A New Approach to All-Pairs Shortest Paths on Real-Weighted Graphs ⋆

Seth Pettie [1]

Department of Computer Sciences,
The University of Texas at Austin,
Austin TX, 78712 USA

Abstract

We present a new all-pairs shortest path algorithm that works with real-weighted graphs in the traditional *comparison-addition* model. It runs in $O(mn+n^2 \log\log n)$ time, improving on the long-standing bound of $O(mn + n^2 \log n)$ derived from an implementation of Dijkstra's algorithm with Fibonacci heaps. Here m and n are the number of edges and vertices, respectively.

Our algorithm is rooted in the so-called *component hierarchy* approach to shortest paths invented by Thorup for integer-weighted undirected graphs, and generalized by Hagerup to integer-weighted directed graphs. The technical contributions of this paper include a method for *approximating* shortest path distances and a method for leveraging approximate distances in the computation of exact ones. We also provide a simple, one line characterization of the class of hierarchy-type shortest path algorithms. This characterization leads to some pessimistic lower bounds on computing single-source shortest paths with a hierarchy-type algorithm.

Key words: comparison-addition model, all-pairs shortest paths, real weights

⋆ A preliminary version of this paper appeared in the *29th Int'l Colloq. on Automata, Languages, and Programming* (ICALP 2002), titled "A Faster All-pairs Shortest Path Algorithm for Real-weighted Sparse Graphs".

Email address: `seth@cs.utexas.edu` (Seth Pettie).
URL: `http://www.cs.utexas.edu/users/seth/` (Seth Pettie).

[1] Supported by Texas Advanced Research Program Grant 003658-0029-1999, NSF Grant CCR-9988160, and an MCD Graduate Fellowship.

1 Introduction

Nearly all known shortest path algorithms can be neatly separated into two groups: those which assume real-weighted graphs, where reals are manipulated only by *comparison* and *addition* operations, and those which assume integer-weighted graphs and a suite of RAM-type operations to act on the edge weights. The standard textbook algorithms, established early on by Dijkstra [1], Bellman-Ford, Floyd-Warshall and others (see [2]), all work in the comparison-addition model with real edge-weights. Since then most progress on shortest paths problems has come by assuming integral edge-weights. Techniques based on scaling [3–5], integer matrix multiplication [6–10], and fast integer sorting (see [11–15] for recent results) only work with integer edge-weights, and until recently it appeared as though the *component hierarchy* approach used in [16] and [17] also required integers. We refer the reader to a recent survey paper [18] for more background and references.

The state of the art in all-pairs shortest paths (APSP) for real-weighted, sparse, directed graphs is — quite surprisingly — a combination of two standard textbook algorithms. For the case of non-negatively weighted graphs, Dijkstra's algorithm [1] computes single-source shortest paths (SSSP) in $O(m + n \log n)$ time, and, by repeated application, APSP in $O(mn + n^2 \log n)$ time. These time bounds assume that a Fibonacci heap [19] is used. Johnson [20] gave an $O(mn)$-time shortest path-preserving reduction from arbitrarily weighted to non-negatively weighted graphs (assuming no negative weight cycles). Combined with Dijkstra's algorithm this implies an $O(mn + n^2 \log n)$ time general APSP algorithm. This is the fastest algorithm to date for *directed* graphs; there is a faster algorithm [21] for *undirected* graphs running in $O(mn \log \alpha(m, n))$ time. [2]

For the case of dense graphs, Fredman [23] gave an APSP algorithm that performs $O(n^{2.5})$ comparisons and additions; however, he did not provide even a polynomial-time implementation of this algorithm. The fastest APSP algorithms for dense graphs [23,24] use Fredman's approach on very small problems. They are faster than the $O(n^3)$ algorithms of Floyd and Dijkstra by only sub-logarithmic factors.

Our algorithm fits into the hierarchy framework for computing shortest paths [16,17,21,25]. The impetus behind this approach is a stubborn fact: Dijkstra's SSSP algorithm [1] is inherently as hard as sorting, making its $O(m+n \log n)$-time implementation with Fibonacci heaps [19] optimal in the comparison-addition model. Thorup [16], who invented the hierarchy-based approach,

[2] [21] claims APSP is solved on a pointer machine [22] in $O(mn\alpha(m, n))$ time. We note here that if constant-time array lookups are allowed, [21] can be implemented in $O(mn \log \alpha(m, n))$ time. Here α is the inverse-Ackermann function.

showed that *undirected* SSSP on non-negative *integer*-weighted graphs can be solved in $O(m)$ time, assuming edge-weights are subject to typical RAM operations. Thorup's algorithm is like Dijkstra's in that it fixes the distance to each vertex, one vertex at a time; however, he circumvents the sorting-bottleneck inherent in Dijkstra's algorithm by not insisting that vertices be visited in order of increasing distance. Hagerup [17] generalized Thorup's approach to directed graphs, though he remained in the same non-negative integer/RAM model. He gave an $O(m \log\log C + n \log\log n)$ time SSSP algorithm and an $O(mn + n^2 \log\log n)$ APSP algorithm, which is the fastest APSP algorithm to date for sparse graphs in the integer/RAM model. Here C represents the largest integer edge weight.

The requirement that edge-lengths be integers seemed, at first, to be essential to the algorithms of Thorup and Hagerup [16,17]. Recently, Pettie and Ramachandran [21] adapted Thorup's algorithm to *real*-weighted graphs and the comparison-addition model, yielding an undirected APSP algorithm running in $O(mn \log \alpha)$ time, where $\alpha = \alpha(m, n)$ is the mind-bogglingly slow-growing inverse of Ackermann's function. Pettie [25] has shown, also with a hierarchy-based algorithm, that the comparison-addition complexity of directed APSP is $O(mn \log \alpha)$; however, this algorithm has a large (though polynomial) overhead for deciding which comparisons and additions to make. The hierarchy-based approach also turns out to be practical in certain situations. An experimental study by Pettie et al. [26] of a simplified version of the algorithm in [21] shows it to be decisively faster than Dijkstra's algorithm, if the one-time cost of constructing a hierarchy is offset by a sufficient number of SSSP computations.

In this paper we generalize the hierarchy-based approach to real-weighted directed graphs and explore its complexity in the comparison-addition model. Our primary result is a new APSP algorithm that runs in $O(mn+n^2 \log\log n)$ time. A second contribution of this paper is a new way to view the commonalities between all hierarchy-type algorithms. We give, in particular, a simple, one-line characterization of all hierarchy-type algorithms which allows us to prove lower bounds on their complexity in the comparison-addition model. The upshot is that for directed and undirected graphs alike, no hierarchy-type SSSP algorithm can break the $\Omega(m + n \log n)$ barrier. A subtler consequence of our lower bound is that no *directed* APSP algorithm can break the $\Omega(mn + n^2 \log n)$ barrier if it follows the traditional hierarchy-based approach [16,17,21], which is to first compute some kind of hierarchy, then to solve n SSSP problems with n *independent* processes. In other words, if we choose to stay in the hierarchy framework, at some point in the algorithm we *must* exploit the *dependencies* that exist between different SSSP computations. Our APSP algorithm introduces a novel method for identifying and representing these dependencies.

1.1 Organization

In Section 2 we define the problems and the comparison-addition model. In Sections 2.2 and 2.3 we outline Dijkstra's classical algorithm and give an informal introduction to the hierarchy approach to shortest paths. In Section 3 we give a lower bound on any hierarchy-based algorithm computing SSSP in the comparison-addition model. In Section 4 we adapt the hierarchy-based approach to real-weighted directed graphs. We present our $O(mn+n^2 \log\log n)$ time APSP algorithm in Section 5.

2 Preliminaries

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

The length of a path is defined to be the sum of its constituent edge lengths. The *distance* from vertex u to vertex v, denoted $d(u, v)$, is defined as the length of the minimum-length path from u to v. The APSP problem is to compute the $d(\cdot,\cdot)$ function, and the SSSP problem is to compute the $d(s,\cdot)$ function for a fixed *source* s. We generalize the d notation to include subgraphs. Define $d(H_1, H_2)$, where H_i can be a vertex or subgraph, to be the minimum distance from any vertex in H_1 to any vertex in H_2. H_i may also be an object associated with a subgraph, not necessarily the subgraph itself.

2.1 The Comparison-Addition Model

In the comparison-addition model, real numbers are only subject to *comparisons* and *additions*. Comparisons determine the larger of two given reals, and addition of existing reals is the *only* means for generating new reals. An algorithm in this model chooses the next operation based on the outcome of previous comparisons.

The comparison-addition model fits naturally with the assumption of real numbers; however, it does not depend on this assumption. The model could just as easily be cast in programming terminology. We would assume an arbitrary numerical data type called $\mathbb{R}$, representing some subset of the reals closed under addition. The actual instantiation of $\mathbb{R}$ is not our concern so long as it supports two operations, $+ : \mathbb{R}\times\mathbb{R} \to \mathbb{R}$, and $< : \mathbb{R}\times\mathbb{R} \to \{\text{true, false}\}$ representing addition and comparison of numbers of type $\mathbb{R}$. The data type $\mathbb{R}$ could represent reals, integers, rationals, or something else. [3]

In the description of our algorithm we frequently make use of subtraction, multiplication by an integer, division and the floor operation. These operations are, of course, unavailable in the comparison-addition model; however they can frequently be simulated cheaply and sometimes without asymptotic penalty. It is shown in [21], for instance, that simulating subtraction incurs at most a constant factor overhead. The idea is to represent each virtual real number x as the difference between two actual reals a and b so that $x = a - b$. A comparison between the virtual reals $x_1 = a_1 - b_1$ and $x_2 = a_2 - b_2$ then reduces to a comparison between the actual reals $(a_1 + b_2)$ and $(a_2 + b_1)$. Multiplication by an integer is also simple. Suppose x is a real and N an integer. We can calculate Nx in $O(\log N)$ time as follows. Produce the set of reals $B = \{x, 2x, 4x, 8x, \ldots, 2^{\lfloor \log N \rfloor} x\}$, using $\log N$ additions, then produce Nx by summing up the appropriate subset of B. Division by an integer is accomplished in a similar fashion. Suppose we set $y = x/N$. If we want to compare y with another number, say z, we can substitute the equivalent comparison between x and Nz.

An operation that comes in very handy is division by a real, followed by the floor operation, i.e. computing the integer $\lfloor \frac{x}{y} \rfloor$. This operation is different from the ones discussed above because the result is an *integer* rather than a real number. We will discuss this difference below. We can compute $\lfloor \frac{x}{y} \rfloor$ in $O(1 + \log \frac{x}{y})$ time using a method similar to our simulation of multiplication. We produce the set $B = \{y, 2y, 4y, 8y, \ldots, 2^{\lfloor \log \frac{x}{y} \rfloor} y\}$ then use the elements of B to implement a binary search to find the integer $\lfloor \frac{x}{y} \rfloor$.

We have already defined the limits of the data type *real*. In what ways does the data type *integer* differ? Like reals, integers are subject to comparisons and additions. We assume integers can take on values between 0 and $O(n^2)$ (this is essentially a minimal assumption since the size of the output is n^2 in the APSP problem) and that they can be used to index an array. That is, if i is an integer and A is an array, then $A[i]$ can be looked up in unit time. We assume *no* primitive operations that convert reals to integers or vice versa.

There are several lower bounds on various shortest path problems in the

[3] At this point one might be tempted to assume a more abstract algebraic structure, such as an ordered semigroup $(S, \oplus, \prec)$. Our algorithm can very likely be adapted to this model, but at the cost of simplicity. For instance, every inference normally made by common sense, such as $x \preceq y \implies (x \oplus z) \preceq (y \oplus z)$, would require explicit justification.

comparison-addition model. However, they are all very weak. Spira and Pan [27] showed that regardless of additions, $\Omega(n^2)$ comparisons are necessary to solve SSSP on the complete graph. Karger et al. [28] proved that all-pairs shortest paths requires $\Omega(mn)$ comparisons if all summations correspond to paths in the graph. Kerr [29] showed that any oblivious APSP algorithm performs $\Omega(n^3)$ operations, and Kolliopoulos and Stein [30] proved that any fixed sequence of edge relaxations solving SSSP must have length $\Omega(mn)$. By "fixed sequence" they mean one which depends on m and n but *not* the graph topology. Graham et al. [31] did not give a lower bound but showed that the standard information-theoretic argument cannot yield a non-trivial ($\omega(n^2)$) lower bound in the APSP problem. Similarly, no information-theoretic argument can provide an interesting lower bound on SSSP.

2.2 Dijkstra's Algorithm

Dijkstra's SSSP algorithm *visits* vertices in order of increasing distance from the source s. It maintains a set S of visited vertices, initially empty, and a *tentative distance* $D(v)$ for all $v \in V$ satisfying the following invariant.

Invariant 0 *For $v \in S$, $D(v) = d(s,v)$ and for $v \notin S$, $D(v)$ is the distance from s to v using only intermediate vertices from S.*

Dijkstra's method for growing the set S while maintaining Invariant 0 is to visit vertices in ascending order of their distance from the source. In each step, Dijkstra's algorithm identifies the vertex $v \notin S$ with minimum tentative distance, sets $S := S \cup \{v\}$, and updates tentative distances. This involves *relaxing* each outgoing edge (v,w) by setting $D(w) := \min\{D(w),\ D(v) + \ell(v,w)\}$. The algorithm halts when $S = V$, implying that the tentative distances equal the actual distances.

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

2.3 An Outline of the Hierarchy Approach

The hierarchy-based approach can be traced back to a simple observation of Dinic [32]: if $t > 0$ is the minimum edge length in the graph, then $d(v, u) \geq t$ for $v \neq u$. Therefore, one can maintain Invariant 0 by visiting any vertex $u \notin S$ minimizing $\lfloor D(u)/t \rfloor$. Thorup's insight [16] was that this idea could be generalized to arbitrary and even multiple values for t. We will give a precise description of the approach in Section 4. For now, consider an illustrative example.

Suppose that the vertex set V can be partitioned into V_1 and V_2 such that all edges crossing the cut (V_1, V_2) have length at least t. We imagine a shortest path algorithm for this graph as composed of three interacting processes: p_1, p_2, and p_h, where p_1 and p_2 govern V_1 and V_2, respectively, and p_h is a high-level process governing the other two. The process p_h operates by passing real intervals to p_1 and p_2. If $[a, b)$ is passed to p_1 this means that p_1 should find and visit all $v \in V_1$ such that $d(s, v) \in [a, b)$. The p_h process might compute SSSP with the following simple algorithm: pass to both p_1 and p_2 the interval $[0, t)$, followed by the intervals $[t, 2t)$, $[2t, 3t)$, $\ldots$, $[it, (i + 1)t), \ldots$ until all vertices are visited. The crucial observation here is that p_1 and p_2 always operate on *independent* subproblems. Specifically, when $[a, a + t)$ is passed to p_1 it can determine those $v \in V_1$ such that $d(s, v) \in [a, a + t)$ based solely on the subgraph induced by V_1 and the tentative distances (D-values) to vertices in V_1. The reason is simple: if $v \in V_1$ is such that $d(s, v) \in [a, a+t)$, the shortest s–to–v path cannot pass through any vertex $w \in V_2$ such that $d(s, w) \geq a$. If w were on the shortest s–to–v path then we would have $d(s, v) = d(s, w) + d(w, v) \geq a + t$, a contradiction. The lower bound $d(w, v) \geq t$ follows since any w–to–v path crosses the cut (V_1, V_2), and therefore must include an edge of length at least t.

The example given above is a bit simpler than the situation encountered in our algorithm. Generally speaking, we partition the vertex set into more than two subsets, and we can only make an asymmetric guarantee on the edges crossing the partition, i.e. for V_1 and V_2 in the partition, we can only lower bound the length of edges going from V_1 to V_2, not the other way. These generalizations do not present much of a problem; the tricky part is implementing this approach efficiently. Although the hierarchy approach does not involve visiting the vertices in increasing distance from the source, we will see in Section 3 that there is a certain inherent sorting bottleneck in the approach.

3 Lower Bounds for SSSP

In this section we give lower bounds on the complexity of any hierarchy-based algorithm in a comparison-based model. The notion of lower bounding an algorithm, rather than the complexity of a *problem*, should make one a little uncomfortable. This is because there is no agreed upon standard for deciding when two programs are really implementations of the same algorithm. Our approach is to succinctly characterize the *extra information* obtained by running some algorithm, then to lower bound the complexity of computing that extra information from scratch (so we are, in effect, lower bounding a problem.) The robustness of this approach depends on how central the extra information is to the algorithm in question.

Let us illustrate the method on Dijkstra's single-source shortest path algorithm [1]. One way to characterize Dijkstra's algorithm, without specifying how it works, is to say that it finds a permutation $\pi_s : V \to V$ such that

$$\forall u, v \in V \; : \quad \pi_s(u) < \pi_s(v) \quad \Rightarrow \quad d(s,u) \leq d(s,v)$$

where $d(\cdot,\cdot)$ is the distance function and s is the source. It is not difficult to show that any algorithm computing such a permutation (and hence any implementation of Dijkstra's algorithm) must make $\Omega(n \log n)$ comparisons [33].

We give a similar characterization of the hierarchy-based shortest path algorithms. Suppose for this discussion that the graph is strongly connected. Let $\text{CYCLES}(u,v)$ be the set of all cycles containing vertices u and v and let $\text{SEP}(u,v)$ be defined as

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

where $\ell(e)$ is the length of edge e. Notice that if the graph is undirected, $\text{SEP}(u,v)$ is the length of the longest edge in the minimum spanning tree path connecting u and v. Regardless of whether the graph is undirected or directed, all hierarchy-based algorithms generate a permutation π_s satisfying Property 1.

Property 1 *Let s be the source. For any pair of vertices u, v, π_s satisfies*
$$d(s,v) \geq d(s,u) + \text{SEP}(u,v) \quad \Rightarrow \quad \pi_s(u) < \pi_s(v)$$

Is there a sorting bottleneck inherent in Property 1? The answer is that it depends. Note that $d(s,v)$ and $d(s,u)$ depend on the source s, whereas $\text{SEP}(u,v)$ does not. From the perspective of a *multi*-source shortest path algorithm, estimating $\textsc{sep}(u, v)$ is a *one-time* cost, whereas computing π_s, given the SEP function, can be thought of as the *marginal* cost of computing SSSP. We are interested in lower bounding the cost of SSSP both when the SEP function is known and unknown. Our results are tabulated in Figure 1. When SEP is unknown, we have $\Omega(m + n \log n)$ lower bounds for computing SSSP on both directed and undirected graphs. However, the undirected bounds become qualitatively weaker if we introduce a new parameter. Let r denote the ratio of the maximum-to-minimum edge length. The directed and undirected bounds become, respectively, $\Omega(m + \min\{n \log r, n \log n\})$ and $\Omega(m + \min\{n \log\log r, n \log n\})$. In other words, to induce an $\Omega(m + n \log n)$ lower bound, r need only be polynomial for directed graphs, but exponential for undirected ones. When the SEP function is known, no non-trivial lower bounds are known for undirected graphs, whereas the same lower bound holds for directed graphs. In light of the results from [21,19], the bounds on directed graphs are tight, and the bounds on undirected graphs are tight to within a $\log \alpha(m, n)$ factor when SEP is known, and an $\alpha(m, n)$ factor when SEP is unknown.

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

Fig. 1. Lower bounds on SSSP algorithms satisfying Property 1 in the comparison-addition model. The parameter r bounds the ratio of the maximum-to-minimum edge length.

In Section 3.1 we prove the lower bound for directed graphs. The lower bound for undirected graphs appears in [21].

3.1 Limits on Hierarchy-Type Algorithms

In this section we consider a model that is more powerful than the comparison-addition model. The model allows the use of *any* function mapping vectors of reals to vectors of reals, as well as comparison between reals.

We will show that any directed SSSP algorithm satisfying Property 1 must make $\Omega(n \log n)$ comparisons, and hence take $\Omega(m + n \log n)$ time. This lower bound extends to randomized algorithms and holds even under the assumption that $\textsc{sep}(u, v)$ is a constant for $u \neq v$.

We consider a graph with fixed structure and a set of possible edge-length functions. A permutation of the vertices is said to be *compatible* with a certain edge-length function if it satisfies Property 1.

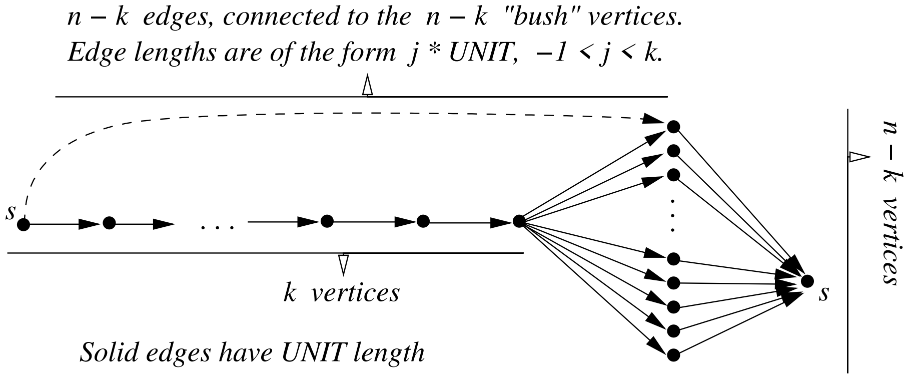

Fig. 2. The “broom” graph.

Our fixed graph, depicted in Figure 2, is organized a little like a broom. It has a “broom stick” of k vertices, whose head is the source s and whose tail connects to the remaining $n-k$ vertices (the “bush”), each of which is connected back to s by an edge (s appears twice to simplify the figure). All these edges have equal length $UNIT$, which is an arbitrary positive real. Additionally, there are $n-k$ edges directed from s to each of the vertices in the bush, having lengths of the form $j \cdot UNIT$, where j is a non-negative integer. One may easily confirm that $\text{SEP}(u,v) = UNIT$ for all distinct u, v. Assuming without loss of generality that k divides n, we define $\mathcal{L}$ to be the set of length functions that assign the edge length $j \cdot \text{UNIT}$ to exactly $(n-k)/k = n/k - 1$ edges from s to the “bush”, for $0 \leq j < k$. We have that $|\mathcal{L}| = (n-k)!/(\frac{n}{k}-1)!^k$. One can also see that for any two length functions $\ell^1, \ell^2 \in \mathcal{L}$, there is always a pair of vertices u, v in the broom's bush such that $d(s,u) < d(s,v)$ with respect to ℓ^1, but $d(s,v) < d(s,u)$ with respect to ℓ^2. Since $d(s,u) < d(s,v)$ implies $d(s,u) \leq d(s,v) + \text{SEP}(u,v)$, no permutation of the vertices can be compatible with both ℓ^1 and ℓ^2. Therefore at least $\log|\mathcal{L}|$ comparisons must be made to choose a compatible permutation. For $k \leq n/2$, $\log|\mathcal{L}| = \Omega(n \log k)$. One can repeat this lower bound argument for any source in the broom's bush. Theorem 1 follows.

Theorem 1 *Suppose* $\text{SEP}(u,v)$ *is already known, for all vertices* u, v. *Any directed SSSP algorithm obeying Property 1 must take time* $\Omega(m + \min\{n \log r, n \log n\})$, *where the source can be any of* $n - o(n)$ *vertices and* r *bounds the ratio of any two edge-lengths.*

4 Fundamentals of the Hierarchy Approach

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

Recall that s denotes the source of the SSSP problem. Let $X \subseteq V$ denote a set of vertices. We define $d_X(s, v)$ to be the distance from s to v in the subgraph induced by X. If I is a real interval, we define X^I to be the set $\{v \in X \ : \ d(s, v) \in I\}$, that is, those vertices in X whose distances from the source lie in I.

Definition 2 *Let X and S be sets of vertices and I be a real interval. We will call X (S, I)-independent if for all $v \in X^I$, $d(s, v) = d_{S \cup X^I}(s, v)$*

To paraphrase Definition 2, if X is (S, I)-independent then one can determine the set X^I by examining only the subgraph induced by $S \cup X^I$. Suppose that we discover that X is (S, I)-independent in the context of a Dijkstra-like algorithm, i.e. one satisfying Invariant 0. Now we can say something stronger: because the D-values for vertices in $X^I - S$ encode all the relevant information about the subgraph induced on S, one can determine X^I by examining only the subgraph induced by $X^I - S$ and the D-values of those vertices.

A *t-partition*, defined below, is a particularly useful tool for transforming one independent subproblem into several smaller ones.

Definition 3 *Let X be a set of vertices. The sequence $(X_1, X_2, \ldots, X_k)$ is a t-partition of X if $\{X_i\}_i$ is a partition of X and for every edge (u, v) where $u \in X_i$, $v \in X_j$, and $j < i$, we have $\ell(u, v) \geq t$.*

Note the asymmetry in Definition 3. In a t-partition only "backward" edges crossing the partition have length at least t; "forward" edges can be any length. Lemma 4 shows the relationship between t-partitions and independent subproblems.

Lemma 4 *Suppose that X is $(S, [a, b))$-independent. Let $(X_1, \ldots, X_k)$ be a t-partition of X, let I be the interval $[a, \min\{a + t, b\})$, and let $S_i = S \cup X_1^I \cup X_2^I \cup \cdots \cup X_i^I$. Then*

(1) X_{i+1} is (S_i, I)-independent

(2) X is $(S_k, [a + t, b))$-independent

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

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

Lemma 4 is essentially describing a divide and conquer scheme for SSSP. One finds an independent subproblem on the vertex set X, divides it into a series of smaller independent subproblems (using some t-partition) then solves the smaller problems recursively. The basis case, when X is a single vertex, is easy to handle. The problem of finding the first independent subproblem is also easy: V (all the vertices) is trivially $(\emptyset, [0, \infty))$-independent. The difficulty lies in implementing this scheme efficiently. In Section 4.1 we define a *stratified hierarchy*, which is a structure for representing all the t-partitions encountered in our algorithm. In Section 4.2 we give a recursive procedure for computing SSSP and discuss some of the details of its implementation. Using off-the-shelf data structures and no fancy techniques our algorithm solves APSP in $O(mn + n^2 \log n)$ time, the same bound as Dijkstra's algorithm. In Section 5 we introduce the techniques required to implement our algorithm more efficiently.

4.1 A Stratified Hierarchy

A *hierarchy* is a rooted tree where there is a one-to-one correspondence between its leaves and the graph's vertices. We will think of leaves in a hierarchy as being identical to the vertices they represent. An internal node x corresponds to the induced subgraph C_x on the descendant leaves of x. We will assign to each internal node x a value $\textsc{norm}(x)$ such that if $x_1, \ldots, x_\nu$ are the children of x in left-to-right order, $(V(C_{x_1}), \ldots, V(C_{x_\nu}))$ is a $\textsc{norm}(x)$-partition of $V(C_x)$.

Thorup [16] and Hagerup [17] always choose their $\textsc{norm}$-values from the set $\{2^i\}_{i \geq 0}$. Their hierarchies can be adapted in a straightforward manner to real-weighted graphs by selecting $\textsc{norm}$-values from the set $\{\ell_1 \cdot 2^i\}_{i \geq 0}$, where ℓ_1 is the minimum edge length in the graph. However, there is a potential problem with this scheme. Generating the value $\ell_1 \cdot 2^i$ requires $\Theta(i)$ time in the comparison-addition model, which is unbounded as a function of m and n. We do not want to place an a priori limit on the ratio of any two edge lengths. Our solution, as in [21], is to build a *stratified hierarchy* where each stratum corresponds to a different normalizing edge length, rather than using

ℓ_1 as the only normalizing edge length. We ensure that the ratio of two NORM-values within a stratum is bounded as a function of n, and that the strata are *well-separated* in a certain sense.

We now define the structure of our stratified hierarchy $\mathcal{SH}$. First, let $\ell_1, \ldots, \ell_m$ be the edge lengths of the graph in sorted order. We choose, as our set of normalizing lengths, $\{\ell_1\} \cup \{\ell_j : \ell_j > 2n \cdot \ell_{j-1}\} \cup \{\infty\}$. That is, every normalizing length is much larger than any shorter edge lengths. Let ℓ_{r_k} be the k^{th} smallest normalizing length. The nodes of $\mathcal{SH}$ are indexed by their stratum and level within the stratum. For stratum k the levels run from 0 to the maximum i such that $\ell_{r_k} \cdot 2^i < \ell_{r_{k+1}}$. If x is a stratum k, level i node then we define $\textsc{norm}(x)$ as:

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

Each $\mathcal{SH}$-node x corresponds to a strongly connected component[4] (SCC) in the graph restricted to edges with length less than $2 \cdot \textsc{norm}(x)$. We denote by C_x the SCC associated with x. A node x is an ancestor of y if x has larger stratum/level and $V(C_y) \subseteq V(C_x)$. We will call x *irrelevant* if $V(C_y) = V(C_x)$ for some descendant $y \neq x$. The *parent* of a node is its nearest relevant ancestor. Similarly, the children of x are those nodes identifying x as their parent. It is only necessary that we represent the relevant nodes; henceforth, "$x \in \mathcal{SH}$" means x is a relevant node in $\mathcal{SH}$. Figure 3 gives an example input graph and its associated $\mathcal{SH}$.

Suppose $x_1, \ldots x_\nu$ are the children of x. We define C_x^c as the graph derived from C_x by contracting the SCCs $C_{x_1}, \ldots, C_{x_\nu}$ and retaining only those edges with length less than $\textsc{norm}(x)$. There is a natural correspondence between the children of x and the vertices of C_x^c. The notation $y \in V(C_x^c)$ means that y is a child of x.

Claim 5 *C_x^c is acyclic.*

PROOF. Let x be at stratum k, level i, and identify the vertices of C_x^c with (possibly irrelevant) $\mathcal{SH}$-nodes at stratum k level $i-1$ (or if $i = 0$, the maximum level in the previous stratum). Stratum k, level $i-1$ nodes represent the *maximal* strongly connected subgraphs on the graph restricted to edges with length less than $\ell_{r_k} \cdot 2^{i-1} = \textsc{norm}(x)$. Since all edges in C_x^c have length less than $\textsc{norm}(x)$, no cycle can exist in C_x^c. If there were a cycle then at least two stratum k, level $i-1$ nodes could be merged to form a strictly larger SCC. □

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

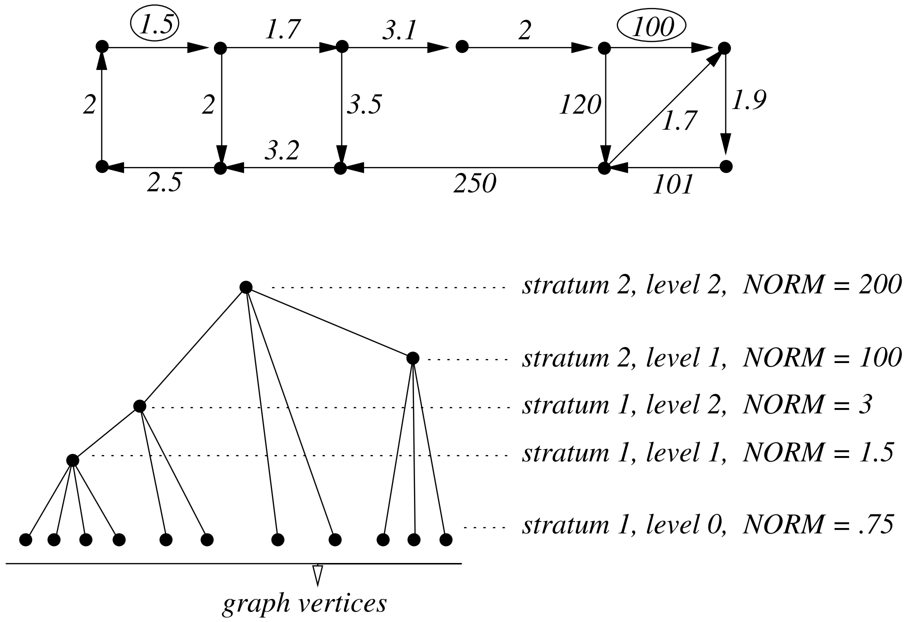

Fig. 3. Above: the input graph. Circled edge lengths represent "normalizing" lengths. Below: the associated $\mathcal{SH}$. It has two strata, based on the normalizing lengths $\ell_{r_1} = 1.5$ and $\ell_{r_2} = 100$. A stratum k, level i node x has $\textsc{norm}(x) = \ell_{r_k} \cdot 2^{i-1}$, and represents a strongly connected component of the graph, when restricted to edges with length less than $2 \cdot \textsc{norm}(x)$. Irrelevant $\mathcal{SH}$-nodes (those having one child) are not shown in the figure.

We assume that the children of x (corresponding to the vertices of C_x^c) appear in some left-to-right order consistent with a topological sorting of the vertices in C_x^c.

Claim 6 *Suppose y, z are children of x, where y is to the left of z. If (u, v) is an edge such that $u \in V(C_z)$, $v \in V(C_y)$, then $\ell(u, v) \geq \textsc{norm}(x)$.*

PROOF. If $\ell(u, v) < \textsc{norm}(x)$ then their exists a corresponding edge $(z, y) \in E(C_x^c)$ with equal length. But $(z, y) \in E(C_x^c)$ means that in any topological sort of $V(C_x^c)$, z preceeds y, a contradiction. $\square$

We use the notation $\textsc{diam}(C_x)$ to denote an upper bound on the diameter of C_x, that is, the length of the longest shortest path between any two vertices in C_x. We calculate $\textsc{diam}(C_x)$ as follows.

$$\textsc{diam}(C_x) \;=\; 2\,\textsc{norm}(x) \cdot (|V(C_x^c)| - 1) \;+\; \sum_{y \,\in\, V(C_x^c)} \textsc{diam}(C_y) \tag{2}$$

Lemma 7, given below, summarizes all the relevant properties of $\mathcal{SH}$ used in our algorithm's analysis and proof of correctness.

Lemma 7 *(1) $\mathcal{SH}$ has a single root, denoted* $\text{ROOT}(\mathcal{SH})$.

(2) Let $x \in \mathcal{SH}$ and $x_1, x_2, \ldots, x_\nu$ be x's children in left-to-right order. Then $(V(C_{x_1}), \ldots, V(C_{x_\nu}))$ is a $\text{NORM}(x)$*-partition of $V(C_x)$.*

(3) If x is the parent of y, then either $\text{NORM}(x)$ is a multiple of $\text{NORM}(y)$, or $\text{DIAM}(C_y) < \text{NORM}(x)$.

(4) $$\sum_{x \in \mathcal{SH}} |V(C_x^c)| \;<\; 2n - 1$$

(5) $$\text{For any } x \in \mathcal{SH},\ \frac{\text{DIAM}(C_x)}{\text{NORM}(x)} \;<\; 2n$$

(6) $$\sum_{x \in \mathcal{SH}} \frac{\text{DIAM}(C_x)}{\text{NORM}(x)} \;<\; 4n$$

(7) $$\left|\left\{ x \,\in\, \mathcal{SH} \;:\; \frac{\text{DIAM}(C_x)}{\text{NORM}(x)} \;\geq\; k \right\}\right| \;<\; \frac{4n}{k}$$

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

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

$$\begin{aligned}
\sum_{x \in \mathcal{SH}} \frac{\text{DIAM}(C_x)}{\text{NORM}(x)} &= \sum_{x \in \mathcal{SH}} \frac{2\,\text{NORM}(x) \cdot (|V(C_x^c)| - 1) + \sum_{y \in V(C_x^c)} \text{DIAM}(C_y)}{\text{NORM}(x)} \\
&= \sum_{x \in \mathcal{SH}} \sum_{z \text{ desc. } x} \frac{2\,\text{NORM}(z) \cdot (|V(C_z^c)| - 1)}{\text{NORM}(x)} \\
&= \sum_{z \in \mathcal{SH}} \sum_{j \geq 0} \frac{2\,\text{NORM}(z) \cdot (|V(C_z^c)| - 1)}{\text{NORM}(z^j)}
\end{aligned}$$

$$
\begin{aligned}
&< \sum_{z\in\mathcal{SH}} \sum_{j=0}^{\infty} \frac{(|V(C_z^c)|-1)}{2^{j-1}} \\
&= \sum_{z\in\mathcal{SH}} 4\cdot(|V(C_z^c)|-1) \;<\; 4n
\end{aligned}
$$

(7) Follows from Part 6. (8) We construct $\mathcal{SH}$ using essentially the same algorithm found in [17]. The idea is to determine those nodes in the "middle" level of $\mathcal{SH}$, then find those nodes above the middle and below the middle recursively. As in [17] we use Tarjan's linear-time algorithm for finding SCCs. We first sort the edge-lengths and determine the $O(m\log n)$ possible $\textsc{norm}$-values in $O(m\log n)$ time. Let $\textsc{norm}_1 < \textsc{norm}_2 < \cdots < \textsc{norm}_k$ be the possible $\textsc{norm}$-values and G' be the input graph G restricted to edges with length less than $2\textsc{norm}_{\lfloor k/2\rfloor}$. We find the SCCs of G' in $O(m+n)$ time; let $\{C_i\}_i$ be the set of SCCs and G^c be derived from G by contracting the $\{C_i\}_i$ into single vertices. The $\{C_i\}_i$ correspond to $\mathcal{SH}$-nodes with $\textsc{norm}$-values equal to $\textsc{norm}_{k/2}$. We proceed recursively on the $\{C_i\}_i$ (finding $\mathcal{SH}$-nodes with $\textsc{norm}$-values in the range $\textsc{norm}_1..\textsc{norm}_{\lfloor k/2\rfloor-1}$) and on the graph G^c (for $\textsc{norm}$-values in the range $\textsc{norm}_{\lfloor k/2\rfloor+1}..\textsc{norm}_k$). There are $\log(m\log n) = O(\log n)$ levels of recursion and for each level the number of edges and vertices for subgraphs at that level is no more than m and $2n$, respectively. Therefore, the total time required is $O(m\log n)$. The procedure described above may find up to $O(n\log n)$ $\mathcal{SH}$-nodes, many of them irrelevant. We perform a final pass over $\mathcal{SH}$ tree, splicing out all irrelevant (one-child) nodes. This takes an additional $O(n\log n)$ time. $\square$

4.2 Computing SSSP

Recall from Section 2.2 that the D-value of a vertex is its tentative distance from the source s. We assign tentative distances to $\mathcal{SH}$-nodes as well as vertices. If $x\in\mathcal{SH}$ we set $D(x)$ to be:

$$D(x) \stackrel{\textbf{def}}{=} \min_{v\in C_x}\{D(v)\}$$

That is, the D-value of leaf node is the same as the D-value of its corresponding vertex.

We compute SSSP with a recursive algorithm called Visit, given in Figure 4. Visit takes two arguments: an $\mathcal{SH}$-node x and an interval I with the guarantee that $V(C_x)$ is (S,I)-independent, where S is the current set of visited vertices. Visit's only task is to visit the vertices in $V(C_x)^I$ and update the tentative distances, restoring Invariant 0. Using the Visit procedure, we can compute SSSP from source s as follows. Set $S := \emptyset$, $D(s) := 0$, $D(v) := \infty$ for all $v \neq s$, and call $\text{VISIT}(\text{ROOT}, [0, \infty))$, where $\text{ROOT} = \text{ROOT}(\mathcal{SH})$. Invariant 0 is clearly satisfied w.r.t. $S = \emptyset$, and $V(C_{\text{ROOT}})$ is clearly $(\emptyset, [0, \infty))$-independent, so the input guarantees for the initial call to VISIT are met. After the call to $\text{VISIT}(\text{ROOT}, [0, \infty))$, Invariant 0 will hold w.r.t. $S \supseteq V(C_{\text{ROOT}})^{[0,\infty)} = V$, implying $D(v) = d(s, v)$ for all $v \in S = V$.

$\text{VISIT}(x, [a, b))$

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

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

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

where t_x is set to

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

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

```
While a_x < b and V(C_x) ⊈ S
    While bucket [a_x, a_x + NORM(x)) is not empty
        Let y be the leftmost child of x in bucket [a_x, a_x + NORM(x))
        VISIT(y, [a_x, a_x + NORM(x)))
        Remove y from its bucket
        If V(C_y) ⊈ S, put y in bucket [a_x + NORM(x), a_x + 2 NORM(x))
    a_x := a_x + NORM(x)
```

Fig. 4. Visit Procedure

In each call to VISIT there are two cases, depending on whether x is a leaf node or an internal node of $\mathcal{SH}$. Suppose x is a leaf and $V(C_x) = \{v\}$. Because we maintain Invariant 0, deciding whether $v \in V(C_x)^I$ is equivalent to deciding if $D(v) \in I$, which is simple to do. In the general case x is an internal node. We determine $V(C_x)^I$ by making a series of recursive calls to children of x, using subintervals of I of width $\text{NORM}(x)$. The crucial property of $\mathcal{SH}$ that we use is that the children of x, in left-to-right order, represent a $\text{NORM}(x)$-partition of $V(C_x)$ — see Lemma 7(2). Together with Lemma 4 we are able to guarantee that each recursive call represents an independent subproblem.

To bound the number of recursive calls, it is important not to make too many trivial ones, that is, calls which cause no vertex to be visited. To that end we associate with x an array of buckets that will contain the children of x.

The buckets represent consecutive real intervals of width $\text{NORM}(x)$ and the bucket array represents an interval spanning $[d(s,x), d(s,x)+\text{DIAM}(C_x)]$ where $d(s,x) = d(s, V(C_x))$ is the distance to C_x. When $\text{VISIT}(x,\cdot)$ is called for the first time we choose a suitable starting point t_x and label each bucket with its associated interval: the i^{th} bucket is assigned the interval $[t_x + i\text{NORM}(x), t_x + (i+1)\text{NORM}(x))$. We frequently refer to buckets by their associated interval. We will choose t_x such that $t_x \leq d(s,x) < t_x + \text{NORM}(x)$. Therefore, at most $\lceil \frac{\text{DIAM}(C_x)}{\text{NORM}(x)} \rceil + 1$ buckets are required.

We will say x is *inactive* until $\text{VISIT}(x,\cdot)$ is called, and *active* afterward. We will assume, for the time being, that Invariant 1 is maintained.

Invariant 1 *Let x be an active $\mathcal{SH}$-node. A child y of x appears in one of x's buckets, unless $D(y) = \infty$ or $V(C_y) \subseteq S$, in which case y appears in no bucket. Every node y appearing in bucket $[a, a + \text{NORM}(x))$ is either an inactive child such that $D(y) \in [a, a + \text{NORM}(x))$, or an active child such that $V(C_y)^{[0,a)} \subseteq S$, but $V(C_y)^{[a,a+\text{NORM}(x))} \not\subseteq S$.*

4.3 Correctness of VISIT

In this section we prove that VISIT works correctly. Specifically, we show that $\text{VISIT}(x, I)$ visits (adds to the set S) all vertices in $V(C_x)^I$. We assume that Invariants 0 and 1 are magically updated behind the scenes. That is, adding a vertex to S causes the D-values of all vertices and $\mathcal{SH}$-nodes to be updated, restoring Invariant 0, and some number of $\mathcal{SH}$-nodes to be moved to different buckets, restoring Invariant 1.

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

Lemma 8 *In any two calls $\text{VISIT}(y, I_1)$ and $\text{VISIT}(y, I_2)$, $|I_1| = |I_2| = \text{NORM}(x)$, where x is the parent of y in $\mathcal{SH}$.*

PROOF. The node x only makes recursive calls on its children and all recursive calls from x are given an interval of width $\text{NORM}(x)$. $\square$

Lemma 9 *If $\text{VISIT}(x, I)$ is the first call to an $\mathcal{SH}$-node x, then we have $D(x) = d(s,x) \in I$.*

PROOF. The lemma clearly holds for the initial call $\text{VISIT}(\text{ROOT}, [0,\infty))$.

For the general case, let z be the parent of x. Before the recursive call $\text{VISIT}(x, I)$, where $I = [a, b)$, x must have been in z's bucket spanning the interval I. Since x was inactive, by Invariant 1 $D(x) \in I$. The equality $D(x) = d(s, x)$ follows from the (S, I)-independence of $V(C_x)$ and the inclusion $V(C_x)^{[0,a)} \subseteq S$. $\square$

Lemma 10 *Consider the variables a_x and b in any call to $\text{VISIT}(x, [a, b))$. Either $\text{NORM}(x)$ divides $b - a_x$ or $V(C_x)^{[0,b)} = V(C_x)$.*

PROOF. In the first call to $\text{VISIT}(x, [a, b))$, a_x is set to t_x. Suppose that $t_x = D(x)$, because $D(x) + \text{DIAM}(x) < b$. By Lemma 9, $D(x) = d(s, x)$, implying that $V(C_x)^{[0,b)} = V(C_x)$. If, on the other hand, t_x is set to $b - \text{NORM}(x)\lceil \frac{b-D(x)}{\text{NORM}(x)} \rceil$, then $\text{NORM}(x)$ divides $b - t_x$ and, at least initially, $b - a_x$ as well. Since a_x is only incremented in units of $\text{NORM}(x)$, $b - a_x$ remains divisible by $\text{NORM}(x)$. We have proved the lemma for the *first* recursive call on x.

Now suppose that $\text{VISIT}(x, [a, b))$ is *not* the first recursive call on x, and therefore we set $a_x := a$ initially. Let z be the parent of x. According to Lemma 7(3) either $\text{NORM}(x)$ divides $\text{NORM}(z)$ or $\text{DIAM}(x) < \text{NORM}(z)$. Suppose $\text{NORM}(x)$ divides $\text{NORM}(z)$. By Lemma 8, $\text{NORM}(z) = b - a$ and therefore $\text{NORM}(x)$ divides $b - a_x$ initially, and, with the observation that a_x is incremented in units of $\text{NORM}(x)$, ever after. Now suppose $\text{DIAM}(x) < \text{NORM}(z)$. Since this is not the first recursive call on x, we know, by Lemma 9, that $d(s, x) < a$ and therefore that $d(s, x) + \text{DIAM}(C_x) < b$, meaning $V(C_x)^{[0,b)} = V(C_x)$. $\square$

Lemma 11 *After the call to $\text{VISIT}(x, [a, b))$, $V(C_x)^{[a,b)} \subseteq S$.*

PROOF. We assume inductively that $V(C_x)$ is $(S, [a, b))$-independent when $\text{VISIT}(x, [a, b))$ is called. This clearly holds for the first recursive call, when $x = \text{ROOT}$, $[a, b) = [0, \infty)$, and $S = \emptyset$.

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

Suppose, now that x is an internal node in $\mathcal{SH}$. We will prove that each time through the outer while loop in Step 3 of VISIT, $V(C_x)^{[0,a_x)} \subseteq S$ and $V(C_x)$ is $(S, [a_x, b))$-independent w.r.t. the current values for a_x and S. Consider the first time through the outer while loop in the call to $\text{VISIT}(x, [a, b))$. If a_x is initially set to a then $V(C_x)$ is $(S, [a_x, b))$-independent and $V(C_x)^{[0,a)} \subseteq S$ by our inductive assumptions. If a_x is initially set to t_x then $a_x = t_x \leq D(x)$. Lemma 9 states that $D(x) = d(s, x)$, implying that $V(C_x)$ is $(S, [a_x, b))$-independent since $V(C_x)^{[a,a_x)} = \emptyset$.

After each iteration through the outer while loop we increment a_x by $\textsc{norm}(x)$. Therefore, the effect of each iteration must be to visit all vertices in $V(C_x)^I$, where $I = [a_x, a_x + \textsc{norm}(x))$. This is exactly what the recursive calls in the inner while loop accomplish. Imagine that we consider *all* the children of x, $\{x_j\}_j$, one at a time in left-to-right order. We will show two things: first, that when x_j is considered $V(C_{x_j})$ is (S, I)-independent for the current value of S. Therefore, *if* the recursive call $\textsc{Visit}(x_j, I)$ is made, we can assume inductively that it visits all vertices in $V(C_{x_j})^I$. Second, if no recursive call is made on x_j (meaning x_j does not appear in the bucket labeled I) then $V(C_{x_j})^I - S = \emptyset$. This will establish the correctness of the inner while loop.

Consider the claim that when x_j is considered $V(C_{x_j})$ is (S, I)-independent. Let S' be the set S just before this iteration of the outer while loop, and assume inductively that when x_j is considered $S = S' \cup V(C_{x_1})^I \cup \cdots \cup V(C_{x_{j-1}})^I$. Lemma 7(2) states that $(V(C_{x_i}))_i$ is a $\textsc{norm}(x)$-partition of $V(C_x)$. Together with the assumption that $V(C_x)$ is $(S', [a_x, b))$-independent and Lemma 4(1), we have that $V(C_{x_j})$ is $(S, [a_x, \min\{a_x + \textsc{norm}(x),\ b\}))$-independent. However, we need to show that it is (S, I)-independent, since it is the interval $I = [a_x, a_x + \textsc{norm}(x))$ that would be passed to the recursive call. By Lemma 10, either $\textsc{norm}(x)$ divides $b - a_x$ or $V(C_x)^{[0,b)} = V(C_x)$. If $\textsc{norm}(x)$ divides $b - a_x$ then $I = [a_x, \min\{a_x + \textsc{norm}(x), b\})$ since we only entered the outer while loop if $a_x < b$, implying $a_x \le b - \textsc{norm}(x)$. On the other hand, if $V(C_x)^{[0,b)} = V(C_x)$, then $V(C_{x_j})$ being $(S, [a_x, \min\{a_x + \textsc{norm}(x), b\}))$-independent implies that it is (S, I)-independent as well, since $V(C_x)^{[b, a_x + \textsc{norm}(x))} = \emptyset$. To complete the induction we must show that *after* x_j is considered, $S = S' \cup V(C_{x_1})^I \cup \cdots \cup V(C_{x_j})^I$. If we perform the recursive call $\textsc{Visit}(x_j, I)$ then we can assume inductively that vertices in $V(C_{x_j})^I$ are visited. Therefore, we must only prove that if no such recursive call is made, then $V(C_{x_j})^I - S = \emptyset$. We perform recursive calls on all children that end up in bucket I. By Invariant 1, if x_j is not in bucket I when it is considered, then $D(x_j) \ge a_x + \textsc{norm}(x)$ or $V(C_{x_j}) \subseteq S$. By the (S, I)-independence of $V(C_{x_j})$, $D(x_j) \ge a_x + \textsc{norm}(x)$ implies $V(C_{x_j})^I = \emptyset$. The other case, $V(C_{x_j}) \subseteq S$, clearly implies $V(C_{x_j})^I - S = \emptyset$. This completes the induction for the inner and outer while loops.

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

The proof of Lemma 11 is rather intricate because there are four distinct inductions, with four base cases. We assume inductively that $V(C_x)$ is $(S, [a, b))$-independent — this is an induction over time, for which the initial call to $\textsc{Visit}(\textsc{root}, [0, \infty))$ is the base case. We assume that recursive calls made in Step 3 visit the right vertices — this is an induction over problem size, where the leaves of $\mathcal{SH}$ form the base cases. Finally, there is a double induction over the two while loops in Step 3.

4.4 *Implementation of* VISIT

In this section we address the details of an implementation of the VISIT routine. The main difficulties are devising data structures to maintain (or simulate) Invariants 0 and 1. Ignoring data structural issues for the moment, we can show that the other costs of computing SSSP with VISIT are linear in n.

Lemma 12 *For each SSSP computation, the total number of recursive calls to* VISIT *is less than 5n.*

PROOF. By Lemma 9, if $\textsc{Visit}(x, I)$ is the first recursive call on x, then $D(x) = d(s, x) \in I$. Together with Invariant 1 and Lemma 8, this implies that each node $x \in \mathcal{SH}$ is passed to at most $\lceil \frac{\textsc{diam}(C_x)}{\textsc{norm}(p(x))} \rceil + 1$ recursive calls, where $p(x)$ is the parent of x in $\mathcal{SH}$. The total number of recursive calls is then

$$\begin{aligned}
\sum_x \left\lceil \frac{\textsc{diam}(C_x)}{\textsc{norm}(p(x))} \right\rceil + 1 \quad &\leq \quad |\mathcal{SH}| + \sum_x \left\lceil \frac{\textsc{diam}(C_x)}{2\textsc{norm}(x)} \right\rceil && (3)\\
&< \quad |\mathcal{SH}| + n - 1 + \tfrac{1}{2} \cdot \sum_x \frac{\textsc{diam}(C_x)}{\textsc{norm}(x)} && (4)\\
&< \quad 5n && (5)
\end{aligned}$$

Line 3 follows from the inequality $\textsc{norm}(p(x)) \geq 2\textsc{norm}(x)$. Line 4 follows since $\lceil \frac{\textsc{diam}(C_x)}{\textsc{norm}(x)} \rceil$ is only strictly greater than $\frac{\textsc{diam}(C_x)}{\textsc{norm}(x)}$ if x is an internal node of $\mathcal{SH}$, of which there are no more than $n - 1$. (If x were a leaf, then $\textsc{diam}(C_x) = 0$.) Line 5 follows from the bounds $|\mathcal{SH}| < 2n$ and, by Lemma 7(6), $\sum_x \frac{\textsc{diam}(C_x)}{\textsc{norm}(x)} < 4n$. $\square$

Lemma 13 *The total time required to find $\{t_x\}_{x \in \mathcal{SH}}$ is $O(n)$.*

PROOF. In Step 2 of VISIT, t_x is set to $D(x)$ if $D(x) + \textsc{diam}(C_x) < b$ and $b - \textsc{norm}(x)\lceil \frac{b - D(x)}{\textsc{norm}(x)} \rceil$ otherwise. Checking whether $D(x) + \textsc{diam}(C_x) < b$ takes $O(1)$ time, and computing $b - \textsc{norm}(x)\lceil \frac{b-D(x)}{\textsc{norm}(x)} \rceil$ takes $O(\frac{b-D(x)}{\textsc{norm}(x)})$ time: one simply counts back from b in units of $\textsc{norm}(x)$ in order to find $\min\{j : b - j \cdot \textsc{norm}(x) \leq D(x)\}$. Given that $b - D(x) \leq \textsc{diam}(C_x)$, the total time to find all $\{t_x\}_{x \in \mathcal{SH}}$ is $\sum_x O(\frac{\textsc{diam}(C_x)}{\textsc{norm}(x)})$, which is $O(n)$ by Lemma 7(6). $\square$

We support an implementation of VISIT with two abstract data structures, denoted $\mathcal{D}$ and $\mathcal{B}$. $\mathcal{D}$ updates the D-values (tentative distances) of $\mathcal{SH}$-nodes as dictated by Invariant 0, and $\mathcal{B}$ maintains the bucket arrays of active $\mathcal{SH}$-nodes in accordance with Invariant 1. Although it is typical to assume that data structures do not talk to each other, it is conceptually simpler here to think of $\mathcal{D}$ and $\mathcal{B}$ making queries to each other. We describe their interactions below, then bound their complexity.

When an edge (u, v) is relaxed in Step 1 of VISIT, we tell $\mathcal{D}$ to set $D(v) := \min\{D(v), D(u) + \ell(u, v)\}$. If this decreases $D(v)$ then it may decrease the D-values of many ancestors of v in $\mathcal{SH}$ as well. Let y be the unique ancestor of v which is an inactive child of an active node. If $D(y)$ is also decreased then to restore Invariant 1 y may have to be moved to a different bucket. If this is the case then $\mathcal{D}$ notifies $\mathcal{B}$ that $D(y)$ has changed. $\mathcal{D}$ also accepts *queries* to D-values. In particular, when an $\mathcal{SH}$-node x becomes active $\mathcal{B}$ files each child y of x in its bucket array based on the value of $D(y)$. The bucketing structure $\mathcal{B}$ must also fulfill the needs of VISIT. Specifically, in a call to $\text{VISIT}(x, \cdot)$, VISIT repeatedly requests the leftmost child of x in the current bucket labeled $[a_x,\, a_x + \text{NORM}(x))$, and possibly moves that node to the next bucket, labeled $[a_x + \text{NORM}(x),\, a_x + 2\,\text{NORM}(x))$. Lemmas 14 and 15 bound the complexities of $\mathcal{D}$ and $\mathcal{B}$, respectively.

Lemma 14 *$\mathcal{D}$ can be implemented to run in $O(m \log \alpha(m, n))$ time.*

See [21] for a proof of Lemma 14. It is a slight improvement over a bound of $O(m\alpha(m, n))$ proved by Gabow [3]. Since managing D-values is not the bottleneck in our algorithm, we can afford to implement $\mathcal{D}$ in $O(m+n \log\log n)$ time using a simplified version of the structure from [3].

Lemma 15 *(see [21,17]) Suppose $\mathcal{B}$ is assigned to maintain the bucket arrays of nodes in $X \subseteq \mathcal{SH}$. Then $\mathcal{B}$ can be implemented in time*

$$O\left(m \,+\, n \log\log n \,+\, \sum_{x \in X} |V(C_x^c)| \cdot \log \frac{\text{DIAM}(C_x)}{\text{NORM}(x)}\right)$$

PROOF. (Sketch) Consider a single $\mathcal{SH}$-node $x \in X$ and an arbitrary child y of x. We insert y into x's bucketing structure upon the activation of x, and perform some number of decrease-key operations on y as $D(y)$ is decreased. Using the lazy bucketing data structure from [21], each insert takes time logarithmic in the number of buckets, which is $\frac{\text{DIAM}(C_x)}{\text{NORM}(x)}$, and each decrease-key takes constant time (all times amortized). These costs correspond to the first and third terms in the claimed running time. The second term reflects the cost of extracting nodes from the current bucket in left-to-right order — see Step 3 of VISIT. As in [17], we use a van Emde Boas heap [34] for this task; the cost per child of x is $O(\log\log|V(C_x^c)|)$, which is $O(n\log\log n)$ overall. $\square$

The third term in the bound of Lemma 15 reflects a definite limitation of the hierarchy approach. Each child y of x can, in principle, appear in any of x's $\textsc{diam}(C_x)/\textsc{norm}(x)$ buckets (and Theorem 1 shows that where y is bucketed is essentially independent of where other children of x are bucketed.) Therefore, to visit x's children in the proper order requires us to extract $|V(C_x^c)|\cdot\log(\textsc{diam}(C_x)/\textsc{norm}(x))$ bits of information, which requires at least that many comparisons. In the case when X consists of all $\mathcal{SH}$-nodes it is not difficult to make the bound of Lemma 15 as bad as $\Omega(m+n\log n)$. However, Lemma 15 is still useful. In Section 5 we will apply it to the set of all low-diameter $\mathcal{SH}$-nodes. For the case when X consists of all $x\in\mathcal{SH}$ such that $\frac{\textsc{diam}(C_x)}{\textsc{norm}(x)}<(\log n)^{O(1)}$, the bound from Lemma 15 becomes $O(m+n\log\log n)$. This allows us the freedom to focus only on large-diameter $\mathcal{SH}$-nodes.

In Section 5 we give a scheme to reduce the bucketing costs of $\mathcal{B}$ to only $O(m+n\log\log n)$ per SSSP computation, provided that we amortize the costs over n such computations.

5 A Faster APSP Algorithm

In this section we present an implementation of the $\textsc{Visit}$ algorithm from Section 4 that computes APSP in $O(mn+n^2\log\log n)$ time. The algorithm is structured around two observations. The first is that $\textsc{Visit}$ can be speeded up if it is supplied with some useful hints, specifically, some discrete (i.e. integral) approximations to certain real quantities related to shortest paths. The second observation is that these discrete approximations are relatively cheap to compute — provided they are computed in bulk.

Before delving into the details of the algorithm, let us first make explicit some notational conventions. Throughout the section x and y are $\mathcal{SH}$-nodes, with y the child of x. Graph vertices are represented by u or v, and $d(u,x)=d(u,V(C_x))$ represents the minimum distance from u to any vertex in C_x. (So if $u\in V(C_x)$ then $d(u,x)=0$.) A hat over a symbol, such as $\hat{\Delta}$, indicates that it is an integer-valued approximation to its unhatted, real-valued twin, in this case Δ. See Section 2.1 for the distinction between real and integer variables.

5.1 Relative Distances and Their Approximations

Recall that the vertices of $V(C_x^c)$ represent the children of x in $\mathcal{SH}$. Define $\Delta_x : V(G) \times V(C_x^c) \to \mathbb{R}$ as

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

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

$$\hat{\Delta}_x(u,y) \stackrel{\textbf{def}}{=} \left\lfloor \frac{\Delta_x(u,y)}{\epsilon_x} \right\rfloor \quad \text{or} \quad \left\lceil \frac{\Delta_x(u,y)}{\epsilon_x} \right\rceil \quad \text{where} \quad \epsilon_x \stackrel{\textbf{def}}{=} \tfrac{\textsc{norm}(x)}{2}$$

It is crucial that $\hat{\Delta}_x$ be represented as an integer, *not* as a real. Lemma 16 and 17 capture the salient features of the $\hat{\Delta}$ function: that it is relatively cheap to compute, and that despite its approximate nature, it is useful for computing exact shortest paths.

Lemma 16 *The $\hat{\Delta}_x$ function can be computed for every $\mathcal{SH}$ node x for which $\frac{\textsc{diam}(C_x)}{\textsc{norm}(x)} \geq \log n$, in $O(mn)$ time total.*

Lemma 17 *If $\hat{\Delta}_x$ is known for all $x \in \mathcal{SH}$ for which $\frac{\textsc{diam}(C_x)}{\textsc{norm}(x)} \geq \log n$, then SSSP can be computed in $O(m + n\log\log n)$ time.*

Together with Lemma 7(8), stating that $\mathcal{SH}$ can be constructed in $O(m \log n)$ time, Lemmas 16 and 17 directly imply our main theorem.

Theorem 18 *The all-pairs shortest path problem on real-weighted directed graphs can be solved in $O(mn + n^2 \log\log n)$ time, where the only operations allowed on reals are comparisons and additions.*

We prove Lemma 17 in Section 5.2. Lemma 16 is addressed in Section 5.3.

5.2 The Algorithm

We show how to implement the bucketing structure $\mathcal{B}$ used in Visit, assuming that $\hat{\Delta}_x$ is already computed for all $x \in \mathcal{SH}$ for which $\textsc{diam}(C_x)/\textsc{norm}(x) \geq \log n$. The remainder of this section will constitute a proof of Lemma 17. As it was observed in Section 4.4, managing the bucket arrays for all $\mathcal{SH}$-nodes x with $\textsc{diam}(x) \leq \log n \cdot \textsc{norm}(x)$ requires $O(m + n \log\log n)$ time by Lemma

15. Therefore, we concentrate on an arbitrary $\mathcal{SH}$-node x for the case when $\hat{\Delta}_x$ is known.

Recall from Section 4.3 that we assumed Invariant 1 was maintained at all times. Consider the following weakened form of Invariant 1:

Invariant 2 *Suppose that y is a child of an active $\mathcal{SH}$-node x. Then y is either bucketed in accordance with Invariant 1, or it is known that $D(y)$ will decrease in the future, in which case y appears in no bucket.*

As a matter of correctness Invariant 2 is just as good as Invariant 1. By Lemma 9, VISIT only extracts a node y from a bucket array if its D-value is finalized, that is, if $D(y) = d(s, y)$. The only question is whether it is possible to tell if a node's D-value will decrease in the future. This is where the $\hat{\Delta}$ function comes into play.

Suppose that we are attempting to bucket an inactive node y by its D-value, either because its parent, x, just became active, or because we just relaxed an edge (u, v), where $v \in V(C_y)$. We know $d(s, x)$ lies in the interval of x's first bucket, that is, $t_x \le d(s, x) < t_x + \text{NORM}(x)$. According to Invariant 1, y belongs in bucket number

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

Therefore, if $D(y)$ does not decrease in the future, then $D(y) = d(s, y)$ and $\Delta_x(s, y) = D(y) - d(s, x)$. This implies that y must be bucketed in either bucket number $\lfloor \frac{\Delta_x(s,y)}{\text{NORM}(x)} \rfloor$ or the following bucket. On the other hand, if $D(y)$ decreases in the future, we have, according to Invariant 2, the freedom not to bucket y at all.

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

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

Thus, the number of eligible buckets is at most three. Since $\hat{\Delta}_x(s, y)$ is represented as an integer, we can identify the three eligible buckets in constant time, and, by checking $D(y)$ against the buckets' labels, we can determine which, if any, should contain y.

In order to implement VISIT we require not only that nodes be bucketed properly, but that they be extracted in the correct left-to-right order. Following Hagerup [17], we prioritize nodes in the same bucket using a van Emde Boas queue [34]. The overhead for using this structure is $O(n \log\log n)$ per SSSP computation. By Lemmas 12, 13, and 14, the other costs of implementing VISIT are $O(m \log \alpha(m,n)) = O(m + n \log\log n)$ per SSSP computation. This concludes the proof of Lemma 17.

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

We show in this section that for any $\mathcal{SH}$ node x, all $\hat{\Delta}_x(\cdot,\cdot)$-values can be computed in $O(m \log n + m|V(C_x^c)| + n\frac{\text{DIAM}(C_x)}{\text{NORM}(x)})$ time. It turns out that this cost is affordable if the $m \log n$ term is not significantly larger than the others. It is for this reason that Lemma 16 only considers $\mathcal{SH}$ nodes x such that $\text{DIAM}(C_x)/\text{NORM}(x) \geq \log n$.

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

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

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

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

We let $G^\delta = (V(G), E(G), \delta)$ denote the graph G under a new length function δ, and let d^δ be the distance function for G^δ. We show that $\Delta_x(u,y)$ is equal to $d^{\delta_x}(u,y)$ and that $d^{\hat{\delta}_x}$ provides a sufficiently good approximation to Δ_x to satisfy the constraints put on $\hat{\Delta}_x$. Our method for computing $\hat{\Delta}_x$ is given in Figure 5. We spend the remainder of this section analyzing its complexity and proving its correctness.

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

COMPUTE-$\hat{\Delta}_x$:

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

(2) For all $u \in V(G)$ and $y \in V(C_x^c)$, compute $d^{\hat{\delta}_x}(u, y)$

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

Fig. 5. A three-step method for computing $\hat{\Delta}_x$.

Lemma 19 *Suppose $x \in \mathcal{SH}$, $y \in V(C_x^c)$ and $u \in V$. Then*

(1) $\Delta_x(u, y) = d^{\delta_x}(u, y)$

(2) $d^{\delta_x}(u, y) \le \textsc{diam}(C_x)$

(3) $d^{\delta_x}(u, y) - \epsilon'_x \cdot d^{\hat{\delta}_x}(u, y) \in [0, \epsilon_x)$

(4) $d^{\hat{\delta}_x}(u, y) < 2n \frac{\textsc{diam}(C_x)}{\textsc{norm}(x)}$

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

$$d^{\delta_x}(u, y) = \min_{j, \langle u = u_1, \ldots, u_j \in C_y \rangle} \left\{ \sum_{i=1}^{j-1} \delta_x(u_i, u_{i+1}) \right\} \tag{6}$$

$$= \min_{j, \langle u = u_1, \ldots, u_j \in C_y \rangle} \left\{ \ell(\langle u_1, \ldots, u_j \rangle) + d(u_j, x) - d(u_1, x) \right\} \tag{7}$$

$$= d(u, y) - d(u, x) = \Delta_x(u, y) \tag{8}$$

Line 6 is simply the definition of d^{δ_x}. Line 7 is derived by cancelling terms in the telescoping sum. Note that $d(u_j, x) = 0$ since $u_j \in C_y \subseteq C_x$, and that $d(u_1, x) = d(u, x)$. Line 8 then follows from the definition of d and Δ_x.

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

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

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

The last inequality follows from the bound $|P_{uy}| < n$ and the definition of

$\epsilon_x = n \cdot \epsilon'_x$. This proves part (3).

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

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

which proves part (4). □

Lemma 20 bounds the time to compute the $\hat{\delta}_x$ function in Step 1.

Lemma 20 *$G^{\hat{\delta}_x}$ is computable in $O(m \log n)$ time.*

PROOF. Let (u,v) be an arbitrary edge. Recall that $\hat{\delta}_x(u,v)$ is either ∞ or $\lfloor \frac{\ell(u,v)+d(v,x)-d(u,x)}{\epsilon'_x} \rfloor$. The original length function ℓ is, of course, already known. We compute the other terms in the numerator with one Dijkstra computation. Let G_1 be derived from G by reversing the direction of all edges and contracting C_x into a vertex called x. Computing SSSP from the source x in G_1 produces the $d(\cdot,x)$ distances. This takes $O(m + n\log n)$ time with Fibonacci heaps. However, we can afford to spend $O(m \log n)$ time using a simpler binary heap.

If $\hat{\delta}_x(u,v) \neq \infty$, which can be checked in constant time, then from the definition of $\epsilon'_x = \frac{\textsc{norm}(x)}{2n}$,

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

The terms $2n \cdot d(u,x)$ and $2n \cdot (\ell(u,v) + d(v,x))$ are easily computable in $O(\log n)$ time — see Section 2.1. We compute $\hat{\delta}_x(u,v)$ in $O(\log \frac{\textsc{diam}(C_x)}{\epsilon'_x}) = O(\log n)$ time by first generating the values

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

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

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

Lemma 21 *Step 2 requires* $O(m \cdot |V(C_x^c)| + n \cdot \frac{\text{DIAM}(C_x)}{\text{NORM}(x)})$ *time.*

PROOF. Let $y \in V(C_x^c)$ be a child of x and let N denote an upper bound on $d^{\hat{\delta}_x}(u, y)$. Let G_1 be the graph derived from $G^{\hat{\delta}_x}$ by reversing the direction of all edges in G. Clearly $d^{\hat{\delta}_x}(u, y)$ is equal to the distance from C_y to u in G_1. Therefore, we can perform Step 2 of COMPUTE-$\hat{\Delta}_x$ by computing SSSP in G_1 from the source C_y (viewed as a single vertex), for each $y \in V(C_x^c)$. To save time we solve each of these $|V(C_x^c)|$ SSSP problems simultaneously, using Dial's implementation of Dijkstra's algorithm. The priority queue is implemented as a bucket array of length N. If the pair $\langle y, u\rangle$ appears in bucket b this indicates that in the SSSP computation with source y, the tentative distance to u is b. Since $\hat{\delta}_x$ is an integer-valued function, edge relaxations take constant time. The overall running time is then $O(\#(\text{edge relaxations}) + \#(\text{buckets scanned})) = O(m \cdot |V(C_x^c)| + N) = O\left(m \cdot |V(C_x^c)| \, + \, n \cdot \frac{\text{DIAM}(C_x)}{\text{NORM}(x)}\right)$. The bound on N follows from Lemma 19(4). $\square$

Lemmas 20 and 21 prove that Steps 1 and 2 take $O(m \log n + m\,|V(C_x^c)| + n\,\frac{\text{DIAM}(C_x)}{\text{NORM}(x)})$ time. Step 3 just involves dividing $d^{\hat{\delta}_x}(u, y)$ by n and rounding up. We did not assume a general integer division operation. However, Step 3 can easily be incorporated into Step 2 by keeping track of the number $\lceil \frac{b}{n} \rceil$ where b is the current bucket number. In Lemma 22 we prove the correctness of COMPUTE-$\hat{\Delta}_x$.

Lemma 22 *Step 3 sets* $\hat{\Delta}_x$ *correctly, i.e.*

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

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

$$\epsilon'_x \cdot d^{\hat{\delta}_x}(u, y) \quad \leq \quad \epsilon_x \cdot \left\lceil \frac{d^{\hat{\delta}_x}(u, y)}{n} \right\rceil \quad < \quad \epsilon'_x \cdot d^{\hat{\delta}_x}(u, y) + \epsilon_x \tag{9}$$

From Lemma 19 parts (1) and (3) we have that:

$$\epsilon'_x \cdot d^{\hat{\delta}_x}(u, y) \quad \leq \quad \Delta_x(u, y) \; = \; d^{\delta_x}(u, y) \quad < \quad \epsilon'_x \cdot d^{\hat{\delta}_x}(u, y) + \epsilon_x \tag{10}$$

Notice that in lines 9 and 10 the upper and lower bounds are identical, and that they are separated from each other by ϵ_x. Therefore,

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

which proves the lemma. □

Now that the correctness of this scheme is established, we are ready to prove the overall time bound of Lemma 16.

PROOF (Lemma 16). Let $T(m,n,k)$ be the time to compute $\hat{\Delta}_x$ for all $\mathcal{SH}$ nodes x for which $\frac{\text{DIAM}(C_x)}{\text{NORM}(x)} \geq k$. From Lemmas 20 and 21 can bound T as follows.

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

hence $T(m,n,\log n) \;=\; O(mn)$ □

6 Discussion

For simplicity, the hierarchy-type shortest path algorithms [16,17,35,25] are usually described as solving either SSSP or APSP. These two extremes, however, obscure the nature of these algorithms. If we think of them as solving the s-sources shortest paths problem, they all have running times of the form $\mathcal{T}(s,m,n) \;=\; s \cdot \mathcal{M} + \mathcal{R}$, where $\mathcal{R}$ is a one-time cost and $\mathcal{M}$ is the marginal cost of one SSSP computation. For our algorithm $\mathcal{R} = O(mn)$ and $\mathcal{M} = O(m + n \log\log n)$. Figure 6 compares the hierarchy-based algorithms from the perspective of their $\mathcal{M}$ and $\mathcal{R}$ complexities.

We believe that it is possible to reduce the marginal cost $\mathcal{M}$ in both our algorithm and Hagerup's [17], though it will be more difficult in our case because the $\log\log n$ bottleneck appears in two places in the algorithm. Reducing the complexity of $\mathcal{M}$ to $O(m)$ in our algorithm and [21,25] *seems* tantamount to

Paper	Notes	$\mathcal{M}$	$\mathcal{R}$
Thorup [16]	integer lengths, undirected graphs	$O(m)$	$O(m)$
Hagerup [17]	integer lengths	$O(m + n \log\log n)$	$O\Big(\min \left\{ \begin{array}{c} m \log\log C, \\ m \log n \end{array} \right\}\Big)$
PR [35]	real lengths, undirected graphs	$O(m \log \alpha(m,n))$	$O\Big(MST + \min \left\{ \begin{array}{c} n \log\log r, \\ n \log n \end{array} \right\}\Big)$
Pettie [25]	real lengths, comp-add compl. (non-uniform)	$O(m \log \alpha(m,n))$	$O(mn)$
This paper	real lengths	$O(m + n \log\log n)$	$O(mn)$

Fig. 6. C is the largest integer edge length, r is the ratio of the maximum-to-minimum edge length, α is the inverse-Ackermann function, and $MST = MST(m,n)$ is the decision-tree complexity of the minimum spanning tree problem [36] (known to be between $\Omega(m)$ and $O(m\alpha)$ [37]). The bound on $\mathcal{R}$ for Hagerup algorithm is slightly stronger than given in [17], as is the bound on $\mathcal{M}$ for [35]. The bounds on $\mathcal{M}$ and $\mathcal{R}$ for Pettie [25] count only comparisons and additions; [25] does not currently admit a fast implementation.

finding a linear-time split-findmin algorithm [3,21].[5] The author has speculated recently [38] that $\Theta(m \log \alpha(m,n))$ may be the actual complexity of split-findmin.

Although reducing $\mathcal{M}$ in previous algorithms is certainly a worthy goal, we think it would be more challenging and interesting to work in other directions. For instance, is there a feasible relaxation of Property 1 that does not have an inherent sorting bottleneck? Can the hierarchy-based approach yield $O(n^2 + o(mn))$-time APSP algorithms? Finally, in our algorithm, can $\mathcal{M}$ be maintained while reducing $\mathcal{R}$ to something more reasonable, say, $O(m \cdot \text{polylog}(n))$? Answering these questions will probably require an insight into the shortest path problem, not just an interesting new data structure.

Acknowledgements

We would like to thank Vijaya Ramachandran and the two anonymous referees. Their comments greatly improved the presentation of this paper.

[5] Thorup & Hagerup [16,17] get around this issue by implementing split-findmin with RAM-based integer sorting.

References

[1] E. W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1 (1959) 269–271.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms., MIT Press, 2001.

[3] H. N. Gabow, A scaling algorithm for weighted matching on general graphs, in: Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS'85), 1985, pp. 90–100.

[4] H. N. Gabow, R. E. Tarjan, Faster scaling algorithms for network problems, SIAM J. Comput. 18 (5) (1989) 1013–1036.

[5] A. V. Goldberg, Scaling algorithms for the shortest paths problem, SIAM J. Comput. 24 (3) (1995) 494–504.

[6] R. Seidel, On the all-pairs-shortest-path problem in unweighted undirected graphs, J. Comput. Syst. Sci. 51 (3) (1995) 400–403.

[7] Z. Galil, O. Margalit, All pairs shortest distances for graphs with small integer length edges, Inform. and Comput. 134 (2) (1997) 103–139.

[8] A. Shoshan, U. Zwick, All pairs shortest paths in undirected graphs with integer weights, in: 40th Annual Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society Press, 1999, pp. 605–614.

[9] N. Alon, Z. Galil, O. Margalit, On the exponent of the all pairs shortest path problem, J. Comput. Syst. Sci. 54 (2, part 1) (1997) 255–262.

[10] U. Zwick, All pairs shortest paths using bridging sets and rectangular matrix multiplication, J. ACM 49 (3) (2002) 289–317.

[11] Y. Han, Improved fast integer sorting in linear space, in: Proceedings 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2001, pp. 793–796.

[12] Y. Han, Deterministic sorting in $O(n \log \log n)$ time and linear space, in: 34th Ann. ACM symp. on Theory of comput., ACM Press, 2002, pp. 602–608.

[13] Y. Han, M. Thorup, Integer sorting in $O(n\sqrt{\log \log n})$ expected time and linear space, in: 43rd Ann. Symp. on Found. of Comput. Sci., 2002, pp. 135–144.

[14] M. Thorup, Equivalence between priority queues and sorting, in: 43rd Ann. Symp. on Found. of Comput. Sci., 2002, pp. 125–134.

[15] M. Thorup, Integer priority queues with decrease key in constant time and the single source shortest paths problem, in: 35th Ann. ACM Symp. on Theory of Comput., 2003, pp. ??–??

[16] M. Thorup, Undirected single-source shortest paths with positive integer weights in linear time, J. ACM 46 (3) (1999) 362–394.

[17] T. Hagerup, Improved shortest paths on the word RAM, in: Proc. 27th Int'l Colloq. on Automata, Languages, and Programming (ICALP'00), LNCS vol. 1853, 2000, pp. 61–72.

[18] U. Zwick, Exact and approximate distances in graphs – a survey, in: Proceedings 9th European Symposium on Algorithms (ESA), 2001, pp. 33–48.

[19] M. L. Fredman, R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithm, J. ACM 34 (3) (1987) 596–615.

[20] D. B. Johnson, Efficient algorithms for shortest paths in sparse networks, J. ACM 24 (1) (1977) 1–13.

[21] S. Pettie, V. Ramachandran, A shortest path algorithm for real-weighted undirected graphs, submitted. (A preliminary version appeared in 13th ACM-SIAM Symp. on Discrete Algs. (2002), 267–276).

[22] R. E. Tarjan, A class of algorithms which require nonlinear time to maintain disjoint sets, J. Comput. Syst. Sci. 18 (2) (1979) 110–127.

[23] M. L. Fredman, New bounds on the complexity of the shortest path problem, SIAM J. Comput. 5 (1) (1976) 83–89.

[24] T. Takaoka, A new upper bound on the complexity of the all pairs shortest path problem, Info. Proc. Lett. 43 (4) (1992) 195–199.

[25] S. Pettie, On the comparison-addition complexity of all-pairs shortest paths, in: Proc. 13th Int'l Symp. on Algorithms and Computation (ISAAC'02), 2002, pp. 32–43.

[26] S. Pettie, V. Ramachandran, S. Sridhar, Experimental evaluation of a new shortest path algorithm, in: 4th Workshop on Algorithm Engineering and Experiments (ALENEX), 2002, pp. 126–142.

[27] P. M. Spira, A. Pan, On finding and updating spanning trees and shortest paths, SIAM J. Comput. 4 (3) (1975) 375–380.

[28] D. R. Karger, D. Koller, S. J. Phillips, Finding the hidden path: Time bounds for all-pairs shortest paths, SIAM J. Comput. 22 (6) (1993) 1199–1217.

[29] L. R. Kerr, The effect of algebraic structure on the computational complexity of matrix multiplication, Technical Report TR70-75, Computer Science Dept., Cornell University (Jun. 1970).

[30] S. G. Kolliopoulos, C. Stein, Finding real-valued single-source shortest paths in $o(n^3)$ expected time, J. Algor. 28 (1) (1998) 125–141.

[31] R. L. Graham, A. C. Yao, F. F. Yao, Information bounds are weak in the shortest distance problem, J. ACM 27 (3) (1980) 428–444.

[32] E. A. Dinic, Finding shortest paths in a network, Transportation Modeling Systems (1978) 36–44.

[33] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, R. E. Tarjan, Faster algorithms for the shortest path problem, J. ACM 37 (2) (1990) 213–223.

[34] P. van Emde Boas, R. Kaas, E. Zijlstra, Design and implementation of an efficient priority queue, Math. Syst. Theory 10 (1977) 99–127.

[35] S. Pettie, V. Ramachandran, Computing shortest paths with comparisons and additions, in: Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02), 2002, pp. 267–276.

[36] S. Pettie, V. Ramachandran, An optimal minimum spanning tree algorithm, J. ACM 49 (1) (2002) 16–34.

[37] B. Chazelle, A minimum spanning tree slgorithm with inverse-Ackermann type complexity, J. ACM 47 (6) (2000) 1028–1047.

[38] S. Pettie, An inverse-Ackermann style lower bound for the online minimum spanning tree verification problem, in: Proc. 43rd Annual Symposium on Foundations of Computer Science (FOCS'02), 2002, pp. 155–163.