A Shortest Path Algorithm for Real-Weighted Undirected Graphs*

Seth Pettie† Vijaya Ramachandran‡

December 13, 2002

Abstract

We present a new algorithm for computing undirected shortest paths in the fundamental *comparison-addition model*. Due to the generality of the model, our algorithm works on *real-weighted* undirected graphs, rather than the integer-weighted graphs assumed by many recent shortest path algorithms. Our algorithm is actually a general scheme for computing shortest paths and, as special cases, implies new bounds on the single-source and all-pairs shortest path problems. In a near-linear time preprocessing phase our algorithm computes a compact structure that facilitates the computation of shortest paths. After the preprocessing phase our algorithm can compute single-source shortest paths in $O(m \log \alpha)$ time, where $\alpha = \alpha(m, n)$ is the slowly growing inverse-Ackermann function, and m and n are the number of edges and vertices, respectively. This immediately implies an $O(mn \log \alpha)$ undirected all-pairs shortest path algorithm; we also show that if the ratio of the maximum-to-minimum edge length is bounded by $n^{(\log n)^{O(1)}}$ then our algorithm solves single-source shortest paths in $O(m + n \log \log n)$ time. Both our single-source and all-pairs shortest path algorithms are theoretical improvements over Dijkstra's algorithm, which was the previous best for real-weighted sparse graphs. Our algorithm takes the hierarchy-based approach to shortest paths invented by Thorup.

1 Introduction

The problem of computing shortest paths is indisputably one of the most well-studied problems in computer science. It is somewhat surprising then, that in the setting of *real*-weighted graphs, many basic shortest path problems have seen little or no progress since the early work by Dijkstra, Bellman-Ford, Floyd-Warshall, and others [CLRS01]. For instance, no algorithm for computing single-source shortest paths (SSSP) in arbitrarily weighted graphs has yet to improve Bellman-Ford's $O(mn)$ time bound, where m and n are the number of edges and vertices, respectively. The fastest uniform all-pairs shortest path (APSP) algorithm for dense graphs [Tak92, F76] requires time $O(n^3\sqrt{\log\log n/\log n})$, which is just a slight improvement over the $O(n^3)$ bound of the Floyd-Warshall algorithm. Similarly, Dijkstra's $O(m + n \log n)$ time algorithm [Dij59, FT87] remains the best for computing SSSP on non-negatively weighted graphs, and until the recent algorithms of Pettie [Pet02a, Pet02b], Dijkstra's algorithm [Dij59, J77, FT87] was also the best for computing APSP on sparse graphs.

In order to improve these bounds most shortest path algorithms depend on a restricted type of input. There are algorithms for geometric inputs (see Mitchell's survey [M00]), planar graphs [F91, H+97, FR01], and graphs with randomly chosen edge weights [S73, FG85, MT87, KKP93, KS98, M01, G01]. In recent years there has also been a focus on computing approximate shortest paths — see Zwick's recent survey [Z01]. One common assumption is that the graph is *integer*-weighted, though structurally unrestricted, and that the machine model is able to manipulate the integer representation of weights. Shortest path algorithms

*This work was supported by Texas Advanced Research Program Grant 003658-0029-1999 and NSF Grant CCR-9988160. A preliminary version of this paper, titled *Computing shortest paths with comparisons and additions*, was presented at the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, January 2002, San Francisco, CA. Authors' address: Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712.

†Also supported by an MCD Graduate Fellowship. E-mail: seth@cs.utexas.edu.

‡E-mail: vlr@cs.utexas.edu.

based on scaling [G85b, GT89, G95] and fast matrix multiplication [S95, GM97, AGM97, Tak98, SZ99, Z02] have running times which depend on the magnitude of the integer edge weights, and therefore yield improved algorithms only for sufficiently small edge weights. In the case of the matrix multiplication based algorithms the critical threshold is rather low: even edge weights sublinear in n can be too large. Dijkstra's algorithm can be sped up in the integer weight model by using an integer-sorting algorithm, and if necessary, Thorup's reduction [Tho00] from monotone priority queues to sorting. The best bounds on integer sorting depend on a number of factors, including whether randomization, superlinear space, and multiplication or other non-AC^0 operations are allowed. For the case of RAMs restricted to AC^0 operations, Dijkstra's algorithm can be implemented in $O(m + n(\log n)^{\frac{1}{3}+\epsilon})$ time [Ram96, Ram97], or in $O(m \log\log n)$ time [A+98, Tho00], which is better for sparse graphs. Other algorithms can improve these bounds when the maximum edge weight is not too large [Ram97, CGS99, vEB+76] or if unit-time multiplication is assumed [HT02]. Thorup [Tho99] considered the restricted case of integer-weighted undirected graphs and showed that on an AC^0 RAM, shortest paths could be computed in linear time. Thorup invented what we call in this paper the *hierarchy-based* approach to shortest paths.

The techniques developed for integer-weighted graphs (scaling, integer matrix multiplication, integer sorting, and Thorup's hierarchy-based approach) *seem* to depend crucially on the graph being integer-weighted. This state of affairs is not unique to the shortest path problem. In the weighted matching [G85b, GT89] and maximum flow problems [GR98], for instance, the best algorithms for real and integer-weighted graphs have running times differing by up to a *polynomial* factor. Therefore, it is of great interest whether an integer-based approach is inherently so, or whether it can yield a faster algorithm for general, real-weighted inputs. In this paper we generalize Thorup's hierarchy-based approach to the comparison-addition model (see Section 2.1), and as a corollary, to real-weighted input graphs. One immediate result is a faster undirected shortest path algorithm. However, we are also optimistic that our techniques could be useful in narrowing the integer/real gap in other optimization problems. Before stating our results we give an overview of the hierarchy-based approach and discuss the recent hierarchy-based shortest path algorithms [Tho99, Hag00, Pet02a, Pet02b].

Hierarchy-based algorithms should be thought of as preprocessing schemes for answering SSSP queries in non-negatively weighted graphs. The idea is to compute a small, non-source-specific structure that can be used to compute single-source shortest paths efficiently, for any given source. We measure the running time of a hierarchy-based algorithm with two quantities: $\mathcal{P}$, the worst case preprocessing cost on the given graph, and $\mathcal{M}$, the marginal cost of one SSSP computation after preprocessing. Therefore, solving the s-sources shortest path problem requires $s \cdot \mathcal{M} + \mathcal{P}$ time. If $s = n$, that is, we are solving APSP, then for all known hierarchy algorithms the $\mathcal{P}$ term becomes negligible. However, the $\mathcal{P}$ term may be dominant (in either the asymptotic or real-world sense) for smaller values of s. In Thorup's original algorithm [Tho99] $\mathcal{P}$ and $\mathcal{M}$ are both $O(m)$; recall that his algorithm works on integer-weighted undirected graphs. Hagerup [Hag00] adapted Thorup's algorithm to integer-weighted *directed* graphs, incurring a slight loss of efficiency in the process. In [Hag00], $\mathcal{P} = O(\min\{m \log\log C, m \log n\})$[1], where C is the maximum edge weight, and $\mathcal{M} = O(m + n \log\log n)$. If we isolate our attention to the marginal complexity $\mathcal{M}$, one can see that Hagerup's algorithm improves on the best SSSP algorithms for integer weighted graphs [A+98, Tho00, Ram96, Ram97]. After the initial publication of our results [PR02a], Pettie [Pet02a, Pet02b] gave an adaptation of the hierarchy-based approach to *real*-weighted directed graphs. The main result of [Pet02a] is an APSP algorithm running in time $O(mn + n^2 \log\log n)$, which improved upon the $O(mn + n^2 \log n)$ bound derived from multiple runs of Dijkstra's algorithm [Dij59, J77, FT87]. (The result of [Pet02a] is stated in terms of the APSP problem because its preprocessing cost $\mathcal{P}$ is $O(mn)$, making it efficient only if s is very close to n.) In [Pet02b] the *non-uniform* complexity of APSP is considered; the main result of [PR02b] is an algorithm performing $O(mn \log\alpha(m, n))$ comparison and addition operations, where α is the incomprehensibly slow-growing inverse-Ackermann function. The bound of [Pet02b] is essentially optimal when $m = O(n)$ due to the trivial $\Omega(n^2)$ lower bound on APSP.

In this paper we give new bounds on computing *undirected* shortest paths in real-weighted graphs. For our algorithm, the preprocessing cost $\mathcal{P}$ is $O(\textsc{mst}(m, n) + \min\{n \log n, n \log\log r\})$, where $\textsc{mst}(m, n)$ is the complexity of the minimum spanning tree problem[2] and r is the ratio of the maximum-to-minimum

[1] Hagerup actually proved it $\mathcal{P} = O(\min\{m \log\log C, mn\})$; see [Pet02a] for the $O(m \log n)$ bound.

[2] One should interpret $\textsc{mst}(m, n)$ as either the complexity of the optimal MST algorithm of Pettie and Ramachandran [PR02a], whose running time is between linear and $O(m\alpha(m, n))$ [Chaz00], or as the randomized complexity of the MST

edge weight. This bound on $\mathcal{P}$ is never worse than $O(m + n \log n)$, though if r is not excessively large, say less than $n^{(\log n)^{O(1)}}$, $\mathcal{P}$ is $O(m + n \log\log n)$. We show that the marginal cost $\mathcal{M}$ of our algorithm is asymptotically equivalent to SPLIT-FINDMIN(m, n), which is the decision-tree complexity of a certain data structuring problem of the same name. It was known that SPLIT-FINDMIN$(m, n) = O(m\alpha(m, n))$ [G85]; we improve this bound to $O(m \log \alpha(m, n))$. Therefore, the marginal cost of our algorithm is essentially (but perhaps not precisely) linear. Theorem 1.1 gives our general result and Corollaries 1.1 and 1.2 relate it to the canonical APSP and SSSP problems, respectively.

Theorem 1.1 *Let $\mathcal{P} =$ MST$(m, n) + \min\{n \log n, n \log\log r\}$, where m and n are the number of edges and vertices, r bounds the ratio of any two edge lengths, and* MST(m, n) *is the cost of computing the graph's minimum spanning tree. In $\Theta(\mathcal{P})$ time a $\Theta(n)$-space structure can be built which allows us to compute undirected single-source shortest paths on the real-weighted graph in $\Theta($*SPLIT-FINDMIN$(m, n))$ *time, where* SPLIT-FINDMIN$(m, n) = O(m \log \alpha(m, n))$ *represents the decision-tree complexity of the split-findmin problem and α is the inverse-Ackermann function.*

Corollary 1.1 *The undirected all-pairs shortest paths problem can be solved on a real-weighted graph in $\Theta(n \cdot$* SPLIT-FINDMIN$(m, n)) = O(mn \log \alpha(m, n))$ *time.*

Corollary 1.2 *The undirected single source shortest path problem can be solved on a real-weighted graph in $\Theta($*SPLIT-FINDMIN$(m, n) +$ MST$(m, n) + \min\{n \log n, n \log\log r\}) = O(m\alpha(m, n) + \min\{n \log n, n \log\log r\})$ *time.*

The running time of our SSSP algorithm (Corollary 1.2) is rather unusual. It consists of three terms, where the first two are unknown (but bounded by $O(m\alpha(m, n))$) and the third depends on a non-standard parameter: the maximum ratio of any two edge lengths. A natural question is whether our SSSP algorithm can be substantially improved. In Section 6 we formally define the class of "hierarchy-based" SSSP algorithms and show that any comparison-based undirected SSSP algorithm in this class must take time $\Omega(m + \min\{n \log n, n \log\log r\})$. This implies that our SSSP algorithm is optimal for this class, up to an inverse-Ackermann factor, and that no hierarchy-based SSSP algorithm can improve on Dijkstra's algorithm (for r unbounded).

Pettie et al. [PRS02] implemented a simplified version of our algorithm. The marginal cost of the [PRS02] implementation is nearly linear, which is in line with our asymptotic analysis. Although it is a little slower than Dijkstra's algorithm in solving SSSP, it is faster in solving the s-sources shortest path problem, in some cases for s as small as 3. In many practical situations it is the s-sources problem, not SSSP, that needs to be solved. For instance, if the graph represents a physical network, such as a network of roads or computers, it is unlikely to change very often. Therefore, in these situations a nearly linear preprocessing cost is a small price to pay for more efficient shortest path computations.

1.1 An Overview

In Section 2 we define the SSSP and APSP problems and review the comparison-addition model and Dijkstra's algorithm [Dij59]. In Section 3 we generalize the hierarchy approach to real-weighted graphs and give a simple proof of its correctness. In Section 4 we propose two implementations of the general hierarchy-based algorithm, one for proving the asymptotic bounds of Theorem 1.1 and one which should be better in practice. The running times of our implementations depend heavily on having a *well balanced* hierarchy. In Section 5 we give an efficient method for constructing balanced hierarchies; it is based on a hierarchical clustering of the graph's minimum spanning tree. In Section 6 we prove a lower bound on the class of hierarchy-based undirected SSSP algorithms. In Section 7 we discuss avenues for further research.

2 Preliminaries

The input is a weighted, undirected graph $G = (V, E, \ell)$ where $V = V(G)$ and $E = E(G)$ are the sets of n vertices and m edges, resp., and $\ell : E \to \mathbb{R}$ assigns a real length to each edge. Since, in an undirected graph,

problem, which is known to be linear [KKT95, PR02c].

the existence of a negative-length edge creates a negative-length cycle, we assume that $\ell : E \to \mathbb{R}^+$ assigns only positive weights. We let $d(u, v)$ denote the distance from u to v, that is, the length of the shortest path from u to v. We may replace v by a subgraph, in which case $d(u, v)$ represents the distance from u to any vertex in the subgraph of v. The single-source shortest paths problem is to compute $d(u, \cdot)$, where the *source* u is fixed, and the all-pairs shortest path problem is to compute $d(u, v)$ for all vertex pairs (u, v).

2.1 The Comparison-Addition Model

We use the term *comparison-addition model* to mean any uniform model in which real numbers are only subject to comparison and addition operations. The term *comparison-addition complexity* refers to the number of comparison and addition operations, ignoring other computational costs. In the comparison-addition model we leave unspecified the machine model used for all data structuring tasks. Our results as stated hold when that machine model is a RAM. If instead we assume a pointer machine [Tar79], our algorithms slow down by at most an inverse-Ackermann factor. The only structure we use whose complexity changes between the RAM and pointer machine models is the split-findmin structure. On a pointer machine there are matching upper and lower bounds of $\Theta(m\alpha)$ [G85, L96], whereas on the RAM the complexity is somewhere between $\Omega(m)$ and $O(m \log \alpha)$ — see Appendix B.

In our algorithm we sometimes use subtraction on real numbers, an operation which is not directly available in the comparison-addition model. Lemma 2.1, given below, shows that simulating subtraction incurs at most a constant factor loss in efficiency.

Lemma 2.1 *C comparisons and A additions and subtractions can be simulated in the comparison-addition model with C comparisons and $2(A + C)$ additions.*

Proof: We represent each real $x_i = a_i - b_i$ as two reals a_i, b_i. An addition $x_i + x_j = (a_i + a_j) - (b_i + b_j)$ or a subtraction $x_i - x_j = (a_i + b_j) - (b_i + a_j)$ can be simulated with two actual additions. A comparison $x_i : x_j$ is equivalent to the comparison $a_i + b_j : b_i + a_j$, which involves two actual additions and a comparison.

□

At a key point in our algorithm we need to approximate the ratio of two numbers. Division is clearly not available for real numbers in the comparison-addition model, and with a little thought one can see that it cannot be simulated exactly. Lemma 2.2, given below, bounds the time to find certain *approximate* ratios in the comparison-addition model, which will be sufficient for our purposes.

Lemma 2.2 *Let $p_1, \ldots, p_k$ be real numbers, and suppose that p_1 and p_k are known to be the smallest and largest, respectively. We can find the set of integers $\{q_i\}$ such that $2^{q_i} \le \frac{p_i}{p_1} < 2^{q_i+1}$ in $\Theta(\log \frac{p_k}{p_1} + k \log\log \frac{p_k}{p_1})$ time.*

Proof: We generate the set $\mathcal{L} = \{p_1, 2 \cdot p_1, 4 \cdot p_1, \ldots, 2^{\lceil \log \frac{p_k}{p_1} \rceil} \cdot p_1\}$ with $\log \frac{p_k}{p_1}$ additions, then for each p_i we find q_i in $\log |\mathcal{L}| = O(\log\log \frac{p_k}{p_1})$ time with a binary search over $\mathcal{L}$.

□

In our algorithm the $\{p_i\}$ correspond to certain edge lengths, and $k = \Theta(n)$. Our need to approximate ratios, as in Lemma 2.2, is the source of the peculiar $n \log\log r$ term in the running time of Theorem 1.1. We note here that the bound stated in Lemma 2.2 is pessimistic in the following sense. If we randomly select the $\{p_i\}$ from a uniform distribution over some interval (or a Poisson or normal distribution), then the time to find approximate ratios can be reduced to $O(k)$ (w.h.p.) using a linear search rather than a binary search.

There are many lower bounds for shortest paths problems in the comparison-addition model though none are truly startling. Spira and Pan [SP75] showed that even if additions are free, $\Omega(n^2)$ comparisons are necessary to solve SSSP on the complete graph. Karger et al. [KKP93] proved that directed all-pairs shortest paths requires $\Omega(mn)$ comparisons if each summation corresponds to a path in the graph.[3] Kerr [K70] showed that any oblivious APSP algorithm performs $\Omega(n^3)$ comparisons, and Kolliopoulos and Stein [KS98] proved that any fixed sequence of edge relaxations solving SSSP must have length $\Omega(mn)$. By "fixed sequence" they mean one which depends only on m and n but not on the graph structure. Ahuja et al. [AM+90] observed that any implementation of Dijkstra's algorithm requires $\Omega(m + n \log n)$ comparison and

[3] However it is not true that all shortest path algorithms satisfy this condition. For example, our algorithm does not and neither do [F76, Tak92, Pet02a, Pet02b].

addition operations. Pettie [Pet02a] gave an $\Omega(m + \min\{n \log r, n \log n\})$ lower bound on computing directed SSSP with a "hierarchy-type" algorithm, where r bounds the ratio of any two edge lengths. In Section 6 we prove a lower bound of $\Omega(m + \min\{n \log\log r, n \log n\})$ on hierarchy-type algorithms for *undirected* SSSP. These last two lower bounds are essentially tight for hierarchy-type algorithms, on directed and undirected graphs respectively.

Graham et al. [GYY80] proved that the information-theoretic argument cannot prove a non-trivial $\omega(n^2)$ lower bound on the *comparison*-complexity of APSP, where additions are granted for free. In many reasonable models, however, additions are not free. The [GYY80] result does not rule out a lower bound of an information-theoretic flavor on the comparison-addition complexity of APSP. One can also see that no information theoretic argument can lower bound the comparison-complexity of SSSP.

2.2 Dijkstra's Algorithm

Our algorithm, and indeed all hierarchy-based shortest path algorithms, are best understood as circumventing the limitations of Dijkstra's algorithm. We give a brief description of Dijkstra's algorithm in order to illustrate its inherent limitations; also, the vocabulary used in hierarchy-based algorithms is drawn mostly from Dijkstra's algorithm.

For a vertex set $S \subseteq V(G)$, let $d_S(u, v)$ denote the distance from u to v in the subgraph induced by $S \cup \{v\}$. Dijkstra's algorithm maintains a *tentative* distance function $D(v)$ and a set of *visited* vertices S satisfying Invariant 2.1. Henceforth, s denotes the source vertex.

Invariant 2.1 *Let s be the source vertex and v be an arbitrary vertex.*

$$D(v) = \begin{cases} d(s,v) & \text{if } v \in S \\ d_S(s,v) & \text{if } v \notin S \end{cases}$$

Choosing $S = \emptyset$, $D(s) = 0$ and $D(v) = \infty$ for $v \neq s$ clearly satisfies the Invariant initially. Dijkstra's algorithm consists of repeating the following step n times: choose a vertex $v \in V(G) - S$ such that $D(v)$ is minimized; let $S := S \cup \{v\}$; finally, update tentative distances to restore Invariant 2.1. This last part involves *relaxing* each edge (v, w) by setting $D(w) = \min\{D(w), D(v) + \ell(v, w)\}$. Invariant 2.1 and the positive-weight assumption implies $D(v) = d(s, v)$ when v is selected. It is also simple to prove that relaxing outgoing edges of v restores Invariant 2.1.

The problem with Dijkstra's algorithm is that vertices are selected in increasing distance from the source, a task which is at least as hard as sorting n numbers. Maintaining Invariant 2.1, however, does not demand such a particular ordering. In fact, it can be seen that selecting *any* vertex $v \notin S$ for which $D(v) = d(s, v)$ will maintain Invariant 2.1. All hierarchy-type algorithms [Tho99, Hag00, Pet02a, Pet02c] (and even a heuristic algorithm [G01]) maintain Invariant 2.1 by generating a weaker certificate for $D(v) = d(s, v)$ than "$D(v)$ is minimal." Any such certificate must show that for all $u \notin S$, $D(u) + d(u, v) \geq D(v)$. For example, Dijkstra's algorithm presumes there are no negative length edges, hence $d(u, v) \geq 0$, and by choice of v ensures $D(u) \geq D(v)$; this is clearly a sufficient certificate. All hierarchy-type algorithms [Tho99, Hag00, Pet02a, Pet02b], ours included, precompute an implicit lower bound on $d(u, v)$ in order to produce such certificates.

3 The Hierarchy Approach and Its Correctness

In this section we generalize the hierarchy-based approach of [Tho99] to real-weighted graphs. Because the algorithm follows directly from its proof of correctness, we will actually give a kind of correctness proof first.

Below, $X \subseteq V(G)$ denotes any set of vertices and s always denotes the source vertex. Let I be a real interval. The notation X^I refers to the subset of X whose distance from the source lies in the interval I, i.e.,

$$X^I = \{\, v \in X \;:\; d(s,v) \in I \,\}$$

Definition 3.1 *A vertex set X is $(S, [a, b))$-*SAFE *if*

(i) $X^{[0,a)} \subseteq S$

(ii) *For* $v \in X^{[a,b)}$, $d_{S \cup X}(s,v) = d(s,v)$

In other words, if a subgraph is (S,I)-SAFE, we can determine the distances that lie in interval I without looking at parts of the graph outside the subgraph and S. Clearly, finding safe subgraphs has the potential to let us compute distances cheaply.

Definition 3.2 *A t-partition of X is a set $\{X_i\}_i$ such that the $\{X_i\}_i$ partition X, and for any edge $e = (v_1, v_2)$, where $v_1 \in X_i$, $v_2 \in X_j$ and $i \neq j$, $\ell(e) \geq t$.*

Note that a t-partition need not be maximal, that is, if $\{X_1, X_2, \ldots, X_k\}$ is a t-partition then $\{X_1 \cup X_2, X_3, \ldots, X_k\}$ is as well.

Lemma 3.1 *Suppose that X is $(S,[a,b))$-SAFE. Let $\{X_i\}_i$ be a t-partition of X and let S' be such that $S \cup X^{[a,\min\{a+t,b\})} \subseteq S'$. Then*

(i) *For X_i in the t-partition, X_i is $(S,[a,\min\{a+t,b\}))$-SAFE*

(ii) *X is $(S',[\min\{a+t,b\},b))$-SAFE*

Proof: We prove part (i) first. Let $v \in X_i^{[a,\min\{a+t,b\})}$ and suppose that the lemma is false, that $d(s,v) \neq d_{S \cup X_i}(s,v)$. From the assumed safeness of X we know that $d(s,v) = d_{S \cup X}(s,v)$. This means that the shortest path to v must pass through $X - (X_i \cup S)$. Let w be the last vertex in $X - (X_i \cup S)$ on the shortest s–v path. By Definition 3.2, the edge from w to X_i has length $\geq t$. Since $d(s,v) < \min\{a+t,b\}$, $d(s,w) < \min\{a+t,b\} - t \leq a$. Since, by Definition 3.1(i), $X^{[0,a)} \in S$, it must be that $w \in S$, which is a contradiction because we specifically selected w from $X - (X_i \cup S)$. Part (ii) claims that X is $(S',[\min\{a+t,b\},b))$-SAFE. Consider first Definition 3.1(i) regarding safeness. By the assumption that X is $(S,[a,b))$-SAFE we have $X^{[0,a)} \subseteq S$, and by definition of S' we have $S \cup X^{[a,\min\{a+t,b\})} \subseteq S'$, therefore $X^{[0,\min\{a+t,b\})} \subseteq S'$, satisfying Definition 3.1(i). Definition 3.1(ii) is easily satisfied. By the assumption that X is $(S,[a,b))$-SAFE we have that for $v \in X^{[a,b)}$, $d_{S \cup X}(s,v) = d(s,v)$; this implies the weaker statement that for $v \in X^{[\min\{a+t,b\},b)}$, $d_{S' \cup X}(s,v) = d_{S \cup X}(s,v) = d(s,v)$.

□

As Thorup noted [Tho99], Lemma 3.1 alone leads to a simple recursive procedure for computing SSSP; however it makes no guarantee as to efficiency. The input to the procedure is an (S,I)-SAFE subgraph X; its only task is to compute the set X^I, which it performs with recursive calls (corresponding to Lemma 3.1 (i) and (ii)) or directly if X consists of a single vertex. There are essentially three major obstacles to making this general algorithm efficient: bounding the number of recursive calls, bounding the *time* to decide what those recursive calls are, and computing good t-partitions. Thorup gave a simple way to choose the t-partitions in integer-weighted graphs so that the number of recursive calls is $O(n)$. However, if adapted directly to the comparison-addition model, the time to decide which calls to make becomes $\Omega(n \log n)$; it amounts to the problem of implementing a general priority queue. We reduce the overhead for deciding which recursive calls to make to linear by using a "well balanced" hierarchy and a specialized priority queue for exploiting this kind of balance. Our techniques rely heavily on the graph being undirected and do not seem to generalize to directed graphs in any way.

As in other hierarchy-type algorithms [Tho99, Hag00, Pet02a, Pet02b], we generalize the distance and tentative distance notation from Dijkstra's algorithm to include not just single vertices but sets of vertices. If X is a set of vertices (or associated with a set of vertices) then

$$D(X) \stackrel{\text{def}}{=} \min_{v \in X} D(v) \qquad \text{and} \qquad d(u,X) \stackrel{\text{def}}{=} \min_{v \in X} d(u,v) \tag{1}$$

The procedure GENERALIZED-VISIT, given in Figure 1, takes a vertex set X which is (S,I)-SAFE and computes the distances to all vertices in X^I, placing these vertices in the set S as their distances become known. We maintain Invariant 2.1 at all times. By Definition 3.1 we can compute the set X^I without looking at parts of the graph outside of $S \cup X$. If $X = \{v\}$ happens to contain a single vertex we can compute X^I directly: if $D(v) \in I$ then $X^I = \{v\}$, otherwise it's $\emptyset$. For the general case, Lemma 3.1 says that we can

compute X^I by first finding a t-partition χ of X, then computing X^I in phases. Let $I = I_1 \cup I_2 \cup \ldots \cup I_k$, where each subinterval is disjoint from the others and has width t, except perhaps I_k which may be a leftover interval of width less than t. Let $S_i = S \cup X^{I_1} \cup \ldots \cup X^{I_i}$ and let $S_0 = S$. By the assumption that X is (S, I)-SAFE and Lemma 3.1, each set in χ is (S_i, I_{i+1})-SAFE. Therefore, we can compute $S_1, S_2, \ldots, S_k = S \cup X^I$ with a series of recursive calls as follows. Assume that the current set of visited vertices is S_i. We determine $X^{I_{i+1}} = \bigcup_{Y \in \chi} Y^{I_{i+1}}$ with recursive calls of the form GENERALIZED-VISIT(Y, I_{i+1}), where $Y \in \chi$. When each recursive call completes, $Y^{I_{i+1}}$ is part of the current set of visited vertices, so if we make one recursive call for each $Y \in \chi$ such that $Y^{I_{i+1}} \neq \emptyset$, the resulting set of visited vertices will be $S_{i+1} = S_i \cup X^{I_{i+1}}$, as called for.

To start things off, we initialize the set S to be empty, set the D-values (tentative distances) in accordance with Invariant 2.1, and call GENERALIZED-VISIT$(V(G), [0, \infty))$. By the definition of safeness, $V(G)$ is clearly $(\emptyset, [0, \infty))$-SAFE. If GENERALIZED-VISIT works according to specification, when it completes $S = V(G)$ and Invariant 2.1 is satisfied, implying that $D(v) = d(s, v)$ for all vertices $v \in V(G)$.

GENERALIZED-VISIT$(X, [a, b))$

Input guarantee: X is $(S, [a, b))$-SAFE and Invariant 2.1 is satisfied

Output guarantee: Invariant 2.1 is satisfied and $S_{post} = S_{pre} \cup X^{[a,b)}$, where S_{pre} and S_{post} are the set S before and after the call.

1. If X contains one vertex, $X = \{v\}$, and $D(v) \in [a, b)$, then $D(v) = d_S(s, v) = d(s, v)$, where the first equality is by Invariant 2.1 and the second by the assumption that X is $(S, [a, b))$-SAFE. Let $S := S \cup \{v\}$. Relax all edges incident on v, restoring Invariant 2.1 and return.

2. Let $a' := a$
   While $a' < b$ and $X \not\subseteq S$
   Let $t > 0$ be any positive real
   Let $\chi = \{X_1, X_2, \ldots, X_k\}$ be an arbitrary t-partition of X
   Let $\chi' = \{X_i \in \chi : D(X_i) < \min\{a' + t, b\} \text{ and } X_i \not\subseteq S\}$
   For each $X_i \in \chi'$, GENERALIZED-VISIT$(X_i, [a', \min\{a' + t, b\}))$
   $a' := \min\{a' + t, b\}$

Figure 1: A generalized hierarchy-type algorithm for real-weighted graphs

Lemma 3.2 *If the input guarantees of* GENERALIZED-VISIT *are met, then after a call to* GENERALIZED-VISIT(X, I)*, Invariant 2.1 remains satisfied and X^I is a subset of the visited vertices S.*

Proof Sketch: The base case, when X is a single vertex, is simple to handle. Turning to the general case, we prove that each time the while statement is examined in Step 2, X is $(S, [a', b))$ safe for the *current* value of S and a'; in what follows we will treat S as a variable, not a specific vertex set. The first time through the while-loop in Step 2, it follows from the input guarantee to GENERALIZED-VISIT that X is $(S, [a', b))$-SAFE. Similarly, the input guarantee for all recursive calls holds by Lemma 3.1. However, to show that X is $(S, [a', b))$-SAFE at the assignment $a' := \min\{a' + t, b\}$, by Definition 3.1 we must show $X^{[0, \min\{a'+t, b\})} \subseteq S$. We assume inductively that the output guarantee of any recursive call to GENERALIZED-VISIT is fullfilled, that is, upon the completion of GENERALIZED-VISIT$(X_i, [a', \min\{a' + t, b\}))$, S includes the set $X_i^{[a', \min\{a'+t, b\})}$. Each time through the while-loop in Step 2 GENERALIZED-VISIT makes recursive calls to all $Y \in \chi'$. To complete the proof we must show that for $Y \in (\chi - \chi')$, $Y^{[a', \min\{a'+t, b\})} - S = \emptyset$. If $Y \in (\chi - \chi')$ it was because $D(Y) \geq \min\{a' + t, b\}$ or because $Y \subseteq S$, both of which clearly imply $Y^{[a', \min\{a'+t, b\})} - S = \emptyset$. The output guarantee for GENERALIZED-VISIT is clearly satisfied if Step 1 is executed; if Step 2 is executed, then when the while loop finishes X is either $(S, [b, b))$-SAFE or $X \subseteq S$, both implying $X^{[0,b)} \in S$.

□

GENERALIZED-VISIT can be simplified in a few minor ways. It can be seen that in Step 1 we do not need to check whether $D(v) \in [a, b)$; the recursive call would not have taken place were this not the case. In Step 2 the final line can be shortened to $a' := a' + t$. However, we cannot change all occurrences of $\min\{a' + t, b\}$ to $a' + t$ because this is crucial to the procedure's correctness. It is not assumed (nor can it be guaranteed) that t divides $(b - a)$ so the procedure must be prepared to deal with fractional intervals of width less than t. In Section 4 we show that for a *proper* hierarchy this fractional interval problem does not arise.

4 Efficient Implementations of GENERALIZED-VISIT

We propose two implementations of the GENERALIZED-VISIT algorithm, called VISIT-A and VISIT-B. The time bound claimed in Theorem 1.1 is proved by analyzing VISIT-A, given later in this section. Although VISIT-A is asymptotically fast it seems too impractical for a real-world implementation. In Section 4.5 we give the VISIT-B implementation of GENERALIZED-VISIT which is simpler than VISIT-A and uses fewer specialized data structures. The asymptotic running time of VISIT-B is just a little slower than that of VISIT-A.

VISIT-A and VISIT-B differ from GENERALIZED-VISIT in their input/output specification only slightly. Rather than accepting a set of vertices, as GENERALIZED-VISIT does, our implementations (like [Tho99, Hag00, Pet02a, Pet02b]) accept a *hierarchy node* x, which represents a set of vertices. Both of our implementations work correctly for any *proper* hierarchy $\mathcal{H}$, defined below. We prove bounds on their running times as a function of m, n, and a certain function of $\mathcal{H}$ (which is different for VISIT-A and VISIT-B). In order to compute SSSP in near-linear time the proper hierarchy $\mathcal{H}$ must in addition satisfy certain *balance* conditions, which are the same for VISIT-A and VISIT-B. In section 5 we give the requisite properties of a balanced hierarchy and show how to construct a balanced proper hierarchy in $O(\text{MST}(m, n) + \min\{n \log n, n \log\log r\})$ time. Definition 4.1, given below, describes exactly what is meant by hierarchy and proper hierarchy.

Definition 4.1 *A* hierarchy *is a rooted tree whose leaf nodes correspond to graph vertices. If x is a hierarchy node, $p(x)$ is its parent,* $\text{DEG}(x)$ *is the number of children of x, $V(x)$ is the set of descendant leaves (or the equivalent graph vertices), and* $\text{DIAM}(x)$ *is an upper bound on the diameter of $V(x)$ (where the diameter of $V(x)$ is defined to be* $\max_{u,v \in V(x)} d(u, v)$*). Each node x is given a value* $\text{NORM}(x)$*. A hierarchy is* proper *if the following hold:*

1. $\text{NORM}(x) \leq \text{NORM}(p(x))$
2. *Either* $\text{NORM}(p(x))/\text{NORM}(x)$ *is an integer or* $\text{DIAM}(x) < \text{NORM}(p(x))$
3. $\text{DEG}(x) \neq 1$
4. *If $x_1, \ldots, x_{\text{DEG}(x)}$ are the children of x, then $\{V(x_i)\}_i$ is a* $\text{NORM}(x)$*-partition of $V(x)$. (Refer to Definition 3.2 for the meaning of "*$\text{NORM}(x)$*-partition.")*

Part (4) of Definition 4.1 is the crucial one for computing shortest paths. Part (3) guarantees that a proper hierarchy has $O(n)$ nodes. The second part of Part (2) is admittedly a little strange. It allows us to replace all occurances of $\min\{a + t, b\}$ in GENERALIZED-VISIT with just $a + t$, which greatly simplifies the analysis of our algorithms. Part (1) will be useful when bounding the total number of recursive calls to our algorithms.

4.1 VISIT-A

We now consider the VISIT-A procedure in Figure 2. VISIT-A is clearly very similar GENERALIZED-VISIT; indeed, the base cases of the two procedures (Step 1) are the same. Since GENERALIZED-VISIT is already proved correct, we only need to show that VISIT-A implements the GENERALIZED-VISIT procedure. In Step 2 of GENERALIZED-VISIT we let χ be any arbitrary t-partition of the subset of vertices given as input. In VISIT-A the input is a hierarchy node x and the associated vertex set is $V(x)$. We represent the t-partition of $V(x)$ (where $t = \text{NORM}(x)$) by the set of bucketed $\mathcal{H}$-nodes $\{x_i\}_i$ (see Step 2), where the sets $\{V(x_i)\}_i$

VISIT-A$(x, [a, b))$

Input: x is a node in a proper hierarchy $\mathcal{H}$; $V(x)$ is $(S, [a, b))$-SAFE and Invariant 2.1 is satisfied

Output guarantee: Invariant 2.1 is satisfied and $S_{post} = S_{pre} \cup V(x)^{[a,b)}$, where S_{pre} and S_{post} are the set S before and after the call.

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 2.1, and return.

2. If VISIT-A$(x, \cdot)$ 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

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

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

   We initialize $a' := a_x$ and insert all the children of x in $\mathcal{H}$ into the bucket array.

   The Bucket Invariant: A node $y \in \mathcal{H}$ in x's bucket array appears (logically) in the bucket whose interval spans $D(y)$. If $\{x_i\}$ are the set of bucketed nodes, then $\{V(x_i)\}$ is a NORM(x)-partition of $V(x)$.

3. While $a' < b$ and $V(x) \not\subseteq S$
   While $\exists y$ in bucket $[a',\ a' + \text{NORM}(x))$ such that $\text{NORM}(y) = \text{NORM}(x)$
   Remove y from the bucket array
   Insert y's children in $\mathcal{H}$ in the bucket array
   For each y in bucket $[a', a' + \text{NORM}(x))$
   and each y such that $D(y) < a'$ and $V(y) \not\subseteq S$
   VISIT-A$(y, [a', a' + \text{NORM}(x)))$
   $a' := a' + \text{NORM}(x)$

Figure 2: The VISIT-A procedure.

partition $V(x)$. Clearly the $\{x_i\}_i$ are descendants of x. The set $\{x_i\}_i$ will begin as x's children though later on $\{x_i\}_i$ may contain a mixture of children of x, grandchildren of x, and so on.

Consider the inner while-loop in Step 3. Assuming inductively that the bucketed $\mathcal{H}$-nodes represent a $\textsc{norm}(x)$-partition of $V(x)$, if y is a bucketed node and $\textsc{norm}(y) = \textsc{norm}(x)$ then replacing y by its children in the bucket array produces a new $\textsc{norm}(x)$-partition. This follows from the definitions of t-partitions and proper hierarchies (Definitions 3.2 and 4.1). Since the bucketed nodes form a $\textsc{norm}(x)$-partition, one can easily see that the recursive calls in Visit-A Step 3 correspond to the recursive calls in Generalized-Visit. However, their interval arguments are different. We sketch below why this change does not affect correctness.

In Generalized-Visit the intervals passed to recursive calls are of the form $[a', \min\{a' + t, b\})$ whereas in Visit-A they are $[a', a' + t) = [a', a' + \textsc{norm}(x))$. We will argue why $a' + t = a' + \textsc{norm}(x)$ is never more than b. The main idea is to show that we are always in one of the three cases portrayed in Figure 3.

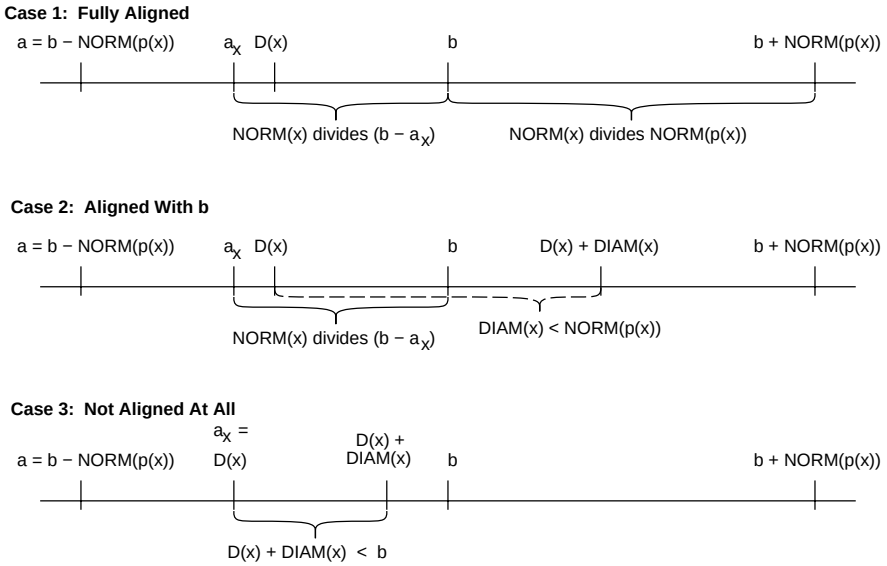

Figure 3: First observe that when a_x is initialized we have $D(x) \geq a_x \geq a$, as in the figure. If a_x is chosen such that $\textsc{norm}(x)$ divides $(b - a_x)$ then by Definition 4.1(2) either $\textsc{norm}(x)$ divides $\textsc{norm}(p(x))$ (which puts us in Case 1) or $\textsc{diam}(x) < \textsc{norm}(p(x))$ (putting us in Case 2), that is, $\textsc{norm}(x)$ does not divide $(b + \textsc{norm}(p(x)) - a_x)$ but it does not matter since we'll never reach $b + \textsc{norm}(p(x))$ anyway. If a_x is chosen so that $\textsc{norm}(x)$ does not divide $(b - a_x)$ then $a_x = D(x)$ and $D(x) + \textsc{diam}(x) < b$ (putting us in Case 3), meaning we will never reach b. Note that by the definition of $\textsc{diam}(x)$ (Definition 4.1) and Invariant 2.1, for any vertex in $u \in V(x)$ we have $d(s, u) \leq d(s, x) + \textsc{diam}(x) \leq D(x) + \textsc{diam}(x)$.

If $\textsc{norm}(x)$ divides $\textsc{norm}(p(x))$ and a_x is chosen in Step 2 so that $t = \textsc{norm}(x)$ divides $(b - a_x)$, then we can freely substitute the interval $[a', a' + t)$ for $[a', \min\{a' + t, b\})$ since they will be identical. Note that in our algorithm $(b - a) = \textsc{norm}(p(x))$.[4] The problems arise when $\textsc{norm}(x)$ does not divide either $\textsc{norm}(p(x))$ or $(b - a_x)$. In order to prove the correctness of Visit-A we must show that the input guarantee (regarding safe-ness) is satisfied for each recursive call. We consider two cases: when we are in the first recursive call to Visit-A$(x, \cdot)$ and any subsequent call. Suppose we are in the first recursive call to Visit-A$(x, \cdot)$. By our choice of a_x in Step 2, either $b = a_x + q \cdot \textsc{norm}(x)$ for some integer q, or $b > D(x) + \textsc{diam}(x) = a_x + \textsc{diam}(x)$. If it is the first case, each time the outer while-loop is entered we have $a' < b$, which, since q is integral, implies $\min\{a' + \textsc{norm}(x), b\} = a' + \textsc{norm}(x)$. Now consider the second case, where $b > D(x) + \textsc{diam}(x) = a_x + \textsc{diam}(x)$, and one of the recursive calls Visit-A$(y, [a', a' + \textsc{norm}(x)))$ made in Step 3. By Lemma 3.1, $V(y)$ is $(S, [a', \min\{a' + \textsc{norm}(x), b\}))$-safe, and it is actually $(S, [a', a' + \textsc{norm}(x)))$-safe as well because $b > D(x) + \textsc{diam}(x)$, implying $V(y)^{[b,\infty)} \subseteq V(x)^{[b,\infty)} = \emptyset$. (Recall from Definition 4.1 that for any

[4] Strictly speaking this does not hold for the initial call because in this case, $x = \textsc{root}(\mathcal{H})$ is the root of the hierarchy $\mathcal{H}$ and there is no such node $p(x)$. The argument goes through just fine if we let $p(\textsc{root}(\mathcal{H}))$ denote a dummy node such that $\textsc{norm}(p(\textsc{root}(\mathcal{H}))) = \infty$.

$u \in V(x)$, $\textsc{diam}(x)$ satisfies $d(s,x) \le d(s,u) \le d(s,x) + \textsc{diam}(x) \le D(x) + \textsc{diam}(x)$.) Now consider the recursive call $\textsc{Visit-A}(x,[a,b))$ which is *not* the first call to $\textsc{Visit-A}(x,\cdot)$. Then by Definition 4.1(2), either $(b-a) = \textsc{norm}(p(x))$ is a multiple of $\textsc{norm}(x)$ or $a + \textsc{diam}(x) < b$; these are identical to the two cases treated above.

There are two data structural problems that need to be solved in order to efficiently implement Visit-A. First, we need a way to compute the tentative distances of hierarchy nodes, i.e., the D-values as defined in Equation 1 in Section 3. For this problem we use an improved version of Gabow's split-findmin structure [G85]. The other problem is efficiently implementing the various bucket arrays, which we solve with a new structure called the *Bucket-Heap*. The specifications for these two structures are discussed below, in Sections 4.2 and 4.3, respectively. The interested reader can refer to Appendices A and B for details about our implementations of split-findmin and the Bucket-Heap, and proofs of their respective complexities.

4.2 The Split-Findmin Structure

The split-findmin structure operates on a collection of disjoint sequences, consisting, in total, of n items with associated keys. The idea is to maintain the smallest key in each sequence under the following operations.

split(x)	Split the sequence containing x into two sequences: the elements up to and including x and the rest.
decrease-key(x,κ)	Set key$(x) = \min\{\text{key}(x), \kappa\}$
findmin(x)	Return the element with minimum key in x's sequence.

Theorem 4.1, given below, establishes some new bounds on the problem which are just slightly better than Gabow's original data structure [G85]. Refer to Appendix B for a proof. Thorup [Tho99] gave a similar data structure for integer keys in the RAM model which runs in linear time. It relies on the RAM's ability to sort small sets of integers in linear time [FW93].

Theorem 4.1 *The split-findmin problem can be solved on a pointer machine in $O(n+m\alpha)$ time while making only $O(n+m\log\alpha)$ comparisons, where $\alpha = \alpha(m,n)$ is the inverse-Ackermann function. Alternatively, split-findmin can be solved on a RAM in time $\Theta(\textsc{split-findmin}(m,n))$, where $\textsc{split-findmin}(m,n) = O(n + m\log\alpha)$ is the (randomized) decision-tree complexity of the problem.*

We use the split-findmin structure to maintain D-values as follows. In the beginning there is one sequence consisting of the n leaves of $\mathcal{H}$ in an order consistent with some depth-first search traversal of $\mathcal{H}$. For any leaf v in $\mathcal{H}$ we maintain, by appropriate decrease-key operations, that key$(v) = D(v)$. During execution of Visit-A we will say an $\mathcal{H}$-node is *unresolved* if it lies in another node's bucket array but its tentative distance (D-value) is not yet finalized. The D-value of an $\mathcal{H}$-node becomes finalized, in the sense that it never decreases again, during Step 3 of Visit-A, either by being removed from some bucket array or passed, for the first time, to a recursive call of Visit-A. (It follows from Definition 3.1 and Invariant 2.1 that $D(y) = d(s,y)$ at the first recursive call to y.) One can verify a couple properties of the unresolved nodes. First, each unvisited leaf has exactly one unresolved ancestor. Second, to implement Visit-A we need only query the D-values of unresolved nodes. Therefore, we maintain that for each unresolved node y, there is some sequence in the split-findmin structure corresponding to $V(y)$, the descendants of y. Now suppose that a previously unresolved node y is *resolved* in Step 3 of Visit-A. The $\textsc{deg}(y)$ children of y will immediately become unresolved, so to maintain our correspondence between sequences and unresolved nodes, we perform $\textsc{deg}(y) - 1$ split operations on y's sequence so that the resulting subsequences correspond to y's children.

We remark that the split-findmin structure we use can be simplified slightly because we know in advance where the splits will occur. However, this knowledge does not seem to affect the asymptotic complexity of the problem. See Appendix B.

4.3 The Bucket-Heap

We now turn to the problem of efficiently implementing the bucket array used in Visit-A. Because of the information-theoretic bottleneck built into the comparison-addition model, we cannot always bucket nodes in constant time; each comparison extracts at most one bit of information, whereas properly bucketing a node in

x's bucket array requires us to extract up to $\log(\text{DIAM}(x)/\text{NORM}(x))$ bits of information. Thorup [Tho99] and Hagerup [Hag00] assume integer edge lengths and the RAM model and therefore do not face this limitation. We now give the specification for the *Bucket-Heap*, a structure which supports the bucketing operations of VISIT-A. This structure logically operates on a sequence of buckets; however, our implementation is actually a simulation of the logical structure. Lemma 4.1, proved in Appendix A, bounds the complexity of our implementation of the Bucket-Heap.

create(μ, δ)	Create a new Bucket-Heap whose buckets are associated with intervals $[\delta, \delta+\mu), [\delta+\mu, \delta+2\mu), [\delta+2\mu, \delta+3\mu), \ldots$ An item x lies in the bucket whose interval spans key(x). All buckets are initially *open*.
insert(x, κ)	Insert a new item x with key$(x) = \kappa$.
decrease-key(x, κ)	Set key$(x) = \min\{\text{key}(x), \kappa\}$. It is guaranteed that x is not moved to a closed bucket.
enumerate	Close the first open bucket and enumerate its contents.

Lemma 4.1 *Let Δ_x denote the number of buckets between the first open bucket at the time of x's insertion and the bucket from which x was enumerated. The Bucket-Heap can be implemented on a pointer machine to run in $O(m + n + b + \sum_x \log(\Delta_x + 1))$ time, where n, m, and b are the number of insertions, decrease-keys, and enumerates, respectively.*

When VISIT-A$(x, \cdot)$ is called for the first time, we initialize the Bucket-Heap at x with a call to create(NORM(x), a_x), followed by a number of insert operations for each of x's children, where the key of a child is its D-value. Here a_x is the beginning of the real interval represented by the bucket array, and NORM(x) the width of each bucket. Every time the D-value of a bucketed node decreases, which can easily be detected with the split-findmin structure, we perform a decrease-key on the corresponding item in the Bucket-Heap. We usually refer to buckets not by their cardinal number but by their associated real interval, e.g. bucket $[a_x, a_x + \text{NORM}(x))$.

4.4 Analysis of VISIT-A

In this section we bound the time required to compute SSSP with VISIT-A as a function of m, n, and the given hierarchy $\mathcal{H}$. We will see later that the dominant term in this running time corresponds to the split-findmin structure, whose complexity is no more than $O(m \log \alpha)$ but could turn out to be linear.

Lemma 4.2 *Let $\mathcal{H}$ be a proper hierarchy. Computing single-source shortest paths with* VISIT-A *on $\mathcal{H}$ takes time $O(\text{SPLIT-FINDMIN}(m, n) + \phi(\mathcal{H}))$ where* SPLIT-FINDMIN(m, n) *is the complexity of the split-findmin problem and*

$$\phi(\mathcal{H}) \;=\; \sum_{\substack{x \in \mathcal{H} \text{ such that} \\ \text{NORM}(x) \neq \text{NORM}(p(x))}} \frac{\text{DIAM}(x)}{\text{NORM}(x)} \;+\; \sum_{x \in \mathcal{H}} \log\Big(\frac{\text{DIAM}(p(x))}{\text{NORM}(p(x))} + 1\Big)$$

Proof: The SPLIT-FINDMIN(m, n) term represents the time to relax edges (in Step 1) and update the relevant D-values of $\mathcal{H}$-nodes, as described in Section 4.2. Except for the costs associated with updating D-values, the overall time of VISIT-A is linear in the number of recursive calls and the bucketing costs. The two terms of $\phi(\mathcal{H})$ represent these costs. Consider the number of calls to VISIT-A(x, I) for a particular $\mathcal{H}$-node x. According to Step 3 of VISIT-A there will be *zero* calls to x unless NORM$(x) \neq$ NORM$(p(x))$. If it is the case that NORM$(x) \neq$ NORM$(p(x))$, then for all recursive calls on x, the given interval I will have the same width: NORM(z) for some ancestor z of x. By Definition 4.1(1) NORM$(z) \geq$ NORM(x), therefore the number of such recursive calls on x is $\leq$ DIAM(x)/NORM$(x) + 2$; the extra 2 counts the first and last recursive calls, which may cover negligible parts of the interval $[d(s, x), d(s, x) + \text{DIAM}(x)]$. By Definition 4.1(3), $|\mathcal{H}| < 2n$ therefore the total number of recursive calls is bounded by $4n + \sum_x \text{DIAM}(x)/\text{NORM}(x)$, where the summation is over $\mathcal{H}$-nodes whose NORM-values differ from their parents NORM-values.

Now consider the bucketing costs of VISIT-A if implemented with the Bucket-Heap. According to Steps 2 and 3, a node y is bucketed either because VISIT-A$(p(y), \cdot)$ was called for the first time, or its parent $p(y)$ was removed from the first open bucket (of some bucket array), say bucket $[a, a + \text{NORM}(p(y)))$. In either case, this means that $d(s, p(y)) \in [a, a+\text{NORM}(p(y)))$ and that $d(s, y) \in [a, a+\text{NORM}(p(y))+\text{DIAM}(p(y)))$. To use the terminology of Lemma 4.1, $\Delta_y \leq \lceil \text{DIAM}(p(y))/\text{NORM}(p(y)) \rceil$ and the total bucketing costs would be $\#(\text{buckets scanned}) + \#(\text{insertions}) + \#(\text{dec-keys}) + \sum_x \log(\text{DIAM}(p(x))/\text{NORM}(p(x)) + 1)$, which is $O(\phi(\mathcal{H}) + m + n)$.

□

In Section 5 we give a method for constructing a proper hierarchy $\mathcal{H}$ such that $\phi(\mathcal{H}) = O(n)$. This bound together with Lemma 4.2 shows that we can compute SSSP in $O(\text{SPLIT-FINDMIN}(m, n))$ time, given a suitable hierarchy. Asymptotically speaking, this bound is the best we are able to achieve. However, the promising experimental results of a simplified version of our algorithm [PRS02] have led us to design an alternate implementation of GENERALIZED-VISIT that is both theoretically fast and easier to code.

4.5 A Practical Implementation of GENERALIZED-VISIT

In this Section we present another implementation of GENERALIZED-VISIT called VISIT-B. Although VISIT-B is a bit slower than VISIT-A in the asymptotic sense, it has other advantages. Unlike VISIT-A, VISIT-B treats all internal hierarchy nodes in the same way and is generally more streamlined. VISIT-B also works with any optimal off-the-shelf priority queue, such as a Fibonacci heap [FT87], unlike VISIT-A, which relies on the somewhat specialized Bucket-Heap. We will prove later that the asymptotic running time of VISIT-B is $O(m + n\log^* n)$. Therefore, if $m/n = \Omega(\log^* n)$, both VISIT-A and VISIT-B run in optimal $O(m)$ time.

The pseudocode for VISIT-B is given in Figure 4.

VISIT-B$(x, [a, b))$

Input: x is a node in a proper hierarchy $\mathcal{H}$; $V(x)$ is $(S, [a, b))$-SAFE and Invariant 2.1 is satisfied

Output guarantee: Invariant 2.1 is satisfied and $S_{post} = S_{pre} \cup V(x)^{[a,b)}$, where S_{pre} and S_{post} are the set S before and after the call.

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 2.1, and return.
2. If VISIT-B$(x, \cdot)$ is being called for the first time, put x's children in $\mathcal{H}$ in a heap associated with x, where the key of a node is its D-value. Choose a_x as in VISIT-A and initialize $a' := a_x$ and $\chi := \emptyset$.
3. While $a' < b$ and either χ or x's heap is non-empty,
   While there exists a y in x's heap with $D(y) \in [a', a' + \text{NORM}(x))$
   Remove y from the heap
   $\chi := \chi \cup \{y\}$
   For each $y \in \chi$,
   VISIT-B$(y, [a', a' + \text{NORM}(x)))$
   If $V(y) \subseteq S$ then set $\chi := \chi - \{y\}$.
   $a' := a' + \text{NORM}(x)$

Figure 4: The VISIT-B procedure.

The proof of correctness for VISIT-B follows the same lines as VISIT-A. It is easy to establish that before the for-loop in Step 3 is executed, $\chi = \{y : p(y) = x, D(y) < a' + \text{NORM}(x), \text{ and } V(y) \not\subseteq S\}$, so VISIT-B is actually a more straightforward implementation of GENERALIZED-VISIT than VISIT-A. In VISIT-B the NORM(x)-partition for x corresponds to x's children, whereas in VISIT-A the partition begins with x's children but is decomposed progressively.

Lemma 4.3 *Let $\mathcal{H}$ be a proper hierarchy. Computing single-source shortest paths with* VISIT-B *on $\mathcal{H}$ takes time $O(\text{SPLIT-FINDMIN}(m,n) + \psi(\mathcal{H}))$ where $\text{SPLIT-FINDMIN}(m,n)$ is the complexity of the split-findmin problem and*

$$\psi(\mathcal{H}) = \sum_{x \in \mathcal{H}} \left(\frac{\text{DIAM}(x)}{\text{NORM}(x)} + \text{DEG}(x) \log \text{DEG}(x) \right)$$

Proof: The SPLIT-FINDMIN term plays the same role in VISIT-B as in VISIT-A. VISIT-B is different than VISIT-A in that it makes recursive calls on *all* hierarchy nodes, not just those with different NORM-values than their parents. Using the same argument as in Lemma 4.3 we can bound the number of recursive calls of the form $\text{VISIT-B}(x,\cdot)$ as $\text{DIAM}(x)/\text{NORM}(x) + 2$; this gives the first summation in $\psi(\mathcal{H})$. Assuming an optimal heap is used (for example, a Fibonacci heap [FT87]), all decrease-keys take $O(m)$ time and all deletions take $\sum_x \text{DEG}(x) \log \text{DEG}(x)$ time. The bound on deletions follows since each of the $\text{DEG}(x)$ children of x are inserted into and deleted from x's heap at most once.

□

In Section 5 we construct a hierarchy $\mathcal{H}$ such that $\psi(\mathcal{H}) = \Theta(n\log^* n)$, implying an overall bound on VISIT-B of $O(m + n\log^* n)$, since $\text{SPLIT-FINDMIN}(m,n) = O(m\alpha(m,n)) = O(m + n\log^* n)$. Even though $\psi(\mathcal{H}) = \Omega(n\log^* n)$ in the worst case, we are only able to construct very contrived graphs for which this lower bound is tight.

5 Efficient Construction of Balanced Hierarchies

In this section we construct a hierarchy that works well for both VISIT-A and VISIT-B. The construction procedure has three distinct phases. In Phase 1 we find the graph's minimum spanning tree, denoted M, and classify its edges by length. This classification immediately induces a coarse hierarchy, denoted $\mathcal{H}_0$, which is analogous to Thorup's [Tho99] "component hierarchy", which was defined for integer-weighted graphs. Although $\mathcal{H}_0$ is proper, using it to run VISIT-A or VISIT-B may result in a slow SSSP algorithm. In particular, $\phi(\mathcal{H}_0)$ and $\psi(\mathcal{H}_0)$ can easily be $\Theta(n \log n)$, giving no improvement over Dijkstra's algorithm. Phase 2 facilitates Phase 3, in which we produce a *refinement* of $\mathcal{H}_0$, called $\mathcal{H}$; this is the "well balanced" hierarchy we referred to earlier. The refined hierarchy $\mathcal{H}$ is constructed so as to minimize the $\phi(\mathcal{H})$ and $\psi(\mathcal{H})$ terms in the running times of VISIT-A and VISIT-B. In particular, $\phi(\mathcal{H})$ will be $O(n)$ and $\psi(\mathcal{H})$ will be $O(n\log^* n)$. Although $\mathcal{H}$ could be constructed directly from M (the graph's minimum spanning tree) we would not be able to prove the time bound of Theorem 1.1 using this method. The purpose of Phase 2 is to generate a collection of small auxiliary graphs that — loosely speaking — capture the structure and edge lengths of certain subtrees of the minimum spanning tree. Using the auxiliary graphs in lieu of M, we are able to construct $\mathcal{H}$ in Phase 3 in $O(n)$ time.

In Section 5.1 we define all the notation and properties used in Phases 1, 2, and 3 (Sections 5.2, 5.3, and 5.4, respectively). In Section 5.5 we prove that $\phi(\mathcal{H}) = O(n)$ and $\psi(\mathcal{H}) = O(n\log^* n)$.

5.1 Some Definitions and Properties

5.1.1 The Coarse Hierarchy

Our refined hierarchy $\mathcal{H}$ is derived from a *coarse hierarchy* $\mathcal{H}_0$, which is defined here and in Section 5.2. Although $\mathcal{H}_0$ is typically very simple to describe, the general definition of $\mathcal{H}_0$ is rather complicated since it must take into account certain extreme circumstances. Therefore, for this section we give an abstract definition of $\mathcal{H}_0$. $\mathcal{H}_0$ is defined with respect to an increasing sequence of NORM-values: $\text{NORM}_1, \text{NORM}_2, \ldots$, where all edge lengths are at least as large as NORM_1. (Typically $\text{NORM}_{i+1} = 2 \cdot \text{NORM}_i$; however, this is not true in general.) We will say an edge e is at level i if $\ell(e) \in [\text{NORM}_i, \text{NORM}_{i+1})$, or alternatively, we may write $\text{NORM}(e) = \text{NORM}_i$ to express that e is at level i. A level i subgraph is a maximal connected subgraph restricted to edges with level i or less, that is, with length strictly less than NORM_{i+1}. Therefore, the level zero subgraphs consist of single vertices. A level i node in $\mathcal{H}_0$ corresponds to a non-redundant level i subgraph, where a level i subgraph is redundant if it is also a level $i-1$ subgraph. This non-redundancy property guarantees that all non-leaf $\mathcal{H}_0$-nodes have at least two children. The ancestor relationship in

$\mathcal{H}_0$ should be clear: x is an ancestor of y if and only if the subgraph of y is contained in the subgraph of x, i.e. $V(y) \subseteq V(x)$. The leaves of $\mathcal{H}_0$ naturally correspond to graph vertices and the internal nodes to subgraphs. The coarse hierarchy $\mathcal{H}_0$ clearly satisfies Definition 4.1(Parts 1,3,4); however, we have to be careful in choosing the NORM-values if we want it to be a *proper* hierarchy, that is, for it to satisfy Definition 4.1(2) as well. Our method for choosing the NORM-values is deferred to Section 5.2.

5.1.2 The Minimum Spanning Tree

By the *cut property* of minimum spanning trees (see [CLRS01, PR02a]) the $\mathcal{H}_0$ w.r.t. G is identical to the $\mathcal{H}_0$ w.r.t. M, the minimum spanning tree of G. Therefore, the remainder of this section is mainly concerned with M, not the graph itself. If $X \subseteq V(G)$ is a set of vertices, we let $M(X)$ be the minimal connected subtree of M containing X. Notice that $M(X)$ can include vertices outside of X. Later on we will need M to be a rooted tree in order to talk coherently about a vertex's parent, ancestors, children, and so on. Assume that M is rooted at an arbitrary vertex. The notation $\text{ROOT}(M(X))$ refers to the root of the subtree $M(X)$.

5.1.3 Mass and Diameter

The MASS of a vertex set $X \subseteq V(G)$ is defined as

$$\text{MASS}(X) \stackrel{\text{def}}{=} \sum_{e \in E(M(X))} \ell(e)$$

Extending this notation, we let $M(x) = M(V(x))$ and $\text{MASS}(x) = \text{MASS}(V(x))$, where x is a node in *any* hierarchy. Since the MST path between two vertices in $M(x)$ is an upper bound on the shortest path between them, $\text{MASS}(x)$ is an upper bound on the diameter of $V(x)$. Recall from Definition 4.1 that $\text{DIAM}(x)$ denoted any upper bound on the diameter of $V(x)$; henceforth, we will freely substitute $\text{MASS}(x)$ for $\text{DIAM}(x)$.

5.1.4 Refinement of the Coarse Hierarchy

We will say $\mathcal{H}$ is a *refinement* of $\mathcal{H}_0$ if all nodes in $\mathcal{H}_0$ are also represented in $\mathcal{H}$. An equivalent definition, which provides us with better imagery, is that $\mathcal{H}$ is derived from $\mathcal{H}_0$ by *replacing* each node $x \in \mathcal{H}_0$ with a rooted sub-hierarchy $H(x)$, where the root of $H(x)$ corresponds to (and is also referred to as) x and the leaves of $H(x)$ correspond to the children of x in $\mathcal{H}_0$. Consider a refinement $\mathcal{H}$ of $\mathcal{H}_0$ where each internal node y in $H(x)$ satisfies $\text{DEG}(y) \neq 1$ and $\text{NORM}(y) = \text{NORM}(x)$. One can easily verify from Definitions 3.2 and 4.1 that if $\mathcal{H}_0$ is a proper hierarchy, so too is $\mathcal{H}$. Of course, in order for $\phi(\mathcal{H})$ and $\psi(\mathcal{H})$ to be linear or near-linear $H(x)$ must satisfy certain properties. In particular it must be sufficiently short and balanced. By balanced we mean that a node's mass should not be *too* much smaller than its parent's mass.

5.1.5 Lambda Values

We will use the λ-values, defined below, in order to quantify precisely our notion of balance.

$$\lambda_0 = 0, \quad \lambda_1 = 12 \quad \text{and} \quad \lambda_{q+1} = 2^{\lambda_q \cdot 2^{-q}}$$

Lemma 5.1 gives a lower bound on the growth of the λ-values; we give a short proof before moving on.

Lemma 5.1 $\min\{q : \lambda_q \geq n\} \leq 2\log^* n$

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

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

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

5.1.6 Ranks

Recall from Section 5.1.4 that our refined hierarchy $\mathcal{H}$ is derived from $\mathcal{H}_0$by replacing each node $x \in \mathcal{H}_0$ with a subhierarchy $H(x)$. We assign to all nodes in $H(x)$ a non-negative integer *rank*. The analysis of our construction would become very simple if for every rank j node y in $H(x)$, $\text{MASS}(y) = \lambda_j \cdot \text{NORM}(x)$. Although this is our ideal situation, the nature of our construction does not allow us to place any non-trivial lower or upper bounds on the mass of y. We will assign ranks in order to satisfy Property 5.1, given below, which ensures us a sufficiently good approximation to the ideal. It is mainly the internal nodes of $H(x)$ that can have sub-ideal ranks; we assign ranks to the leaves of $H(x)$ (representing children of x in $\mathcal{H}_0$) to be as close to the ideal as possible.

We should point out that the assignment of ranks is mostly for the purpose of analysis. Rank information is never stored explicitly in the hierarchy nodes, nor is rank information used, implicitly or explicitly, in the computation of shortest paths. We only refer to ranks in the construction of $\mathcal{H}$ and when analyzing their effect on the ϕ and ψ functions.

Property 5.1 *Let $x \in \mathcal{H}_0$ and $y, z \in H(x) \subseteq \mathcal{H}$.*

1. *If y is an internal node of $H(x)$ then $\text{NORM}(y) = \text{NORM}(x)$ and $\text{DEG}(y) > 1$.*
2. *If y is a leaf of $H(x)$ (i.e., a child of x in $\mathcal{H}_0$) then y has rank j, where j is maximal s.t. $\text{MASS}(y)/\text{NORM}(x) \geq \lambda_j$.*
3. *Let y be a child of a rank j node. We call y stunted if $\text{MASS}(y)/\text{NORM}(x) < \lambda_{j-1}/2$. Each node has at most one stunted child.*
4. *Let y be of rank j. The children of y can be divided into three sets: Y_1, Y_2, and a singleton $\{z\}$ such that $(\text{MASS}(Y_1) + \text{MASS}(Y_2))/\text{NORM}(x) < (2 + o(1)) \cdot \lambda_j$.*
5. *Let $\mathcal{X}$ be the nodes of $H(x)$ of some specific rank. Then $\sum_{y \in \mathcal{X}} \text{MASS}(y) \leq 2 \cdot \text{MASS}(x)$.*

Before moving on let us examine some features of Property 5.1. Part (1) is asserted to guarantee that $\mathcal{H}$ is proper. Part (2) shows how we set the rank of leaves of $H(x)$. Part (3) says that at most one child of any node is less than half its ideal mass. Part (4) is a little technical but basically says that for a rank j node y, although $\text{MASS}(y)$ may be huge the children of y can be divided into sets $Y_1, Y_2, \{z\}$ such that Y_1 and Y_2 are of reasonable mass — around $\lambda_j \cdot \text{NORM}(x)$. However, no bound is placed on the mass contributed by z. Part (5) says that if we restrict our attention to the nodes of a particular rank, their subgraphs do not overlap in too many places. To see how two subgraphs might overlap, consider $\{x_i\}$, the set of nodes of some rank in $H(x)$. By our construction it will always be the case that the vertex sets $\{V(x_i)\}$ are disjoint; however, this does not imply that the subtrees $\{M(x_i)\}$ are edge-disjoint because $M(x_i)$ can, in general, be much larger than $V(x_i)$.

We show in Section 5.5 that if $\mathcal{H}$ is a refinement of $\mathcal{H}_0$ and $\mathcal{H}$ satisfies Property 5.1, then $\phi(\mathcal{H}) = O(n)$ and $\psi(\mathcal{H}) = O(n \log^* n)$. Recall from Lemmas 4.2 and 4.3 that $\phi(\mathcal{H})$ and $\psi(\mathcal{H})$ are terms in the running times of VISIT-A and VISIT-B, respectively.

5.2 Phase 1: The MST and The Coarse Hierarchy

Pettie and Ramachandran [PR02a] recently gave an MST algorithm which runs in time proportional to the decision-tree complexity of the problem. As the complexity of MST is trivially $\Omega(m)$ and only known to be $O(m\alpha(m,n))$ [Chaz00], it is unknown whether this cost will dominate or be dominated by the $\text{SPLIT-FINDMIN}(m,n)$ term. (This issue is mainly of theoretical interest.) In the analysis we use $\text{MST}(m,n)$ to denote the cost of computing the MST. The term $\text{MST}(m,n)$ can be interpreted in several ways: as the decision-tree complexity of MST [PR02a], as the randomized complexity of MST, which is known to be linear [KKT95, PR02c], or even as the comparison-addition complexity of MST. It is unlikely that additions help; nonetheless, [PR02a] can be modified to run in the comparison-addition complexity of MST as well.

Recall from Section 5.1.1 that $\mathcal{H}_0$ was defined with respect to an arbitrary increasing sequence of NORM-values. We describe below exactly how the NORM-values are chosen, then prove that $\mathcal{H}_0$ is a proper hierarchy. Our method depends on how large r is, which is the ratio of the maximum-to-minimum edge length in the minimum spanning tree. If $r < 2^n$, which can easily be determined in $O(n)$ time, then the possible NORM-values are $\{\ell_{\min} \cdot 2^i \ : \ 0 \le i \le \log r + 1\}$, where $\ell_{\min}$ is the minimum edge length in the graph. If $r \ge 2^n$ then let $e_1, \ldots, e_{n-1}$ be the edges in M in non-decreasing order by length and let $J = \{1\} \cup \{j : \ell(e_j) > n \cdot \ell(e_{j-1})\}$. The possible NORM-values are then $\{\ell(e_j) \cdot 2^i \ : \ j \in J \text{ and } \ell(e_j) \cdot 2^i < \ell(e_{j+1})\}$

Under either definition, NORM_i is the ith largest NORM-value and for an edge $e \in E(M)$, $\text{NORM}(e) = \text{NORM}_i$ if $\ell(e) \in [\text{NORM}_i, \text{NORM}_{i+1})$. Notice that if no edge length falls within the interval $[\text{NORM}_i, \text{NORM}_{i+1})$ then NORM_i is an unused NORM-value. We only need to keep track of the NORM-values in use, of which there are no more than $n-1$.

Lemma 5.2 *The coarse hierarchy $\mathcal{H}_0$ is a proper hierarchy.*

Proof: As we observed before, parts (1), (3), and (4) of Definition 4.1 are satisfied for any monotonically increasing sequence of NORM-values. Definition 4.1(2) states that if x is a hierarchy node, either $\text{NORM}(p(x))/\text{NORM}(x)$ is an integer or $\text{DIAM}(x)/\text{NORM}(p(x)) < 1$. Suppose x is a hierarchy node and $\text{NORM}(p(x))/\text{NORM}(x)$ is not integral; then $\text{NORM}(x) = \ell(e_{j_1}) \cdot 2^{i_1}$ and $\text{NORM}(p(x)) = \ell(e_{j_2}) \cdot 2^{i_2}$, where $j_2 > j_1$. By our method for choosing the NORM-values, the lengths of all MST edges are either at least $\ell(e_{j_2})$ or less than $\ell(e_{j_2})/n$. Since edges in $M(x)$ have length less than $\ell(e_{j_2})$, and hence less than $\ell(e_{j_2})/n$, $\text{DIAM}(x) < (|V(x)| - 1) \cdot \ell(e_{j_2})/n < \ell(e_{j_2}) \le \text{NORM}(p(x))$.

□

Lemma 5.3 *We can compute the minimum spanning tree M, and $\text{NORM}(e)$ for all $e \in E(M)$, in $O(\text{MST}(m,n) + \min\{n \log\log r,\ n \log n\})$ time.*

Proof: $\text{MST}(m,n)$ represents the time to find M. If $r < 2^n$ then by Lemma 2.2 we can find $\text{NORM}(e)$ for all $e \in M$ in $O(\log r + n \log\log r) = O(n \log\log r)$ time. If $r \ge 2^n$ then $n \log\log r = \Omega(n \log n)$, so we simply sort the edges of M and determine the indices J in $O(n \log n)$ time. Suppose there are n_j edges e s.t. $\text{NORM}(e)$ is of the form $\ell(e_j) \cdot 2^i$. Since $\ell(e)/\ell(e_j) \le n^{n_j}$ we need only generate $n_j \log n$ values of the form $\ell(e_j) \cdot 2^i$. A list of the $\sum_j n_j \log n = n \log n$ possible NORM-values can easily be generated in sorted order. By merging this list with the list of MST edge lengths, we can determine $\text{NORM}(e)$ for all $e \in M$ in $O(n \log n)$ time.

□

Lemma 5.4, given below, will come in handy in bounding the running time of our preprocessing and single-source shortest paths algorithms. It says that the total normalized mass in $\mathcal{H}_0$ is linear in n. Variations of Lemma 5.4 are at the core of the hierarchy approach [Tho99, Hag00, Pet02a, Pet02b].

Lemma 5.4

$$\sum_{x \in \mathcal{H}_0} \frac{\text{MASS}(x)}{\text{NORM}(x)} < 4(n-1)$$

Proof: Recall that the notation $\text{NORM}(e) = \text{NORM}_i$ means $\ell(e) \in [\text{NORM}_i, \text{NORM}_{i+1})$ where NORM_i is the ith largest NORM-value. Observe that if $e \in M$ is an MST edge with $\text{NORM}(e) = \text{NORM}_i$, e can be included in $\text{MASS}(x)$ for no more than one x at levels $i, i+1, \ldots$ in $\mathcal{H}_0$. Also, it follows from our definition of the NORM-values that $\text{NORM}_{i+1}/\text{NORM}_i \ge 2$ and for any MST edge, $\ell(e)/\text{NORM}(e) < 2$. Therefore, we can bound the normalized mass in $\mathcal{H}_0$ as:

$$\begin{aligned}
\sum_{x \in \mathcal{H}_0} \frac{\text{MASS}(x)}{\text{NORM}(x)} &\le \sum_{\substack{e \in M \\ \text{NORM}(e) = \text{NORM}_i}} \sum_{j=i}^{\infty} \frac{\ell(e)}{\text{NORM}_j} \\
&\le \sum_{\substack{e \in M \\ \text{NORM}(e) = \text{NORM}_i}} \sum_{j=i}^{\infty} \frac{\ell(e)}{2^{j-i} \cdot \text{NORM}_i} \ < \ 4(n-1)
\end{aligned}$$

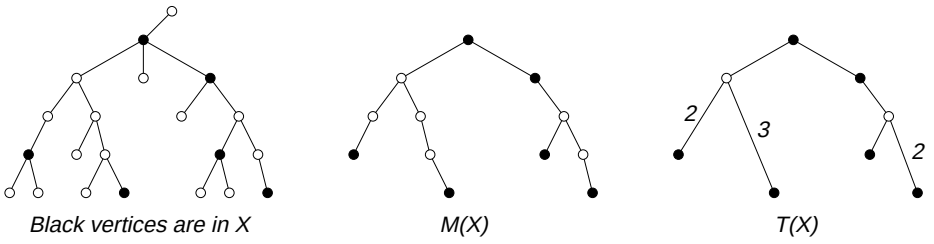

Figure 5: On the left is a subtree of M, the MST, where X is the set of blackened vertices. In the center is $M(X)$, the minimal subtree of M connecting X, and on the right is $T(X)$, derived from $M(X)$ by splicing out unblackened degree 2 nodes in $M(X)$ and adjusting edge lengths appropriately. Unless otherwise marked, all edges are of length 1.

□

Implicit in Lemma 5.4 is a simple accounting scheme where we treat mass, or more accurately normalized mass, as a currency equivalent to computational work. A hierarchy node x "owns" $\text{MASS}(x)/\text{NORM}(x)$ units of currency. If we can then show that the share of some computation relating to x is bounded by k times its currency, the total time for this computation is $O(kn)$, that is, of course, if all computation is attributable to some hierarchy node. Although simple, this accounting scheme is very powerful and can become quite involved [Pet02a, Pet02b].

5.3 Phase 2: Constructing $T(x)$ trees

Although it is possible to construct an $H(x)$ that satisfies Property 5.1 by working directly with the subtree $M(x)$, we are unable to efficiently compute $H(x)$ in this way. The problem is that we have time roughly proportional to the size of $H(x)$ to construct $H(x)$, whereas $M(x)$ could be significantly larger than $H(x)$. Our solution is to construct a *succinct tree* $T(x)$ that preserves the essential structure of $M(x)$ while having size roughly the same as $H(x)$.

For $X \subseteq V(G)$, let $T(X)$ be the subtree derived from $M(X)$ by *splicing* out all single-child vertices in $V(M(X)) - X$. That is, we replace each chain of vertices in $M(X)$, where only the end vertices are potentially in X, with a single edge; the length of this edge is the sum of its corresponding edge lengths in $M(X)$. Since there is a correspondence between vertices in $T(X)$ and M we will refer to $T(X)$ vertices by their names in M. Figure 5 gives examples of $M(X)$ and $T(X)$ trees, where X is the set of blackened vertices.

If $x \in \mathcal{H}_0$ and $\{x_j\}_j$ is the set of children of x, then let $T(x)$ be the tree $T(\{\text{ROOT}(M(x_j))\}_j)$; note that $\text{ROOT}(M(x))$ is included in $\{\text{ROOT}(M(x_j))\}_j$. Since only some of the edges of $M(x)$ are represented in $T(x)$, it is possible that the total length of $T(x)$ is significantly less than the total length of $M(x)$ (the MASS of $M(x)$); however, we will require that any subgraph of $T(x)$ have roughly the same mass as an equivalent subgraph in $M(x)$. In order to accomplish this we attribute certain amounts of mass to the *vertices* of $T(x)$ as follows. Suppose y is a child of x in $\mathcal{H}_0$ and $v = \text{ROOT}(y)$ is the corresponding root vertex in $T(x)$. We let $\text{MASS}(v) = \text{MASS}(y)$. All other vertices in $T(x)$ have zero mass. The mass of a subtree of $T(x)$ is then the sum of its edge lengths plus the collective mass of its vertices.

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

Lemma 5.5 *For $x \in \mathcal{H}_0$,*

1. $\text{DEG}(x) \leq |V(T(x))| < 2 \cdot \text{DEG}(x)$
2. *Let T_1 be a subtree of $T(x)$ and T_2 be the corresponding tree in $M(x)$. Then* $\text{MASS}(T_2) \leq \text{MASS}(T_1) \leq 2 \cdot \text{MASS}(T_2)$

Proof: Part (1) follows from two observations. First, $T(x)$ has no degree two vertices. Second, there are at most $\text{DEG}(x)$ leaves of $T(x)$ since each such leaf corresponds to a vertex $\text{ROOT}(M(y))$ for some child y of x

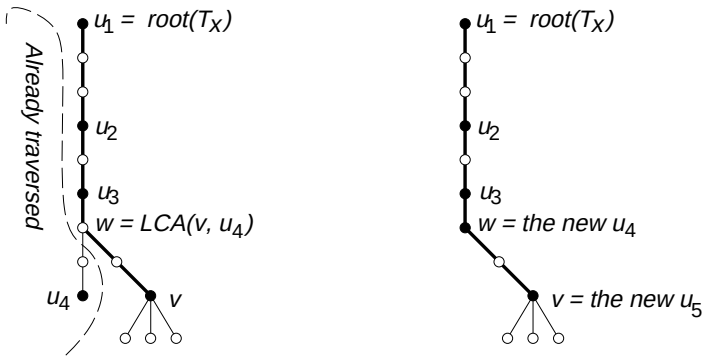

Figure 6: The blackened vertices represent those known to be in $T(x)$. The *active path* of the traversal is shown in bold edges. Before v is processed (left) the stack consists of $\langle u_1, u_2, u_3, u_4\rangle$, where u_4 is the last vertex in the traversal known to be in $T(x)$ and $w = LCA(v, u_4)$, which implies $w \in T(x)$. After processing v (right) the stack is set to $\langle u_1, u_2, u_3, w, v\rangle$ and w is blackened.

in $\mathcal{H}_0$. Part (2) follows since all mass in T_2 is represented in T_1, and each edge in T_2 contributes to the mass of at most one edge and one vertex in T_1.

□

We construct $T(x)$ with a kind of depth first traversal of the minimum spanning tree, using the procedure SUCCINCT-TREE, given in Figure 7. SUCCINCT-TREE focusses on some fixed $\mathcal{H}_0$-node x. We will explain how SUCCINCT-TREE works with the aid of the diagram in Figure 6. At every point in the traversal we maintain a stack of vertices $\langle u_1, \ldots, u_q\rangle$ consisting of all vertices known to be in $T(x)$ whose parents in $T(x)$ are not yet fixed. The stack has the following properties: u_i is ancestral to u_{i+1}, $\langle u_1, \ldots, u_{q-1}\rangle$ are on the *active path* of the traversal, and u_q is the last vertex known to be in $T(x)$ encountered in the traversal.

In Figure 6 the stack consists of $\langle u_1, \ldots, u_4\rangle$ where $\langle u_1, u_2, u_3\rangle$ are on the active path of the traversal, marked in bold edges. The preprocessing of v (before making recursive calls) is to do nothing if $v \notin \{\text{ROOT}(M(x_j))\}_j$. Otherwise, we update the stack to reflect our new knowledge about the edges and vertices of $T(x)$. The vertex $w = LCA(u_q, v) = LCA(u_4, v)$ must be in $T(x)$. There are three cases: either w is the ultimate or penultimate vertex in the stack (u_q or u_{q-1}), that is, we already know $w \in T(x)$, or w lies somewhere on the path between u_q and u_{q-1}. Figure 6 diagrams the third situation. Because no $T(x)$ vertices were encountered in the traversal between $u_q = u_4$ and v, there can be no new $T(x)$ vertices discovered on the path between u_q and w. Therefore, we can pop u_q off the stack, designating its parent in $T(x)$ to be w, and push w and v onto the stack. The other two situations, when $w = u_q$ or $w = u_{q-1}$, are simpler. If $w = u_q$ then we simply push v on the stack and if $w = u_{q-1}$ we pop u_q off the stack and push v on. Now consider the postprocessing of v (performed after all recursive calls), and let u_{q-1}, u_q be the last two vertices in the stack. Suppose that $v = u_{q-1}$. We cannot simply do nothing, because when the active path retracts there will be two stack vertices ($v = u_{q-1}$ and u_q) outside of the active path, contrary to the stack properties. However, because no $T(x)$ vertices were discovered between u_q and u_{q-1} we can safely say u_{q-1} is the parent of u_q in $T(x)$. So, to maintain the stack properties we pop off u_q and add the edge (u_q, u_{q-1}) to $T(x)$.

Lemma 5.6 *Given the MST and a list of its edges ordered by level, $\mathcal{H}_0$ and $\{T(x)\}_x$ can be constructed in $O(n)$ time.*

Proof: We construct $\mathcal{H}_0$ with a union-find structure and mark all vertices in M as roots of $M(x)$ for (one or more) $x \in \mathcal{H}_0$. It is easy to see that we can construct all $T(x)$ for $x \in \mathcal{H}_0$, with one tree traversal given in Figure 7. We simply maintain a different stack for each T_x under construction. So if v is the root of several $M(y_1), M(y_2), \ldots$, where $y_i \in \mathcal{H}_0$, we simply reexecute Lines 1–8 and 10–13 of SUCCINCT-TREEfor each of v's roles. Using a well-known union-find based least common ancestors algorithm [AHU76, Tar79b], we can compute the LCAs in Line 2 in $O(n\alpha(n))$ time, since the number of finds is linear in the number of nodes in $\mathcal{H}_0$. If we use the scheme of Buchsbaum et al. [B+98] instead, the cost of finding LCAs is linear;

```
SUCCINCT-TREE(v)
    The argument v is a vertex in the MST M.

    The stack for T(x) consists of vertices ⟨u_1, ..., u_q⟩, which
    are known to be in T(x) but whose parents in T(x) are not
    yet known. All but u_q are on the active path of the DFS
    traversal. Initially the stack for T(x) is empty.

1.  If v = ROOT(y) where y is a child of x in H_0, then
2.      Let w = LCA(v, u_q)
3.      If w ≠ u_q
4.          POP u_q off the stack
5.          Designate (u_q, w) an edge in T(x)
6.          If w ≠ u_{q-1}
7.              PUSH w on the stack
8.      PUSH v on the stack
9.  Call SUCCINCT-TREE on all the children on v
10. Let u_{q-1}, u_q refer to the last two elements in the current stack
11. If v = u_{q-1}
12.     POP u_q off the stack
13.     Designate (u_q, u_{q-1}) = (u_q, v) an edge in T(x)
```

Figure 7: A procedure for constructing $T(x)$, for a given $x \in \mathcal{H}_0$.

however, since this algorithm is offline (it does not handle LCA queries in the middle of a tree traversal, unlike [AHU76, Tar79b]) we would need to determine what the LCA queries are with an initial pass over the tree. Finally, we determine the length of an edge $(u, v) \in T(x)$ as follows. Let $\rho(u)$ be the distance from u to $\text{ROOT}(M)$ in M. Suppose v is ancestral to u in M; then the length of (u, v) is $\rho(u) - \rho(v)$. See Lemma 2.1 for a simulation of subtraction in the comparison-addition model.

□

5.4 Phase 3: Constructing the Refined Hierarchy

We show in this section how to construct an $H(x)$ from $T(x)$ which is consistent with Property 5.1.

The REFINE-HIERARCHY procedure, given as pseudocode in Figure 8, constructs $H(x)$ in a bottom-up fashion by traversing the tree $T(x)$. A call to REFINE-HIERARCHY(v), where $v \in T(x)$ will produce an array of sets $v[\cdot]$ whose elements are nodes in $H(x)$ that represent (collectively) the subtree of $T(x)$ rooted at v. The set $v[j]$ holds rank j nodes, which, taken together, are not yet massive enough to become a rank $j+1$ node. We extend the mass notation to sets $v[\cdot]$ as follows. Bear in mind that this mass is w.r.t. the tree $T(x)$, not $M(x)$. By Lemma 5.5(2), mass w.r.t. $T(x)$ is a good approximation to the mass of the equivalent subtree in $M(x)$.

$$\text{MASS}(v[j]) \quad = \quad \text{MASS}\left(\bigcup_{j' \le j} \bigcup_{y \in v[j']} V(y)\right)$$

The structure of REFINE-HIERARCHY is fairly simple. To begin with, we initialize $v[\cdot]$ be an array of empty sets. Then, if v is a root vertex of a child y of x in $\mathcal{H}_0$, we create a node representing y and put it in the proper set in $v[\cdot]$; which set receives y depends only on $\text{MASS}(y)$. Next we process the children of v. Each pass through the loop we pick an as yet unprocessed child w of v, recurse on w, producing sets $w[\cdot]$ representing the subtree rooted at w, then merge the sets $w[\cdot]$ into their counterparts in $v[\cdot]$. At this point, the mass of some sets may be beyond a critical threshold: the threshold for $v[j]$ is $\lambda_{j+1} \cdot \text{NORM}(x)$. In order to restore a quiescent state in the sets $v[\cdot]$ we perform *promotions* until no set's mass is above threshold.

Definition 5.1 *Promoting the set $v[j]$ involves removing the nodes from $v[j]$, making them the children of a new rank $j+1$ node, then placing this node in $v[j+1]$. There is one exception: if $|v[j]| = 1$ then to comply with Definition 4.1(3), we simply move the node from $v[j]$ to $v[j+1]$. Promoting the sets $v[0], v[1], \ldots, v[j]$ means promoting $v[0]$, then $v[1]$, up to $v[j]$, in that order.*

Suppose that after merging $w[\cdot]$ into $v[\cdot]$, j is maximal such that $\text{MASS}(v[j])$ is beyond its threshold of $\lambda_{j+1} \cdot \text{NORM}(x)$ (there need not be such a j). We promote the sets $v[0], \ldots, v[j]$ which has the effect of emptying the sets $v[0], \ldots, v[j]$ and adding a new node to $v[j+1]$ representing the nodes formerly in $v[0], \ldots, v[j]$. Lemma 5.7, given below, shows that we can compute the $H(x)$ trees in linear time.

Lemma 5.7 *Given $\{T(x)\}_x$, $\{H(x)\}_x$ can be constructed to satisfy Property 5.1 in $O(n)$ time.*

Proof: We first argue that REFINE-HIERARCHY produces a refinement $\mathcal{H}$ of $\mathcal{H}_0$ which satisfies Property 5.1. We then look at how to implement it in linear time.

Property 5.1(1) states that internal nodes in $H(x)$ must have NORM-values equal to that of x, which we satisfy by simply assigning them the proper NORM-values, and that no node of $H(x)$ have one child. By our treatment of one-element sets in the promotion procedure of Definition 5.1, it is simply impossible to create a one-child node in $H(x)$. Property 5.1(5) follows from Lemma 5.5(2) and the observation that the mass (in $T(x)$) represented by nodes of the same rank is disjoint. Now consider Property 5.1(3), regarding stunted nodes. We show that whenever a set $v[j]$ accepts a new node z, either $v[j]$ is immediately promoted, or z is not stunted, or the promotion of z into $v[j]$ represents the *last* promotion in the construction of $H(x)$. Consider the pattern of promotions in Line 9. We promote the sets $v[0], \ldots, v[j]$ in a cascading fashion: $v[0]$ to $v[1]$, $v[1]$ to $v[2]$ and so on. The only set accepting a new node which is not immediately promoted is $v[j+1]$, so in order to prove Property 5.1(3) we must show that the node derived from promoting $v[0], \ldots, v[j]$ is not stunted. By choice of j, $\text{MASS}(v[j]) \ge \lambda_{j+1} \cdot \text{NORM}(x)$, where mass is w.r.t. the tree $T(x)$. By Lemma 5.5(2) the mass of the equivalent tree in $M(x)$ is at least $\lambda_{j+1} \cdot \text{NORM}(x)/2$, which is exactly the threshold

REFINE-HIERARCHY(v) where v is a vertex in $T(x)$.

1. Initialize $v[j] := \emptyset$ for all j.
2. If $v = \text{ROOT}(M(y))$ for some child y of x in $\mathcal{H}_0$
3. Let j be maximal s.t. $\text{MASS}(y)/\text{NORM}(x) \geq \lambda_j$.
4. $v[j] := \{y\}$; (i.e., y is implicitly designated a rank j node)
5. For each child w of v in $T(x)$:
6. REFINE-HIERARCHY(w)
7. For all i, $v[i] := v[i] \cup w[i]$
8. Let j be maximal such that $\text{MASS}(v[j])/\text{NORM}(x) \geq \lambda_{j+1}$
9. *Promote* $v[0], \ldots, v[j]$ (see Definition 5.1)
10. If v is the root of $T(x)$, promote $v[0], v[1], \ldots$ until one node remains. (This final node is the root of $H(x)$.)

Figure 8: Constructing $H(x)$, for a given $x \in \mathcal{H}_0$.

for this node being stunted. Finally, consider Property 5.1(4). Before the merging step in Line 7 none of the sets in $v[\cdot]$ or $w[\cdot]$ is massive enough to be promoted. Let $v[\cdot]$ and $w[\cdot]$ denote the sets associated with v and w before the merging in step 7, and let $v'[\cdot]$ denote the set associated with v after the step 7. By the definition of MASS we have:

$$\text{MASS}(v'[j]) = \text{MASS}(v[j]) + \text{MASS}(w[j]) + \ell(v,w) < 2 \cdot \lambda_{j+1} \cdot \text{NORM}(x) + \ell(v,w)$$

Since (v,w) is an edge in $T(x)$ it can be arbitrarily large compared to NORM(x), meaning we cannot place any reasonable bound on MASS($v'[j]$) after the merging step. Let us consider how Property 5.1(4) is maintained. Suppose that $v'[j]$ is promoted in Lines 9 or 10 and let y be the resulting rank $j+1$ node. Using the terminology from Property 5.1(4), let $Y_1 = v[j], Y_2 = w[j]$ and let z be the node derived by promoting $v'[0], \ldots, v'[j-1]$. Since neither $v[j]$ nor $w[j]$ were sufficiently massive to be promoted before they were merged, we have $(\text{MASS}(Y_1) + \text{MASS}(Y_2))/\text{NORM}(x) < 2\lambda_{j+1}$. This is slightly stronger than what Property 5.1(4) calls for, which is the inequality $< (2 + o(1))\lambda_{j+1}$. We'll see why the $(2 + o(1))$ is needed below.

Suppose that we implemented REFINE-HIERARCHY in a straightforward manner. Let L be the (known) maximum possible index of any non-empty set $v[\cdot]$ during the course of REFINE-HIERARCHY. One can easily see that the initialization in Lines 1–4 take $O(L+1)$ time and that exclusive of recursive calls, each time through the for loop in Line 5 takes $O(L+1)$ amortized time. (The bound on Line 5 is *amortized* since promoting a set $v[j]$ takes worst case $O(|v[j]|+1)$ time but only constant amortized time. The cost of examining a node in $v[j]$ can be charged to the promotion that created it.) The only hidden costs in this procedure are updating the MASS of sets, which is done as follows. After the merging step in Line 7 we simply set $\text{MASS}(v[j]) := \text{MASS}(v[j]) + \ell(v,w) + \text{MASS}(w[j])$ for each $j \leq L$. Therefore the total cost of computing $H(x)$ from $T(x)$ is $O((L+1) \cdot |T(x)|)$. We can bound L as $L \leq 2\log^*(4n)$ as follows. The first node placed in any previously empty set is unstunted; therefore, by Lemma 5.1, the maximum non-empty set has rank at most $2\log^*(\text{MASS}(T(x))/\text{NORM}(x))$. By Lemma 5.5(2), and the construction of $\mathcal{H}_0$, $\text{MASS}(T(x)) \leq 2 \cdot \text{MASS}(M(x)) < 4(n-1) \cdot \text{NORM}(x)$.

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

$$t_1 = O\left(1 + \log^* \frac{\text{MASS}(v)}{\text{NORM}(x)}\right)$$

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

where $\text{MASS}(v[\cdot])$ is just the total mass represented by the $v[\cdot]$ sets. We only update the mass of the first $t_1 + t_2$ sets in $v[\cdot]$, and, as a rule, we update $v[j+1]$ half as often as $v[j]$. It is routine to show that REFINE-HIERARCHY will have a lower bound on the mass of $v[j]$ which is off by a $1+o(1)$ factor, where the $o(1)$ is a function of j.[5] This leads to the conspicuous $2 + o(1)$ in Property 5.1(4). To bound the cost of REFINE-HIERARCHY we model its computation as a binary tree: leaves represent the creation of nodes in Lines 1–4 and internal nodes represent the merging events in Line 7. The cost of a leaf f is $\log^*(\text{MASS}(f)/\text{NORM}(x))$ and the cost of an internal node f with children f_1 and f_2 is $1 + \log^*(\min\{\text{MASS}(f_1)/\text{NORM}(x), \text{MASS}(f_2)/\text{NORM}(x)\})$. We can think of charging the cost of f collectively to the mass in the subtree of f_1 or f_2, whichever is smaller. Therefore, no unit of mass can be charged for two nodes f and g if the total mass under f is within twice the total mass under g. The total cost is then:

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

The last equality follows since $|T(x)| = O(\text{MASS}(T(x))/\text{NORM}(x)) = O(\text{MASS}(M(x))/\text{NORM}(x))$. Summing over all $x \in \mathcal{H}_0$, the total cost of constructing $\{H(x)\}_{x \in \mathcal{H}_0}$ is, by Lemma 5.4, $O(n)$.

□

Lemma 5.8 *In $O(\text{MST}(m,n) + \min\{n \log\log r, n \log n\})$ time we can construct both the coarse hierarchy $\mathcal{H}_0$ and a refinement $\mathcal{H}$ of $\mathcal{H}_0$ satisfying Property 5.1.*

Proof: Follows from Lemmas 5.3, 5.6, 5.7.

□

5.5 Analysis

In this section we prove bounds on the running times of VISIT-A and VISIT-B, given an appropriate refined hierarchy, such as the one constructed in Section 5.4. Theorem 1.1 follows directly from Lemma 5.9, given below, and Lemma 5.8.

Lemma 5.9 *Let $\mathcal{H}$ be any refinement of $\mathcal{H}_0$ satisfying Property 5.1. Using $\mathcal{H}$,* VISIT-A *computes SSSP in $O(\text{SPLIT-FINDMIN}(m,n))$ time and* VISIT-B *computes SSSP in $O(m + n\log^* n)$ time.*

Proof: We prove that $\phi(\mathcal{H}) = O(n)$ and $\psi(\mathcal{H}) = O(n\log^* n)$. Together with Lemmas 4.2 and 4.3, this will complete the proof.

With the observation that $\text{MASS}(x)$ is an upper bound on the diameter of $V(x)$, we will substitute $\text{MASS}(x)$ for $\text{DIAM}(x)$ in the functions ϕ and ψ. By Lemma 5.4, the first sum in ϕ is $O(n)$. The first sum of $\psi(\mathcal{H})$ is much like in ϕ, except we sum over *all* nodes in $\mathcal{H}$, not just those nodes which also appear in $\mathcal{H}_0$. By Property 5.1(1,3,4) and Lemma 5.1, the maximum rank of any node in $H(x)$ is $2\log^*(\text{MASS}(x)/\text{NORM}(x)) \leq 2\log^* n$. By Property 5.1(5) the total mass of nodes of one rank in $H(x)$ is bounded by $2 \cdot \text{MASS}(x)$. Therefore, we can bound the first sum in $\psi(\mathcal{H})$ as $\sum_x \text{MASS}(x)/\text{NORM}(x) \leq 4\log^* n \cdot \sum_{x \in \mathcal{H}_0} \text{MASS}(x)/\text{NORM}(x)$, which is $O(n\log^* n)$ by Lemma 5.4.

We now turn to the second summations in $\phi(\mathcal{H})$ and $\psi(\mathcal{H})$, which can be written as $\sum_x \text{DEG}(x) \log(\text{MASS}(x)/\text{NORM}(x))$ and $\sum_x \text{DEG}(x) \log \text{DEG}(x)$, respectively. Since $\text{DEG}(x) \leq 1 + \text{MASS}(x)/\text{NORM}(x)$, any bound established on the first summation will extend to the second.

Let y be a rank j node. Using the terms from Property 5.1(4), let $\alpha = (\text{MASS}(Y_1) + \text{MASS}(Y_2))/\text{NORM}(y)$ and $\beta = \text{MASS}(y)/\text{NORM}(y) - \alpha$. Property 5.1(3,4) implies that $\alpha < (2 + o(1)) \cdot \lambda_j$ and that $\text{DEG}(y) \leq 2\alpha/\lambda_{j-1} + 2$, where the $+2$ represents the stunted child and the child z exempted from Property 5.1(4).

[5] The proof of this is somewhat tedious. Basically one shows that for $i < j$, the mass of $v[i]$ can be updated at most $2^{j-i} - 1$ times before the mass of $v[j]$ is updated. Since $2^{j-1} - 1 \cdot \lambda_i << \lambda_j$, our neglecting to update the mass of $v[j]$ causes only a negligible error.

$$\begin{aligned}
\text{DEG}(y)\log\frac{\text{MASS}(y)}{\text{NORM}(y)} &\leq \left(\frac{2\alpha}{\lambda_{j-1}}+2\right)\log(\alpha+\beta) \qquad \{\text{See explanations below}\}\\
&= O\left(\frac{\max\{\alpha\log(2\lambda_j),\beta\}}{\lambda_{j-1}}\right)\\
&= O\left(\frac{\alpha+\beta}{2^{j-1}}\right) \;=\; O\left(\frac{\text{MASS}(y)}{\text{NORM}(y)\cdot 2^{j-1}}\right)
\end{aligned}$$

The first line follows from our bound on $\text{DEG}(y)$ and the definition of α and β. The second line follows since $\alpha < (2+o(1))\lambda_j$, and $\alpha\log(\alpha+\beta) = O(\max\{\alpha\log\alpha,\beta\})$. The last line follows since $\log\lambda_j = \lambda_{j-1}/2^{j-1} > 1$. By the above bound and Property 5.1(5), $\sum_{y\in H(x)}\text{DEG}(y)\log(\text{MASS}(y)/\text{NORM}(y)) = O(\text{MASS}(x)/\text{NORM}(x))$. Therefore, by Lemma 5.4, the second summations in both $\phi(\mathcal{H})$ and $\psi(\mathcal{H})$ are bounded by $O(n)$.

□

6 Limits of Hierarchy-Type Algorithms

In this section we state a simple property (Property 6.1) of all hierarchy-type algorithms and give a lower bound on any undirected SSSP algorithm satisfying that property. The upshot is that our SSSP algorithm is optimal (up to an inverse-Ackermann factor) for a fairly large class of SSSP algorithms, which includes all hierarchy-type algorithms, variations on Dijkstra's algorithm, and even a recent heuristic SSSP algorithm [G01].

We will state Property 6.1 in terms of directed graphs. Let $\text{CYCLES}(u,v)$ denote the set of all cycles, including non-simple cycles, that pass through both u and v, and let $\text{SEP}(u,v) = \min_{C\in\text{CYCLES}(u,v)}\max_{e\in C}\ell(e)$. Note that in undirected graphs $\text{SEP}(u,v)$ corresponds exactly to the longest edge on the MST path between u and v.

Property 6.1 *An SSSP algorithm with the* hierarchy property *computes, aside from shortest paths, a permutation $\pi_s : V(G) \to V(G)$ such that for any vertices u, v, $d(s,u) \geq d(s,v) + \text{SEP}(u,v) \Longrightarrow \pi_s(u) > \pi_s(v)$, where s is the source and d the distance function.*

The permutation π_s corresponds to the order in which vertices are visited when the source is s. Property 6.1 says that π_s is loosely sorted by distance, but may invert pairs of vertices if their relative distance is less than their SEP-value. To see that our hierarchy-based algorithm satisfies Property 6.1, consider two vertices u and v. Let x be the least common ancestor of u and v in $\mathcal{H}$ and let u' and v' be the ancestors of u and v, respectively, which are children of x. By our construction of $\mathcal{H}$, $\text{NORM}(x) \leq \text{SEP}(u,v)$. If $d(s,u) \geq d(s,v) + \text{SEP}(u,v)$ then $d(s,u) \geq d(s,v) + \text{NORM}(x)$ and therefore the recursive calls on u' and v' which cause u and v to be visited are not passed the same interval argument, since both intervals have width $\text{NORM}(x)$. The recursive call on u' must, therefore, preceed the recursive call on v' and u must be visited before v.

Theorem 6.1 *Suppose that our computational model allows any set of functions from $\mathbb{R}^{O(1)} \to \mathbb{R}$ and comparison between two reals. Any single-source shortest path algorithm for real-weighted graphs satisfying Property 6.1 makes $\Omega(m + \min\{n\log\log r,\, n\log n\})$ operations in the worst case, where r is the ratio of the maximum to minimum edge length.*

Proof: Let $q = \log r + 1$ be an integer. Assume without loss of generality that $2q$ divides $n-1$. The MST of the input graph is as depicted in Figure 9. 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. All the vertices (except s) are divided into disjoint *groups*, where group i consists of exactly p/q randomly chosen pairs of vertices. There are exactly $p!/(p/q)!^q = q^{\Omega(p)}$ possible group arrangements. We will show that any algorithm satisfying Property 6.1 must be able to distinguish them.

We choose edge lengths as follows. All edges in group i have length 2^i. This includes edges from s to the group's top row and between the two rows. Other non-MST edges are chosen so that shortest paths from

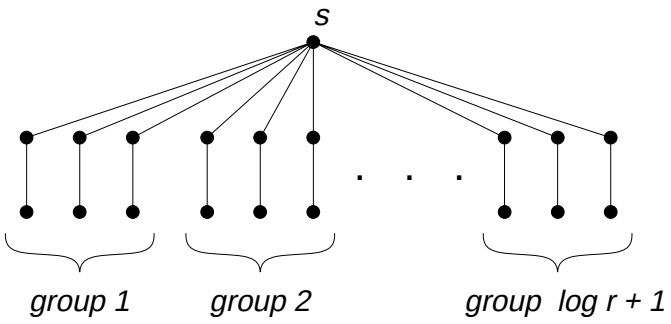

Figure 9: The minimum spanning tree of the graph

s correspond to paths in the MST. The ratio between the maximum and minimum edge length is at least $2^{\log r+1}/2^1 = r$, as called for. Let v_i denote any vertex in the bottom row of group i. Then $d(s, v_i) = 2 \cdot 2^i$ and $\text{SEP}(v_i, v_j) = 2^{max\{i,j\}}$. By Property 6.1, v_i must be visited before v_j if $d(s, v_i) + \text{SEP}(v_i, v_j) \leq d(s, v_j)$, which is true for $i < j$ since $2 \cdot 2^i + 2^j \leq 2 \cdot 2^j$. Therefore, any algorithm satisfying Property 6.1 must be prepared to visit vertices in $q^{\Omega(p)}$ distinct permutations and make at least $p \log q = \Omega(n \log\log r)$ comparisons in the worst case. It also must include every non-MST edge in at least one operation, which gives the lower bound.

□

Theorem 6.1 shows that our SSSP algorithm is optimal among hierarchy-type algorithms, to within a tiny inverse-Ackermann factor. A lower bound on *directed* SSSP algorithms satisfying Property 6.1 is given in [Pet02a]. Theorem 6.1 differs from that lower bound in two respects. First, the [Pet02a] bound is $\Omega(m + \min\{n \log r, n \log n\})$, which is $\Omega(m + n \log n)$ for even reasonably small values of r. Second, the [Pet02a] bound holds even if the algorithm is allowed to compute the SEP function (and sort the values) for free. Contrast this with our SSSP algorithm, where the main obstacle to achieving linear time is the need to sort the SEP-values.

7 Discussion

We have shown that with a near-linear time investment in preprocessing, SSSP queries can be answered in very close to linear time. Furthermore, among a natural class of SSSP algorithms captured by Property 1, our SSSP algorithm is optimal, aside from a tiny inverse-Ackermann factor. We can imagine several avenues for further research. The most interesting, in our opinion, is developing feasible alternatives to Property 1 which do not have intrinsic sorting bottlenecks. This approach could be termed algorithm design in reverse: first one defines a desirable property, then one looks for algorithms with that property. Another avenue, which might have some real-world impact, is to reduce the preprocessing cost of the *directed* shortest path algorithm in [Pet02a] from $O(mn)$ to near-linear, as it is in our algorithm.

The marginal cost of computing SSSP with our algorithm may or may not be linear; it all depends on the complexity of the split-findmin structure. This data structure, invented first by Gabow [G85] for use in a weighted matching algorithm, actually has connections with other fundamental problems. For instance, it can be used to solve the minimum spanning tree (and shortest path tree) sensitivity analysis problems[6]. Therefore, by Theorem 4.1 these problems have complexity $O(m \log \alpha(m, n))$, an $\alpha / \log \alpha$ improvement over Tarjan's path-compression-based algorithm [Tar82]. If we consider the *offline* version of the split-findmin problem, where all splits and decrease-keys are given in advance, one can show that it is reducible to *both* the MST problem and the MST sensitivity analysis problem. None of these reductions proves whether $\text{MST}(m, n)$ dominates $\text{SPLIT-FINDMIN}(m, n)$ or vice versa; however, they do suggest that we have no hope of solving the MST problem [PR02c] without first solving the manifestly simpler split-findmin and MST sensitivity analysis problems.

[6] This is an unpublished result that will appear in the Ph.D. thesis of the first author. It uses the split-findmin structure in a straightforward way. See [Tar79, Tar82, DRT92] for other sensitivity analysis algorithms.

The experimental study of Pettie et al. [PRS02] shows that our algorithm is very efficient in practice. However, the [PRS02] study did not explore all possible implementation choices, such as the proper heap to use, the best preprocessing algorithm, or different implementations of the split-findmin structure. To our knowledge, no one has investigated whether the other hierarchy-type algorithms [Tho99, Hag00, Pet02a] are competitive in real-world scenarios.

Another open research problem is the parallel time-work complexity of SSSP. There are several published algorithms on the subject [BTZ98, C+98, KS97, M02, TZ96], though none runs in worst-case polylogarithmic time using work comparable to Dijkstra's algorithm. There is clearly some parallelism in the hierarchy-based approach. Whether this approach can lead to a parallel algorithm which improves upon existing ones is an open question.

APPENDIX

A The Bucket-Heap

The bucket-heap structure consists of an array of buckets, where the ith bucket spans the interval $[\delta + i\mu, \delta + (i+1)\mu)$, for fixed reals δ and μ. Logically speaking, a heap item with key κ appears in the bucket whose interval spans κ. We are never concerned about the relative order of items within the same bucket. We restate below the specification of the bucket-heap then prove the bounds claimed in Lemma 4.1.

create(μ, δ)	Create a new Bucket-Heap whose buckets are associated with intervals $[\delta, \delta+\mu), [\delta+\mu, \delta+2\mu), [\delta+2\mu, \delta+3\mu), \ldots$ An item x lies in the bucket whose interval spans key(x). All buckets are initially *open*.
insert(x, κ)	Insert a new item x with key$(x) = \kappa$.
decrease-key(x, κ)	Set key$(x) = \min\{$key$(x), \kappa\}$. It is guaranteed that x is not moved to a closed bucket.
enumerate	Close the first open bucket and enumerate its contents.

(Lemma 4.1) *Let Δ_x denote the number of buckets between the first open bucket at the time of x's insertion and the bucket from which x was enumerated. The Bucket-Heap can be implemented on a pointer machine to run in $O(m + n + b + \sum_x \log(\Delta_x + 1))$ time, where n, m, and b are the number of insertions, decrease-keys, and enumerates, respectively.*

Proof: Our 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 intervals of 2^i level zero buckets. The interval represented by a higher-level bucket is the union of its component level zero buckets. Only one bucket at each level is *active*: it is the first one which presides over no closed level zero buckets. See Figure 10. Suppose that an item x should *logically* be in the level zero bucket B. We maintain the invariant that x is either *descending* and in the lowest active bucket presiding over B, or *ascending* and in some active bucket presiding over level zero buckets before B.

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

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

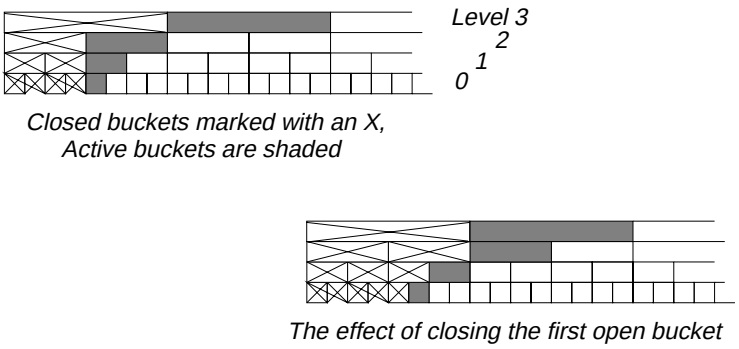

Figure 10: Active buckets are shaded.

From the invariant it follows that no node x appears in more than $2\log(\Delta_x + 1) + 1$ distinct buckets: $\log(\Delta_x + 1) + 1$ buckets as an ascending node and another $\log(\Delta_x + 1)$ as a descending node. Aside from this cost of moving nodes around, the other costs are clearly $O(m + n + b)$, giving the lemma.

□

We remark that the bucket heap need not actually label the items. Whether an item is ascending or descending can be inferred from context.

B The Split-Findmin Problem

The split-findmin problem is to maintain a collection of sequences of weighted elements under the following operations.

split(x)	Split the sequence containing x into two sequences: the elements up to and including x and the rest.
decrease-key(x, κ)	Set key(x) = min{key(x), κ}
findmin(x)	Return the element in x's sequence with minimum key.

Gabow [G85] gave an elegant algorithm for this problem that is nearly optimal. On an initial sequence of n elements, it handles up to $n-1$ splits and m decrease-keys in $O((m+n)\alpha(m,n))$ time. Gabow's algorithm runs on a pointer machine [Tar79]. We now prove Theorem 4.1 from Section 4.2.

(Theorem 4.1) The split-findmin problem can be solved on a pointer machine in $O(n+m\alpha)$ time while making only $O(n + m\log\alpha)$ comparisons, where $\alpha = \alpha(m,n)$ is the inverse-Ackermann function. Alternatively, split-findmin can be solved on a RAM in time $\Theta(\text{SPLIT-FINDMIN}(m,n))$, where $\text{SPLIT-FINDMIN}(m,n)$ is the decision-tree complexity of the problem, or the expected decision-tree complexity for randomized algorithms.

Proof: In Gabow's decrease-key routine a sequence of roughly α variables need to be updated, although it is already known that their values are monotonically decreasing. We observe that, on a pointer machine, the same task can be accomplished in $O(\alpha)$ time using $O(\log\alpha)$ comparisons for a binary search. Using a simple two-level scheme one can easily reduce the $n\alpha$ term in the running time to n. This gives the split-findmin algorithm that performs $O(m\log\alpha(m,n) + n)$ comparisons.

To get a potentially faster algorithm on the RAM model we construct all possible split-findmin solvers on inputs with at most $q = \log\log n$ elements and choose one which is close to optimal for all problem sizes. We then show how to compose this optimal split-findmin solver on q elements with Gabow's structure to get an optimal one on n elements.

We consider only instances with $m' < q^2$ decrease-keys. If more decrease-keys are actually encountered we can revert to Gabow's algorithm [G85] or a trivial one that runs in $O(m')$ time.

We represent the state of the solver with three components: a bit-vector with length $q-1$ representing where the splits are, a directed graph H on no more than $q + m' < q(q+1)$ vertices representing known inequalities between current keys *and* older keys retired by decrease-key operations, and finally, a mapping

from elements to vertices in H. One may easily confirm that the state can be represented in no more than $3q^4 = o(\log n)$ bits. One may also confirm that a split or decrease-key can update the state in $O(1)$ time. We now turn to the findmin operation. Consider the *findmin-action* function, which determines the next step in the findmin procedure. It can be represented as:

$$\text{findmin-action} \ : \ \text{state} \times \{1,\ldots,q\} \ \rightarrow \ \Big(V(H_X) \times V(H_X)\Big) \cup \{1,\ldots,q\}$$

where the first $\{1,\ldots,q\}$ represents the argument to the findmin query. The findmin-action function can either perform a comparison (represented by $V(H_X) \times V(H_X)$) which, if performed, will alter the state, or return an answer to the findmin query, represented by the second $\{1,\ldots,q\}$. One simply applies the findmin-action function until it produces an answer. We will represent the findmin-action function as a table. Since the state is represented in $o(\log n)$ bits we can keep it in one machine word; therefore, computing the findmin-action function (and updating the state) takes constant time on a RAM.

One can see that any split-findmin solver can be converted into another with equal amortized complexity but which performs comparisons only during calls to findmin. Therefore, finding the optimal findmin-action function is tantamount to finding the optimal split-findmin solver.

We have now reduced the split-findmin problem to a brute force search over the findmin-action function. There are less than $F = 2^{3q^4} \cdot q \cdot (q^4+q) < 2^{4q^4}$ distinct findmin-action functions, most of which do not produce correct answers. There are less than $I = (2q + q^2(q+1))^{q^2+3q}$ distinct instances of the problem, because the number of decrease-keys is $< q^2$, findmins $< 2q$ and splits $< q$. Furthermore, each operation can be a split or findmin, giving the $2q$ term, or a decrease-key, which requires us to chose an element and where to fit its new key into the permutation, giving the $q^2(q+1)$ term. Each findmin-action/problem instance pair can be tested for correctness in $V = O(q^2)$ time, therefore all correct findmin-action functions can be chosen in time $F \cdot I \cdot V = 2^{\Omega(q^4)}$. For $q = \log\log n$ this is $o(n)$, meaning the time for this brute force search does not affect the other constant factors involved.

How do we choose the optimal split-findmin solver? This is actually not a trivial question because of the possibility of there not being one solver which dominates all others on all input sizes. Consider charting the worst-case complexity of a solver $\mathcal{S}$ as a function $g_{\mathcal{S}}$ of the number of operations p in the input sequence. It is plausible that certain solvers are optimal for only certain densities p/q. We need to show that for some solver $\mathcal{S}^*$, $g_{\mathcal{S}^*}$ is within a constant factor of the lower envelope of $\{g_{\mathcal{S}}\}_{\mathcal{S}}$, where $\mathcal{S}$ ranges over all correct solvers. Let $\mathcal{S}_k$ be the optimal solver for 2^k operations. We let $\mathcal{S}^*$ be the solver which mimics $\mathcal{S}_k$ from operations $2^{k-1}+1$ to 2^k. At operation 2^k it resets its state, reexecutes all 2^k operations under $\mathcal{S}_{k+1}$, and continues using $\mathcal{S}_{k+1}$ until operation 2^{k+1}. Since $g_{\mathcal{S}_{k+1}}(2^{k+1}) \leq 2 \cdot g_{\mathcal{S}_k}(2^k)$ it follows that $g_{\mathcal{S}^*}(p) \leq 4 \cdot \min_{\mathcal{S}}\{g_{\mathcal{S}}(p)\}$.

Our overall algorithm is very simple. We divide the n elements into $n' = n/q$ super-elements, each representing a contiguous block of q elements. Each unsplit sequence then consists of three parts: two subsequences in the leftmost and rightmost super-elements and a third subsequence consisting of unsplit super-elements. We use Gabow's algorithm on the unsplit super-elements, where the key of a super-element is the minimum over constituent elements. For the super-elements already split, we use the $\mathcal{S}^*$ split-findmin solver constructed as above. The cost of Gabow's algorithm is $O((m+n/q)\alpha(m,n/q)) = O(m+n)$ and the cost of using $\mathcal{S}^*$ on each super-element is $\Theta(\text{SPLIT-FINDMIN}(m,n)$ by construction; therefore the overall cost is $\Theta(\text{SPLIT-FINDMIN}(m,n))$.

One can easily extend the proof to randomized split-findmin solvers by defining the findmin-action as selecting a distribution over actions.

□

We note that the *time* bound of Theorem 4.1 on pointer machines is provably optimal. La Poutré [L96] gave a lower bound on the pointer machine complexity of the split-find problem, which is subsumed by the split-findmin problem. This results in this section address the RAM complexity and decision tree complexity of split-findmin, which are unrelated to La Poutré's result.

References

[AHU76] A. V. Aho, J. E. Hopcroft, J. D. Ullman. On finding lowest common ancestors in trees. *SIAM J. Comput.* 5 (1976), no. 1, 115–132.

[AM+90] R. Ahuja, K. Mehlhorn, J. Orlin, R. E. Tarjan. Faster algorithms for the shortest path problem. *J. Assoc. Comput. Mach.* 37 (1990), no. 2, 213–223.

[AGM97] N. Alon, Z. Galil, O. Margalit. On the exponent of the all pairs shortest paths problem. *J. Comput. Sys. Sci.* 54 (1997), 255-262.

[A+98] A. Andersson, T. Hagerup, S. Nilsson, R. Raman. Sorting in linear time? J. Comput. System Sci. 57 (1998), no. 1, 74–93.

[BTZ98] G. S. Brodal, J. L. Träff, C. D. Zaroliagis. A parallel priority queue with constant time operations. J. Parallel Distr. Comput. 49 (1998), no. 1, pp. 4–21.

[B+98] A. L. Buchsbaum, H. Kaplan, A. M. Rogers, J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In *Proc. STOC 1998*, 279–288.

[Chaz00] B. Chazelle. A minimum spanning tree algorithm with inverse-Ackermann type complexity. In *JACM* 47 (2000), no. 6, 1028–1047.

[CKT00] D. Z. Chen, K. S. Klenk, H. T. Tu. Shortest path queries among weighted obstacles in the rectilinear plane. *SIAM J. Comput.* 29 (2000), no. 4, 1223–1246.

[CGS99] B. V. Cherkassky, A. V. Goldberg, C. Silverstein. Buckets, heaps, lists, and monotone priority queues. SIAM J. Comput. 28 (1999), no. 4, pp. 1326–1346.

[CLRS01] T. Cormen, C. Leiserson, R. Rivest, C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[C+98] A. Crauser, K. Mehlhorn, U. Meyer, P. Sanders. A parallelization of Dijkstra's shortest path algorithm. Proc. 23rd Intl. Symp. Math. Found. of Comp. Sci. (MFCS 1998), LNCS 1450, pp. 722-731.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. In *Numer. Math.*, 1 (1959), 269-271.

[Din78] E. A. Dinic. Economical algorithms for finding shortest paths in a network. *Transportation Modeling Systems*, Y.S. Popkov and B.L. Shmulyian (eds), Institute for System Studies, Moscow, pp. 36–44, 1978.

[DRT92] B. Dixon, M. Rauch, R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. SIAM J. Computing 21 (1992), no. 6, pp. 1184–1192.

[FR01] J. Fakcharoenphol, S. Rao. Planar graphs, negative weight edges, shortest paths and near linear time. In *Proc. FOCS 2001.*

[FG85] A. M. Frieze, G. R. Grimmett. The shortest-path problem for graphs with random arc-lengths. Discrete Appl. Math. 10 (1985), no. 1, 57–77.

[F91] G. N. Frederickson. Planar graph decomposition and all pairs shortest paths. *J. Assoc. Comput. Mach.* 38 (1991), no. 1, 162–204.

[F76] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.* 5 (1976), no. 1, 83–89.

[FT87] M. L. Fredman, R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *JACM* 34 (1987), 596–615.

[FW93] M. L. Fredman, D. E. Willard Surpassing the information-theoretic bound with fusion trees. *J. Comput. System Sci.* 47 (1993), no. 3, 424–436.

[FW94] M. L. Fredman, D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. of Comput. and Syst. Sci.* 48 (1994), no. 3, 533–551.

[G85] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *Proc. FOCS 1985*, 90–99.

[G85b] H. N. Gabow. Scaling algorithms for network problems. *J. Comput. System Sci.* 31 (1985), no. 2, 148–168.

[GT89] H. N. Gabow, R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.* 18 (1989), no. 5, 1013–1036.

[GM97] Z. Galil, O. Margalit. All pairs shortest paths for graphs with small integer length edges. *J. Comput. Sys. Sci.* 54 (1997), 243-254.

[G01] A. V. Goldberg. A simple shortest path algorithm with linear average time. Proc. European Symp. Algorithms (ESA), LNCS 2161, 230–241, 2001.

[G95] A. V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comput.* 24 (1995), no. 3, 494–504.

[GR98] A. V. Goldberg, S. Rao. Beyond the flow decomposition barrier. J. ACM 45 (1998), no. 5, 783–797.

[GYY80] R. L. Graham, A. C. Yao, F. F. Yao. Information bounds are weak in the shortest distance problem. *J. Assoc. Comput. Mach.* 27 (1980), no. 3, 428–444.

[Hag00] T. Hagerup. Improved shortest paths on the word RAM. In *Proc. ICALP 2000*, LNCS volume 1853, 61–72.

[H+97] M. R. Henzinger, P. N. Klein, S. Rao, S. Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.* 55 (1997), no. 1, 3–23.

[HT02] Y. Han, M. Thorup. Integer sorting in $O(n\sqrt{\log\log n})$ expected time and linear space. Proc. 43rd Ann. IEEE Symp. on Found. of Comp. Sci. (FOCS), 2002.

[J77] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. J. Assoc. Comput. Mach. 24 (1977), no. 1, 1–13.

[K70] L. R. Kerr. The effect of algebraic structure on the computational complexity of matrix multiplications. Ph.D. thesis, Cornell University, Ithaca, N.Y., 1970.

[KKP93] D. R. Karger, D. Koller, S. J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM J. on Comput.* 22 (1993), no. 6, 1199–1217.

[KKT95] D. R. Karger, P. N. Klein, R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. J. Assoc. Comput. Mach. 42, 321–328.

[KS97] P. N. Klein, S. Sairam. A randomized parallel algorithm for single-source shortest paths. J. Algorithms 25 (1997), no. 2, 205–220.

[KS98] S. G. Kolliopoulos, C. Stein. Finding real-valued single-source shortest paths in $o(n^3)$ expected time. *J. Algorithms* 28 (1998), no. 1, pp. 125–141.

[L96] H. La Poutré. Lower bounds for the union-find and the split-find problem on pointer machines. *J. Comput. and Syst. Sci.* 52 (1996), no. 1, 87–99.

[M01] U. Meyer. Single-source shortest paths on arbitrary directed graphs in linear average-case time. Proc. 12th Ann. Symp. on Discrete Algorithms (SODA 2001), 797–806.

[M02] U. Meyer. Buckets strike back: improved parallel shortest-paths. Proc. 16th Intl. Par. Distr. Process. Symp. (IPDPS), 2002.

[M00] J. S. B. Mitchell. Geometric shortest paths and network optimization. Handbook of computational geometry, 633–701, North-Holland, Amsterdam, 2000.

[MT87] A. Moffat, T. Takaoka. An all pairs shortest path algorithm with expected time $O(n^2 \log n)$. SIAM J. Comput. 16 (1987), no. 6, 1023–1031.

[MS94] B. M. E. Moret, H. D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science 15 (1994), pp. 99–117.

[Pet02a] S. Pettie. A faster all-pairs shortest path algorithm for real-weighted sparse graphs. In 29th Internat. Symp. on Automata, Languages, and Programming, LNCS vol. 2380, Malaga, Spain, 2002, pp. 85–97.

[Pet02b] S. Pettie. On the comparison-addition complexity of all-pairs shortest paths. In 13th Annual Internat. Symp. on Algorithms and Computation, Vancouver, Canada, 2002, pp. ??–??.

[Pet02c] S. Pettie. An inverse-Ackermann style lower bound for the online minimum spanning tree verification problem. In Proc. 43rd Annual Symp. on the Foundations of Computer Science, Vancouver, Canada, 2002, pp. ??–??.

[PR02a] S. Pettie, V. Ramachandran. An optimal minimum spanning tree algorithm. J. ACM 49 (2002), pp. 16–34.

[PR02b] S. Pettie, V. Ramachandran. Computing shortest paths with comparisons and additions. In Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, 2002, pp. 267–276.

[PR02c] S. Pettie, V. Ramachandran. Minimizing randomness in minimum spanning tree, parallel connectivity, and set maxima algorithms. In Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms, San Francisco, CA, 2002, pp. 713–722

[PRS02] S. Pettie, V. Ramachandran, S. Sridhar. Experimental evaluation of a new shortest path algorithm. In Proceedings of the 4th Workshop on Algorithm Engineering and Experiments, LNCS vol. 2409, San Francisco, CA, 2002, pp. 126–142.

[Ram96] R. Raman. Priority queues: small monotone, and trans-dichotomous. In Proceedings of European Symposium on Algorithms, 1996, pp. 121–137.

[Ram97] R. Raman. Recent results on the single-source shortest paths problem. SIGACT News 28 (1997), 81–87.

[S95] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. J. Comput. Syst. Sci., vol. 51 (1995), pp. 400–403.

[SZ99] A. Shoshan, U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In Proc. *FOCS 1999*, pp. 605–614.

[S73] P. M. Spira. A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$. SIAM J. Comput. 2 (1973), 28–32.

[SP75] P. M. Spira, A. Pan. On finding and updating shortest paths and spanning trees. *SIAM J. Comput.* 4 (1975), no. 3, pp. 375–380.

[Tak92] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Inform. Process. Lett.* 43 (1992), no. 4, 195–199.

[Tak98] T. Takaoka. Subcubic cost algorithms for the all pairs shortest path problem. Algorithmica 20 (1998), no. 3, 309–318.

[Tar79] R. E. Tarjan. A class of algorithms which require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.* 18 (1979), no. 2, 110–127.

[Tar79b] R. E. Tarjan. Applications of path compression on balanced trees. *J. Assoc. Comput. Mach.* 26 (1979), no. 4, 690–715.

[Tar82] R. E. Tarjan. Sensitivity analysis of minimum spanning trees andshortest path trees. Infor. Process. Lett. 14 (1982), no. 1, pp. 20–33. Corrigendum: Infor. Process. Lett. 23 (1986), no. 4, pp. 219.

[Tho99] M. Thorup. Undirected single source shortest paths with positive integer weights in linear time. *J. Assoc. Comput. Mach.* 46 (1999), no. 3, 362–394.

[Tho00] M. Thorup. On RAM priority queues. SIAM J. Comput. 30 (2000), no. 1, 86–109.

[TZ96] J. L. Träff, C. D. Zaroliagis. A simple parallel algorithm for the single-source shortest path problem on planar digraphs. Parallel algorithms for irregularly structured problems, LNCS 1117 (1996), 183–194.

[vEB+76] P. van Emde Boas, R. Kaas, E. Zijlstra. Design and implementation of an efficient priority queue. Math. Systems Theory 10 (1976/77), no. 2, 99–127.

[Z01] U. Zwick. Exact and approximate distances in graphs—a survey. European Symp. on Algorithms (ESA), LNCS 2161, 33–48, 2001.

[Z02] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication J. ACM 49 (2002), no. 3, 289–317.