A Faster All-pairs Shortest Path Algorithm for Real-weighted Sparse Graphs*

Seth Pettie
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
seth@cs.utexas.edu

TR-02-13

February 26, 2002

Abstract

We present a faster all-pairs shortest paths algorithm for arbitrary real-weighted directed graphs. The algorithm works in the fundamental *comparison-addition model* and runs in $O(mn + n^2 \log\log n)$ time, where m and n are the number of edges & vertices, respectively. This is strictly faster than Johnson's algorithm (for arbitrary edge-weights) and Dijkstra's algorithm (for positive edge-weights) when $m = o(n \log n)$ and matches the running time of Hagerup's APSP algorithm, which assumes *integer* edge-weights and a more powerful model of computation.

Our algorithm is based on a generalization of Hagerup's "component hierarchy" to *real*-weighted directed graphs. The component hierarchy approach, first initiated by Thorup, is a non-greedy method for computing single-source shortest paths, and, by repeated application, all-pairs shortest paths. Component hierarchy-based algorithms have been designed for integer-weighted graphs, directed and undirected, and real-weighted undirected graphs. We show that in a precise sense, adapting the component hierarchy approach to the general case (arbitrary real-weighted directed graphs) is more complicated that for the restricted graph classes treated earlier.

1 Introduction

Nearly all known shortest path algorithms can be easily 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 algorithms, established early on by Dijkstra [Dij59], Bellman-Ford, Floyd-Warshall and others (see [CLR90]), 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, integer matrix multiplication and fast integer sorting only work with integer edge-weights, and until recently [PR02] it appeared as though the *component hierarchy* approach used in [Tho99, Hag00] also required integers. We refer the reader to a recent survey paper [Z01] for more background and references.

The state of the art in APSP for real-weighted, sparse directed graphs is (surprisingly) a combination of two standard textbook algorithms. Johnson [J77] showed that an arbitrarily-weighted graph is reducible to a positively-weighted graph such that shortest paths remain unchanged (assuming no negative cycles

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

and hence 'shortest path' is well-defined.) The reduction makes one call to the Bellman-Ford algorithm and takes $O(mn)$ time. For positively-weighted graphs, Dijkstra's algorithm [Dij59, FT87] solves the single-source shortest path problem (SSSP) in $O(m + n \log n)$ time, implying an APSP algorithm for *arbitrarily*-weighted graphs running in $O(mn + n^2 \log n)$ time. This is the best bound to date for directed graphs, however there is a faster algorithm [PR02] for *undirected* APSP running in $O(mn\alpha(m, n))$ time. It is worth mentioning that several algorithms with good *average-case* performance (e.g., [MT87, KKP93, M01, G01]) can improve upon this bound (either in the mn or $n^2 \log n$ term), however each assumes random edge-weights and many assume the complete graph.

The recent component hierarchy (CH) based algorithms [Tho99, Hag00, PR02] either reduce or eliminate the sorting bottleneck in Dijkstra's algorithm. Thorup, who first described the CH approach, 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. This immediately implies an $O(mn)$ time APSP algorithm. Hagerup [Hag00] assumed the same non-negative integer/RAM model and showed SSSP on *directed* graphs can be solved in $O(m \log\log C + n \log\log n)$ time, and APSP in $O(mn + n^2 \log\log n)$ time. Here C is the largest integer edge weight. Recently Pettie & Ramachandran [PR02] adapted Thorup's algorithm to *real*-weighted graphs and the pointer machine model [Tar79], which is weaker than the RAM, yielding an undirected APSP algorithm running in $O(mn\alpha(m, n))$ time, where α is the inverse-Ackermann function. Pettie [Pet01] gave a CH-based APSP algorithm for directed graphs that performs $O(mn \log \alpha(m, n))$ comparison/addition operations, however there is no known implementation of this algorithm with the same asymptotic overhead. The CH approach also turns out to be practical; an experimental study of Pettie et al. [PRS02] of a simplified version of [PR02] shows it to be decisively faster than Dijkstra's algorithm, if the one-time cost of constructing the CH is offset by a sufficient number of SSSP computations.

In this paper we adapt the component hierarchy approach to real-weighted directed graphs, giving an APSP algorithm running in $O(mn + n^2 \log\log n)$ time. Our algorithm differs from previous CH-based algorithms in its overall structure. In [Tho99, Hag00, PR02], a component hierarchy is constructed in one phase, and in the next phase SSSP can be computed any number of times in (nearly-)linear time per SSSP computation. In Section 4.3 we show that *if we adhere to the basic CH framework*, the structure of the component hierarchy does *not* provide enough information to compute SSSP in $O(m) + o(n \log n)$ time. In our algorithm we have a three-phase structure. After the CH is constructed (phase 1), we take the time to gather a slew of approximate shortest path-related statistics (phase 2) which will allow us to compute APSP faster in phase 3. For a parameter k, if we spend $O(mn\lceil \frac{\log n}{k} \rceil)$ time in phase 2, the s-sources shortest paths problem is solved in phase 3 in $O(s \cdot (m + n \log k + n \log\log n))$ time. Setting $k = \log n,\ s = n$ gives us the claimed APSP result, though improvements can still be had for $s = \omega(m/\log n)$.

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. The length of a path is defined to be the sum of its constituent edge lengths. We let $d(u, v)$ denote the length of the shortest path from u to v, or ∞ if none exists. The single-source shortest paths problem is to compute $d(s, v)$ for some *source* vertex s and every vertex v while the all-pairs shortest path problem is to compute $d(u, v)$ for all u, v. Generalizing the d notation, let $d(u, H)$ (resp. $d(H, u)$) be the shortest distance from u to any vertex in the subgraph H (from any vertex in H to u). H may also be an object that is associated with a subgraph, not necessarily the subgraph itself. It was mentioned in the introduction that the APSP problem is reducible in $O(mn)$ time to one of the same size but having only positive edge lengths. We therefore assume that $\ell : E \to \mathbb{R}^+$ assigns only positive lengths.

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. A comparison-addition based algorithm, which is modeled as a decision tree with additions, chooses which operations to make based on the outcomes of previous comparisons.

This model cannot distinguish between integers and arbitrary reals, and cannot produce a specific integer in a real variable. Therefore, when we say some variable or quantity is an *integer*, we mean that it is kept in

an integer variable. The only additional property assumed of integers is that they may be used to index an array. We will only produce polynomially-bounded integers, whereas reals are assumed to take on arbitrary values.

This model is elegant and sufficiently powerful to solve shortest path problems (the standard textbook algorithms of Dijkstra, Bellman-Ford, Floyd-Warshall, and min-plus matrix multiplication assume nothing more). The practicality of this model stems from its universality: comparison-addition based algorithms inherently work with a variety of numerical data types and can be ported to different platforms with little or no modification — see [PRS02].

3 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 shortest 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 *greedily*. 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), setting $D(w) := \min\{D(w), D(v) + \ell(v,w)\}$. The algorithm halts when $S = V$, and therefore $D(v) = d(s,v)$ — the tentative distances equal the shortest distances.

Component hierarchy-based algorithms also maintain Invariant 0, though in a non-greedy fashion. Throughout the paper D, S, s mean the same thing as in Dijkstra's algorithm, and the terms "visit" and "relax" are essentially the same.

4 The Component Hierarchy

We assume a familiarity with the component hierarchy approach [Tho99, Hag00, PR02]. See Appendix A for an overview.

Hagerup [Hag00] constructed a CH for directed graphs and positive integer edge lengths in $O(m \log\log C)$ time, where C is the largest edge weight. For real edge lengths his method can be adapted, using techniques akin to those in [PR02], to run in $O(m \log\log r + \log r)$ time, where r is the ratio of the maximum-to-minimum edge length. Below we define a component hierarchy for real-weighted directed graphs; it can be constructed in $O(m \log n)$ time using a combination of the techniques from [Hag00, PR02]. We omit the proof.

4.1 The CH for Real-weighted Directed Graphs

Assume w.l.o.g. that G is strongly connected. This can be enforced without altering the finite distances by adding an n-cycle with very long edges. As in [PR02] we first produce the edge lengths in sorted order: $\ell_1, \ldots, \ell_m$. We then find a set of "normalizing" edge lengths $\{\ell_j : \ell_j > n \cdot \ell_{j-1}\} \cup \{\ell_1\}$. Let r_k be the k^{th} smallest normalizing edge. For each edge j between r_k and $r_{k+1} - 1$ we determine the i s.t. $2^i \ell_{r_k} \le \ell_j < 2^{i+1}\ell_{r_k}$. In other words, we find a factor 2 approximation of every edge length divided by its associated normalizing edge length. The CH is composed of layered *strata*, where stratum k, level i nodes correspond to the strongly connected components (SCCs) of the graph restricted to edges with length less than $\ell_{r_k} \cdot 2^i$. If x is a stratum k, level i node we let $norm(x) = \ell_{r_k} \cdot 2^{i-1}$; Most quantities relating to x will be measured in units of $norm(x)$. Let C_x denote the SCC associated with a CH node x, and let $diam(C_x)$ (the diameter) be the longest shortest path between two distinct vertices in C_x. The children of x, $\{x_1, \ldots, x_\nu\}$ are those stratum k, level $i-1$ CH nodes whose SCCs $\{C_{x_1}, \ldots, C_{x_\nu}\}$ are subgraphs of C_x. (If $i = 0$, i.e. x is at the "bottom" of its stratum, then its children are the stratum $k-1$ nodes of maximum level, $\{x_j\}$, s.t. the $\{C_{x_j}\}$ are subgraphs of C_x.) Let C_x^c be derived from C_x by contracting the SCCs $\{C_{x_j}\}$, and C_x^{DAG} be derived from C_x^c by removing edges with length at least $norm(x)$. That is, there is a correspondence between

vertices in C_x^c and the children of x in the component hierarchy; *we will frequently use the same notation to refer to both.* It is convenient to think of single-child nodes in the CH being spliced out, hence the children of a node are not necessarily all on the same stratum/level, but the CH is linear in size.

The following lemma, variants of which were used in [Tho99, Hag00, PR02], is useful for associating the running time of our algorithm with certain CH statistics. Its proof is straightforward; it appears in Appendix B.

Lemma 4.1

$$\textit{(i)} \qquad \sum_{x \in CH} |V(C_x^c)| \leq 2n$$

$$\textit{(ii)} \qquad \sum_{x \in CH} \frac{diam(C_x)}{norm(x)} \leq 8n$$

$$\textit{(iii)} \qquad \left| \left\{ x \in CH : \frac{diam(C_x)}{norm(x)} > k \right\} \right| \leq \frac{8n}{k}$$

4.2 Computing SSSP

Component hierarchy-based algorithms also maintain Dijkstra's Invariant 0. However, they do not necessarily visit vertices in increasing distance from the source. Recall that the D-value of a vertex was its tentative distance from the source s. We extend the D notation to CH nodes by letting $D(x) = \min_{v \in C_x}\{D(v)\}$ (i.e. minimum over leaf descendants of x.) The Visit procedure, given below, takes a CH node x and some interval $[a, b)$ and visits all vertices $v \in C_x$ whose $d(s, v)$-values lie in $[a, b)$. If C_x is a single vertex and $D(x) \in [a, b)$, we mark C_x as visited and relax all its outgoing edges. Otherwise we delegate the responsibility of visiting vertices in $[a, b)$ to the children of x. SSSP are computed from s by setting $S = \emptyset$, $D(s) = 0$, $D(v) = \infty$ for $v \neq s$ and calling Visit$(root(CH), [0, \infty))$. One may refer to [Tho99, Hag00, PR02] for more detailed descriptions of the basic component hierarchy algorithm or proofs of its correctness. We will call a node *active* if it has been visited at least once, and *inactive* otherwise.

```
    Visit(x, [a, b))
        If C_x is a single vertex and D(x) ∈ [a, b) then
            Visit C_x:
              Let S := S ∪ {C_x}
(1)           Relax C_x's outgoing edges
              Return.
        If Visit(x, ·) is being called for the first time, then
(2)         Initialize x's bucket array:
              Create ⌈diam(C_x)/norm(x)⌉ + 1 buckets
              Let the first bucket start at t_0, a real number s.t. a ≤ t_0 ≤ D(x).
              Label bucket j with its associated interval:
                  [t_0 + j · norm(x), t_0 + (j + 1) · norm(x)).
(3)         Bucket x's children by their D-values.
        t refers to the start of the current bucket's interval (Initially t = t_0.)
        While S ∩ C_x ≠ C_x and t < b
            While bucket [t, t + norm(x)) is not empty
(4)             Choose a suitable node y from bucket [t, t + norm(x))
                Visit(y, [t, t + norm(x)))
                If S ∩ C_y ≠ C_y, put y in bucket [t + norm(x), t + 2 · norm(x))
            t := t + norm(x)
```

Some lines which need elaboration are marked by a number.

1. Visiting vertices and relaxing edges is done just as in Dijkstra's algorithm. Relaxing an edge (u, v) may cause an inactive ancestor of v in the CH to be bucketed or *re-bucketed* if relaxing (u, v) caused its D-value (tentative distance) to decrease.

2. Buckets in the bucket array represent consecutive intervals of width $norm(x)$, which together form an interval that contains $d(s, v)$ for all $v \in C_x$. We will refer to buckets by their place in the bucket array (e.g. the first bucket) or by the endpoints of the interval they represent (e.g. bucket $[t, t + norm(x))$). There is some subtlety to choosing the starting point t_0 of the first bucket. The concern is that we may have a *fractional* interval left over[1] if b, the end of the given interval, is not aligned with $t_0 + q \cdot norm(x)$ for some q. As in [PR02], we choose the initial t_0 as follows: if $D(x) + diam(C_x) < b$ then we will not reach b anyway and the alignment problem does not arise; set $t_0 = D(x)$. Otherwise, count back from b in units of $norm(x)$; find the minimum q s.t. $t_0 = b - q \cdot norm(x) \leq D(x)$. One can also show that, because of the wide separation in edge-lengths between strata, the fractional interval problem does not arise when Visit makes inter-stratum recursive calls. Indeed, this motivated our definition of strata.

3. The only time we ask for the D-value of a CH node is when its parent has been visited, but it has yet to be visited. Gabow's [G85] split-findmin data-structure handles updating and querying D-values.

4. Hagerup noted that Invariant 0 is not maintained if nodes from the same bucket are visited in any order; this is in contrast to the [Tho99, PR02] algorithms for undirected graphs, where nodes may be visited in arbitrary order. In [Hag00] it is shown that Invariant 0 can be maintained if nodes from the same bucket are visited in an order consistent with a topological ordering of C_x^{DAG}. Hagerup first assigns numbers in $\{1, 2, \ldots, |V(C_x^c)|\}$ to the vertices in C_x^c consistent with such an ordering, then uses a van Emde Boas heap [vEKZ77] to prioritize nodes within the same bucket. The overhead for managing the van Emde Boas structure is $O(n \log\log n)$ in total.

A CH node y is bucketed on two occasions: when its parent node x is first visited (item 2) or when some edge $(\cdot, v)$, $v \in C_y$ is relaxed (item 1). We will actually think of the first kind of bucketing operation as an edge relaxation too. When x is first visited, $D(y)$ corresponds to a path P_{sy} from s to y, hence bucketing y according to its D-value is tantamount to *re-relaxing* the last edge in P_{sy}. We are concerned with both kinds of edge relaxations, of which there are no more than $m + 2n = O(m)$.

4.3 A Lower Bound on Hagerup's Algorithm

It is not difficult to show that Dijkstra's algorithm is "just as hard as sorting", that is, producing the vertices in order of their d-values is just as hard as sorting n numbers. This implies an $\Omega(m + n \log n)$ lower bound on the complexity of Dijkstra's algorithm *in the comparison-addition model*, and tells us that we must alter our approach or strengthen the model in order to obtain faster shortest path algorithms. In this section we give a similar lower bound on Hagerup's algorithm [Hag00] in the comparison-addition model: we show that, even given the graph's component hierarchy, it too requires $\Omega(m + n \log n)$ operations.

All CH-based algorithms satisfy the following Property:

Property 1 *If $u, v \in V(C_x)$ and $d(s, v) \geq d(s, u) + norm(x)$, then u must be visited before v.*

A permutation of the vertices is *compatible* with a certain edge-length function if visiting the vertices in that order does not violate Property 1. We show that there is a directed graph and a family of $n^{\Omega(n)}$ distinct edge-length functions, no two of which share a compatible permutation. It is worth noting that this lower bound does not extend to undirected graphs — see [PR02].

Consider the graph depicted in Figure 1. It consists of the source vertex s and a large strongly connected component C_x containing the remaining $n-1$ vertices. The C_x subgraph is organized a little like a broom; it has a "broom stick" of $k-1$ vertices, whose head is w and whose tail connects to $n-k$ vertices (the "bush"), each of which is connected back to w (in the Figure w is drawn twice to avoid crossing lines.) All these edges have length $norm(x)$. The source s has one edge of length zero connecting it to w, and $n-k$ edges connecting it to the $n-k$ vertices in the broom's bush. Each of these $n-k$ edges takes on lengths of the form

[1] Having fractional intervals left over is not a problem in terms of correctness, but it does complicate the analysis.