On the Comparison-Addition Complexity of All-Pairs Shortest Paths*

Seth Pettie

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

Abstract. We present an all-pairs shortest path algorithm for arbitrary graphs that performs $O(mn \log \alpha)$ comparison and addition operations, where m and n are the number of edges and vertices, resp., and $\alpha = \alpha(m, n)$ is Tarjan's inverse-Ackermann function. Our algorithm eliminates the sorting bottleneck inherent in approaches based on Dijkstra's algorithm, and for graphs with $O(n)$ edges our algorithm is within a tiny $O(\log \alpha)$ factor of optimal. The algorithm can be implemented to run in polynomial time (though it is not a pleasing polynomial). We leave open the problem of providing an efficient implementation.

1 Introduction

In 1975 Fredman [F76] presented a simple and elegant algorithm for the all-pairs shortest paths problem that performs only $O(n^{2.5})$ comparison and addition operations, rather than the $O(n^3)$ bound of Floyd's algorithm (see [CLRS01]). However, Fredman gave no polynomial-time implementation of this algorithm, illustrating that the notion of *comparison-addition complexity* in shortest paths problems can be studied apart from the usual notion of *algorithmic complexity*, that is, the actual running times of shortest path programs. We present, in the same vein, an APSP algorithm that makes $O(mn \log \alpha(m, n))$ comparisons and additions, where m and n are the number of edges and vertices, resp., and α is the mind-bogglingly slow growing inverse-Ackermann function. For sparse graphs, the best comparison-addition-based algorithm to date was established very recently [Pet02]; it runs in $O(mn + n^2 \log \log n)$ time, improving on the long-standing bound of $O(mn + n^2 \log n)$ [Dij59,FT87,J77]. A trivial lower bound on the APSP problem is $\Omega(n^2)$, implying that our algorithm is tantalizingly close to optimal for edge-density $m/n = O(1)$. For dense graphs, the best implementable algorithm is due to Takaoka [Tak92], running in time $O(n^3 \sqrt{\log \log n / \log n})$. We refer the reader to Zwick's survey [Z01] for a summary of other shortest path algorithms.

It is still an open question whether there are $O(n^2) + o(mn)$ algorithms for APSP when $m = O(n^{1.5})$. Karger et al. [KKP93] have shown that $\Omega(mn)$ is

* This work was supported by Texas Advanced Research Program Grant 003658-0029-1999, NSF Grant CCR-9988160, and an MCD Graduate Fellowship.

a lower bound among algorithms that only compare path-lengths. Fredman's algorithm obviously does not fit into this class, and neither does our algorithm. This raises the interesting possibility that our techniques could be used to obtain $O(n^2) + o(mn)$ APSP algorithms for sparse graphs.

Our APSP algorithm is based on the *component hierarchy* (CH) approach to single source shortest paths invented by Thorup [Tho99] for the special case of undirected graphs, and generalized by Hagerup [Hag00] to directed graphs. The [Tho99,Hag00] algorithms were designed for *integer*-weighted graphs in the RAM model of computation. Their improved running times depended crucially on the ability of RAMs to sort n integers in $o(n \log n)$ time. It was, therefore, not obvious whether these algorithms could be translated into good algorithms for real-weighted graphs in the comparison-addition model. Pettie & Ramachandran [PR02] gave an adaptation of Thorup's algorithm to real-weighted undirected graphs; it solves APSP in $O(mn\alpha)$ time.[1] Pettie et al. implemented a simplified version of [PR02]; in their experiments with real-weighted graphs it consistently outperformed Dijkstra's algorithm. The techniques used in [PR02] are specific to undirected graphs and simply have no analogues in directed graphs. Pettie [Pet02], using a different set of techniques, gave a version of Hagerup's algorithm for real-weighted directed graphs. It solves APSP in $O(mn + n^2 \log\log n)$ time. Table 1 summarizes the state of the art in APSP for real-weighted graphs.

APSP Algorithms for Real-Weighted Graphs

Citation	Complexity	Uniform?
Fredman [F76]	$O(n^{2.5})$	NO
Takaoka [Tak92]	$O(n^3\sqrt{\frac{\log\log n}{\log n}})$	yes
Pettie-Ramachandran [PR02] (undirected graphs only)	$O(mn\alpha(m,n))$	yes
Pettie [Pet02]	$O(mn + n^2 \log\log n)$	yes
this paper	$O(mn \log \alpha(m,n))$	NO

Table 1. The best comparison-addition APSP algorithms to date, both uniform and non-uniform. Excluded from this table are algorithms for integer-weighted graphs and average-case algorithms. See [Z01] for a good survey on shortest path algorithms.

In this paper we build on the techniques introduced in [Pet02]. Specifically, our algorithm leverages *approximate* shortest path distances in the computation of exact distances, and it uses a novel mechanism for different SSSP computations to share information. In the next Section we give a technical introduction to the component hierarchy approach which focusses on a high-level feature of all CH-type algorithms [Tho99,Hag00,PR02,Pet02] and not on the algorithmic particulars. This high-level characterization also turns out to be useful in *lower bounding* the complexity of the CH approach in a comparison-based model.

[1] Actually, it solves the s-sources shortest path problem, $s > \log n$, in $O(sm\alpha)$ time.

1.1 Technical Introduction

One way to characterize Dijkstra's SSSP algorithm [Dij59] is to say that it finds a permutation π_s of the vertices such that

$$\pi_s(u) < \pi_s(v) \quad \Rightarrow \quad d(s,u) \leq d(s,v)$$

where $d(\cdot,\cdot)$ is the distance function and s is the source. We give a similar characterization of the shortest path algorithms based on component hierarchies [Tho99,Hag00,PR02,Pet02].

Suppose for this discussion that the graph is strongly connected. Let $circ(u,v)$ be the set of all cycles containing vertices u and v and let $sep(u,v)$ be defined as

$$sep(u,v) \quad = \quad \min_{\mathcal{C} \,\in\, circ(u,v)} \quad \max_{e \,\in\, \mathcal{C}} \; length(e)$$

All component hierarchy-based algorithms [Tho99,Hag00,PR02,Pet02] generate a permutation π_s satisfying Property 1.

Property 1. $\forall u,v: \; d(s,v) \geq d(s,u) + sep(u,v) \quad \Rightarrow \quad \pi_s(u) < \pi_s(v)$

It is not obvious whether there is a sorting bottleneck inherent in Property 1. In [Pet02] it is proved that any directed SSSP algorithm obeying Property 1 must make $\Omega(m + \min\{n\log n, n\log r\})$ operations, where r is the ratio of the maximum to minimum edge length, even if the *sep* function is already known. Interestingly, these bounds become significantly weaker for undirected graphs. In an upcoming full version of [PR02] it is proved that any Property 1 undirected SSSP algorithm must perform $\Omega(m+\min\{n\log n, n\log\log r\})$ operations (notice the weaker dependence on r); however, if the *sep* function is already known there is only a trivial $\Omega(m)$ lower bound for undirected graphs.

What conclusions should be made from these lower bounds? First, one should not waste time trying to develop substantially faster SSSP algorithms obeying Property 1: the directed & undirected SSSP algorithms in [PR02] are tight to within α factors. Second, any directed APSP algorithm that first computes a component hierarchy (read: computes the *sep* function) then performs n *independent* SSSP computations obeying Property 1 must make $\Omega(mn + n^2\log n)$ operations since each SSSP computation is subject to the lower bound of [Pet02]. The key technique to improving this bound, which was used to a lesser extent in [Pet02], is to make the SSSP computations *dependent.* In the algorithm presented here, we perform a sequence of n SSSP computations in such a way that later SSSP computations learn from the time-consuming mistakes of earlier ones.

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 is well-known [J77] that the shortest path problem is reducible in $O(mn)$ time to one of the same size but having only non-negative edge lengths. We therefore assume that $\ell : E \rightarrow \mathbb{R}^+$ assigns only non-negative lengths. We let $d(u, v)$ denote the length of the shortest path from u to v, or ∞ if none exists. The all-pairs shortest path problem is to compute $d(\cdot, \cdot)$ and the single-source shortest paths problem is to compute $d(s, \cdot)$ where the first argument, the *source*, is fixed. Generalizing the d notation, let $d(u, H)$ be the shortest distance from u to H, where H is a subgraph or an object associated with a subgraph.

2.1 The Comparison-Addition Model

In the comparison-addition model real numbers are only subject to comparisons and additions and *comparison-addition complexity* refers to the number of such operations. In order to specify an *implementation* of a comparison-addition-based algorithm one would also need to fix some kind of underlying model governing non-real number computation such as a pointer machine or RAM; implementations, however, are not the focus of this paper. We frequently use subtraction in our algorithms; refer to [PR02] for a simulation of subtraction.

There are several lower bounds for shortest paths in the comparison-addition model though they are all for restricted classes of algorithms. See [PR02] for a summary.

3 The Component Hierarchy Approach

Dijkstra's classic algorithm [Dij59] computes SSSP by visiting vertices by increasing distance from the source s. It can be thought of as simulating a physical process. Suppose the graph-edges represent water pipes and at time zero we begin releasing water from vertex s. Dijkstra's algorithm simulates the flow of water at unit-speed through the graph. Component hierarchy-based algorithms can also be thought of as simulating this process, though in a much coarser way. Instead of maintaining the same simulated time throughout the whole graph, as Dijkstra's algorithm does, CH-based algorithms decompose the graph into a hierarchy of subgraphs (the component hierarchy), where each subgraph maintains its *own* local simulated time. Progress is made by giving a well-selected subgraph, say at simulated time a, permission to advance its clock to simulated time $b > a$. The correctness of this scheme is not obvious, and depends upon the subgraphs and intervals $[a, b)$ being chosen carefully. Due to space constraints we can only sketch the basic component hierarchy algorithm; refer to [Pet02,Pet02b] for a complete description.

The component hierarchy we use is the same CH given in [Pet02,Pet02b]. Below we describe a generalized CH meant solely for understanding our APSP algorithm; it leaves out many important details from [Pet02,Pet02b]. Assume w.l.o.g. that the graph is strongly connected. The CH is defined w.r.t. an increasing sequence of real lengths $(\ell_1, \ldots, \ell_k)$ where ℓ_1 is the minimum edge length in the graph. Let G_{i-1} denote the graph G restricted to edges with length less than ℓ_i, so for instance, G_0 contains no edges. A level i component hierarchy node x corresponds to a strongly connected component C_x of G_i. The notation $diam(C_x)$ refers to the diameter of C_x (the longest shortest path length) and $norm(x) = \ell_i$ by definition. A node x is an ancestor of y if C_y is a subgraph of C_x. Since we would like to ignore CH nodes with only one child, define the 'parent' of a CH node to be its nearest ancestor with a strictly larger strongly connected component. If $\{x_j\}_j$ is the set of children of x then C_x^c denotes the subgraph derived from C_x by contracting the subgraphs $\{C_{x_j}\}_j$. Because of the nice correspondence between component hierarchy nodes and subgraphs, we frequently treat them as equivalent in our notation, so $d(s,x)$ refers to the distance from s to C_x, and $y \in V(C_x^c)$ is understood to mean y is a child of x.

The basic idea of the component hierarchy approach [Tho99,Hag00,PR02,Pet02] is to compute $d(s,x)$ for all $x \in CH$. Since the leaves of the component hierarchy represent graph vertices, this solves SSSP from source s as well. One can imagine that there is a separate process identified with each CH node where the job of y is to compute $d(s,y)$. If x is the parent of y, y simply waits for x to compute $d(s,x)$, then y computes $d(s,y) - d(s,x)$. The key observation from [Pet02] is that this scheme can be made very efficient in the comparison-addition model if y is supplied with a key piece of information: an integer approximation to $(d(s,y)-d(s,x))/norm(x)$ that is accurate to within some absolute constant. We summarize the important aspects of the high-level CH algorithm [Pet02,Pet02b], from the point of view of the process of a CH node y, child of x.

First, all CH algorithms, like Dijkstra's, maintain a set S of visited vertices whose distance from the source has been fixed. Let $d_S(s,u)$ be the distance from s to u using only intermediate vertices from S. Define $D(y)$ as $\min\{d_S(s,u) : u \in V(C_y)\}$.[2] Note that d_S is simply Dijkstra's tentative distance function; D represents the tentative distance to whole subgraphs represented by CH nodes.

As soon as x (parent of y) discovers $d(s,x)$ it creates a bucket array of at least $diam(C_x)/norm(x) + 1$ buckets, each representing a real interval of width $norm(x)$. The first bucket begins at t_0, a real such that $t_0 \le d(s,x) < t_0 + norm(x)$. (We will refer to buckets by their place in the array or by their associated interval, whichever is more convenient.) It would be nice to guarantee that y always appears in the correct bucket, namely bucket number $\lfloor (D(y) - t_0)/norm(x) \rfloor$. This "ideal" invariant is maintained in the CH-based algorithms of [Tho99,Hag00]; however, we do not know how to maintain it efficiently in the comparison-addition model. Our solution is to simulate the ideal bucket array with an *actual* bucket array and a heap, denoted H_x.

Invariant 1 *If $d(s,y)$ has not yet been fixed, then either y appears in an actual bucket between $\lfloor \frac{d(s,y)-t_0}{norm(x)} \rfloor - 2$ and $\lfloor \frac{D(y)-t_0}{norm(x)} \rfloor$ inclusive, or in the heap H_x.*

The purpose of the heap H_x is to hold nodes until enough information is available to bucket them in accordance with Invariant 1. The efficiency of this

[2] Updating and querying D-values is a non-trivial task, requiring $O(m \log \alpha(m,n))$ comparisons per SSSP computation using Gabow's split-findmin structure [G85], as modified in [PR02].

scheme depends on there being relatively few heap insertions, since heap deletion is a non-constant time operation. We will not go into why the "-2" is a tolerable error (see [Pet02b]).

Our algorithm should be thought of as consisting of two levels, the "high-level" component hierarchy algorithm (see [Pet02b]) and a "low-level" algorithm that maintains Invariant 1 behind the scenes, which we give in Section 4. The low-level algorithm uses some simple, though non-obvious properties of shortest paths.

4 Our Algorithm

In Section 4.1 we define a set of new length functions $\{\gamma_x\}_{x \in CH}$ and a set of *relative distance* functions $\{\Gamma_x\}_{x \in CH}$. A relative distance is just the difference between two distances. In Section 4.2 we show that, using *discrete approximations* of the length and relative distance functions, it can be possible to stitch together new shortest paths from previously computed ones, in a manner that is cheaper than computing them from scratch. There is a tradeoff between the accuracy of the discrete approximations and their usefulness; our amortized analysis depends on the degree of accuracy being chosen carefully.

Naming conventions. The letters x, y, z will refer to CH nodes and u, v, w, s to graph vertices. A hat ($\hat{}$) or tilde ($\tilde{}$) indicates a discrete approximation to a real quantity.

4.1 Approximating Relative Distances

Let $anchor_x(u)$ be some vertex in $V(C_x^c)$ (recall, $V(C_x^c)$ corresponds to the children of x) and $a_x(u) = d(u, anchor_x(u))$. The anchor of u is specified as soon as possible. That is, as soon as $d(u, y)$ is known for *some* $y \in V(C_x^c)$, $anchor_x(u)$ is set to y. The edge-labeling functions γ_x and $\hat{\gamma}_x$ are also calculated as soon as possible. As soon as $a_x(v)$ and $a_x(u)$ are known, $\gamma_x(u, v)$ is set to:

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

It follows that for edges $(u, v) \in E(C_x)$, $\gamma_x(u, v) = \ell(u, v)$ is fixed immediately, since $a_x(u) = a_x(v) = 0$ is known a priori. As far as conserving comparisons & additions, it turns out that γ_x is not as useful as a discrete approximation to γ_x. If $\gamma_x(u, v) > 2 \cdot diam(C_x)$ then $\hat{\gamma}_x(u, v) \stackrel{\mathbf{def}}{=} \infty$. Otherwise, define $\hat{\gamma}_x(u, v)$ as

$$\hat{\gamma}_x(u, v) \stackrel{\mathbf{def}}{=} \epsilon_x \cdot \lfloor \frac{\gamma_x(u, v)}{\epsilon_x} \rfloor \qquad \text{where} \qquad \epsilon_x \stackrel{\mathbf{def}}{=} \frac{norm(x)}{4 \cdot |V(C_x^c)|}$$

Why is $\hat{\gamma}_x$ better than γ_x? The difference is in how they are represented. We represent γ_x in the natural way, as a real number kept in a real variable. On the other hand $\hat{\gamma}_x$ is represented *implicitly.* That is, the statement "$\hat{\gamma}_x(u, v)$ is known" means the *integer* $\hat{\gamma}_x(u, v)/\epsilon_x$ can be derived from previous comparisons and additions. Clearly two γ_x-values require one operation to be compared or added. Manipulating $\hat{\gamma}_x$-values is really just a mental exercise; there are no comparisons or additions involved in computing functions of the $\hat{\gamma}_x$ values.

Define $\mathit{\Gamma}_x, \hat{\mathit{\Gamma}}_x : V(G) \times V(C_x^c) \rightarrow \mathbb{R}$. As above, the integer $\hat{\mathit{\Gamma}}_x/\epsilon_x$ will be represented implicitly.

$$\mathit{\Gamma}_x(u,y) \overset{\textbf{def}}{=} d(u,y) - a_x(u) \qquad \text{and} \qquad \hat{\mathit{\Gamma}}_x(u,y) \overset{\textbf{def}}{=} \epsilon_x \cdot \lfloor \frac{\mathit{\Gamma}_x(u,y)}{\epsilon_x} \rfloor$$

Lemma 1. *Let $x, y \in CH$, $y \in V(C_x^c)$ and u, v be vertices.*

(i) Given $\gamma_x(u,v)$ (resp., $\mathit{\Gamma}_x(u,y)$), $\hat{\gamma}_x(u,v)$ (resp., $\hat{\mathit{\Gamma}}_x(u,y)$) can be computed with $O(\log \frac{|V(C_x^c)| \cdot diam(C_x)}{norm(x)})$ comparisons and additions.
(ii) Computing $\hat{\gamma}_x(e)$, over all x and edges e, takes $O(mn)$ comparisons and additions.

$\hat{\mathit{\Gamma}}_x$-values are very useful for conserving on comparison-addition operations in the component hierarchy algorithm; however, even given $\mathit{\Gamma}_x$, $\hat{\mathit{\Gamma}}_x$ is fairly expensive to compute. Lemma 1(ii) illustrates that we can afford to compute all $\hat{\gamma}$-values; however, it will become clear in the analysis that we can only afford to compute a small fraction of the $\hat{\mathit{\Gamma}}$-values Our solution is to introduce another approximation of $\mathit{\Gamma}_x$ which is significantly less accurate than $\hat{\mathit{\Gamma}}_x$. Define $\tilde{\mathit{\Gamma}}_x : V(G) \times V(C_x^c) \rightarrow \mathbb{R}$ to be any function that satisfies:

$$\mathit{\Gamma}_x(u,y) - \tilde{\mathit{\Gamma}}_x(u,y) \;\in\; [0, |V(C_x^c)|\, \epsilon_x) \qquad \text{and} \qquad \frac{\tilde{\mathit{\Gamma}}_x}{\epsilon_x} \text{ is represented as an integer}$$

Notice that $\epsilon_x \, |V(C_x^c)| = norm(x)/4$ which is just about as accurate as we will ever need. Lemma 2 illustrates the relationship between $\hat{\gamma}_x, \hat{\mathit{\Gamma}}_x, \tilde{\mathit{\Gamma}}_x$, and ϵ_x.

Lemma 2. *Suppose $\mathcal{P} = \langle u_0, u_1, \ldots, u_k \rangle$, $u_k \in V(C_y)$, $y \in V(C_x^c)$, is known to be the shortest path from u_0 to C_y, and u_h is the first vertex in $\mathcal{P}$ for which $\hat{\mathit{\Gamma}}_x(u_h, y)$ is known. Then if $h < |V(C_x^c)|$, $\tilde{\mathit{\Gamma}}_x(u_i, y)$ is known as well, for $i \leq h$.*

Proof. Because $\mathcal{P}$ is known to be the shortest path to some vertex in $V(C_x^c)$, it follows that $anchor_x(w)$ has been chosen for all vertices $w \in \mathcal{P}$ and that $\gamma_x(e), \hat{\gamma}_x(e)$ are computed for all edges $e \in \mathcal{P}$. We now prove that for $i \leq h$

$$\mathit{\Gamma}_x(u_i, y) - \Big(\hat{\mathit{\Gamma}}_x(u_h, y) + \sum_{j=i}^{h-1} \hat{\gamma}(u_j, u_{j+1})\Big) \in [0, (h-i+1)\epsilon_x)$$

Hence $\hat{\mathit{\Gamma}}_x(u_h, y) + \sum_{j=i}^{h-1} \hat{\gamma}(u_j, u_{j+1})$ is a good enough approximation to $\mathit{\Gamma}_x(u_i, y)$ to satisfy the constraints put on $\tilde{\mathit{\Gamma}}_x(u_i, y)$, so long as $h - i < |V(C_x^c)|$. Let $[a, b)$ denote some number in that interval. Then, in general, $\hat{\gamma}_x = \gamma_x - [0, \epsilon_x)$ and $\hat{\mathit{\Gamma}}_x = \mathit{\Gamma}_x - [0, \epsilon_x)$. Let $\xi = [0, \epsilon_x(h-i+1))$. Then

$$\mathit{\Gamma}_x(u_i, y) - \Big(\hat{\mathit{\Gamma}}_x(u_h, y) + \sum_{j=i}^{h-1} \hat{\gamma}(u_j, u_{j+1})\Big)$$

$$= \Gamma_x(u_i, y) - \Big(d(u_h, y) - a_x(u_h) + \sum_{j=i}^{h-1}(\ell(u_j, u_{j+1}) + a_x(u_{j+1}) - a_x(u_j))\Big) + \xi$$
$$= \Gamma_x(u_i, y) - d(u_i, y) + a_x(u_i) + \xi \quad = \quad \xi$$

The above equalities follow directly from the definitions of $\gamma, \hat{\gamma}, \Gamma$, and $\hat{\Gamma}$. □

4.2 The Algorithm

The algorithm is best described as a list of triggers of the form $\mathcal{P} \longrightarrow \mathcal{A}$, where $\mathcal{P}$ is a precondition and $\mathcal{A}$ an action to be performed. The high-level component hierarchy algorithm can only proceed if none of the triggers are applicable. We have already informally defined a few triggers. Let us state them formally.

Trigger 1 *$anchor_x(u)$ is unspecified but $d(u,y)$ is known, $y \in V(C_x^c) \longrightarrow$ Set $anchor_x(u) := y,\ \ a_x(u) := d(u,y)$*

Trigger 2 *(u,v) is an edge and both $a_x(u)$ and $a_x(v)$ are known $\longrightarrow$ Compute $\gamma_x(u,v)$ and $\hat{\gamma}_x(u,v)$*

Trigger 3 *Some edge is relaxed, decreasing $D(y)$, where $y \in V(C_x^c)$, $d(s,x)$ is known. $\longrightarrow$ If possible, bucket y according to Invariant 1*

Lemma 2 suggests another trigger. Let $OUT(u)$ and $IN(u)$ denote the *known* trees of shortest paths out of, and into u, respectively.[3] So, for instance, $IN(y)$, $y \in CH$, initially has no edges because we do not know any non-trivial shortest paths to C_y. After each SSSP computation, from say, source s, one can see that $IN(y)$ grows the minimal amount to incorporate s. It will be important to know the $\hat{\Gamma}_x(u,y)$ function for vertices $u \in IN(y)$; by Lemma 2 it is enough to compute $\hat{\Gamma}_x(\cdot,y)$-values for a sufficiently large and well-chosen subset of the vertices in $IN(y)$. It can be proved that Trigger 4 fits the bill.

Trigger 4 *The closest ancestor of u in $IN(y)$ for which $\hat{\Gamma}_x(\cdot,y)$ is known is at distance exactly $|V(C_x^c)| \longrightarrow$ Compute $\hat{\Gamma}_x(w,y)$, where w is the ancestor of u in $IN(y)$ at distance exactly $\lfloor |V(C_x^c)|/2 \rfloor$*

Lemma 3. *For CH nodes x and $y \in V(C_x^c)$*
(i) The $\tilde{\Gamma}_x(\cdot,y)$ function is known for vertices in $IN(y)$.
(ii) At most $2n/|V(C_x^c)|$ different $\hat{\Gamma}_x(\cdot,y)$-values are computed.
(iii) The total comparison-addition cost of (ii), over all x,y, is $O(n^2)$.

Proof. Part (i): Trigger 4 ensures that every vertex in $IN(y)$ has some ancestor at distance at most $|V(C_x^c)| - 1$ whose $\hat{\Gamma}_x(\cdot,y)$-value is known. By Lemma 2 the $\tilde{\Gamma}_x(\cdot,y)$-value of every vertex in $IN(y)$ is also known. Part (ii) is straightforward [Pet02b]. Part (iii) can be shown to follow from Part (ii) and Lemma 1(i) [Pet02b]. □

[3] For $y \in CH$, $IN(y)$ is really an in-forest.

Let $G^{\hat{\gamma}_x}$ be the subgraph of G consisting of edges whose $\hat{\gamma}_x$-values are known. Every time $G^{\hat{\gamma}_x}$ grows we can better estimate shortest distances. Trigger 5, given below, attempts to bucket nodes residing in the heap as soon as possible.

Trigger 5 *Edge(s) are added to $G^{\hat{\gamma}_x}$ $\longrightarrow$ If possible, migrate nodes from H_x to the bucket array, consistent with Invariant 1.*

We now clarify exactly what is meant by "if possible" in Triggers 3 and 5, that is, how we decide if it is possible to bucket a node $y \in V(C_x^c)$. Suppose s is the source in the current SSSP computation. The first moment we are concerned about the distance from s to C_y, $y \in V(C_x^c)$, is when $d(s,x)$ becomes known. At this moment the shortest path from s to C_y consists of a *head* in $OUT(s)$, a *bridge*, and a *tail* in $IN(y)$. We show that if the head, bridge, and tail satisfy certain conditions, then a good, discrete approximation of $d(s,x) - d(s,y)$ is known implicitly, allowing us to bucket y in constant time. Every time new edges are added to $G^{\hat{\delta}_x}$ or when $D(y)$ decreases (Triggers 5 and 3, resp.) we have a new opportunity to bucket y.

We now describe the bucketing procedure for $y \in V(C_x^c)$ more carefully. Let $f \in C_z$, $z \in V(C_x^c)$ be such that $d(s,f) = d(s,x)$, that is, f is the closest vertex to s in C_x — Figure 1 diagrams our situation. Because we are attempting to bucket y, $d(s,x)$ must already be known. Let P_{sf} denote the shortest s-to-f path (which is also the shortest s-to-C_x path).

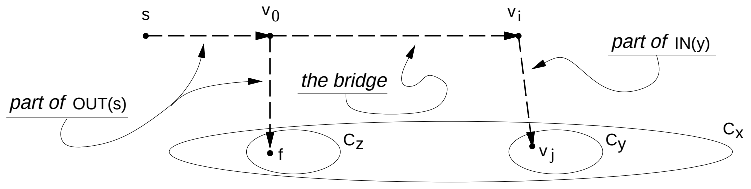

Fig. 1. The path $\langle s, \ldots, v_j \rangle$, divided into a head $\langle s, \ldots, v_0 \rangle$, a bridge $\langle v_0, \ldots, v_i \rangle$, and a tail $\langle v_i, \ldots, v_j \rangle$.

Definition 1. *Let $\mathcal{Q}_y$ be the set of paths $\{\langle v_0, \ldots, v_i, \ldots, v_j \rangle\}$ satisfying*

(i) $v_0 \in P_{sf} \subseteq OUT(s)$
(ii) $i \leq |V(C_x^c)|$ *and* $\langle v_0, \ldots, v_i \rangle \in G^{\hat{\gamma}_x}$
(iii) $v_j \in C_y$ *and* $\langle v_i, \ldots, v_j \rangle \in IN(y)$

Bucketing y by its D-value is equivalent to estimating $D(y) - d(s,f)$, however we generally will not have enough information to do this. Our solution is not to focus solely on the current path with length $D(y)$, but to estimate the distance of *many* hypothetically shortest paths from s to C_y.

For $Q \in \mathcal{Q}_y$, $Q = \langle v_0, \ldots, v_i, \ldots, v_j \rangle$, define *diff*$(Q)$ and *diff*$(\mathcal{Q}_y)$ as

$$\mathit{diff}(Q) \stackrel{\mathbf{def}}{=} \tilde{\Gamma}_x(v_i, y) + \sum_{k=0}^{i-1} \hat{\gamma}_x(v_k, v_{k+1}) - \tilde{\Gamma}_x(v_0, z)$$

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

Lemma 4. *$diff(Q)$ requires no comparisons or additions to compute, and*

$$diff(Q) \;=\; \ell(Q) - d(v_0, x) + \left(-\tfrac{norm(x)}{2}, \tfrac{norm(x)}{4}\right)$$

Proof. Let $\xi \;=\; (-2\epsilon_x\,|V(C_x^c)|\,, \epsilon_x\,|V(C_x^c)|) \;=\; \left(-\frac{norm(x)}{2}, \frac{norm(x)}{4}\right)$

Recall that $f \in C_z,\;\; z \in V(C_x^c)$ were such that $d(s,f) = d(s,z) = d(s,x)$.

$$\begin{aligned}
diff(Q) &= \sum_{k=0}^{i-1} \gamma_x(v_k, v_{k+1}) + \mathit{\Gamma}_x(v_i, y) - \mathit{\Gamma}_x(v_0, z) + \xi & (1)\\
&= \ell(\langle v_0, \ldots, v_i\rangle) - a_x(v_0) + a_x(v_i) + \mathit{\Gamma}_x(v_i, y) - \mathit{\Gamma}_x(v_0, z) + \xi & (2)\\
&= \ell(\langle v_0, \ldots, v_j\rangle) - a_x(v_0) - \mathit{\Gamma}_x(v_0, z) + \xi & (3)\\
&= \ell(Q) - d(v_0, x) + \xi & (4)
\end{aligned}$$

Line 1 follows from the equalities $\tilde{\mathit{\Gamma}}_x = \mathit{\Gamma}_x - [0, \epsilon_x\,|V(C_x^c)|)$ and $\hat{\gamma}_x = \gamma_x - [0, \epsilon_x)$, and the bound $i \leq |V(C_x^c)|$. Line 2 is derived by cancelling the terms in the telescoping sum. Line 3 follows from the equality $\mathit{\Gamma}_x(v_i, y) = d(v_i, y) - a_x(v_i) = \ell(\langle v_i, \ldots, v_j\rangle) - a_x(v_i)$, and Line 4 from the equality $\mathit{\Gamma}_x(v_0, z) = d(v_0, z) - a_x(v_0) = d(v_0, x) - a_x(v_0)$.

The $\hat{\gamma}_x$ terms in $diff(Q)$ are known from the fact that $Q \in \mathcal{Q}_y$. By Lemma 3 the $\tilde{\mathit{\Gamma}}_x(v_i, y)$ and $\tilde{\mathit{\Gamma}}_x(v_0, z)$ terms are also implicitly known. Therefore, $diff(Q)$ can be computed with no real number operations. $\square$

Our procedure for bucketing $y \in V(C_x^c)$ is as follows. Recall that we denote the beginning of x's bucket array with t_0. Let $[\beta, \beta + norm(x))$ be the bucket s.t. $\beta \leq t_0 + diff(\mathcal{Q}_y) < \beta + norm(x)$. Since $\beta - t_0$, $norm(x)$, and $diff(\mathcal{Q}_y)$ are all known multiples of ϵ_x, this bucket can be identified without real number operations.

1. If $D(y) \;\geq\; \beta$, put y in bucket $\;[\beta,\, \beta + norm(x))$ and stop.
2. If $D(y) \;\geq\; \beta - norm(x),\;$ put y in bucket $\;[\beta - norm(x),\, \beta)$ and stop.
3. Otherwise, put y in H_x (or keep y in H_x if it is already there).

Lemma 5. *The bucketing procedure does not violate Invariant 1, and if $\mathcal{Q}_y$ contains a suffix of a shortest s-to-C_y path, then y is bucketed in Line 1 or 2.*

Proof. Recall from Section 3 that t_0 was chosen so that $d(s,x) \in [t_0, t_0 + norm(x))$. Lines 1 and 2 guarantee that y is never bucketed in a higher bucket than $\lfloor \frac{D(y) - t_0}{norm(x)} \rfloor$. We only need to show that in Line 1, y is not bucketed before bucket $\lfloor \frac{d(s,y) - t_0}{norm(x)} \rfloor - 2$. Because $diff(\mathcal{Q}_y)$ cannot correspond to a path shorter than $d(s,y)$, we have, from Lemma 4, $diff(\mathcal{Q}_y) > d(s,y) - d(s,x) - \frac{1}{2}norm(x)$. Using the inequality $d(s,x) < t_0 + norm(x)$, we also have $diff(\mathcal{Q}_y) > d(s,y) - t_0 - \frac{3}{2}norm(x)$. So bucketing y according to $diff(\mathcal{Q}_y)$ can put it at most $\lceil \frac{3}{2} \rceil = 2$ buckets before bucket $\lfloor \frac{d(s,y) - t_0}{norm(x)} \rfloor$. For the last part of the Lemma, assume that some

$Q \in \mathcal{Q}_y$ is a suffix of the shortest s-to-C_y path. It follows from Lemma 4 that $\mathit{diff}(\mathcal{Q}_y) < d(s,y) - d(s,x) + \frac{1}{4}\mathit{norm}(x)$. This, together with the inequalities $d(s,x) \geq t_0$ and $\beta \leq \mathit{diff}(\mathcal{Q}_y) + t_0$, implies $D(y) > \beta - \frac{1}{4}\mathit{norm}(x)$, meaning y must be bucketed in Line 1 or 2. □

We address the efficiency of our bucketing procedure in Lemma 6. Since deleting items from a heap is a non-constant operation, we must show that the percentage of times Step 3 is reached in the bucketing procedure is sufficiently low to counterbalance the cost of the heap operations. Lemma 6 does not depend on any fancy heap implementation. It holds if H_x supports constant time insert and decrease-key operations, and deletion of any subset of H_x in time linear in $|H_x| \leq |V(C_x^c)|$. These are very weak assumptions.

Lemma 6. *The bucketing and heap costs over n SSSP computations are $O(mn)$.*

Proof. (sketch) We prove in [Pet02b] that the bucketing procedure above is called only $O(mn)$ times, requiring a constant number of comparisons per invocation, and that the total cost of heap operations is $O(n^2)$. We briefly outline how one would bound the number of heap operations.

Suppose in the SSSP computation from source s, $y \in V(C_x^c)$ is inserted into H_x. Let $P_{sy} = \langle P_1, P_2, P_3 \rangle$ be the shortest s-to-C_y path, where P_1 and P_3 are maximal such that $P_1 \subseteq P_{sx} \subseteq \mathit{OUT}(s)$ and $P_3 \subseteq \mathit{IN}(y)$. From Lemma 5 we know that $\langle P_2, P_3 \rangle \notin \mathcal{Q}_y$, otherwise y would have been bucketed properly. Why wasn't $\langle P_2, P_3 \rangle \in \mathcal{Q}_y$? From Definition 1 there can be only two reasons: either (a) $|P_2| > |V(C_x^c)|$ or (b) some edge in P_2 is not in $G^{\hat{\gamma}_x}$, which by Trigger 1 means some vertex in P_2 is unanchored. One can easily bound the number of times (a) occurs, since after the current SSSP computation with source s, at least $|V(C_x^c)|$ edges are added to $\mathit{IN}(y)$. A sufficient bound on (b) is n times for each CH node x, since any unanchored vertex in P_2 will, by Trigger 1, be anchored by the end of the current SSSP computation. One can, using other properties of the component hierarchy, prove that the heap costs due to (a) and (b) are $O(n^2)$ [Pet02b]. □

The only costs not covered by Lemma 6 are constructing the component hierarchy, which is $O(m \log n)$ [Pet02b], computing the $\hat{\Gamma}$ and $\hat{\gamma}$ functions, which is $O(mn)$ by Lemma 1 and 3, and maintaining the D-values of CH nodes, which is $O(m \log \alpha(m,n))$ for each SSSP computation [G85,PR02]. Regarding an actual implementation of this algorithm, the tricky part is maintaining shortest distances in the graphs $\{G^{\hat{\gamma}_x}\}_{x \in CH}$ under insertion of new edges. Simply running Bellman-Ford every time a batch of new edges are inserted gives a bound of $O(mn^3)$. However, if we use the dynamic shortest path algorithm from [RR96] the upper bound can be reduced to $\tilde{O}(mn^2)$. Theorem 1 follows.

Theorem 1. *The all-pairs shortest path problem on arbitrarily weighted, directed graphs can be solved with $O(mn \log \alpha(m,n))$ comparisons & additions in $\tilde{O}(mn^2)$ time, where m and n are the number of edges & vertices, resp., and α is the inverse-Ackermann function.*

Acknowledgment. We thank Vijaya Ramachandran for her comments and Camil Demetrescu for references on dynamic shortest paths.

References

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

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

[F76] M. 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.

[G85] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *26th Ann. Symp. on Foundations of Computer Science (FOCS 1985)*, 90–99.

[Hag00] T. Hagerup. Improved shortest paths on the word RAM. In *Proceedings 27th Int'l Colloq. on Automata, Languages and Programming (ICALP 2000)*, LNCS volume 1853, 61–72.

[J77] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *JACM* 24 (1977), 1–13.

[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.

[Pet02] S. Pettie. A faster all-pairs shortest path algorithm for real-weighted sparse graphs. *Proceedings 29th Int'l Colloq. on Automata, Languages and Programming (ICALP 2002)*, LNCS 2380, 85–97.

[Pet02b] S. Pettie. On the comparison-addition complexity of all-pairs shortest paths. UTCS Technical Report TR-02-21, May 2002.

[PR02] S. Pettie, V. Ramachandran. Computing shortest paths with comparisons and additions (extended abstract). *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002, 267–276.

[PRS02] S. Pettie, V. Ramachandran, S. Sridhar. Experimental evaluation of a new shortest path algorithm. *4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2002.

[RR96] G. Ramalingam, T. Reps. An incremental algorithm for a generalization of the shortest path problem. *J. Algorithms* 21 (1996), 267–305.

[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.

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

[Z01] U. Zwick. Exact and approximate distances in graphs – A survey. Updated version at `http://www.cs.tau.ac.il/~zwick/`, *Proc. of 9th ESA* (2001), 33–48.