step, the set of parent pointers associated with live vertices induce a number of trees. Let $T(a)$ be the tree containing a. We assume inductively that after $\lceil \log i \rceil$ modified Borůvka steps, the $(i-1)$-Min tree of each vertex in the original graph has been found (this is clearly true for $i = 1$). For any live vertex a let (x, y) be the minimum weight edge s.t. $x \in T(a)$, $y \notin T(a)$. By the inductive hypothesis, the $(i-1)$-Min trees of a and y are subsets of $T(a)$ and $T(y)$ respectively. By Claim 7, (x, y) is the first external edge of $T(a)$ chosen by the DJP algorithm, starting at a. As every edge in $(i-1)$-Min(y) is lighter than (x, y), $(2(i-1)+1)$-Min(a) is a subset of $T(a) \cup \{(x, y)\} \cup T(y)$. Since edge (x, y) is chosen in the $(\lceil \log i \rceil + 1)^{th}$ modified Borůvka step, $(2i-1)$-Min(a) is a subset of $T(a)$ after $\lceil \log i \rceil + 1 = \lceil \log 2i \rceil$ modified Borůvkasteps. Thus after $\log(k_0 + 1)$ steps, the k_0-Min tree of each vertex has been found.

□

Lemma 60 *After b modified Borůvka steps, the length of any edge-list is bounded by ${k_0}^{{k_0}^b}$.*

Proof: This is true for $b = 0$. Assuming the lemma holds for $b - 1$ modified Borůvka steps, the length of any edge-list after that many steps is $\le {k_0}^{{k_0}^{b-1}}$. Since we only contract trees of height $< k_0$, the length of any edge-list after b steps is less than $({k_0}^{{k_0}^{b-1}})^{k_0} = {k_0}^{{k_0}^b}$.

□

It is shown in the next section that our algorithm only deals with graphs that are the result of $O(\log k_0)$ modified Borůvka steps. Hence the maximum length edge-list is ${k_0}^{{k_0}^{O(\log k_0)}}$.

The costliest step in Borůvka-A is calculating the depth of each vertex. After parent pointers are chosen (min weight incident edges), the root of each induced tree will broadcast its depth to all depth 1 vertices, which in turn broadcast to depth 2 vertices, etc. Once a vertex knows it is at depth $k_0 - 1$, it may stop, letting all its descendents infer that they are at depth $\ge k_0$.

Lemma 61 *Let G_1 have m_1 edges. Then a call to Borůvka-A$(G_1, l;\ F)$ can be executed in time $O({k_0}^{O(\log k_0)} + \log l \cdot \log n \cdot (m_1/m))$ with $m/\log n$ processors.*

Proof: Let G_1 be the result of b modified Borůvka steps. By Lemma 60, the maximum degree of any vertex after the i^{th} modified Borůvka step in the current call to Borůvka-A is ${k_0}^{{k_0}^{b+i}}$. Let us now look at the required time of the i^{th} modified Borůvka step. Selecting the minimum cost incident edge takes time $O(\log {k_0}^{{k_0}^{b+i}})$, while the time to determine the depth of each vertex is $O(k_0 \cdot \log {k_0}^{{k_0}^{b+i}})$. Summing over the $\log l$

modified Borůvka steps, the total time is bounded by $\sum_i^{\log l} {k_0}^{O(b+i)} = {k_0}^{O(b+\log l)}$. As noted above, the algorithm performs $O(\log k_0)$ modified Borůvka steps on any graph, hence the time is ${k_0}^{O(\log k_0)}$.

The work performed in each modified Borůvka step is linear in the number of edges. Summing over $\log l$ such steps and dividing by the number of processors, we arrive at the second term in the stated running time.

□

10.5.3 Filtering Edges via The Filter Forest

We will maintain, concurrent with the operation of Borůvka-A, a structure called the *Filter forest.* This collection of rooted trees records which vertices merged together and the edge weights involved. (This structure appeared first in [133].) If v is a vertex of the original graph, or a new vertex resulting from contracting a set of edges, there is a corresponding vertex $\phi(v)$ in the Filter forest. During a Borůvka step, if a vertex v becomes dead, a new vertex x is added to the Filter forest, as well as a directed edge $(\phi(v), x)$ having the same weight as $(v, p(v))$. If live vertices $v_1, v_2, \ldots, v_j$ are contracted into a live vertex v, a vertex $\phi(v)$ is added to the Filter forest in addition to edges $(\phi(v_1), \phi(v))$, $(\phi(v_2), \phi(v))$, $\ldots$, $(\phi(v_j), \phi(v))$, having the same edge-weights as $(v_1$,p$(v_1))$, $(v_2$,p$(v_2))$, $\ldots$, $(v_j$,p$(v_j))$, respectively. We make the simple observation that the edge weights on the path from $\phi(u)$ to $root(\phi(u))$ are exactly the weights of those edges chosen by u (or its representative) in previous Borůvka steps.

It is shown in [133] that the heaviest weight in the path from u to v in the MST is the same as the heaviest weight in the path from $\phi(u)$ to $\phi(v)$ in the Filter forest (if there is such a path). We extend this scheme to handle k-Min-lightness.

Let $P_{\text{ff}}(u, v)$ be the path from u to v in the Filter forest, if it exists. Define $w_{\text{ff}}(u, v)$ as:

$$w_{\text{ff}}(u,v) \overset{\text{def}}{=} \begin{cases} \text{maxweight}\{P_{\text{ff}}(\phi(u), \phi(v))\} & \text{if } \phi(u), \phi(v) \text{ in same filter tree} \\ \text{max}\{ & \text{otherwise} \\ \quad \text{maxweight}\{P_{\text{ff}}(\phi(u), \text{ root}(\phi(u)))\} & \\ \quad \text{maxweight}\{P_{\text{ff}}(\phi(v), \text{ root}(\phi(v)))\} & \\ \} & \end{cases}$$

The value $w_{\text{ff}}(u, v)$ represents our best lower bound on maxweight$\{P_{MST}(u, v)\}$, given the information contained in the filter forest. The w_{ff} values are used by the procedure k-Min-Filter (called from Find-k-min, Section 10.5.1) to remove, or *filter* edges known not to be in the k-min tree of any vertex. In a call to k-Min-Filter(H, F), where F is a forest of H derived by some number of modified Borůvka steps, each edge e of H is examined and possibly filtered. If either endpoint of e is in a tree in F with fewer than k_0 edges then e is trivially k_0-min-light and is retained. Otherwise, e is filtered precisely when $w(e) < w_{\mathit{ff}}(e)$. We show below that all k_0-min-light edges are retained and that not too many edges are kept in total.

To implement the k-Min-Filter procedure we use a slight modification to the $O(\log n)$ time, $O(m)$ work MST verification algorithm of [134]. If $e = (u, v)$ is the edge being tested and $\phi(u), \phi(v)$ are not in the same Filter tree, we test the pairs $(\phi(u), root(\phi(u))$ and $(\phi(v), root(\phi(v))$ instead and delete e if both of these pairs are identified to be deleted. A finer analysis of the [134] algorithm actually shows that it runs in $O(\log r)$ time, where r is the size of the largest filter tree.

Lemmas 62 and 63, proved below, establish the correctness of the filtering procedure.

Lemma 62 *Suppose b modified Borůvka steps were applied to a graph, then for any vertex u and some $k \geq \min\{k_0, 2^b - 1\}$,*

$$\text{maxweight}\{P_{\mathit{ff}}(\phi(u), root(\phi(u)))\} = \text{maxweight}\{k\text{-}Min(u)\}$$

Before proving this we first prove a necessary technical lemma.

Lemma 63 *Let T be a tree of MST edges after an arbitrary number of Borůvka steps and let $T' = T \cup \{(v, w)\}$, where (v, w), $v \in T$, $w \notin T$ is the edge chosen by T in the next Borůvka step. For any $u \in T$, the maximum weight edge in $P_{T'}(u, w)$ was chosen by the tree containing u in some Borůvka step.*

Proof: Let T be formed after b Borůvka steps. Suppose, without loss of generality, that the lemma is falsified for the first time after the b^{th} Borůvka step. That is, the heaviest edge in $P_{T'}(u, w)$, say f, was chosen in the b^{th} step. Let $g \neq f$ be the edge chosen by u or u's representative tree in this step. If f lies between g and the root of T then by Claim 4 it is lighter than g, and similarly, if it lies between vertex v and the root of T then it is lighter than (v, w). Both cases are contradictions.

□

We are now ready to prove Lemma 62.

Proof: Let $e(u, b)$ be the maximum weight edge chosen by u's tree in the first b Borůvka steps. Assume inductively that $w(e(u, b-1)) = \text{maxweight}\{k(u, b-1)\text{-Min}(u)\}$ where $k(u, b-1) \geq 2^{b-1} - 1$ if u is live, $k(u, \cdot) \geq k_0$ if u is dead, and $k(u, b-1)$-Min(u) is contained in a tree of MST edges after $b-1$ Borůvkasteps. If u is dead it already satisfies the inductive claim for b Borůvka steps, so assume u is alive. Let (z_1, z_2) be the edge chosen by the tree containing u in the b^{th} Borůvka step and let P be the MST path connecting $k(u, b-1)$-Min(u) to z_1 — see Figure 10.3 for a schematic diagram. We have that $w(z_1, z_2) > w(e(z_2, b-1))$, because (z_1, z_2) was not already chosen by z_2 in the first $b-1$ steps, and $\text{maxweight}\{e(u, b-1) + P + (z_1, z_2)) = w(e(u, b)\}$. This is true because

$w(e(u,b)) = \text{maxweight}\{e(u,b-1),\ (z_1,z_2)\} > \text{maxweight}\{P\}$, where the equality is by definition and the inequality by Lemma 63. Let D be the subgraph

$$D = k(u,b-1)\text{-Min}(u) + P + (z_1,z_2) + k(z_2,b-1)\text{-Min}(z_2)$$

and $k(u,b)$ be the smallest number such that $k(u,b)\text{-Min}(u) \supseteq D$. It follows that $e(u,b)$ is the heaviest edge in $k(u,b)\text{-Min}(u)$ because when the DJP algorithm is started from u, until all edges from D are chosen there is *some* eligible edge from D weighing no more than the edge $e(u,b)$.

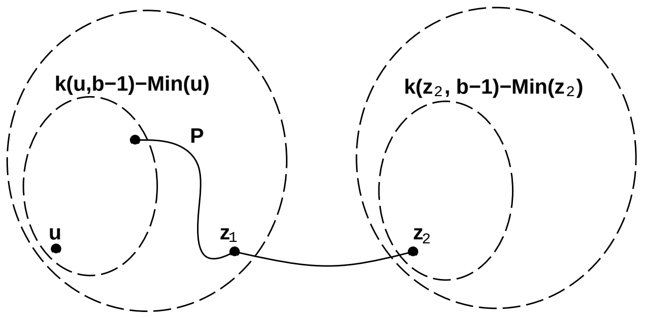


Figure 10.3: The larger ovals represent the trees of MST edges after $b-1$ Borůvka steps containing u and z_2, resp. The smaller ovals are the $k(u,b-1)$-Min(u) tree and the $k(z_2,b-1)$-Min(z_2) tree.

If u and z_2 remain live then $k(u,b) \geq 2 \cdot (2^{b-1}-1)+1 \geq 2^b - 1$. On the other hand, if u becomes dead after the b^{th} Borůvka step then (z_1,z_2) is the heaviest edge at the end of a chain C of length at least k_0 and $k(u,b) \geq 2^{b-1} - 1 + |C| \geq k_0$. In either case our inductive claim is proved for b Borůvka steps.

□

Lemma 64 *Suppose the k-Min-Filter procedure is only called on graphs after performing at least* $\log(k_0+1)$ *Borůvka steps. Then no* k_0*-Min-light edges are filtered, and all unfiltered edges are k-Min-light for some* $k \geq k_0$*.*

Proof: Consider an edge (u,v) examined by the k-Min-Filter procedure that is not trivially k_0-min-light. Note that if $\phi(u)$ is in the same filter tree as $\phi(v)$, by King's observation [133], $w_{\text{ff}}(u,v) = \text{maxweight}\{P_{MST}(u,v)\}$. Therefore, by Claim 2, if $w(u,v) > w_{\text{ff}}(u,v)$ then (u,v) is k_0-Min-heavy and may be safely filtered. On the other hand, if $w(u,v)$ is less than $w_{\text{ff}}(u,v)$, then (u,v) is k-Min-light for $k = n-1$. We therefore focus on the case when $\phi(u)$ and $\phi(v)$ are in different filter trees. By Lemma 62, for some k_1, k_2 we have that:

$$\text{maxweight}\{P_{\text{ff}}(\phi(u), root(\phi(u)))\} = \text{maxweight}\{k_1\text{-Min}(u)\}$$

$$\text{maxweight}\{P_{\text{ff}}(\phi(v), root(\phi(v)))\} = \text{maxweight}\{k_2\text{-Min}(v)\}$$

Thus $w_{\text{ff}}(u,v) = \text{maxweight}\{k_1\text{-Min}(u) \cup k_2\text{-Min}(v)\}$ Since maxweight$\{k$-Min$(u)\}$ is a non-decreasing function of k, if (u,v) is not filtered out then by Claim 2 it must be k_3-Min-light where $k_3 = \max\{k_1, k_2\}$. On the other hand, if (u,v) is filtered out then it must be k_4-Min-heavy where $k_4 = \min\{k_1, k_2\}$. Because the k-Min-Filter procedure is applied only after performing at least $\log(k_0+1)$ Borůvka steps, by Lemma 62 $k_3, k_4 \geq k_0$.

□

Remark. k-Min-Filter is responsible for updating the threshold variables – see Section 10.5.2. When an edge (u,v) is discarded, threshold(u) is updated to reflect the weight of the lightest discarded edge incident to u; threshold(v) is updated similarly.

10.5.4 Finding a k-Min Forest

We are now ready to present the main procedure of Phase 1, Find-k-min, which is given in Figure 10.4. (Recall that the initial call – given in Section 10.4 – is Find-k-min$(G_t, \log^* n)$, where G_t is the graph obtained from G by removing all but the k_0 lightest edges on each adjacency list.)

```
Find-k-Min(H, i)
        $H_c$ := Borůvka-A($H, (\log^{(i-1)} n)^4, F$)
        if $i = 3$, return($F$)
        $H_s$ := sample edges of $H_c$ with prob. $1/(\log^{(i-1)} n)^2$
        $F_s$ := Find-k-min($H_s, i-1$)
        $H_f$ := k-Min-Filter($H_c, F_s$)
        $F'$ := Find-k-min($H_f, i-1$)
        Return($F + F'$)
```

Figure 10.4: The Find-k-min procedure.

H is a graph with some vertices possibly marked as dead; i is a parameter that indicates the level of recursion (which determines the number of Borůvka steps to be performed and the sampling probability). Lemmas 65 and 66 establish the correctness of this procedure. The performance of Find-k-Min is analyzed in Section 10.5.5.

Lemma 65 *Let H' be a graph formed by sampling each edge in H with probability p, and F be a k_0-Min forest of H' (derived from at least $\log(k_0+1)$ Borůvka steps). The call to k-Min-Filter(H,F) returns a graph containing a k_0-Min forest of H, whose expected number of edges is no more than n/p.*

Proof: By Claim 2 any edge in the k_0-Min forest of H is k_0-Min-light w.r.t. H'. By Lemma 64, no edges k_0-Min-light w.r.t. H' are filtered; this establishes the first part of the lemma. By the second part of Lemma 64, all edges not filtered are k-Min-light w.r.t. H', for some k. According to Lemma 57 the expected number of edges in H that are k-Min-light w.r.t. H', for any k, is no more than n/p. This establishes the rest of the lemma.

□

Lemma 66 *The call Find-k-min($G_t, \log^* n$) returns a set of edges that includes the k_0-Min tree of each vertex in G_t.*

Proof: The proof is by induction on i. For $i = 3$ (the base case) Find-k-min$(H, 3)$ returns F, which by Lemma 59 contains the k_0-min tree of each vertex. Now assume inductively that Find-k-min$(H, i-1)$ returns the k_0-min tree of H. Consider the call Find-k-min(H, i). By the induction assumption the call to Find-k-min$(H_s, i-1)$ returns the k_0-min tree of each vertex in H_s. By Lemma 65 the call to k-Min-Filter(H_c, F_s) returns in H_f a set of edges that contains the k_0-Min trees of all vertices in H_c. Finally, by the inductive assumption, the set of edges returned by the call to Find-k-min$(H_f, i-1)$ contains the k_0-Min trees of all vertices in H_f. Since F' contains the $(\log^{(i-1)} n)$-Min tree of each vertex in H, and Find-k-min(H, i) returns $F + F'$, it returns the edges in the k_0-Min tree of each vertex in H.

□

10.5.5 Performance of Find-k-min

In this section we bound the time and work required by the Find-k-min procedure.

Claim 8 *The following invariants are maintained at each call to Find-k-min. The number of live vertices in $H \le n/(\log^{(i)} n)^4$, and the expected number of edges in $H \le m/(\log^{(i)} n)^2$, where m and n are the number of edges and vertices in the original graph.*

Proof: These hold for the initial call, when $i = \log^* n$. By Lemma 59, the contracted graph H_c has $\le n/(\log^{(i-1)} n)^4$ live vertices. Since H_s is derived by sampling edges with probability $1/(\log^{(i-1)} n)^2$, the expected number of edges in H_s is $\le m/(\log^{(i-1)} n)^2$, maintaining the invariants for the first recursive call.

By Lemma 57, the expected number of edges in $H_f \le \frac{n(\log^{(i-1)} n)^2}{(\log^{(i-1)} n)^4} = \frac{n}{(\log^{(i-1)} n)^2}$. Since H_f has the same number of vertices as H_c, both invariants are maintained for the second recursive call.

□

Lemma 67 *Find-k-min$(G_t, \log^* n)$ runs in expected $O(\log n)$ time and $O(m)$ work.*

Proof: Since recursive calls to Find-k-min proceed in a sequential fashion, the total running time is the sum of the local computation performed in each invocation. Aside from randomly sampling the graph the local computation consists of calls to k-Min-Filter and Borůvka-A.

In a given invocation of Find-k-min, the number of Borůvka steps performed on graph H is the sum of all Borůvka steps performed in all ancestral invocations of Find-k-min, i.e. $\sum_{i=3}^{\log^* n} O(\log^{(i)} n)$, which is $O(\log^{(3)} n)$. From our bound on the maximum length of edge lists (Lemma 60), we can infer that the size of any tree in the filter forest is ${k_0}^{{k_0}^{O(\log^{(3)} n)}}$. Thus the time needed for each modified Borůvka step and each call to k-Min-Filter is ${k_0}^{O(\log^{(3)} n)}$. Summing up, the total time required is $o(\log n)$.

The work required by the k-Min-Filter procedure and *each* Borůvka step is linear in the number of edges. By Claim 8 the expected number of edges in an invocation at level i is $O(m/(\log^{(i)} n)^2)$. Since there are $O(\log^{(i)} n)$ Borůvka steps performed in this invocation, the work required is $O(m/\log^{(i)} n)$. There are $2^{\log^* n - i}$ invocations with depth parameter i, therefore the total work is given by $\sum_{i=3}^{\log^* n} 2^{\log^* n - i} O(m/\log^{(i)} n)$, which is $O(m)$.

$\square$

10.6 Phase 2

Recall the Phase 2 portion of our overall algorithm **High-Level**:

```
(the number of vertices in $G_s$ is $\leq n/k_0$)
$G_s$ :=Sample edges of $G'$ with prob. $1/\sqrt{k_0} = 1/\log^{(2)} n$
$F_s$ :=Find-MST($G_s$)
$G_f$ := Filter($G', F_s$)
$F$ := Find-MST($G_f$)
```

The procedure Filter(G, F) returns the F-light edges of G; it runs in logarithmic time and linear work [134]. The procedure Find-MST(G_1), described below, finds the MST of G_1 in $O(\frac{m_1}{m} \log n \log^{(2)} n)$ time, where m_1 is the number of edges in G_1.

The graphs G_s and G_f each have expected $m/\sqrt{k_0} = m/\log^{(2)} n$ edges since G_s is derived by sampling each edge with probability $1/\sqrt{k_0}$, and by Lemma 56, the expected number of edges in G_f is $(m/k_0)/(1/\sqrt{k_0}) = m/\sqrt{k_0}$. Because we call Find-MST on graphs having expected size $O(m/\log^{(2)} n)$, each call takes $O(\log n)$ time.

10.6.1 The Find-MST Procedure

The procedure Find-MST(H), given in Figure 10.5, is similar to previous randomized parallel algorithms except it uses no recursion. In place of recursive calls we use a special *base case* algorithm. We also use slightly different Borůvka steps in order to reduce the work.

As its Base-case, we use the simplest version of the algorithm of Chong et al. [36], which takes time $O(\log n)$ using $m \log n$ processors. By guaranteeing that it is only called on graphs of expected size $O(m/\log^2 n)$, the running time remains $O(\log n)$ with $m/\log n$ processors.

```
Find-MST($H$)
    $H_c$ := Borůvka-B($H, \log^4 n$; $F$)
    $H_s$ := Sample edges of $H_c$ with prob. $p = 1/\log^2 n$
    $F_s$ := BaseCase($H_s$)
    $H_f$ := Filter($H_c, F_s$)
    $F'$ := BaseCase($H_f$)
    Return($F + F'$)
```

Figure 10.5: The Find-MST procedure.

After the call to Borůvka-B, the graph H_c has $< m/\log^4 n$ vertices. Since H_s is derived by sampling the edges of H_c with probability $1/\log^2 n$, the expected number of edges to the first BaseCase call is $O(m/\log^2 n)$. By the sampling lemma of [127], the expected number of edges to the second BaseCase call is $< (m/\log^4 n)/(1/\log^2 n)$, thus the total time spent in these subcalls is $O(\log n)$. Assuming the size of H conforms to its expectation of $O(m/\log^{(2)} n)$, the calls to Filter and Borůvka-B also take $O(\log n)$ time, as described below.

The Borůvka-B(H, l, F) procedure, Figure 10.6, returns a contracted version of H with $O(m/l)$ vertices. It uses a simple growth control schedule, designating vertices as *inactive* if their degree exceeds l. We can determine if a vertex is inactive by performing list ranking on its edge list for $\log l$ time steps. If the computation has not stopped after this much time, then its edge list has length $> l$.

The last step takes $O(\log n)$ time; all other steps take $O(\log l)$ time, as they deal with edge lists of length at most l. Consequently, the total running time is $O(\log n + \log^2 l)$. For each iteration of the main loop, the work is linear in the number of edges. Assuming the graph conforms to its expected size of $O(m/\log^{(2)} n)$, the total work is linear.

Remark. Borůvka-B is essentially one phase of the Johnson-Metaxas algorithm [120]. They introduced the edge-plugging technique, as well as the idea of a growth control schedule.

```
Borůvka-B$(G, l, F)$
    Repeat $\log l$ times
        For each vertex, let it be inactive if its edge list
        has more than $l$ edges, and active otherwise.
        For each active vertex $v$
            Choose min. weight incident edge $e$
            $F := F + e$
        Using the edge-plugging technique, build a
        single edge list for each induced tree ($O(1)$ time)
    Contract all trees of inactive vertices
```

Figure 10.6: The Borůvka-B procedure.

10.7 Proof of Theorem 16

Correctness

Consider the set of edges F returned by Find-k-Min. F is a k-Min forest and by definition a subset of the MST of G. By contracting the edges of F to produce G_c, we have $MST(G) = F \cup MST(G_c)$. The call to Filter in Phase 2 returns a graph G_f derived from G_c by removing edges known not to be in the MST of G_c, thus $MST(G_c) = MST(G_f)$. Assuming the correctness of Find-MST, it returns the MST T of G_f; thus, $T + F = MST(G)$. The correctness of Find-MST is established by a nearly identical argument, and relies on the correctness of our BaseCase algorithm [36].

High Probability Bounds

We have already proved that each step of our algorithm runs in logarithmic time assuming (a) we have a constant-time perfect processor allocation procedure, and (b) that the sizes of all graphs encountered by the algorithm are roughly as large as expected. We address (b) below. In particular we give an upper bound on the size of every graph examined by our algorithm, that holds with probability $1 - \exp(-\Omega(m/log^2 n))$. This is significantly better than the $n^{-\omega(1)}$ failure probability claimed in Theorem 16. It turns out that the bottleneck, in terms of failure probability, is the randomized processor allocation procedure we use, given in Section 10.8.

Phase 1: Consider a single invocation Find-k-min(H, i), where H has m' edges and n' vertices. We bound the size of the two graphs passed to recursive calls of Find-k-min in terms of m' and i. For the first recursive call, the edges of H are sampled independently with probability $p_i = 1/(\log^{(i-1)} n)^2$, which depends on the recursion depth $\log^* n - i$.

Call the sampled graph H_1. By applying a Chernoff bound [10], the probability that the size of H_1 is less than twice its expectation of $p_i m'$ is $1 - \exp(-\Omega(p_i m'))$.

Turning to the second recursive call, let $F = MST(H_1)$. The sampling lemma from [127] states that the number of F-light edges of H is dominated by the negative binomial distribution with parameters n' and p_i. As we saw in the proof of Lemma 57, every k-Min-light edge must also be F-light. Using this observation, we will analyze the size of the second recursive call in terms of F-light edges, and conclude that any bounds we attain apply equally to k-Min-light edges. The probability of at least r F-light edges is at most the probability of fewer than n' "heads" in a sequence of r coin flips, with probability p_i of heads (see [127]). By applying a Chernoff bound, this is $\exp(-\Omega(r))$ for $r \geq 2n'/p_i$. In this particular instance of Find-k-min, $n' \leq p_i^2 m$, so the probability that fewer than $2p_i m$ edges are F-light is $1 - \exp(-\Omega(p_i m))$.

Given a single invocation of Find-k-min(H, i), we can bound the probability that H has more than $2^{\log^* n - i} p_{i+1} m$ edges by $\exp(-\Omega(p_{i+1} m)$. This follows from applying the argument used above to each invocation of Find-k-min from the initial call to the current call at depth $\log^* n - i$. Summing over all recursive calls to Find-k-min, the total number of edges in all graphs (and thus the total work) is bounded by:

$$\sum_{i=3}^{\log^* n} 2^{2\log^* n - 2i} p_{i+1} m = O(m)$$

with probability $1 - \exp(-\Omega(p_4 m) = 1 - \exp(-\Omega(m/(\log\log\log n)^2))$. Using a nearly identical argument, one can show the probability that Phase 2 uses $O(m)$ work is $1 - \exp(-\Omega(m/\log^2 n))$.

10.8 Processor Allocation

Halperin and Zwick [99] give a processor allocation algorithm well-suited to our algorithm; however, one drawback is that it uses superlinear space. In [100] they claim a linear-space allocation procedure; however, it is not given a clear description, and more seriously, it makes heavy use of a non-trivial linked-list based sorting algorithm of Goodrich and Kosaraju [93]. In this Section, we give a simple, self-contained processor allocation scheme that occupies linear space and does not use any sorting procedure. It works on a variety of "tree-structured" computations.

Let M be a set of m processes which perform some computation. So long as the computation is *tree structured*, in the sense given below, its exact nature is unimportant. At any point in the computation there is a set $D \subseteq M$ of *dead processes* and a stack $\mathcal{S} = (S_0, S_1, \ldots, S_d)$ where $S_0 = M$ and $S_{j+1} \subseteq S_j$. In the i^{th} round of computation, the stack is potentially changed (with a push or pop) and some set $R_i \subseteq M$ of the processes compute for t_i time steps. Round i follows these steps:

1. Either a) $\mathcal{S}$ is unchanged, $R_i := S_d - D$
   or b) $\mathcal{S} := (S_0, \ldots, S_d, S_{d+1})$, $S_{d+1} \subseteq S_d$, $R_i := S_{d+1} - D$
   or c) $\mathcal{S} := (S_0, \ldots, S_{d-1})$, $R_i := S_{d-1} - D$
2. The R_i do something for $t_i \geq 1$ steps
3. $D := D + \{\text{some subset of } R_i\}$

This is a rather technical characterization of a class of algorithms. Informally, any recursive algorithm fits into this scheme if the active processes in one recursive call are a subset of those inherited from its parent call.

Let $p \leq m$ be the number of EREW processors available. Ideally we would like to simulate round i in $O(\lceil R_i/p \rceil)$ time (i.e. with zero overhead). Like [99] our overhead is non-constant but usually negligible.

Theorem 17 *For some tree computation, let r be the total number of rounds, $T = \sum_i t_i$ be the total time for all rounds, $W = \sum_i t_i \cdot |R_i|$ be the total work for all rounds, d_{max} be the maximum depth of the stack, and $q = \Omega(\log(mr))$ be a parameter. Then with probability $1 - e^{-\Omega(q)}$ the computation of m processes can be simulated with p EREW processors in $O(T + W/p + r\log q + \log p)$ time. The space required is $O(m + p \cdot d_{max})$. It is assumed there is some (easily computable) bound r_i such that $r_i \geq |R_i|$ and $r_i = O(|R_i|)$.*

In our MST algorithm the number of rounds $r = O(2^{\log^* n} k_0 \log k_0) = O((\log\log n)^3)$, the time $T = O(\log n)$, work $W = O(m)$, and $d_{max} = O((\log\log n)^3)$. Plugging these values into Theorem 17, our MST algorithm can be simulated in $O(m/p + \log n + \log q(\log\log n)^3)$ time with probability at least $1 - e^{-\Omega(q)}$, using space $O(m + p(\log\log n)^3)$. Since $p < m/\log n$ the space is linear in m. We could set $q = \Theta(\log(mr)) = \Theta(\log n)$ and achieve a polynomially small error probability, or set q as high as $2^{\log n/(\log\log n)^3}$ for an error probability of $n^{-\omega(1)}$, as promised in Theorem 16. Also, the "dead processes" in Theorem 17 correspond to those edges known to be k_0-Min-heavy (in Phase 1) or not in the MST at all (in Phase 2).

As in [99] we organize the processes into *blocks* of size $b = qm/p$, (q processors per block,) as follows. We imagine placing the processes deterministically into an $m/b \times b$ array, then performing a random rotation on each column. The processes that end up in the same row are in the same block. Computing this initial allocation is easily done in $O(m/p + \log p)$ time. Since processors from different blocks do not communicate we will isolate our discussion to a single arbitrary block. Let B denote the set of processes in this block; initially $|B| = b$.

We maintain the invariant that the block is represented as a linked list $L = L_d, L_{d-1}, \ldots, L_0$ where $L_d = S_d - D$, and in general, $L_j = S_j - S_{j+1} - D$. That is, $L = B - D$: no dead processes appear in this list and L_j lists those processes that do not appear higher up in the stack. We also maintain that for all j, L_j has been fairly allocated. What this means is that the ℓ^{th} processor ($0 \leq \ell < q$) assigned to this block "owns" a sublist $L_{\ell,j}$ of L_j extending from element $\ell\lceil\frac{|L_j|}{q}\rceil$ to element $(\ell+1)\lceil\frac{|L_j|}{q}\rceil - 1$ (if they exist). We assume processor ℓ has a pointer to $L_{\ell,j}$. (These pointers contribute the pd_{max} term to the space in Theorem 17. The other space requirements are linear in m.)

Suppose in round i, Step 1 is of type (a) — the stack is not altered. Then $R_i \cap L = L_d$ and we already have a fair allocation of L_d. Provided $|L_d|$ is about the same in this block as in any other, Step 2 can be simulated optimally, in $O(t_i \cdot \lceil |L_d|/q \rceil)$ time. This will be discussed later. To restore our invariants after Step 3 we simply need to splice out newly dead processes from L_d and compute a fair allocation for the new list. Let L_d and L'_d be the list before and after Step 3. Processor ℓ will find all $L'_{w,d}$ which lie in $L_{\ell,d}$, sending a pointer of $L'_{w,d}$ to processor w. For this task processor ℓ must know $|L'_d|$ and the number of elements from L'_d which lie before $L_{\ell,d}$, both of which can be computed in $O(|L_d|/q + \log q)$ time with a prefix-sums computation. Finally we compute L'_d by splicing out all dead elements, also in $O(|L_d|/q + \log q)$ time.[6] The other two cases for Step 1, (b) and (c), involve either splitting L_d into two lists or combining L_d and L_{d-1} into one list, followed by a step to compute a fair allocation for the new list(s). We omit a discussion of these two cases; the techniques used are the same as in Step 3.

In implementing Step 2 we use the assumption that there is a known upper bound $r_i \geq |R_i|$ on the number of processes taking part in the i^{th} round. (In our MST algorithm, for instance, this upper bound would hold with high probability – see Section 10.7.) We argue that with a certain probability (that depends on q), for every round i, every processor is given no more than $(1+\epsilon)(1 + r_i/p)$ active processes. Each of the t_i time steps in Step 2 is then easily simulated in $(1+\epsilon)(1+r_i/p)$ time. Consider the $m/b \times b$ array used in the initial allocation, and an arbitrary block and round. Let X_k be 1 if the process initially placed in the k^{th} column is active in the round, and 0 otherwise. Because the rotations on different columns were *independent*, so too are the X_k's. Let $X = \sum_{k=1}^{q} X_k$ be the number of active processes appearing in the block; clearly $E(X) = |R_i|b/m \leq r_i b/m$. Since each processor can be thought to have a "dummy" process associated with it which is active in every round, assume, without loss of generality, that $E(X) \geq q$. Noting that X is the sum of independent Bernoulli trials, we can bound the probability that X deviates too far from its expectation using a Chernoff bound [10]. For $0 < \epsilon < 1$, $\Pr[X > (1+\epsilon)E(X)] < e^{-\Omega(\epsilon^2 E(X))}$, and for constant ϵ, the probability that *any* block in any round gets more than $1+\epsilon$ times its expectation is $< \frac{mr}{b} e^{-\Omega(q)} = e^{-\Omega(q)}$ since $q = \Omega(\log(mr))$. The analysis of our scheme is very similar to that of [99] but considerably more efficient in terms of time. In [99] ϵ is increased in order to reduce the probability of failure. In our scheme we would set ϵ to

[6]The prefix-sums and splicing can, of course, be performed in one pass.

be a small constant and increase q (number of processors per block) as necessary. It is crucial to keep ϵ small because in either scheme nearly all processors spend an $\epsilon/(1+\epsilon)$ fraction of their time doing nothing! On the other hand, the q parameter can usually be increased dramatically with negligible effects on the overall running time. Hence our scheme achieves a low failure probability without excessive processor idling.

Remark. The space claimed in Theorem 17 can be easily reduced to $O(m)$ at the expense of simplicity. The idea is to compute fair allocations only when necessary. Very frequently, a previously computed fair allocation is "fair enough". For instance, in Step 1(b) L_d is split into two lists, L'_d and L'_{d+1}. If L'_{d+1} contains more than half of the elements from L_d, we might as well use the fair allocation of L_d instead of computing new ones for L'_{d+1} and L'_d.

10.9 Adaptations to Practical Parallel Models

Our results imply good MST algorithms for the QSM [85] and BSP [202] models, which are more realistic models of parallel computation than the PRAM models. Theorem 18 given below follows directly from results mapping EREW and QRQW computations on to QSM given in [85]. Theorem 19 follows from the QSM to BSP emulation given in [85] in conjunction with the observation that the slowdown in that emulation due to hashing does not occur for our algorithm since the assignment of edges to processors made by our allocation scheme achieves the same effect.

Theorem 18 *Using a p-processor QSM machine, the MST of an edge-weighted graph on n nodes and m edges can be found in $O(g \log n + m/p)$ time with probability $1-n^{-\omega(1)}$, where g is the gap parameter of the QSM.*

Theorem 19 *Using a p-processor BSP machine, the MST of an edge-weighted graph on n nodes and m edges can be found in $O((L+g)\log n + m/p)$ time with probability $1-n^{-\omega(1)}$, where g and L are the gap and periodicity parameters of the BSP.*

10.10 Discussion

Our algorithm settles the *randomized* time-work complexity of the MST problem. We note that the deterministic parallel work-complexity of both MST and connectivity are open. The best implementation of Borůvka's algorithm [42, 39] uses $O(m \log\log\log n)$ work, and the best connectivity algorithm [43] uses $O(m\alpha(m,n))$ work. It seems plausible that any method based solely on hook-and-contract/Borůvka steps can never do better than $\Theta(m\alpha(m,n))$: such algorithms seem to be forced into implementing a kind of online Union-Find algorithm.

We doubt that our optimal sequential MST algorithm [Chapter 8] can be parallelized, and doubt that any of the randomized parallel algorithms can be fully derandomized. However, in Chapter 11 we give parallel expected linear-work algorithms for MST and connectivity that use just a polylogarithmic number of random bits. This is in contrast to the linear number of random bits used by previous parallel algorithms.

Chapter 11

A Reduced Randomness MST Algorithm

Randomized algorithms are frequently simpler and asymptotically faster than their deterministic counterparts. Take, for example, algorithms for primality testing [163, 174, 2], finding medians [19, 64], and minimum spanning trees [127, 28] [Chapter 8]. For the sake of simplicity, most randomized algorithms assume a great abundance of independent perfectly random bits, a commodity that, in reality, is scarce or non-existent. The problem of *derandomization* — reducing or eliminating the dependence on randomness — is well studied; see [154] for a survey.[1]

Derandomization is both a big-picture, theoretical problem (e.g., does $P = BPP$?) as well as a practical one. A number of results have shown that the so-called pseudorandom number generators used in many computers can cause certain randomized algorithms to perform poorly. Karloff and Raghavan [129] showed that quicksort can perform worse than expected, and Bach [13] showed that the success probability of some number-theoretic algorithms is less than expected. In [62, 111, 110] some peculiarities of popular pseudorandom generators were noted, in the context of Monte Carlo physics simulations [62] and parallel computation [111, 110]. Although debunking pseudorandom generators is a worthy endeavor, we would prefer to design algorithms that use little or no random bits at all.

In this chapter we present a parallelizable, expected linear-time MST algorithm that uses only $O(\log^2 n \log^* n)$ random bits. Our algorithm also solves the simpler parallel connectivity problem, and provides an alternative to Karger et al.'s [127] sequential, randomized expected linear-time algorithm. The previous best algorithms for parallel connectivity [83, 100] and MST [Chapter 10] [127, 40, 41, 172] used a *linear* number

[1]This chapter's results appeared in: S. Pettie and V. Ramachandran, Minimizing randomness in minimum spanning tree, parallel connectivity, and set maxima algorithms, Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 713–722, 2002.

of perfect random bits; thus our algorithm represents an exponential reduction in the usage of random bits. (Note that the low-randomness MST algorithm from Section 8.6 is not parallelizable.)

Our techniques may be of more interest in isolation. We introduce a novel sampling strategy (different from Karger et al.'s [127]) and give a surprisingly simple scheme for random sampling and processor allocation on the EREW PRAM. (Random sampling typically complicates the problem of processor allocation.)

Organization

In Section 11.1 we prove bounds on the behavior of k-wise independent samplers, when sampling from a total order. In Section 11.2 we give a new, expected linear-work MST algorithm for the EREW PRAM (see Chapter 10 for a discussion of the PRAM model.) In Section 11.3 we discuss the relationship between our MST algorithm and the existing non-greedy MST algorithms, including those of Karger et al. [127], Chazelle [28], and the one presented in Chapter 8. We also discuss some other randomized algorithms whose dependence on randomness can be reduced exponentially without affecting asymptotic performance.

11.1 Limited Independence Sampling

In this section we establish a fairly general result on k-wise independent sampling which suggests that $O(1)$-wise independence is nearly as good in situations common to many randomized sorting-type algorithms. The situation is this: we have a set of elements from a total order χ and wish to find an element on the cheap whose rank is close to some desired rank t. Assuming an abundance of randomness we simply select each element of χ independently with probability p and take the rank pt sampled element as a decent approximation of the actual rank t element. A tradeoff between the efficiency and accuracy of this scheme can be had by manipulating the sampling probability p.

We show that by using just pairwise independence the expected rank (w.r.t. χ) of the rank $\lfloor pt \rfloor$ sampled element is $O(t \log n)$, and using $2k$-wise independence, $k > 1$, its expectation is $O(t)$. There is then a natural tradeoff between k and the concentration of the distribution around its mean.

The following Lemma is just an extension of Chebyshev's inequality for 0/1 random variables. A more complex proof of this result appears in [179]; our proof is elementary.

Definition 8 *Random variables $X_1, \ldots, X_n$ are k-wise independent if for any distinct indices $i_1, \ldots, i_k$ and values $x_1, \ldots, x_k$:*

$$\Pr[X_{i_1} = x_1 \wedge \cdots \wedge X_{i_k} = x_k] \quad = \quad \Pr[X_{i_1} = x_1] \times \cdots \times \Pr[X_{i_k} = x_k]$$

Lemma 68 *Let $X_1, \ldots, X_n$ be $2k$-wise independent $0/1$ random variables, each with mean μ, and let $\mu_X = \mu n$ be the mean of $X = \sum_i X_i$.*

$$\Pr\left[|X - \mu_X| \geq t\right] < \left(\frac{4k\mu_X}{t^2}\right)^k$$

Proof: Along the lines of Chebyshev's inequality we have $\Pr[|X - \mu_X| \geq t] = \Pr[(X - \mu_X)^{2k} \geq t^{2k}]$ which is $\leq \frac{\mathrm{E}(X-\mu_X)^{2k}}{t^{2k}}$ by Markov's inequality. The numerator can be expanded into an expression of the form $\mathrm{E}\sum\prod(X_i - \mu)$ — the expectation of a sum of products. By identifying duplicate factors in each product we can simplify them to be of the form $\prod_i (X_i - \mu)^{a_i}$ where $a_i \geq 0$ and $\sum_i a_i = 2k$. Notice that because the X_i's are $2k$-wise independent the factors of each product are also independent. We may then rewrite the numerator in the form $\sum\prod \mathrm{E}(X_i - \mu)^{a_i}$ — a sum of products of expectations. Observe that in any term, if some $a_i = 1$ then $\mathrm{E}(X_i - \mu)^{a_i} = 0$ and the term disappears. We bound the numerator by first bounding a single term and then bounding the number of non-zero terms. For the first, note that $\mathrm{E}(X_i - \mu)^{a_i} = \mu(1-\mu)[(1-\mu)^{a_i - 1} - (-\mu)^{a_i - 1}] \in (-\mu, \mu)$, hence each term is bounded by μ^k.

Bounding the number of non-zero terms is equivalent to a balls-and-bins problem: how many ways are there to put $2k$ balls in n bins (order counts!) such that all bins have zero or ≥ 2 balls? Let N be the number of non-zero terms, we have that:

$$N \leq \sum_{i=1}^{k} \binom{n}{i} \binom{2k-i-1}{i-1} \frac{(2k)!}{2^k}$$

Here i represents the number of non-empty bins and $(2k)!/2^k$ is an upper bound on the number of distinct realizations for each balls-in-bins pattern.

$$\begin{aligned}
N &\leq \sum_{i=1}^{k} \binom{n}{i} \binom{2k-i-1}{i-1} \frac{(2k)!}{2^k} \\
&\leq \sum_{i} \frac{n^i}{i!} \frac{(2k)^i}{i!} \frac{(2k)!}{2^k} \\
&\leq \tfrac{4}{3} \frac{n^k}{k!} \cdot \frac{(2k)^k}{k!} \cdot \frac{(2k)!}{2^k} \qquad \{(i+1)\text{th term } \geq 4 \cdot i\text{th term}\} \\
&\leq \tfrac{4}{3} \frac{1 + \frac{1}{24k-1}}{\sqrt{k\pi}} \cdot \left(\frac{ne}{k}\right)^k \cdot \left(\frac{2ke}{k}\right)^k \cdot \left(\frac{2k^2}{e^2}\right)^k \qquad \{\text{Stirling's approx.}\} \\
&< (4kn)^k
\end{aligned}$$

We conclude that:

$$\Pr[|X - \mu_X| \geq t] \leq \frac{\mathrm{E}(X - \mu_x)^{2k}}{t^{2k}} \leq \frac{\mu^k (4kn)^k}{t^{2k}} = \frac{(4k\mu_x)^k}{t^{2k}}$$

□

The main lemma of this section is given below.

Lemma 69 *Let χ be a set of totally ordered elements and χ_p be a subset of χ derived by sampling each element with probability p using a $2k$-wise independent sampler. Let Z be the number of unsampled elements less than $\min \chi_p$. Then*

$$\mathrm{E}(Z) \le \begin{cases} 4p^{-1}\ln n + 1 & \text{for } k = 1 \\ \\ 16p^{-1} & \text{for } k > 1 \end{cases}$$

and

$$\Pr[Z \ge \ell] \le \min_{\nu \le k} \left\{ \left(\frac{4\nu}{p\ell}\right)^{\nu} \right\}$$

Proof: Let $X_i = 1$ if the element of χ with rank i is sampled, and 0 otherwise. So $\mathrm{E}[X_i] = p$ and for any distinct indices $i_1, \ldots, i_{2k}$, $X_{i_1}, \ldots, X_{i_{2k}}$ are independent. Let $S_\ell = \sum_{i=1}^{\ell} X_i$ count the number of ones in $X_1, \ldots, X_\ell$. We have that $\mathrm{E}[S_\ell] = p\ell$ and

$$\Pr[Z \ge \ell] = \Pr[S_\ell = 0] \le \Pr[|S_\ell - \mathrm{E}(S_\ell)| \ge p\ell]$$

Using Lemma 68 we can bound $\Pr[Z \ge \ell]$ as follows.

$$\begin{aligned} \Pr[Z \ge \ell] &\le \Pr[|S_\ell - \mathrm{E}(S_\ell)| \ge p\ell] \\ &\le \left(\frac{4p\ell k}{(p\ell)^2}\right)^k \qquad \{\text{ Lemma 68}\} \\ &= \left(\frac{4k}{p\ell}\right)^k \end{aligned}$$

The second part of the Lemma follows from the simple observation that any $2k$-wise independent distribution is also 2ν-wise independent for $\nu \le k$.

We now bound $\mathrm{E}[Z]$:

$$\begin{aligned} \mathrm{E}[Z] &= \sum_{i=1}^{\infty} \Pr[Z \ge i] \\ &\le \Delta + \sum_{i=\Delta+1}^{n} \Pr[Z \ge i] \qquad \{\text{for any integer } \Delta \ge 0\} \\ &\le \Delta + (4kp^{-1})^k \sum_{i=\Delta+1}^{n} i^{-k} \end{aligned}$$

For $k = 1$ (pairwise independence) and $\Delta = 0$ we have $\mathrm{E}[Z] \leq 4p^{-1} \ln n + O(1)$ and for $k = 2$, we have $\mathrm{E}[Z] \leq \Delta + (8p^{-1})^2/\Delta$. Setting $\Delta = 8p^{-1}$ gives us $\mathrm{E}[Z] \leq 16p^{-1}$.
□

The proof of Lemma 70 follows the same lines as Lemma 69.

Lemma 70 *Let χ be a set of totally ordered elements and χ_p be a subset of χ derived by sampling each element with probability p using a $2k$-wise independent sampler. Let x_t be the element of χ_s with rank t and let Z_t be the number of elements in χ less than x_t. Then*

$$\mathrm{E}(Z_t) = \begin{cases} O(tp^{-1} \log n) & \textit{for } k = 1 \\ \\ O(tp^{-1}) & \textit{for } k > 1 \end{cases}$$

11.1.1 Pairwise Independent Sampling on the EREW PRAM

In Section 11.2 we need a method for sampling a set that takes time linear *in the size of the sample.* Furthermore, we would like to assign the sampled elements to EREW PRAM processors without burdening one processor more than the others. We solve both of these problems using Joffe's [117] method for generating k-wise independent variables, given below.

Lemma 71 *(Joffe [117]) Let q be prime, $a_0, a_1, \ldots, a_{k-1}$ be chosen uniformly at random from $\mathbb{Z}_q$, and $X(i) = \sum_{j=0}^{k-1} a_j \cdot i^j \pmod{q}$. Then $X(0), \ldots, X(q-1)$ are uniformly distributed over $\mathbb{Z}_q$ and k-wise independent.*

That is, for generating pairwise independent variables we require only two random coefficients, a_0 and a_1. We sample a set of size q as follows. Let p be the desired sampling probability. If $X(i) = a_1 i + a_0 \pmod{q} \in [0, \lceil pq \rceil - 1]$ then element i is sampled; otherwise it is not. Evaluating the polynomial X on q points is too expensive because the number of sampled elements could be sublinear in q. Under the assumption that $a_1 \neq 0$ we can generate the sampled set by generating all solutions to $i = (j - a_0)a_1^{-1} \pmod{q}$ for $j \in [0, \lceil pq \rceil - 1]$. This leads to a simple scheme for assigning sampled elements to processors — refer to Figure 11.1.

Lemma 72 *Suppose a parallel algorithm requires s pairwise independent samples from a set of size q (a prime), with perhaps different sampling probability for each sample. After $O(s + \log P + \log q)$-time preprocessing on a P processor EREW PRAM, the samples can be generated in optimal time and work.*

Proof: Our EREW sampling algorithm is given in Figure 11.1. For it to work, every processor must know two random elements $a_0, a_1 \in Z_q$ and a_1^{-1} (if $a_1 \neq 0$) We assign

s processors to generate one (a_0, a_1) pair each and compute a_1^{-1}. This takes $O(\log q)$ time. In $O(s + \log P)$ time we distribute the (a_0, a_1, a_1^{-1}) tuples to all processors.
□

EREW-Pairwise-Sampler:

Specs: There are P EREW processors and q elements. Each element is to be sampled with probability p (pairwise independently) and the sampled elements distributed fairly among the processors.

We assume a_0, a_1 have been selected uniformly at random from $\mathbb{Z}_q$, and that each processor knows p, q, a_0, a_1, a_1^{-1} and its processor ID.

Case 1. If $a_1 = 0$ and $a_0 \geq \lceil pq \rceil$ then $X(i) = a_0$. No elements are sampled.

Case 2. If $a_1 = 0$ and $a_0 < \lceil pq \rceil$ then all elements are sampled. Processor k is assigned elements $\lceil \frac{q}{P} \rceil k$ through $\lceil \frac{q}{P} \rceil (k+1) - 1$.

Case 3. If $a_1 \neq 0$, then processor k is assigned elements of the form

$$(j - a_0)a_1^{-1} \pmod q$$

for all j from $\lceil \frac{pq}{P} \rceil k$ through $\lceil \frac{pq}{P} \rceil (k+1) - 1$.

Figure 11.1: A combination pairwise-independent sampler and allocation scheme.

In situations where pairwise independent sampling suffices, our processor allocation scheme is significantly more desirable than the alternatives (for instance, that of Section 10.8). It is quick, uses minimal communication, and distributes the sampled set perfectly: every processor's load is less than the average load plus 1.

One problem that remains is finding a prime q in parallel for use with Joffe's construction. This is not so hard: we simply run the randomized Miller-Rabin [163, 174] primality test on a sequence of integers known to contain a prime. Baker and Harman [14, p. 225] showed that if p_n is the nth prime, then $p_n - p_{n-1} \leq n^{.535+o(1)}$.

Lemma 73 *Let q be the smallest prime such that $q \geq m$. Then with probability at least $1 - m^{-2c+1}$, q can be found on the EREW PRAM in $O(\log m)$ time, using $cm^{.535+o(1)} \log m$ processors and $c \log^2 m$ random bits.*

Proof: We run the Miller-Rabin [163, 174] primality test $c \log m$ times on each integer in the interval $[m, m + b(m)]$, where $b(m) = m^{.535+o(1)}$, reusing the same random bits for each number tested. The probability that Miller-Rabin reports the wrong answer for any of the numbers is $\leq b(m)\left(\frac{1}{4}\right)^{c\log m} \leq m^{-2c+1}$. Each test uses $\log m$ random bits and takes time $O(\log m)$, hence finding the first prime $\geq m$ takes $c\log^2 m$ random bits and $O(\log m)$ time using $b(m)\cdot c\log m$ processors.

$\square$

Remark. One can reduce the number of random bits assumed in Lemma 73 to $O(\log m)$ using the random walk technique — see [6, 113].

11.2 A Low-Randomness MST Algorithm

Our algorithm is based on an *approximate* version of the standard Borůvka step. Rather than having each vertex select its least-weight incident edge (a normal Borůvka step), we first choose a random subset of the edges, denoted H. Each vertex then selects its least-weight incident *in H*. Suppose that vertex u selected edge (u,v) in such a step. If $w(u,x) < w(u,v)$ then the edge (u,x) *bears witness* to the fact that u chose the wrong edge; we call (u,x) a *tainted edge* and set it aside for later consideration. (For technical reasons, we always designate (u,v) tainted.) If we continue to perform approximate (or exact) Borůvka steps, eventually every edge will become tainted or a self-loop (due to edge contractions). The usefulness of approximate Borůvka steps follows from a few claims. We claim that any untainted self-loops are not in the MST, and that all such edges can be found in linear time with an MST verification algorithm. Second, if the random subgraphs are chosen properly then all the approximate Borůvka steps take linear time in total. Moreover, the tainted edges form a small minority. Therefore, each phase of the algorithm (consisting of exact Borůvka steps, then approximate steps, then MST verification) takes linear time and reduces the number of edges by a constant factor.

Using a linear number of random bits, our algorithm can be shown to run in linear time with all but exponentially small probability — the same as the Karger et al. [127] algorithm. However, since the focus of this chapter is reducing dependence on randomness, we analyze our algorithm under the assumption that all sampled graphs are generated using a pairwise independent sampler. We show our algorithm requires expected linear-work and can be made to run in polylogarithmic time. In Section 11.2.1 we introduce a number of basic techniques; many of them are specialized towards a parallel implementation of our algorithm and can be simplified, slightly, in a sequential model of computation.

11.2.1 Techniques

We shall assume, without any loss of generality, that the graph has maximum degree 3. (Any vertices with higher degree can be separated into a chain of degree-3 vertices.)

We make this assumption for the sake of a simpler exposition.

Relocation Lemma of MSTs

MST algorithms are generally proved correct by appealing to the cut and cycle properties (take for example, Lemmas 33 and 36). Those properties, in turn, can be shown to follow from a general property we call the *endpoint relocation property.*

Lemma 74 *(Endpoint relocation property) Suppose that $w(u,v) > w(v,w)$, where $(u,v),(v,w)$ are edges in a weighted graph G. Let G' be derived from G by reassigning the endpoints of (u,v) to (u,w); call this edge e, whether it appears in G or G'. Then $e \in MST(G)$ if and only if $e \in MST(G')$.*

Proof: Suppose that e is the heaviest edge on a cycle Q in G, that is, $e \notin MST(G)$. Then in G', e is the heaviest edge on the unique cycle in $Q \cup \{(v,w)\}$, since $w(e) > w(v,w)$. Note that the edge (v,w) may or may not be in the cycle in $Q \cup \{(v,w)\}$. The reverse direction is proved in an identical manner, switching the roles of G and G'.
□

Pairing Up

In this Chapter we force all of our Borůvka steps to contract vertices in pairs — *binary Borůvka steps* for short. As usual, each vertex identifies its least-weight incident edge, forming a forest F of rooted trees. We then use the procedure Pair-Up(F) to identify a subset $F' \subseteq F$ of independent edges. To obtain a sizable set F', Pair-Up may relocate edges of F consistent with Lemma 74 – see Figure 11.2 for the details of the procedure.

Lemma 75 *Pair-Up(F) returns a set of independent edges F', such that* $\mathrm{E}[|F'|] \geq |F|/4$*, even if the "coin flips" are only pairwise independent. Moreover, all edge relocations made by Pair-Up are minimum spanning tree preserving (see Lemma 74).*

Proof: By Lemma 74 all edge relocations do not affect which edges are in the MST. Each relocated edge in F' removes from consideration at most two other edges from F. Because the coin flips are pairwise independent, all remaining edges in F are included in F' with probability 1/4. By linearity of expectations, the $\mathrm{E}[|F'|] \geq |F|/4$.
□

Thus, if F is the set of edges chosen in a normal Borůvka step, Pair-Up(F) returns at least $n/8$ (expected) independent edges. This can be increased at the price of more complication. Pair-Up can clearly be implemented in linear time sequentially. We address a parallel implementation in Lemma 76.

```
Pair-Up($F$)
    $F' := \emptyset$
    For each vertex $v$:
        Let $v_1, v_2, \ldots, v_k$ be the children of $v$ in $F$
        For $i := 1..\lfloor \frac{k}{2} \rfloor$
            Assume, w.l.o.g., that $w(v_{2i}, v) > w(v_{2i-1}, v)$
            Relocate $(v_{2i}, v)$ to $(v_{2i-1}, v_{2i})$
            $F' := F' \cup \{(v_{2i-1}, v_{2i})\}$
    ($F - F'$ now consists of a collection of paths)
    Each vertex not incident to $F'$ flips a fair coin
    For each edge $(u, v) \in F - F'$
        If $u$ has heads and $v$ has tails,
            $F' := F' \cup \{(u, v)\}$
    Return($F'$)
```

Figure 11.2: Pair-Up(F) returns a set of independent edges from F, after relocating some endpoints.

Borůvka Steps, Borůvka Trees

A *Borůvka tree* is a rooted tree modeling the graph contractions encountered in Borůvka's algorithm. The leaf-nodes correspond to graph vertices and the nodes at height i correspond to the vertices after i Borůvka steps. This structure was introduced by King [133] as part of her MST verification algorithm, and is essentially the same as the *filter forest* defined in Chapter 10.

Suppose that our current graph was derived from the original by performing b binary Borůvka steps; therefore, we assume there is a partially constructed Borůvka tree with height b. We execute the next Borůvka step in $O(b+1)$ time, as follows. Assign one (virtual) processor to each edge. The processor for edge (u, v) begins at the leaves corresponding to u and v in the Borůvka tree and crawls to their respective roots, in $O(b+1)$ time. If $\text{root}(u) = \text{root}(v)$ then (u, v) is a self-loop and should not participate in any more Borůvka steps. Otherwise, the processor begins again at leaves u, v, moving towards the root. If it meets another processor at an internal node, only the lighter edge proceeds upward. Thus, exactly one edge remains at each root: this is the lightest edge incident to some vertex in the current contracted graph. We then use the Pair-Up procedure to find a set of independent edges. Contracting them adds a layer of new nodes to the Borůvka tree, each having one or two children from the previous layer.

Lemma 76 *The* b^{th} *Borůvka step can be implemented to run in* $O(b \cdot \lceil \frac{m}{P} \rceil)$ *time with*

P EREW processors, where m is the number of edges participating in the Borůvka step. (Perfect processor allocation is assumed.)

Proof: Let F consist of the lightest edge incident on each vertex. Finding F in the stated time bounds is straightforward on the EREW PRAM. We show how to implement Pair-Up(F) in $O(b)$ time, assuming one processor per edge. If (u, v) is the edge selected by u, i.e. v is the parent of u in F, then the processor associated with (u, v) begins at the leaf associated with v in the Borůvka tree and crawls upward. If it meets another processor, say representing (x, v), where $w(u, v) > w(x, v)$, then *both* processors stop climbing, and the edge (u, v) is relocated to (u, x); both u and x are *matched.* Any edge (i.e. processor) that reaches its root without encountering another edge checks to see if either of its endpoints were matched, and if so, stops. All edges whose processors havn't yet stopped form a set of disjoint paths. We then use the coin flipping technique to select an independent subset of the edges. All edges in the independent set are contracted, adding another layer to the Borůvka tree.

□

Remark. Since the graph has degree-3, at most three (virtual) processors simultaneously attempt to start climbing from a leaf. In the EREW PRAM model it is very simple to resolve such reading conflicts, provided the number of competing processors is constant. This is the reason behind contracting vertices in pairs (to bound the degree of the Borůvka tree) and restricting the graph to have constant degree.

Lemma 77 *After b binary Borůvka steps the expected number of vertices is at most $n(\frac{7}{8})^b$.*

Proof: Let X_i be the number of vertices after the ith Borůvka step. By Lemma 75 $\mathrm{E}[X_b] \leq 7X_{b-1}/8$, hence $\mathrm{E}[X_b] \leq X_0(\frac{7}{8})^b = n(\frac{7}{8})^b$.

□

Approximate Borůvka Steps

An approximate Borůvka step is just a normal Borůvka step w.r.t. a sampled graph H. Each vertex identifies the least-weight *eligible* edge incident to it, inducing a forest, where an edge is eligible if it is in H, it is not a self-loop, and it was not *tainted* in a previous approximate Borůvka step. An edge (u, v) (not necessarily in H) becomes tainted if its weight is at most that of the edges chosen by either u or v. (Notice that by this definition, all edges selected in an approximate Borůvka step are tainted.) The procedure Approx-Borůvka-Step(H) performs a Borůvka step w.r.t. H (inducing a forest $F \subseteq H$), finds an independent subset $F' \subseteq F$ using the Pair-Up procedure, then contracts F', adding one layer to the Borůvka tree. It is implemented just like a normal Borůvka step, except that already tainted edges in H need to be identified and discarded. We can identify tainted edges as follows. Label each edge $(x, p(x))$ in the Borůvka tree by the weight of the edge selected by x in the appropriate (aproximate) Borůvka step. Consider an edge (u, v). After it is certified that u and v are distinct in the current graph (that is, have distinct roots in the Borůvka tree), we check whether (u, v) is strictly heavier than every edge on the path from u to root(u) and from v to root(v). If it is, (u, v) is untainted and eligible.

Lemma 78 *Let H be a subset of the edges in a graph obtained from $b-1$ (approximate) Borůvka steps. Then the call to Approx-Borůvka-Step(H) takes time $O(b \cdot \lceil \frac{|H|}{P} \rceil)$ using P EREW processors. (Perfect processor allocation is assumed.)*

Lemma 79 *Consider a sequence of (approximate) Borůvka steps that causes all edges to become tainted and/or self-loops (due to edge contractions). Then any untainted edge is not in the minimum spanning tree.*

Proof: Let F be the forest of edges selected and contracted by the approximate Borůvka steps, and let U be the set of untainted edges. We claim that F is the minimum spanning forest of $U \cup F$. To see this, just consider a re-execution of the (approximate) Borůvka steps on $U \cup F$. By definition, every edge contracted is the lightest edge incident to some vertex (all other edges being untainted) and by the cut property must belong to the MST of $U \cup F$.

$\square$

11.2.2 The Algorithm

Our Low-Randomness MST algorithm is given in Figure 11.3. It uses the constants $\gamma, \bar{\gamma}, p_j, \lambda_i$ to determine how many exact Borůvka steps to perform, and the sampling probability for the approximate Borůvka steps.

$$\gamma = \tfrac{7}{8}, \quad \bar{\gamma} = \tfrac{8}{7}, \quad p_j = \gamma^{j/2}$$

$$\lambda_0 = 0 \quad \lambda_1 = C \quad \lambda_{j+1} = \bar{\gamma}^{\lambda_j/3}$$

Here C represents a constant large enough to make the sequence $(\lambda_j)_{j\geq 0}$ increasing.[2] The constant $\gamma = 7/8$ reflects the expected number of vertices contracted in an (approx) Borůvka step. (Many of the constants in our algorithm could be improved if γ were set closer to $1/2$; this would require a more sophisticated pairing-up procedure than Pair-Up.)

The correctness of Low-Randomness-MST is immediate. Any edges discarded (in Line 3) are by Lemma 79 not in the MST. Therefore any edges contracted in Line 1

[2]For instance, if $C = 3\log_{\bar{\gamma}} C$ then there is no growth. Setting $C = 120$ suffices.

```
**Low-Randomness-MST**$(G)$
    $G_0 := G$
    Execute Phases $1, 2, \ldots$ until the graph is trivial
    The edges contracted in Line 1 (below) form $MST(G)$.

Phase $i$:

  1.   Perform $\lceil \lambda_i - \lambda_{i-1} \rceil$ exact Borůvka steps on $G_{i-1}$
  2.   For $j = \lambda_i + 1,\ \lambda_i + 2, \ldots$
  2.1      Let $H_{ij}$ be derived by sampling edges of $G_{i-1}$ with prob. $p_j$
  2.2      Approx-Borůvka-Step$(H_{ij})$
  2.3      Break out of for-loop if all edges are tainted or self-loops
  3.   Identify all untainted edges; $G_i := G_{i-1} - \{$untainted edges$\}$
  4.   Retract the approximate Borůvka steps made in Step 2.
       All edges tainted in those steps become untainted again.
```

Figure 11.3: An expected linear-time minimum spanning tree algorithm.

are definitely in the MST. We implement Line 3 (finding tainted edges) with the MST verification algorithm of [134] for the EREW PRAM. It runs in logarithmic time and linear work.

In Lemma 80 we establish various bounds on the expected size of the graphs encountered in each phase of Low-Randomness-MST.

Lemma 80 *Assume that a pairwise independent sampler is used to generate the graphs H_{ij} in Line 2.1, and to perform the random pairing implicit in Lines 1 and 2.2. Let n_i and m_i be the expected number of vertices and edges, respectively, in G_i. Then $n_i \leq n\gamma^{\lambda_i}$ and $m_i \leq n_i \cdot (C_1\lambda_{i-1} + 2)$*

Proof: G_i is derived by applying $\sum_{i'=1}^{i}(\lambda_{i'} - \lambda_{i'-1}) = \lambda_i$ Borůvka steps. Hence, by Lemma 77, $n_i \leq n\gamma^{\lambda_i}$. Notice that m_i is precisely the number of edges tainted in the ith Phase. Before bounding m_i, consider the number of edges tainted in some approximate Borůvka step, with sampling probability p, on a graph with m' edges and n' vertices. Let d_k be the degree of the kth vertex. By Lemma 69 and the concavity of the log function, the expected number of tainted edges is bounded by

$$\sum_{k=1}^{n'} c_1\, p^{-1} \log d_k \quad \leq \quad c_1 n' p^{-1} \log \frac{2m'}{n'}$$

We will bound m_i with an inductive argument using the above inequality. Since we assumed the graph to have degree at most 3, $m_0 \leq 3n_0/2 \leq n_0(C_1\lambda_0 + 2)$. Assume inductively that $m_{i'} \leq n_{i'} \cdot (C_1\lambda_{i'} + 2)$ for $i' < i$. (The $+2$ is needed because $\lambda_0 = 0$.) We bound m_i as follows:

$$\begin{aligned}
m_i &\leq \sum_{j=\lambda_i+1}^{\infty} c_1 n\gamma^{j-1} p_j^{-1} \log\left(\frac{2m_{i-1}}{n\gamma^{j-1}}\right) \\
&\leq c_1 n_i \sum_{j=1}^{\infty} \gamma^{j/2-1} \log\left(\frac{2n_{i-1}[C_1\lambda_{i-1}+2]}{n_i\gamma^{j-1}}\right) \\
&\leq c_1 n_i \sum_{j=1}^{\infty} \gamma^{j/2-1} \log\left(\bar{\gamma}^{\lambda_i+j-\lambda_{i-1}} \cdot [2C_1\lambda_{i-1}+4]\right) \\
&\leq c_1 n_i \sum_{j=1}^{\infty} \gamma^{j/2-1} c_2(\lambda_i + j)\log C_1 \\
&\leq c_1c_2c_3 n_i\lambda_i \log C_1 \;\;\leq\;\; C_1 n_i \lambda_i \qquad \{\text{for } C_1 \text{ large enough}\}
\end{aligned}$$

□

Theorem 20 *Let G be a graph on m edges and n vertices. On an EREW PRAM, Low-Randomness-MST(G) computes the minimum spanning tree of G with $O(\log^2 n \log^* n)$ random bits, in expected $O(\log^2 n \log^* n)$ time and linear work.*

Proof: Recall that if G is not initially of degree three, we force it to be by introducing $O(m)$ edges and vertices. Thus, we are seeking work bounds linear in the number of vertices.

Let T and W be the expected time and work, respectively. By Lemma 77 the expected number of (approx) Borůvka steps needed to cause all edges to become tainted or contracted is $\log_{\bar{\gamma}} n = O(\log n)$. The expected number of Phases is at most $\min\{j : n\gamma^{\lambda_j} < 1\} = O(\log^* n)$. Since each Borůvkastep takes $O(\log n)$ time, $T = O(\log^2 n \log^* n)$. We now bound the expected work W. By Lemmas 76, 78 and 80, we have:

$$\begin{aligned}
W &\leq c_4 \sum_{i=1}^{\infty}\left[m_{i-1}\lambda_i^2 + \sum_{j=\lambda_i+1}^{\infty} j m_{i-1} p_j\right] \\
&\leq c_4 \sum_{i=1}^{\infty}\left[n\gamma^{\lambda_{i-1}}(C_1\lambda_{i-1}+2)\lambda_i^2 + \sum_{j=\lambda_i+1}^{\infty} j n\gamma^{\lambda_{i-1}}[C_1\lambda_{i-1}+2]\gamma^{j/2}\right]
\end{aligned}$$

$$
\begin{aligned}
&\le \quad O(n) + c_5 \sum_{i=1}^{\infty} \left[n\gamma^{(\lambda_{i-1}/3)}\lambda_{i-1} + n[C_1\lambda_{i-1} + 2] \sum_{j=\lambda_i+1}^{\infty} j\gamma^{j/2+\lambda_{i-1}} \right] \\
&\le \quad O(n) + \sum_{i=2}^{\infty} O(n\lambda_{i-1}\gamma^{\lambda_{i-1}/2}) \quad = \quad O(n)
\end{aligned}
$$

The number of random bits we use to find prime numbers (for our pairwise independent sampler) is $O(\log^2 n)$ – see Lemma 73. Once they are found we use at most $4\log n$ random bits per Borůvka step: $2\log n$ to generate the sample (if it is approximate) and another $2\log n$ for the random pairing, for a total of $O(\log^2 n \log^* n)$ random bits.

□

Corollary 3 *The connected components of a graph can be found with an EREW PRAM using $O(\log^2 n \log^* n)$ random bits, in expected $O(\log^2 n \log^* n)$ time and linear work.*

11.3 Discussion

The recent MST algorithms [127, 28], [Chapter 8] are all said to be non-greedy, but they are not really alike. Karger et al.'s algorithm [127] uses random sampling and MST verification, and the deterministic MST algorithms [28], [Chapter 8] are based on Chazelle's Soft Heap [29]. The reduced-randomness MST algorithm we presented in this chapter can be seen as a happy medium between these two approaches. Like [127] it uses random sampling and MST verification. On the other hand, one can think of our approximate Borůvka steps as implementing a certain kind of Soft Heap, in particular, one where the tradeoff between *running time* and *error rate* can be changed at will. There is also an obvious similarity between our "tainted" edges and the "corrupted" edges of Chapter 8.

It is possible to obtain similar reduced-randomness algorithms for a number of other problems, either with our sampling techniques, or, in some cases, more direct methods. For instance, the *local sorting* and *set maxima* algorithms from [86] can be reorganized to use a polylogarithmic number of random bits [171]. One can also obtain reduced-randomness versions of Floyd and Rivest's selection algorithm [64], and Hoare's Quicksort [108]; see [171].

Appendix A

Split-Findmin and Its Applications

In this Chapter we propose some minor improvements to Gabow's split-findmin data structure [76], then show how split-findmin can be used to establish new bounds on the complexity of the minimum spanning tree (and shortest path tree) sensitivity analysis problems.[1]

A.1 Background

The *split-findmin* problem was introduced by Gabow in [76] for use in his weighted matching algorithm. His data structure handled m operations on a universe of size n in $O(m\alpha(m,n))$ time, which is nearly linear. Split-findmin data structures are used in all the hierarchy based shortest path algorithms to date [196, 98] [Chapters 3–5], though the versions assumed in [196, 98] are specialized to integer-weighted inputs, and the one in [98] is much slower than Gabow's structure [76], though supposedly simpler.

Split-findmin is a weighted version of the simpler split-find problem, which is itself a variation on union-find. The pointer-machine complexity of split-find and union-split-find were studied in [109, 158, 25, 147]. Burago [25] proved a lower bound of $\Omega(n \log \alpha(n))$ on the problem, and LaPoutré gave a tight lower bound of $\Omega(m\alpha(m,n))$. However, unlike the union-find problem [193, 71], split-find admits a relatively simple linear-time implementation on the RAM — see [79] for the method.

[1]Our results on the split-findmin data structure will appear in the journal version of [170]. The new sensitivity analysis algorithms have yet to be published.

A.2 An Optimal Split-Findmin Structure

The split-findmin data structure operates on a collection of disjoint sequences of elements. Initially, there is one sequence containing all n elements, and each element has key ∞. The following operations are supported.

split(u) Splits the sequence containing u into two sequences, one consisting of those elements up to and including u, the other sequence taking the rest.

findmin(u) Returns the element in u's sequence with minimum key.

decrease-key(u, κ) sets key(u) := min{key(u), κ}.

Theorem 21 *The split-findmin problem can be solved on a pointer machine in $O(n + m\alpha)$ time while making only $O(n + m \log \alpha)$ comparisons, where n is the number of elements, m the number of operations, and $\alpha = \alpha(m, n)$ is the inverse-Ackermann function. Alternatively, split-findmin can be solved on a RAM in time $\Theta(\text{SPLIT-FINDMIN}(m, n))$, where* SPLIT-FINDMIN *represents the decision-tree complexity of the problem, or, if random bits are available, the expected randomized decision-tree complexity of the problem.*

The remainder of this Section constitutes a proof of Theorem 21.

The $O(n + m \log \alpha)$ bound is simply an observation about Gabow's structure. His decrease-key(u, κ) routine just updates a series $x_0, x_1, \ldots, x_\alpha$ of real variables, setting $x_i = \min\{x_i, \kappa\}$. However, it is already guaranteed that $x_0 \geq x_1 \geq \cdots \geq x_\alpha$, so once the variables are found, updating them can be accomplished in $O(\alpha)$ time with $1 + \log \alpha$ comparisons.

To get a potentially faster algorithm on the RAM model we construct all possible split-findmin solvers on inputs with at most $q = \log\log n$ elements and choose one that is close to optimal for all problem sizes. We then show how to compose this optimal split-findmin solver on q elements with Gabow's structure to get an optimal one on n elements.

We consider only instances with $m' < q^2$ decrease-keys. If more decrease-keys are actually encountered we can revert to Gabow's algorithm [76] or a trivial one that runs in $O(m')$ time.

We represent the state of the solver with three components: a bit-vector with length $q - 1$ representing where the splits are, a directed graph H on no more than $q + m' < q(q + 1)$ vertices representing known inequalities between current keys *and* older keys retired by decrease-key operations, and finally, a mapping from elements to vertices in H. One may easily confirm that the state can be represented in no more than $3q^4 = o(\log n)$ bits. One may also confirm that a split or decrease-key can update the state in $O(1)$ time. We now turn to the findmin operation. Consider the *findmin-action* function, which determines the next step in the findmin procedure. It can be represented as:

$$\text{findmin-action} \; : \; \text{state} \times \{1, \ldots, q\} \; \rightarrow \; \left(V(H_X) \times V(H_X) \right) \cup \{1, \ldots, q\}$$

where the first $\{1, \ldots, q\}$ represents the argument to the findmin query. The findmin-action function can either perform a comparison (represented by $V(H_X) \times V(H_X)$) which, if performed, will alter the state, or return an answer to the findmin query, represented by the second $\{1, \ldots, q\}$. One simply applies the findmin-action function until it produces an answer. We will represent the findmin-action function as a table. Since the state is represented in $o(\log n)$ bits we can keep it in one machine word; therefore, computing the findmin-action function (and updating the state) takes constant time on a RAM.

One can see that any split-findmin solver can be converted into another with equal amortized complexity but which performs comparisons only during calls to findmin. Therefore, finding the optimal findmin-action function is tantamount to finding the optimal split-findmin solver.

We have now reduced the split-findmin problem to a brute force search over the findmin-action function. There are less than $F = 2^{3q^4} \cdot q \cdot (q^4 + q) < 2^{4q^4}$ distinct findmin-action functions, most of which do not produce correct answers. There are less than $I = (2q + q^2(q+1))^{q^2+3q}$ distinct instances of the problem, because the number of decrease-keys is $< q^2$, findmins $< 2q$ and splits $< q$. Furthermore, each operation can be a split or findmin, giving the $2q$ term, or a decrease-key, which requires us to chose an element and where to fit its new key into the permutation, giving the $q^2(q+1)$ term. Each findmin-action/problem instance pair can be tested for correctness in $V = O(q^2)$ time, therefore all correct findmin-action functions can be chosen in time $F \cdot I \cdot V = 2^{\Omega(q^4)}$. For $q = \log \log n$ this is $o(n)$, meaning the time for this brute force search does not affect the other constant factors involved.

How do we choose the optimal split-findmin solver? This is actually not a trivial question because of the possibility of there not being one solver which dominates all others on all input sizes. Consider charting the worst-case complexity of a solver $\mathcal{S}$ as a function $g_{\mathcal{S}}$ of the number of operations p in the input sequence. It is plausible that certain solvers are optimal for only certain densities p/q. We need to show that for some solver $\mathcal{S}^*$, $g_{\mathcal{S}^*}$ is within a constant factor of the lower envelope of $\{g_{\mathcal{S}}\}_{\mathcal{S}}$, where $\mathcal{S}$ ranges over all correct solvers. Let $\mathcal{S}_k$ be the optimal solver for 2^k operations. We let $\mathcal{S}^*$ be the solver which mimics $\mathcal{S}_k$ from operations $2^{k-1}+1$ to 2^k. At operation 2^k it resets its state, reexecutes all 2^k operations under $\mathcal{S}_{k+1}$, and continues using $\mathcal{S}_{k+1}$ until operation 2^{k+1}. Since $g_{\mathcal{S}_{k+1}}(2^{k+1}) \leq 2 \cdot g_{\mathcal{S}_k}(2^k)$ it follows that $g_{\mathcal{S}^*}(p) \leq 4 \cdot \min_{\mathcal{S}}\{g_{\mathcal{S}}(p)\}$.

Our overall algorithm is very simple. We divide the n elements into $n' = n/q$ super-elements, each representing a contiguous block of q elements. Each unsplit sequence then consists of three parts: two subsequences in the leftmost and rightmost super-elements and a third subsequence consisting of unsplit super-elements. We use Gabow's algorithm on the unsplit super-elements, where the key of a super-element is the minimum over constituent elements. For the super-elements already split, we use the $\mathcal{S}^*$ split-findmin solver constructed as above. The cost of Gabow's algorithm is $O((m + n/q)\alpha(m, n/q)) = O(m + n)$ and the cost of using $\mathcal{S}^*$ on each super-element is $\Theta(\textsc{split-findmin}(m, n))$ by construction; therefore the overall cost is $\Theta(\textsc{split-findmin}(m, n))$.

One can easily extend the proof to randomized split-findmin solvers by defining the findmin-action as selecting a distribution over actions.

Remark. The *time*-bounds in Theorem 21 are clearly tight, for both pointer machines [147] and RAMs. The open question is whether SPLIT-FINDMIN is $\Theta(m \log \alpha(m, n))$ or $\Theta(m)$ or something else.

A.3 MST and SSSP Sensitivity Analysis

Tarjan's sensitivity analysis algorithms [194] run in $O(m\alpha(m, n))$ time. Dixon et al. [57] gave two sensitivity algorithms, a randomized one running in expected linear-time, and a provably optimal deterministic one with unknown running time. In Theorems 22 and 23 we give explicit deterministic algorithms running in $O(\textsc{split-findmin}(m, n)) = O(m \log \alpha(m, n))$ time.

The MST sensitivity analysis problem is, given a graph G and $T = MST(G)$, to compute the largest amount that each edge-weight can be perturbed without affecting the equality $T = MST(G)$. Clearly, if $e \notin T$ then increasing $w(e)$ cannot invalidate $T = MST(G)$, and if $e \in T$ then decreasing $w(e)$ cannot invalidate $T = MST(G)$. Therefore, there are two disjoint problems: one for the non-tree edges and the other for the tree edges. Komlós's [141] MST verification algorithm solves the non-tree edge problem in linear time. Given [141] and some straightforward properties of MSTs (see Section 7.2.2), we will restate the MST sensitivity analysis problem as follows:

Given $T = MST(G)$, compute, for each edge $e \in T$, $MST(G - \{e\}) - T$.

One can easily see that $MST(G - \{e\}) - T$ consists of a single edge, say e', which is the best *replacement edge* for $e \in T$. So if $w(e)$ is perturbed in isolation, the identity $T = MST(G)$ is falsified precisely when $w(e)$ becomes greater than $w(e')$.

Theorem 22 *The MST sensitivity analysis problem can be solved deterministically in* $\textsc{split-findmin}(m, n) = O(m \log \alpha(m, n))$ *time, where m and n are the number of edges and vertices, respectively.*

Proof: Root T at an arbitrary vertex; let $p(u)$ be the parent of u in T. We initialize a split-findmin structure, where the initial n-element sequence consists of the vertices of T in post-order. We maintain two invariants: (1) that every sequence in the split-findmin structure corresponds to some rooted subtree in T, and (2) that key(u) corresponds to the weight of the min-weight edge connecting u to a vertex outside u's sequence.

By invariants (1) and (2), if S is a sequence in the split-findmin structure and u is the root of the subtree corresponding to S, then findmin$(u) = w(e')$ where e' is the edge $MST(G - \{(u, p(u))\}) - T$. Once e' is determined, we proceed to solve the problem on the subtree beneath u. Because of the post-order arrangement of the vertices, u is the right-most element in its sequence. We perform one split, severing u, and if u has k children we perform $k-1$ more splits to reestablish invariant (1). For each new sequence created, say corresponding to a child v of u, we restore invariant (2) by performing a number of decrease-keys. For each edge (v', z) such that $LCA(v', z) = u$ and v' is a descendant of v, we issue the operations decrease-key$(v', w(v', z))$.

In total there are $2m$ operations ($n-1$ splits, $n-1$ findmins, and $2[m-(n-1)]$ decrease-keys, 2 for each non-tree edge) thus the cost of the split-findmin structure is SPLIT-FINDMIN$(2m, n) = O(m \log \alpha(m, n))$. The rest of the costs, such as the post-order traversal and finding least common ancestors, are linear. Thus, the dominant term is SPLIT-FINDMIN(m, n).

□

The SSSP sensitivity analysis problem is easily reduced to MST sensitivity analysis — see [194].

Theorem 23 *The single-source shortest path tree sensitivity analysis problem can be solved deterministically in* SPLIT-FINDMIN$(m, n) = O(m \log \alpha(m, n))$ *time.*

Appendix B

Publications Arising from this Dissertation

In reverse chronological order of original publication:

S. Pettie. On the comparison-addition complexity of all-pairs shortest paths. In Proceedings 13th Int'l Symp. on Algorithms and Computation (ISAAC), pp. 32–43, 2002.

S. Pettie. An inverse-Ackermann style lower bound for the online minimum spanning tree verification problem. In Proceedings 43rd Annual Symposium on Foundations of Computer Science (FOCS), pp. 155–163, 2002.

S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, to appear. Preliminary version: A faster all-pairs shortest path algorithm for real-weighted sparse graphs. In Proceedings 29th Int'l Colloq. on Automata, Languages, and Programming (ICALP), pp. 85–97, 2002. (Received ICALP Best Student Paper award, 2002.)

S. Pettie, V. Ramachandran. Computing shortest paths with comparisons and additions. In Proceedings 13th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 267–276, 2002.

S. Pettie, V. Ramachandran. Minimizing randomness in minimum spanning tree, parallel connectivity and set maxima algorithms. In Proceedings 13th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 267–276, 2002.

S. Pettie, V. Ramachandran and S. Sridhar. Experimental evaluation of a new shortest path algorithm. Proceedings 4th Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 126–142, 2002.

S. Pettie, V. Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM* 49(1):16–34. Preliminary version appeared in Proceedings 27th Int'l Colloq. on Automata, Languages, and Programming (ICALP), LNCS 1853, pp. 49–60, 2000. (Received ICALP Best Paper award, 2000.)

S. Pettie, V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. on Computing* 31(6):1879–1895, 2002. Preliminary version appeared in Proceedings 7th Int'l Workshop on Randomization and Approximation Techniques (RANDOM), LNCS 1671, pp. 233–244, 1999.

Bibliography

[1] W. Ackermann. Zum Hilbertshen Aufbau der reelen Zahlen. *Math. Ann.*, 99:118–133, 1928.

[2] M. Agarwal, N. Saxena, and N. Kayal. PRIMES is in P. Manuscript, 2002.

[3] P. K. Agarwal and J. Erikson. Geometric range searching and its relatives. Advances in Discrete and Computational Geometry (B. Chazelle, J. E. Goodman, R. Pollack, eds.). *Contemporary Mathematics*, 223:1–56, 1998.

[4] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, Reading, MA, USA, 1975.

[5] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *J. ACM*, 37(2):213–223, 1990.

[6] M. Ajtai, J. Komlós, and E. Szemerédi. Deterministic simulation in Logspace. In *Ann. ACM Symposium on the Theory of Computation (STOC'87)*, pages 132–140, 1987.

[7] A. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical Report TR-71/87, Institute of Computer Science, Tel Aviv University, 1987.

[8] N. Alon, P. Erdős, and J. H. Spencer. *The Probabilistic Method.* John Wiley and Sons, New York, 1992.

[9] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Syst. Sci.*, 54(2):255–262, 1997.

[10] N. Alon and J. H. Spencer. *The Probabilistic Method.* John Wiley and Sons, New York, 2000.

[11] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Ann. ACM Symp. on Theory of Computing (STOC)*, pages 268–276, 2002.

[12] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Comput.*, 36(10):1258–1263, 1987.

[13] E. Bach. Realistic analysis of some randomized algorithms. *J. Comput. Syst. Sci.*, 42:30–53, 1991.

[14] E. Bach and J. Shallit. *Algorithmic Number Theory.* The MIT Press, 1996.

[15] R. Bar-Yehuda and S. Fogel. Partitioning a sequence into few monotone subsequences. *Acta Informatica*, 35:421–440, 1998.

[16] C. F. Bazlamaçci and K. S. Hindi. Verifying minimum spanning trees in linear time. In *Proc. Symp. on Operations Research (SOR)*, pages 139–144, 1997.

[17] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

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

[19] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448, 1972.

[20] B. Bolobás. *Random Graphs.* Academic Press, London, 1985.

[21] O. Borůvka. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti*, 3:37–58, 1926. In Czech.

[22] O. Borůvka. Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí. *Elektrotechnický obzor*, 15:153–154, 1926.

[23] G. S. Brodal and R. Fagerberg. Funnel heap – A cache oblivious priority queue. In *13th Int'l Symp. on Algorithms and Computation (ISSAC), LNCS 2518*, pages 219–230, 2002.

[24] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time pointer-machine algorithms for LCAs, MST verification, and dominators. In *Proc. 30th ACM Symposium on Theory of Computing (STOC'98)*, pages 279–288, May 23–26 1998.

[25] D. Yu. Burago. A lower bound for the complexity of the split-find problem for a pointer machine (in russian). *Trudy Sankt-Peterburgskogo Matematicheskogo Obshchestva*, 2:23–38, 1993. English version: American Mathematical Society Translations Series 2, Volume 159 (*Proceedings of the St. Petersburg Mathematical Society. Vol. II*), Simeon Ivanov, ed., pages 15–24, 1994.

[26] B. Chazelle. Computing on a free tree via complexity-preserving mappings. *Algorithmica*, 2(3):337–361, 1987.

[27] B. Chazelle. A faster deterministic algorithm for minimum spanning trees. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 22–31, Silver Spring, MD, October 20–22 1997. IEEE Computer Society Press.

[28] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000.

[29] B. Chazelle. The soft heap: an approximate priority queue with optimal error rate. *J. ACM*, 47(6):1012–1027, 2000.

[30] B. Chazelle. The power of nonmonotonicity in geometric searching. In *Proceedings 18th Annual ACM Symp. Comput. Geom.*, pages 88–93, 2002.

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

[32] D. Cheriton and R. E. Tarjan. Finding minimum spanning trees. *SIAM J. Comput.*, 5:724–742, 1976.

[33] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Math. Programming*, 73(2):129–174, 1996.

[34] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. *SIAM J. Comput.*, 28(4):1326–1346, 1999.

[35] F. Y. Chin, J. Lam, and I. N. Chen. Efficient parallel algorithms for some graph problems. *Commun. ACM*, 25(9):659–665, 1982.

[36] K. W. Chong, Y. Han, and T. W. Lam. Concurrent threads and optimal parallel minimum spanning trees algorithm. *J. ACM*, 48(2):297–323, 2001.

[37] K. W. Chong and T. W. Lam. Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM. *J. Algor.*, 18(3):378–402, 1995.

[38] G. Choquet. Etude de certains r'eseaux de routes. *Comptes Rendus Acad. Sci.*, 206:310–313, 1938.

[39] R. Cole. Personal communication. 1999.

[40] R. Cole, P. N. Klein, and R. E. Tarjan. A linear-work parallel algorithm for finding minimum spanning trees. In *Proc. SPAA'94*, pages 11–15, 1994.

[41] R. Cole, P. N. Klein, and R. E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work using random sampling. In *Proc. SPAA'96*, pages 243–250, 1996.

[42] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree, and graph problems. In *Proc. FOCS'86*, pages 478–491, 1986.

[43] R. Cole and U. Vishkin. Approximate parallel scheduling. II. Applications to logarithmic-time optimal parallel graph algorithms. *Information and Computation*, 92(1):1–47, 1991.

[44] C. Cooper, A. Frieze, K. Mehlhorn, and V. Priebe. Average-case complexity of shortest-paths problems in the vertex-potential model. *J. Random Struct. Alg.*, 16(1):33–46, 2000.

[45] C. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proc. 19th Ann. ACM Symp. on the Theory of Computing*, pages 1–6, 1987.

[46] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 1990.

[47] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[48] C. A. Crane. Linear lists and priority queues as balanced binary trees. Technical Report STAN-CS-72-259, Computer Science Dept., Stanford University, 1972.

[49] J. Czekanowski. Zur differentialdiagnose der neandertalgruppe, korrespondez-blatt der deutschen gesellschaft für anthropologie. *Ethn. u Urg.*, 40:44–47, 1909.

[50] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proc. 35th ACM Symp. on the Theory of Computing*, 2003.

[51] C. Demetrescu, M. Thorup, R. A. Chowdhury, and V. Ramachandran. Oracles for distances avoiding a node or link failure. Manuscript, 2003.

[52] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[53] E. W. Dijkstra. Some theorems on spanning subtrees of a graph. *Indag. Math.*, XXII(2):196–199, 1960.

[54] E. W. Dijkstra. Some beatiful arguments using mathematical induction. *Acta Informatica*, 13:1–8, 1980.

[55] E. W. Dijkstra. Personal communication. 2001.

[56] E. A. Dinic. Finding shortest paths in a network. *Transportation Modeling Systems*, pages 36–44, 1978.

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

[58] J. Edmonds. Matroids and the greedy algorithm. *Math. Programming*, 1:127–136, 1971.

[59] P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 5:17–61, 1960.

[60] P. Erdős and G. Szekeres. A combinatorial problem in geometry. *Composito Mathematica*, 2:464–470, 1935.

[61] J. Fakcharoenphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proc. 42nd IEEE Symp. on Found. of Computer Science (FOCS)*, pages 232–241, 2001.

[62] A. M. Ferrenberg, D. F. Landau, and Y. J. Wong. Monte Carlo simulations: Hidden errors from “good” random number generators. *Physical Review Letters*, 69(23):3382–3388, 1992.

[63] K. Florek, L. Lukasziewicz, J. Perkal, H. Steinhaus, and S. Zubrzycki. Sur la liaison et la division des points d’un ensemble fini. *Colloq. Math.*, 2:282–285, 1951.

[64] R. Floyd and R. Rivest. Expected time bounds for selection. *Commun. ACM*, 18(3):165–172, 1975.

[65] L. R. Ford and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, 1962.

[66] G. N. Frederickson. Fast algorithms for shortest paths in planar networks, with applications. *SIAM J. Comput.*, 16(6):1004–1022, 1989.

[67] G. N. Frederickson. Planar graph decomposition and all pairs shortest paths. *J. ACM*, 38(1):162–204, 1991.

[68] M. L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11:29–35, 1975.

[69] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, 5(1):83–89, 1976.

[70] M. L. Fredman. On the efficiency of pairing heaps and related data structures. *J. ACM*, 46(4):473–501, 1999.

[71] M. L. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st annual ACM Symposium on Theory of Computing*, pages 345–354, 1989.

[72] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[73] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[74] M. L. Fredman and D. E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. In *Proc. STOC'90*, pages 1–7, 1990.

[75] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, June 1994.

[76] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, pages 90–100, 1985.

[77] H. N. Gabow. Scaling algorithms for network problems. *J. Comput. Syst. Sci.*, 31(2):148–168, 1985.

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

[79] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985.

[80] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989.

[81] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph-matching problems. *J. ACM*, 38(4):815–853, 1991.

[82] Z. Galil and O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Inform. and Comput.*, 134(2):103–139, 1997.

[83] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Comput.*, 20(6):1046–1067, 1991.

[84] P. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. *SIAM J. Comput.*, 28(2):733–769, 1999.

[85] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? *Theory of Comput. Syst.*, 32:327–359, 1999.

[86] W. Goddard, C. Kenyon, V. King, and L. Schulman. Optimal randomized algorithms for local sorting and set-maxima. *SIAM J. Comput.*, 22(2):272–283, 1993.

[87] A. V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.*, 24(3):494–504, 1995.

[88] A. V. Goldberg. Shortest path algorithms: Engineering aspects. In *Int'l Symp. on Algorithms and Computation (ISAAC), LNCS 2223*, pages 502–513, 2001.

[89] A. V. Goldberg. A simple shortest path algorithm with linear average time. In *9th European Symposium on Algorithms (ESA), LNCS 2161*, pages 230–241, 2001.

[90] A. V. Goldberg and T. Radzik. A heuristic improvement of the Bellman-Ford algorithm. *Appl. Math. Lett.*, 6(3):3–6, 1993.

[91] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998.

[92] A. V. Goldberg and C. Silverstein. Implementations of Dijkstra's algorithm based on multi-level buckets. In *Network optimization*, volume 450 of *Lecture Notes in Econom. and Math. Systems*, pages 292–327. Springer, Berlin, 1997.

[93] M. T. Goodrich and S. R. Kosaraju. Sorting on a parallel pointer machine with applications to set expression evaluation. *J. ACM*, 43(2):331–361, 1996.

[94] R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. *Ann. Hist. Comput.*, 7(1):43–57, 1985.

[95] R. L. Graham, A. C. Yao, and F. F. Yao. Information bounds are weak in the shortest distance problem. *J. ACM*, 27(3):428–444, 1980.

[96] J. Gudmundsson, C. Levcopoulos, G. Narahimsan, and M. Smid. Approximate distance oracles for geometric graphs. In *Proc. 13th Ann. ACM-SIAM Symp. On Discrete Algorithms (SODA)*, pages 828–837, 2002.

[97] J. Gustedt. Efficient Union-Find for planar graphs and other sparse graph classes. *Theor. Comp. Sci.*, 203(1):123–141, 1998.

[98] T. Hagerup. Improved shortest paths on the word RAM. In *Proc. 27th Int'l Colloq. on Automata, Languages, and Programming (ICALP), LNCS vol. 1853*, pages 61–72, 2000.

[99] S. Halperin and U. Zwick. An optimal randomised logarithmic time connectivity algorithm for the EREW PRAM. *J. Comput. Syst. Sci.*, 53(3):395–416, 1996.

[100] S. Halperin and U. Zwick. Optimal randomized EREW PRAM algorithms for finding spanning forests. *J. Algor.*, 39(1):1–46, 2001.

[101] Y. Han, V. Y. Pan, and J. H. Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. *Algorithmica*, 17(4):399–415, 1997.

[102] Y. Han and M. Thorup. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. In *43rd Ann. Symp. on Found. of Comput. Sci.*, pages 135–144, 2002.

[103] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[104] S. Hart and M. Sharir. Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes. *Combinatorica*, 6(2):151–177, 1986.

[105] M. R. Henzinger, P. N. Klein, S. Rao, and S. Subramanian. Faster shortest path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, 1997.

[106] D. S. Hirschberg. Parallel algorithms for the transitive closure and the connected component problems. In *8th Annual ACM Symposium on Theory of Computing*, pages 55–57, 1976.

[107] D. S. Hischberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commun. ACM*, 22(8):461–464, 1979.

[108] C. A. R. Hoare. Algorithm 63 (PARTITION), algorithm 64 (QUICKSORT) and algorithm 65 (FIND). *Commun. ACM*, 4(7):321–322, 1961.

[109] J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM J. Comput.*, 2:294–303, 1973.

[110] T-s Hsu. *Graph Augmentation and Related Problems: Theory and Practice.* Ph.D. Thesis, University of Texas at Austin, 1993.

[111] T-s Hsu and Vijaya Ramachandran. Efficient massively parallel implementation of some combinatorial algorithms. *Theor. Comp. Sci.*, 162(2):297–322, 1996.

[112] J. Iacono. Improved upper bounds for pairing heaps. In *Scandinavian Workshop on Algorithm Theory (SWAT, LNCS 1851*, pages 32–43, 2000.

[113] R. Impagliazzo and D. Zuckerman. How to recycle random bits. In *30th Ann. Symp. on Foundations of Computer Science (FOCS)*, pages 248–253, 1989.

[114] J. JàJà. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1992.

[115] V. Jarník. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 6:57–63, 1930. In Czech.

[116] V. Jarník and M. Kössler. O minimálních grafech obsahujících n daných bodu. *Časopis Pěst. Mat.*, 63:223–235, 1934.

[117] A. Joffe. On a set of almost deterministic k-independent random variables. *Ann. Probability*, 2(1):161–162, 1974.

[118] D. B. Johnson. Priority queues with update and finding minimum spanning trees. *Info. Proc. Lett.*, 4(3):53–57, 1975.

[119] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.

[120] D. B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. *J. Algor.*, 19(3):383–401, 1995.

[121] D. B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} n)$ parallel time for the CREW PRAM. *J. Comput. Syst. Sci.*, 54(2):227–242, 1997.

[122] N. D. Jones. *Computability and Complexity from a Programming Perspective.* MIT Press, Boston, London, 1997.

[123] R. Kalaba. On some communication network problems. In *Proc. Symp. Applied Mathematics*, pages 261–280, 1960.

[124] D. R. Karger. Random sampling in matroids, with applications to graph connectivity and minimum spanning trees. In *34th Annual Symposium on Foundations of Computer Science*, pages 84–93, 1993.

[125] D. R. Karger. *Random Sampling in Graph Optimization Problems.* Ph.D. thesis, Stanford University, 1995.

[126] D. R. Karger. Random sampling and greedy sparsification for matroid optimization problems. *Math. Programming*, 82(1-2):41–81, 1998.

[127] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *J. ACM*, 42:321–329, 1995.

[128] D. R. Karger, D. Koller, and S. J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. *SIAM J. Comput.*, 22(6):1199–1217, 1993.

[129] H. J. Karloff and P. Raghavan. Randomized algorithms and pseudorandom numbers. *J. ACM*, 40(3):454–476, 1993.

[130] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Computer Science*, pages 869–942. MIT Press/Elsevier, 1990.

[131] R. M. Karp and R. E. Tarjan. Linear expected-time algorithms for connectivity problems. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing (STOC '80)*, pages 368–377, New York, April 1980. ACM Press.

[132] L. R. Kerr. The effect of algebraic structure on the computational complexity of matrix multiplication. Technical Report TR70-75, Computer Science Dept., Cornell University, June 1970.

[133] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997.

[134] V. King, C. K. Poon, V. Ramachandran, and S. Sinha. An optimal EREW PRAM algorithm for minimum spanning tree verification. *Info. Proc. Lett.*, 62(3):153–159, 1997.

[135] M. M. Klawe. Superlinear bounds for matrix searching problems. *J. Algor.*, 13(1):55–78, 1992.

[136] P. N. Klein. Preprocessing an undirected planar network to enable fast approximate distance queries. In *Proc. 13th Ann. ACM-SIAM Symp. On Discrete Algorithms (SODA)*, pages 820–827, 2002.

[137] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *J. Algor.*, 25(2):205–220, 1997.

[138] P. N. Klein and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. In *Proc. 26th Annual ACM Symp. on Theory of Computing*, pages 9–15, 1994.

[139] D. E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[140] S. G. Kolliopoulos and C. Stein. Finding real-valued single-source shortest paths in $o(n^3)$ expected time. *J. Algor.*, 28(1):125–141, July 1998.

[141] J. Komlós. Linear verification for spanning trees. *Combinatorica*, 5(1):57–65, 1985.

[142] B. Korte and J. Nešetřil. Vojtěch jarník's work in combinatorial optimization. *Discrete Mathematics*, 235:1–17, 2001.

[143] A. Kotzig. Súvisl'e grafy s minimálnou hodnotou v konečnom súvislom grafe. *Časopis Pěst. Mat.*, 86:1–6, 1961.

[144] L. Kowalik and M. Kurowski. Short path queries in planar graphs in constant time. In *Proc. 35th ACM Symp. on the Theory of Computing*, 2003.

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

[146] J. B. Kruskal. A reminiscence about shortest spanning subtrees. *Archivum Mathematicum*, 33:13–14, 1997.

[147] H. LaPoutré. Lower bounds for the union-find and the split-find problem on pointer machines. *J. Comput. Syst. Sci.*, 52:87–99, 1996.

[148] L. L. Larmore. An optimal algorithm with unknown time complexity for convex matrix searching. *Info. Proc. Lett.*, 36(3):147–151, November 1990.

[149] E. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart & Winston, New York, 1976.

[150] V. Liberatore. Matroid decomposition methods for the set maxima problem. In *Proc. SODA'98*, pages 400–409, 1998.

[151] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36(2):177–189, 1979.

[152] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9(3):615–627, 1980.

[153] H. Loberman and A. Weinberger. Formal procedures for connecting terminals with a minimum total wire length. *J. ACM*, 4(4):428–437, 1957.

[154] M. Luby and A. Wigderson. Pairwise independence and derandomization. Technical Report CSD-95-880, University of California, Berkeley, 1995.

[155] A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proc. 13th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 372–381, 2002.

[156] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. 10th Ann. Euro. Symposium on Algorithms (ESA), LNCS 2461*, pages 723–735, 2002.

[157] K. Mehlhorn and S. Näher. *LEDA : A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 2000.

[158] K. Mehlhorn, S. Näher, and H. Alt. A lower bound on the complexity of the union-split-find problem. *SIAM J. Comput.*, 17(6):1093–1102, 1988.

[159] Guan Mei-Gu. The method of eliminating cycles for finding minimum spanning trees (in chinese). *Shuxue de Shihian yu Renshi*, 4, 1975.

[160] U. Meyer. Single source shortest paths on arbitrary directed graphs in linear average-case time. In *12th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 797–806, 2001.

[161] U. Meyer and P. Sanders. Parallel shortest path for arbitrary graphs. In *Proc. 6th Int'l Euro-Par Conference, LNCS 1900*, pages 461–470, 2000.

[162] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proc. 11th Ann. Euro. Symposium on Algorithms (ESA)*, 2003.

[163] G. L. Miller. Riemann's hypothesis and tests for primality. *J. Comput. Syst. Sci.*, 13(3):300–317, 1976.

[164] J. S. B. Mitchell. Geometric shortest paths and network optimization. In *Handbook of computational geometry*, pages 633–701. North-Holland, Amsterdam, 2000.

[165] A. Moffat and T. Takaoka. An all pairs shortest path algorithm with expected time $O(n^2 \log n)$. *SIAM J. Comput.*, 16(6):1023–1031, 1987.

[166] B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. In *DIMACS Series on Discrete Math. and Theor. Comp. Sci.*, 1994.

[167] J. Nešetřil. A few remarks on the history of MST problem. *Archivum Mathematicum*, 33:15–22, 1997.

[168] J. Nešetřil, E. Milková, and H. Nešetřilová. Otakar Borůka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233:3–36, 2001.

[169] S. Pettie. Finding minimum spanning trees in $O(m\alpha(m,n))$ time. Technical Report CS-TR-99-23, University of Texas, Austin, 1999.

[170] S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 267–276, 2002.

[171] S. Pettie and V. Ramachandran. Minimizing randomness in minimum spanning tree, parallel connectivity and set maxima algorithms. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 713–722, 2002.

[172] C. K. Poon and V. Ramachandran. A randomized linear-work EREW PRAM algorithm to find a minimum spanning forest. *Algorithmica*, 35(3):257–268, 2003. (Special issue of selected papers from 8th Ann. Int'l Symp. on Algorithms and Computation, Singapore, 1997).

[173] R. C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, pages 1389–1401, 1957.

[174] M. O. Rabin. Probabilistic algorithm for testing primality. *J. Number Theory*, 12(1):128–138, 1980.

[175] V. Ramachandran. Personal communication. 1999.

[176] P. Rosenstiehl. L'arbre minimum d'un graphe. In *Theory of Graphs (Intl. Symp., Rome, 1966)*, pages 357–368. Gordon and Breach, New York, 1967.

[177] P. Sanders. Fast priority queues for cached memory. *J. of Experimental Algs.*, 5, 2000.

[178] C. Savage and J. JáJá. Fast, efficient parallel algorithms for some graph problems. *SIAM J. Comput.*, 10(4):682–691, 1981.

[179] J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM J. Discr. Math.*, 8(2):223–250, 1995.

[180] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Syst. Sci.*, 51(3):400–403, 1995.

[181] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algor.*, 3(1):57–67, 1982.

[182] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *40th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 605–614. IEEE Computer Society Press, 1999.

[183] G. Sollin. Problèm de l'arbre minimum (unpublished). 1961.

[184] P. M. Spira. A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$. *SIAM J. Comput.*, 2:28–32, 1973.

[185] P. M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM J. Comput.*, 4(3):375–380, 1975.

[186] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 14(3):354–356, 1969.

[187] C. R. Subramanian. A generalization of Janson inequalities and its application to finding shortest paths. In *Proc. 10th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 795–804, 1999.

[188] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Info. Proc. Lett.*, 43(4):195–199, 1992.

[189] T. Takaoka. Subcubic cost algorithms for the all pairs shortest path problem. *Algorithmica*, 20(3):309–318, 1998.

[190] R. E. Tarjan. Efficiency of a good but not linear set merging algorithm. *J. ACM*, 22:215–225, 1975.

[191] R. E. Tarjan. Complexity of monotone networks for computing conjunctions. *Ann. Discrete Math.*, 2:121–133, 1978.

[192] R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.

[193] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.*, 18(2):110–127, 1979.

[194] R. E. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path problems (see also corrigendum IPL **23**(4), p. 219). *Info. Proc. Lett.*, 14(1):30–33, 1982.

[195] R. E. Tarjan. *Data structures and network algorithms*, volume 44 of *CBMS-NSF Reg. Conf. Ser. Appl. Math.* SIAM, 1983.

[196] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.

[197] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. In *Proc. 42nd IEEE Symp. on Found. of Computer Science (FOCS)*, pages 242–251, 2001.

[198] M. Thorup. Quick k-median, k-center, and facility location for sparse graphs. In *Proceedings Int'l Colloq. on Automata, Languages, and Programming (ICALP), LNCS 2076*, pages 249–260, 2001.

[199] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *35th Ann. ACM Symp. on Theory of Comput.*, pages ??–??, 2003.

[200] M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. 33rd Ann. ACM Symp. on Theory of Computing*, pages 183–192, 2001.

[201] J. L. Träff and C. D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. In *Parallel algorithms for irregularly structured problems, LNCS 1117*, pages 183–194, 1996.

[202] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[203] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[204] A. Wiernik and M. Sharir. Planar realizations of nonlinear Davenport-Schinzel sequences by segments. *Discrete Comput. Geom.*, 3(1):15–47, 1988.

[205] J. W. J. Williams. Algorithm 232 (heapsort). *Commun. ACM*, 7:347–348, 1964.

[206] A. C. Yao. An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees. *Info. Proc. Lett.*, 4(1):21–23, 1975.

[207] A. C. Yao. Space-time tradeoff for answering range queries. In *Proc. 14th ACM Symposium on Theory of Computing (STOC'82)*, pages 128–136, 1982.

[208] U. Zwick. Exact and approximate distances in graphs — a survey. In *Proceedings 9th European Symposium on Algorithms (ESA)*, pages 33–48, 2001. See updated version at http://www.cs.tau.ac.il/~zwick/.

[209] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002.

Vita

Seth Pettie was born in Summertown, Tennessee, on October 14, 1976, the son of David and Sherry Pettie. After completing his studies at the Washington Waldorf high school, in 1994, he headed off to Brandeis University with no certain plans. He graduated from Brandeis in 1998 with a degree in computer science and entered The University of Texas at Austin to pursue a Ph.D. in computer sciences.

Permanent Address: 2216 40th Street Northwest #3
Washington, DC 20007 USA

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1]LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.