Copyright

by

Seth Pettie

2003

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

Acknowledgments v

Abstract vi

Chapter 1 Introduction 1
1.1 Overview of the Results . . . 2
1.1.1 Shortest Paths . . . 2
1.1.2 Minimum Spanning Trees . . . 3
1.2 Preliminaries . . . 4
1.2.1 Asymptotic Notation . . . 4
1.2.2 Chapter Dependencies . . . 4

I Shortest Paths 6

Chapter 2 Introduction to Shortest Paths 7
2.1 History . . . 7
2.1.1 Single-Source Shortest Paths . . . 7
2.1.2 All-Pairs Shortest Paths . . . 8
2.1.3 Variations . . . 9
2.1.4 Organization . . . 10
2.2 Our Contributions . . . 10
2.3 Problem Definition . . . 11
2.4 The Comparison-Addition Model . . . 12
2.4.1 Non-Uniform Complexity . . . 13
2.4.2 Basic Techniques . . . 13
2.4.3 Lower Bounds . . . 14
2.5 Dijkstra's Algorithm . . . 14
2.6 The Hierarchy Approach . . . 16

Chapter 3 Hierarchies & Shortest Paths **18**
3.1 Independent Subproblems 18
3.2 A Stratified Hierarchy 20
3.3 A Generalized Hierarchy-Type Algorithm 25
3.4 Correctness 27
3.5 Implementation Details 31
3.6 Lower Bounds 34
3.6.1 Characterization of Hierarchy-Type Algorithms 35
3.6.2 Lower Bound: Directed Graphs 38
3.6.3 Lower Bound: Undirected Graphs 39

Chapter 4 Shortest Paths on Directed Graphs **41**
4.1 A Faster APSP Algorithm 42
4.1.1 Relative Distances and Their Approximations 42
4.1.2 GENERALIZED-VISIT and Relative Distances 43
4.1.3 Computing Relative Distances 45
4.2 A Non-Uniform APSP Algorithm 49
4.2.1 Preliminaries 49
4.2.2 Lengths, Distances, etc. 50
4.2.3 Buckets, Heaps, etc. 53

Chapter 5 Shortest Paths on Undirected Graphs **58**
5.1 An Undirected Shortest Path Algorithm 59
5.1.1 Refined Hierarchies 59
5.1.2 The UNDIRECTED-VISIT Algorithm 59
5.1.3 A Lazy Bucketing Structure 61
5.1.4 Analysis of UNDIRECTED-VISIT 63
5.2 An Algorithm for a Refined Hierarchy 64
5.2.1 Phase 1: Computing the MST and $\mathcal{SH}$ 64
5.2.2 Definitions and Properties 65
5.2.3 Phase 2: Computing Shortcut Trees 69
5.2.4 Phase 3: Computing a Refined Hierarchy 71
5.3 Variations on the Algorithm 75
5.3.1 Simpler and Slower 75
5.3.2 Sort-of-Undirected Graphs 76
5.4 Discussion 77

Chapter 6 Experimental Evaluation of a Shortest Path Algorithm **79**
6.1 Introduction 79
6.1.1 Previous Work 80

6.1.2 Scope of the Experiment 80
6.2 Design Choices 81
6.2.1 Dijkstra's Algorithm 81
6.2.2 Pettie-Ramachandran Algorithm 82
6.2.3 Breadth First Search 83
6.3 Experimental Set-up 83
6.3.1 Graph Classes 83
6.4 Results 84
6.5 Discussion 94

II Minimum Spanning Trees 95

Chapter 7 Introduction to Minimum Spanning Trees 96
7.1 History 96
7.2 Preliminaries 100
7.2.1 The Model 100
7.2.2 Cut and Cycle Properties 100
7.2.3 Basic Algorithms 103

Chapter 8 An Optimal Minimum Spanning Tree Algorithm 104
8.1 Introduction 104
8.1.1 Uniform vs. Non-Uniform Complexity 105
8.1.2 Speculation about MST* 106
8.2 The Soft Heap 107
8.3 Corruption and Contractibility 108
8.3.1 A Robust Contraction Lemma 108
8.3.2 The Partition Procedure 109
8.4 The Algorithm 111
8.4.1 Overview 111
8.4.2 MST Decision Trees 111
8.4.3 The Dense Case Algorithm 112
8.4.4 An Optimal Algorithm 112
8.4.5 Analysis 114
8.5 Avoiding Pointer Arithmetic 115
8.6 Introducing A Little Randomness 116
8.7 Performance on Random Graphs 117
8.7.1 The Edge-Addition Martingale 118
8.7.2 Analysis 119
8.8 Discussion 121

Chapter 9 A Lower Bound on MST Verification **125**
9.1 Introduction . . . 125
9.1.1 Related Work . . . 126
9.1.2 Organization . . . 128
9.2 Preliminaries . . . 129
9.2.1 A Variation on Ackermann's Function . . . 129
9.2.2 The Input Distribution . . . 130
9.2.3 A Measure of Information . . . 131
9.3 The Lower Bound . . . 134
9.3.1 Proof of Main Theorem . . . 138
9.4 Upper Bounds . . . 141

Chapter 10 A Time-Work Optimal Parallel MST Algorithm **144**
10.1 The PRAM Model . . . 144
10.2 History . . . 145
10.3 Techniques . . . 146
10.4 The High-Level Algorithm . . . 148
10.5 Phase 1 . . . 149
10.5.1 The k-Min Forest . . . 150
10.5.2 Borůvka-A Steps . . . 151
10.5.3 Filtering Edges via The Filter Forest . . . 155
10.5.4 Finding a k-Min Forest . . . 158
10.5.5 Performance of Find-k-min . . . 159
10.6 Phase 2 . . . 160
10.6.1 The Find-MST Procedure . . . 160
10.7 Proof of the Main Theorem . . . 162
10.8 Processor Allocation . . . 163
10.9 Adaptations to Practical Parallel Models . . . 166
10.10Discussion . . . 166

Chapter 11 A Reduced Randomness MST Algorithm **168**
11.1 Limited Independence Sampling . . . 169
11.1.1 Pairwise Independent Sampling on the EREW PRAM . . . 172
11.2 A Low-Randomness MST Algorithm . . . 174
11.2.1 Techniques . . . 174
11.2.2 The Algorithm . . . 178
11.3 Discussion . . . 181

Appendix A Split-Findmin and Its Applications **182**
A.1 Background 182
A.2 An Optimal Split-Findmin Structure 183
A.3 MST and SSSP Sensitivity Analysis 185

Appendix B Publications **187**

Bibliography **189**

Vita **204**

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.

$$f(n) = O(g(n)) \quad \equiv \quad \exists c_1, c_2 \, \forall n > 0 \; : \; f(n) \leq c_1 \cdot g(n) + c_2$$

$$f(n) = \Omega(g(n)) \quad \equiv \quad g(n) = O(f(n))$$

$$f(n) = \Theta(g(n)) \quad \equiv \quad f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

$$f(n) = \omega(g(n)) \quad \equiv \quad \lim_{n \to \infty} g(n)/f(n) = 0$$

$$f(n) = o(g(n)) \quad \equiv \quad g(n) = \omega(f(n))$$

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 algo-

rithm 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 in-

formation 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

$$\texttt{if } a < b \texttt{ then } \ldots \texttt{ else } \ldots$$

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 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 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 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 NORM-values from the set $\{2^i\}_{i \geq 0}$; a node with 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 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 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 k^{th} 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 NORM(x) is defined as:

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

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 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 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 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 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* NORM(x)*-partition of $V(x)$. Moreover, if the graph is undirected then then all such permutations give* 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.

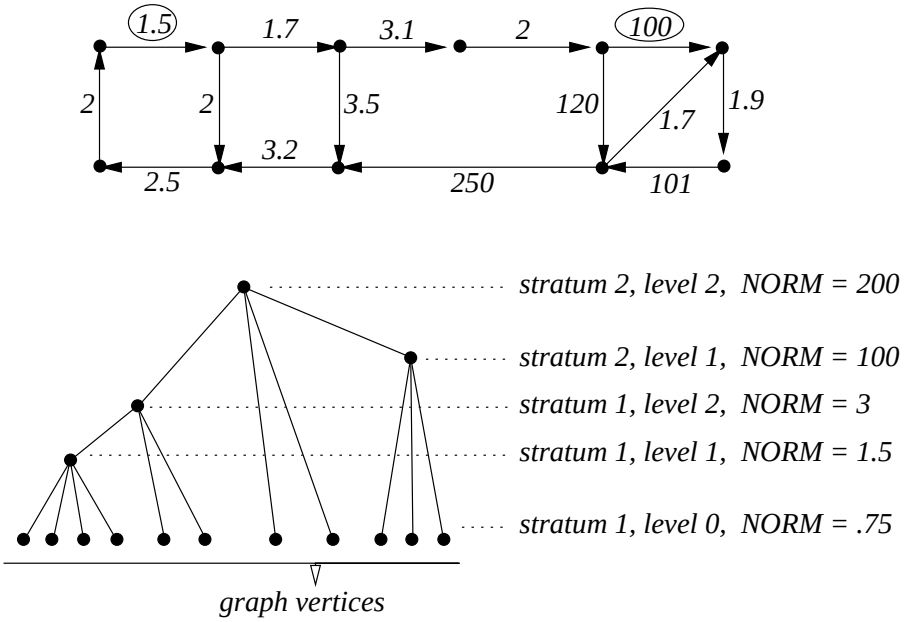

Figure 3.1: Above: the input graph. Circled edge lengths represent "normalizing" lengths. Below: the associated $\mathcal{SH}$. It has two strata, based on the normalizing lengths $\ell_{r_1} = 1.5$ and $\ell_{r_2} = 100$. A stratum k, level i node x has $\text{NORM}(x) = \ell_{r_k} \cdot 2^{i-1}$, and 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. $$\textit{For any } x \in \mathcal{SH},\ \frac{\textsc{diam}(x)}{\textsc{norm}(x)} \;<\; 2n$$
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 $\text{NORM}(x) = \infty$ and $V(x) = V$.
(2) See Lemma 2.
(3) If $p(x)$ and x are in the same stratum, then clearly $\text{NORM}(p(x))$ is a multiple of $\text{NORM}(x)$. If $\text{NORM}(p(x)) = \ell_{r_k} \cdot 2^i$, where $i \geq -1$, and x is not in stratum k, then $\text{DIAM}(x) < (|V(x)| - 1) \cdot 2\text{NORM}(x) < n \cdot \ell_{r_k}/2n \leq \text{NORM}(p(x))$.
(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 $2\text{NORM}(x)$. Therefore, $\text{DIAM}(x) < |V(x) - 1| \cdot 2\text{NORM}(x) < 2n \cdot \text{NORM}(x)$.
(6) Let z^j denote the j^{th} ancestor of $z \in \mathcal{SH}$. Since the NORM-value of a node is no more than half that of its parent (see Equation 3.1), we have $\text{NORM}(z)/\text{NORM}(z^j) \leq 2^{-j}$. We write z desc. x to mean z is a (not necessarily proper) descendant of x in $\mathcal{SH}$. Using the definition of DIAM from Equation 3.2 we can bound the sum as follows.

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

(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 NORM-values in $O(m \log n)$ time. Let $\text{NORM}_1 < \text{NORM}_2 < \cdots < \text{NORM}_k$ be the possible NORM-values and G' be the input graph G restricted to edges with length less than

$2\text{NORM}_{\lfloor k/2 \rfloor}$. 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 $\text{NORM}_{\lfloor k/2 \rfloor}$. We proceed recursively on the $\{C_i\}_i$ (finding $\mathcal{SH}$-nodes with NORM-values in the range $\text{NORM}_1..\text{NORM}_{\lfloor k/2 \rfloor - 1}$) and on the graph G^c (for NORM-values in the range $\text{NORM}_{\lfloor k/2 \rfloor + 1}..\text{NORM}_k$). 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:

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

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 $V(\text{ROOT}) = V$ 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. $S \supseteq V(\text{ROOT})^{[0,\infty)} = V$, 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 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

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

where t_x is set to

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

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 $[a_x,\ a_x + \text{NORM}(x))$ is not empty
    Let y be the leftmost child of x in bucket $[a_x,\ a_x + \text{NORM}(x))$
    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 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 CHILD(x) represents a 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 NORM(x) and the bucket array represents an interval spanning $[d(s,x), d(s,x) + \text{DIAM}(x)]$ where $d(s,x) = d(s,V(x))$ is the distance to any node in $V(x)$. When 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 $[t_x + i\text{NORM}(x), t_x + (i+1)\text{NORM}(x))$. We will choose t_x such that $t_x \leq d(s,x) < t_x + \text{NORM}(x)$. Therefore, at most $\left\lceil \frac{\text{DIAM}(x)}{\text{NORM}(x)} \right\rceil + 1$ buckets are required. For notational convenience we may refer to a bucket by its associated interval.

We will say x is *inactive* until 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 $[q, q + \text{NORM}(x))$ is either an inactive child such that $D(y) \in [q, q + \text{NORM}(x))$, or an active child such that $V(y)^{[0,q)} \subseteq S$, but $V(y)^{[q,q+\text{NORM}(x))} \not\subseteq S$.*

Suppose that in the call to GENERALIZED-VISIT(x,I), I spans the intervals of k of x's buckets, say, buckets $b_{j+1}, b_{j+2}, \ldots, b_{j+k}$. 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 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 GENERALIZED-VISIT.

3.4 Correctness of GENERALIZED-VISIT

In this section we prove that GENERALIZED-VISIT works correctly. Specifically, we show that 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 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 $\text{NORM}(x)$, ever after. Now suppose $\text{DIAM}(x) < \text{NORM}(p(x))$. Since this is not the first recursive call on x, we know, by Lemma 5, that $d(s,x) < a$ and therefore that $d(s,x) + \text{DIAM}(x) < b$, 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 $\text{NORM}(p(x))$ and $\text{NORM}(x)$ respectively, and that they each cover the interval $[d(s,x),\ d(s,x) + \text{DIAM}(x)]$. Furthermore, each interval in $\mathcal{I}_2$ must be wholly contained in one interval from $\mathcal{I}_1$. Because we use a *stratified* hierarchy, $\text{NORM}(p(x))$ is not necessarily a multiple of $\text{NORM}(x)$. Therefore, these requirements can only be satisfied if $\text{DIAM}(x) < \text{NORM}(p(x))$, i.e., if $\text{NORM}(x)$ does not divide $\text{NORM}(p(x))$ 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 $x = \text{ROOT}$, $[a,b) = [0,\infty)$, 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 $\text{NORM}(x)$ 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 $V(x)^{[a_x, a_x + \text{NORM}(x))}$ to be visited.

Consider the entry into the inner while loop in GENERALIZED-VISIT, and let $I = [a_x, a_x + \text{NORM}(x))$, that is, the current bucket is labeled I. Imagine that we consider each node in $\text{CHILD}(x) = (x_j)_j$ 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 $\text{NORM}(x)$-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 $(S, [a_x, \min\{a_x + \text{NORM}(x),\ b\}))$-independent. However, we need to show that it is (S, I)-independent, since it is the interval $I = [a_x, a_x + \text{NORM}(x))$ that would be passed to the recursive call. By Lemma 6, either $\text{NORM}(x)$ divides $b - a_x$ or $V(x)^{[0,b)} = V(x)$. If $\text{NORM}(x)$ divides $b - a_x$ then $I = [a_x, \min\{a_x + \text{NORM}(x), b\})$ since we only entered the outer while loop if $a_x < b$, implying $a_x \leq b - \text{NORM}(x)$. On the other hand, if $V(x)^{[0,b)} = V(x)$, then $V(x_j)$ being $(S, [a_x, \min\{a_x + \text{NORM}(x), b\}))$-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 $D(x_j) \geq a_x + \text{NORM}(x)$ (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 $a_x \geq b$ 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

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

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*

$$O\left(m + n \log\log n + \sum_{x \in X} \text{DEG}(x) \cdot \log \tfrac{\text{DIAM}(C_x)}{\text{NORM}(x)}\right)$$

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 $O(\log \frac{\text{DIAM}(x)}{\text{NORM}(x)})$, where $\frac{\text{DIAM}(x)}{\text{NORM}(x)}$ 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 $O(\log\log \text{DEG}(x))$, 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 $\sum_{x \in X} \text{DEG}(x) \log \frac{\text{DIAM}(x)}{\text{NORM}(x)}$ can be thought of as a measure of the entropy of a specific hierarchy, under two strong assumptions: first, that each $y \in \text{CHILD}(x)$ can appear in each of x's $\text{DIAM}(x)/\text{NORM}(x)$ 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 $\text{DIAM}(x)/\text{NORM}(x) < (\log n)^{O(1)}$, 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:*

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

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.

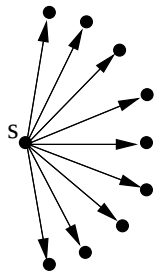

Figure 3.3: The star graph. If edge-lengths are permuted at random, finding a π_s satisfying Property 1 takes $\log((n-1)!) = \Omega(n \log n)$ comparisons.

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 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 SEP(u, v) as:

$$\text{SEP}(u, v) \quad = \quad \min_{\mathcal{C} \in \text{CYCLES}(u,v)} \quad \max_{e \in \mathcal{C}} \quad \ell(e)$$

To see the connection between the SEP-values and $\mathcal{SH}$, notice that $R_t(u, v) \equiv (\text{SEP}(u, v) \leq t)$ 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 SEP-values when the graph is undirected. Thorup [196] makes a similar observation, although he never uses the idea of a SEP function.

Observation 1 *If the graph is undirected,* 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* SEP$(u, v) > 0$ *then* π_s *satisfies:*

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

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 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 SEP-values may be considered a *one-time* cost, whereas computing SSSP given the SEP-values represents the *marginal* cost of computing SSSP.[7]. For directed graphs, we show that our lower bound holds even if all 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 SEP function.

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

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 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 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 $\mathcal{SH}$, NORM$(x) \leq$ SEP(u, v). Now consider the recursive calls on u' and v' that caused u and v to be visited, say GENERALIZED-VISIT(u', I_u) and GENERALIZED-VISIT(v', I_v), where $|I_u| = |I_v| =$ NORM(x). If $d(s, v) \geq d(s, u) +$ SEP$(u, v) \geq d(s, u) +$ NORM(x) then $I_u \neq I_v$, implying 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 SEP-values" as "compute $\mathcal{SH}$" or "compute a good hierarchy" since $\mathcal{SH}$ is just a very compact structure for representing (approximate) SEP-values. In particular, if u, v are leaf-nodes in $\mathcal{SH}$ and $x = LCA(u, v)$ then SEP$(u, v) \in$ [NORM(x), 2NORM(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* $\text{SEP}(u, v)$ *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.

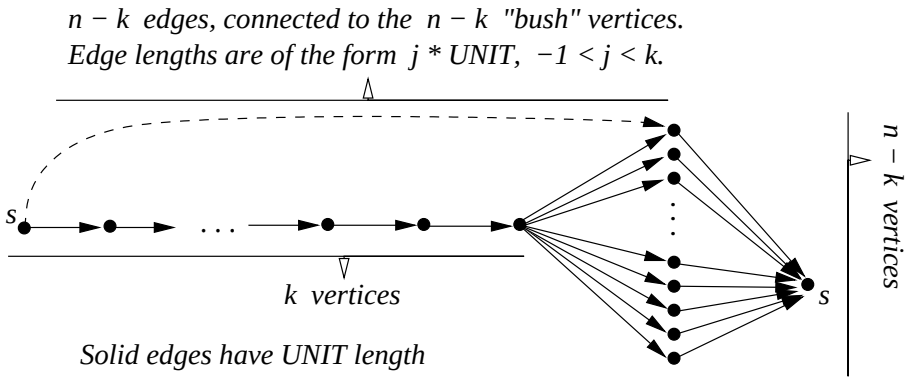

Figure 3.5: The "broom" graph.

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 $j \cdot UNIT$, where j, chosen below, is a non-negative integer. One may easily confirm that $\text{SEP}(u, v) = UNIT$ 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 $0 \leq j < k$. 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 $d(s, u) \leq d(s, v) + \text{UNIT} = d(s, v) + \text{SEP}(u, v)$, 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.

□

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 $q \geq 2$ disjoint *groups* and assign edge lengths based on group. Group i,

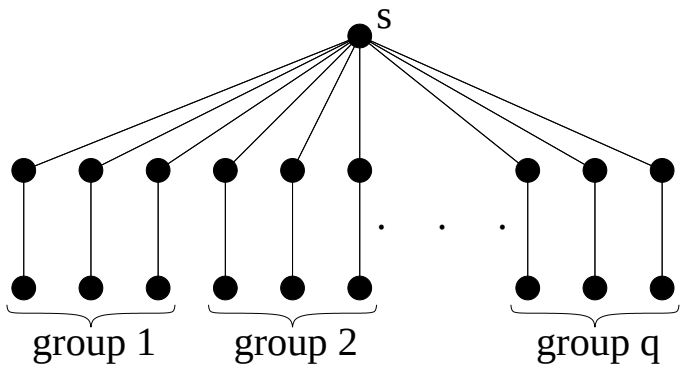

Figure 3.6: The minimum spanning tree of the graph

where $1 \leq i \leq q$, consists of exactly p/q pairs of vertices. Edges in group i have length $2^i \cdot \text{UNIT}$, 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 $2^{O(q)} \cdot \text{UNIT}$ 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 $d(s, v_i) = 2 \cdot 2^i \cdot \text{UNIT}$ and $\text{SEP}(v_i, v_j) = 2^{max\{i,j\}} \cdot \text{UNIT}$. By Property 2, $\pi_s(v_i)$ must be less than $\pi_s(v_j)$ if $d(s, v_i) + \text{SEP}(v_i, v_j) \leq d(s, v_j)$. This is equivalent to $(2 \cdot 2^i + 2^j) \cdot \text{UNIT} \leq 2 \cdot 2^j \cdot \text{UNIT}$, 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

$$\{\, d(s,y) \,\}_{s \in V,\, y \in \text{CHILD}(x)}$$

In other words, we fix an $\mathcal{SH}$-node x and look at the sibling correlations among nodes in 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 CHILD(x) represents the children of x in $\mathcal{SH}$. For $y \in$ CHILD(x) we let $\Delta_x(u,y)$ denote the *relative distance* from u to y,

defined as:

$$\Delta_x(u,y) \stackrel{\text{def}}{=} d(u,y) - d(u,x)$$

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:

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

where

$$\epsilon_x \stackrel{\text{def}}{=} \frac{\text{NORM}(x)}{2}$$

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 $\frac{\text{DIAM}(x)}{\text{NORM}(x)} \geq \log n$, in $O(mn)$ time total.*

Lemma 14 *If $\hat{\Delta}_x$ is known for all $x \in \mathcal{SH}$ for which $\frac{\text{DIAM}(x)}{\text{NORM}(x)} \geq \log n$, 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 $\text{DIAM}(x)/\text{NORM}(x) \geq \log n$. 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 $\text{DIAM}(x) \leq \log n \cdot \text{NORM}(x)$ 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, $t_x \leq d(s, x) < t_x + \text{NORM}(x)$. According to Invariant 2, y belongs in bucket number

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

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

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

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 $O(m \log n + m \cdot \text{DEG}(x) + n \cdot \frac{\text{DIAM}(x)}{\text{NORM}(x)})$. 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 $\text{DIAM}(x)/\text{NORM}(x) \geq \log n$.

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

$$\delta_x(u,v) \stackrel{\textbf{def}}{=} \ell(u,v) + d(v,x) - d(u,x)$$

$$\hat{\delta}_x(u,v) \stackrel{\textbf{def}}{=} \left\lfloor \frac{\delta_x(u,v)}{\epsilon'_x} \right\rfloor \quad \text{or} \quad \infty \quad \text{if} \quad \delta_x(u,v) > \text{DIAM}(x)$$

$$\text{where } \epsilon'_x \stackrel{\textbf{def}}{=} \frac{\epsilon_x}{n} = \frac{\text{NORM}(x)}{2n}$$

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 $y \in \text{CHILD}(x)$, 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 $x \in \mathcal{SH}$, $y \in \text{CHILD}(x)$ and $u \in V$. Then*

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

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

$$d^{\delta_x}(u,y) = \min_{j,\, \langle u = u_1, \ldots, u_j \in V(y) \rangle} \left\{ \sum_{i=1}^{j-1} \delta_x(u_i, u_{i+1}) \right\} \quad (4.1)$$

$$= \min_{j,\, \langle u = u_1, \ldots, u_j \in V(y) \rangle} \left\{ \ell(\langle u_1, \ldots, u_j \rangle) + d(u_j, x) - d(u_1, x) \right\} \quad (4.2)$$

$$= d(u,y) - d(u,x) = \Delta_x(u,y) \quad (4.3)$$

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 $d(u,y) - d(u,x) \leq \textsc{diam}(x)$ 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 $\delta_x(e) > \textsc{diam}(x)$ (i.e., $\hat{\delta}_x(e) = \infty$) or $\epsilon'_x \cdot \hat{\delta}_x(e) \leq \delta_x(e) < \epsilon'_x \cdot (\hat{\delta}_x(e) + 1)$. 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), $d^{\delta_x}(u,y) \leq \textsc{diam}(x)$, implying that for $e \in P_{uy}$, $\hat{\delta}_x(e) \neq \infty$, and

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

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

$$d^{\hat{\delta}_x}(u,y) \leq \frac{d^{\delta_x}(u,y)}{\epsilon'_x} \leq \frac{\textsc{diam}(x)}{\epsilon'_x} \leq \frac{2n \cdot \textsc{diam}(x)}{\textsc{norm}(x)}$$

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 $\delta_x(u,v) \leq \text{DIAM}(x)$, which can be checked in constant time, then $\hat{\delta}_x(u,v)$ can be expressed as:

$$\hat{\delta}_x(u,v) = \max\{j \ : \ 2n \cdot d(u,x) + j \cdot \text{NORM}(x) \leq 2n \cdot (\ell(u,v) + d(v,x))\}$$

which follows from the definition of $\hat{\delta}_x$ and $\epsilon'_x = \text{NORM}(x)/2n$. 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 $O(\log \frac{\text{DIAM}(x)}{\epsilon'_x}) = O(\log n)$ time by first generating the values

$$\left\{\text{NORM}(x), 2\,\text{NORM}(x), 4\,\text{NORM}(x), \ldots, 2^{\left\lceil \log \frac{\text{DIAM}(x)}{\epsilon'_x} \right\rceil} \text{NORM}(x)\right\}$$

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* $O(m \cdot \text{DEG}(x) + n \cdot \frac{\text{DIAM}(x)}{\text{NORM}(x)})$ *time.*

Proof: Let $y \in \text{CHILD}(x)$ 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 $y \in \text{CHILD}(x)$. To save time we solve each of these $\text{DEG}(x)$ 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 $O(\#(\text{edge relaxations}) + \#(\text{buckets scanned})) = O(m \cdot \text{DEG}(x) + N) = O\left(m \cdot \text{DEG}(x) + n \cdot \frac{\text{DIAM}(x)}{\text{NORM}(x)}\right)$. The bound on N follows from Lemma 15(4).

□

Lemmas 16 and 17 prove that Steps 1 and 2 take $O(m \log n + m\,\text{DEG}(x) + n\,\frac{\text{DIAM}(x)}{\text{NORM}(x)})$ 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:

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

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:

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

where

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

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. *$\rho_x \cdot \hat{\gamma}_x(u,v) \in (-\text{DIAM}(x) - \rho_x,\ 2\text{DIAM}(x)] \cup \{\infty\}$. 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 $w_x(u) \in [d(u,x),\ d(u,x) + \text{DIAM}(x)]$. 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 $\ell(u,v) + d(v,x) - d(u,x) \in [0,\ \text{DIAM}(x)]$. Thus:

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

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 $\rho_x \cdot \hat{\gamma}_x(u,v) = \gamma_x(u,v) + (-\rho_x, 0] = (-\text{DIAM}(x) - \rho_x,\ 2\text{DIAM}(x)]$.

(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.

$$\log\left(\frac{3 \cdot \text{DIAM}(x)}{\rho_x}\right) = \log\left(\frac{12 \cdot \text{DEG}(x) \cdot \text{DIAM}(x)}{\text{NORM}(x)}\right) \leq \text{DEG}(x) + \frac{\text{DIAM}(x)}{\text{NORM}(x)} + O(1)$$

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.

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

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

$$\rho_x \cdot \hat{\Gamma}_x(u,y) \stackrel{\text{def}}{=} \Gamma_x(u,y) - [0, \rho_x \cdot \text{DEG}(x))$$

$\hat{\Gamma}_x$ 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,

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

□

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 $\hat{\Gamma}$-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 $\text{IN}(y)$), 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.

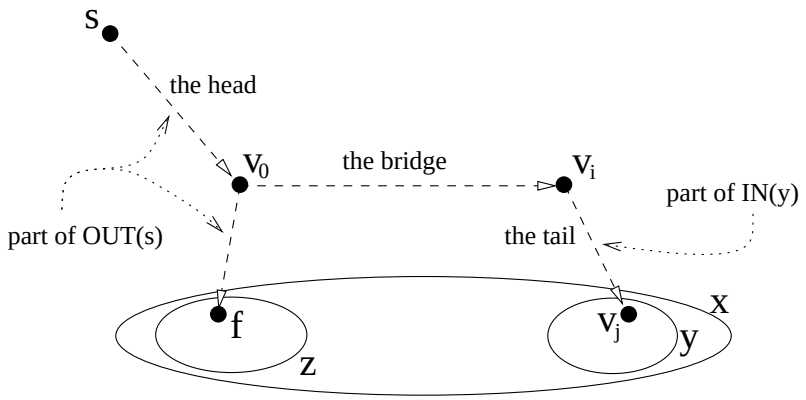

Figure 4.2: The path $\langle s, \ldots, v_j \rangle$, broken into a head $\langle s, \ldots, v_0 \rangle$, a bridge $\langle v_0, \ldots, v_i \rangle$, and a tail $\langle v_i, \ldots, v_j \rangle$.

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 $z \in \text{CHILD}(x)$ 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 $y \in \text{CHILD}(x)$, to be the set of paths of the form $\langle v_0, \ldots, v_i, \ldots, v_j \rangle$ that satisfy:*

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

We define the *integer* $\text{DIFF}(\mathcal{Q}_y)$ 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 $\text{DIFF}(\mathcal{Q}_y)$ in terms of $d(s, y)$; however, in general $\text{DIFF}(\mathcal{Q}_y)$ 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:

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

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

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 $[t_x,\ t_x + \text{NORM}(x))$. Let $[\beta,\ \beta + \text{NORM}(x))$ be the bucket in x's bucket array such that $t_x + \rho_x \cdot \text{DIFF}(\mathcal{Q}_y) \in [\beta,\ \beta + \text{NORM}(x))$.

2. If $D(y) \geq \beta$, put y in bucket $[\beta,\ \beta + \text{NORM}(x))$ and stop.

3. If $D(y) \geq \beta - \text{NORM}(x)$, put y in bucket $[\beta - \text{NORM}(x),\ \beta)$ 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 $d(s,x) \in [t_x, t_x + \text{NORM}(x))$. Lines 2 and 3 of the bucketing procedure guarantee that y is never bucketed in a higher bucket than $\left\lfloor \frac{D(y)-t_x}{\text{NORM}(x)} \right\rfloor$. To show that Invariant 4 is preserved, we need only prove that in Line 2, y is not bucketed before bucket $\left\lfloor \frac{d(s,y)-t_x}{\text{NORM}(x)} \right\rfloor - 2$. Lemma 22(2) states that $\rho_x \cdot \text{DIFF}(\mathcal{Q}_y) > d(s,y) - d(s,x) - \frac{1}{2}\text{NORM}(x)$, which implies that $\rho_x \cdot \text{DIFF}(\mathcal{Q}_y) > d(s,y) - t_x - \frac{3}{2}\text{NORM}(x)$. So bucketing y according to $\rho_x \cdot \text{DIFF}(\mathcal{Q}_y)$ can put it at most $\lceil \frac{3}{2} \rceil = 2$ buckets before bucket $\left\lfloor \frac{d(s,y)-t_x}{\text{NORM}(x)} \right\rfloor$, 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 $\rho_x \cdot \text{DIFF}(\mathcal{Q}_y) < d(s,y) - t_x + \frac{1}{4}\text{NORM}(x)$. By choice of β, we have $\beta - t_x \leq \rho_x \cdot \text{DIFF}(\mathcal{Q}_y)$, which implies that $\beta - \frac{1}{4}\text{NORM}(x) < d(s,y) \leq D(y)$ 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 $O(\text{DEG}(x))$ 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 $\sum_{x \in \mathcal{SH}} O(\text{DEG}(x)) \cdot n$, 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 $\sum_x O(\text{DEG}(x)) \cdot n = O(n^2)$. 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 $y \in \text{CHILD}(x)$, 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 $P_1 \subseteq P_{sx} \subseteq \text{OUT}(s)$ and $P_3 \subseteq \text{IN}(y)$. 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) $|P_2| > \text{DEG}(x)$ or (b) $w_x(u)$ is not known, for some $u \in P_2$. Case (a) can only happen $n/\text{DEG}(x)$ times for y, because after the SSSP computation from source s, $\text{IN}(y)$ will have absorbed P_2 (and P_1 for that matter). Thus the total cost for (a) is $\sum_x O(\text{DEG}(x))^2 \cdot n/\text{DEG}(x) = O(n^2)$. 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