Computing Undirected Shortest Paths with Comparisons and Additions[*]

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

TR-01-12

May 2, 2001

Abstract

We study undirected shortest paths problems in a natural model of computation, namely one which gives us two numerical operations: comparisons and additions. This is the model assumed by such standards as Dijkstra's algorithm, the Bellman-Ford algorithm, and the Floyd-Warshall algorithm, and is the usual model for proving lower bounds on shortest path problems.

We present an algorithm for undirected single source shortest paths (SSSP) with arbitrary real edge weights which performs $O(SSSP(m,n) + m \log \alpha(m,n) + n \log \log r)$ comparisons and additions. Here $SSSP$ is the comparison-addition complexity of the problem, r the ratio of the maximum-to-minimum edge length, and α Tarjan's inverse-Ackermann function. By the usual convention, m and n are the number of edges and vertices respectively. Our algorithm is implementable on a pointer machine with time complexity $O(m\alpha(m,n) + n \log \log r)$. This represents an improvement over Dijkstra's algorithm so long as $r < 2^{n^{o(1)}}$.

For the undirected all pairs shortest paths (APSP) problem we present an algorithm that runs in time $O(mn\alpha(m,n))$ time while performing $O(mn \log \alpha(m,n))$ comparisons and additions. This time bound improves on the best results known for this problem when the input graph is sparse, i.e., when $m = o(n \log n)$.

Our algorithms make extensive use of the graph's minimum spanning tree in order to compute SSSP quickly, and our approach is based on a refinement of Thorup's component hierarchy data structure, which was developed under the more powerful RAM model.

[*]This work was supported by Texas Advanced Research Program Grant 003658-0029-1999 and NSF Grant CCR-9988160. Seth Pettie was also supported by an MCD Fellowship.

1 Introduction

The computation of shortest paths, either single-source (SSSP) or all-pairs (APSP), is one of the oldest and most basic graph optimization problems studied in computer science. It is then a testament to the solid foundations of our field that the standard textbook algorithms of Dijkstra [Dij59] and Bellman-Ford [CLR90] are *still* the best algorithms for the positively weighted and general SSSP problem, resp. For the all pairs shortest paths problem, the method of repeatedly applying Dijkstra's algorithm starting from each vertex has stood the test of time for sparse graphs with positive edge weights. Aside from their good asymptotic and real performance these algorithms are particularly appealing for their simplicity. They make no extra assumptions on the graph or edge weights, and they require only two operations to act on edge weights: comparisons and additions. Nearly all recent advances in the SSSP and APSP problems have come by either assuming integral edge weights, possibly bounded in magnitude, or a powerful random access machine. As an intellectual pursuit, if not a practical one, we believe that shortest paths problems are properly studied under the comparison-addition model.

Let us review some developments on the SSSP problem. For arbitrary edge weights the Bellman-Ford algorithm runs in $O(mn)$ time, where m and n are the number of edges and vertices, respectively. For arbitrary *integral* edge weights, Gabow and Tarjan [GT89] (see also [G85b]) gave an SSSP algorithm running in time $O(m\sqrt{n}\log(nU))$, where the magnitude of all edge weights is bounded by U. For reasonable U, say polynomial, this is a substantial improvement over the Bellman-Ford algorithm. For the case of positively weighted directed graphs, Dijkstra's algorithm [Dij59] can be implemented using Fibonacci heaps [FT87] in $O(m + n\log n)$ time. As Dijkstra's algorithm sorts the vertices by distance from the source, it can be shown that no implementation of Dijkstra's algorithm in the comparison-addition model can do better. (For graphs that are not very sparse – i.e., when $m = \Omega(n\log n)$ – Dijktra's algorithm runs optimally in $O(m)$ time.)

One way around the $n\log n$ sorting bottleneck for sparse graphs is to increase the power of the model. Beginning with the algorithm in [AM+90], there has been a concerted effort to obtain faster SSSP algorithms in the RAM model. The current fastest algorithms in this model are due to Thorup [Tho96], running in time $O(m\log\log n)$, and Raman [Ram97], running in time $O(m+n(\log n)^{\frac{1}{3}+\epsilon})$. Other RAM algorithms can match or improve these bounds when the machine word size is polylogarithmic or if the maximum edge weight is not too large [Tho96, Ram96, Ram97, CGS97, Hag00].

For the APSP problem with arbitrary positive edge weights, Dijkstra's algorithm can be applied n time to obtain an $O(mn + n^2\log n)$ time algorithm. An algorithm running in time $O(m^*n + n^2\log n)$ time is given in Karger et al. [KKP93], where m^* is the number of edges that appear in some shortest path. The Floyd-Warshall and 'min-plus' matrix multiplication algorithms run in $O(n^3)$ time [AHU74, CLR90], and these give good performance for dense graphs. Fredman [F76] presented an elegant, though difficult to implement min-plus matrix multiplication algorithm which makes $O(n^{2.5})$ comparisons and additions. The best implementations of Fredman's algorithm [F76, Tak92] are only marginally better than n^3. There is a large body of work on fast APSP algorithms for dense graphs that restrict the range of values allowed for edge weights (see [S95, GM97] for undirected graphs and [AGM97, Z98] for directed graphs).

In this paper we study undirected shortest paths problems. Since these problems are only interesting on positive edge lengths[1], Dijkstra's algorithm may be used with its usual time bounds. Until recently no SSSP techniques specific to undirected graphs were known. Then, Thorup [Tho99]

[1]If a negative length edge appears in the source's connected component, the shortest distance to all vertices in that component is $-\infty$ if, as is usual, edges may be repeated. If we disallow repeated edges on a shortest path the problem then becomes NP-hard.

showed that by assuming integral edge weights and a random access machine, undirected SSSP could be solved in linear time. His algorithm, based on traversing the graph's *component hierarchy (C.H.)*, is a departure from Dijkstra's approach in that it is non-greedy. Vertices are visited one by one, as in Dijkstra's algorithm, though not necessarily in increasing distance from the source. Since Thorup's C.H. relies on the RAM's power for its efficiency[2], it was unclear whether this approach to SSSP was limited to the RAM, or if it represented a technique of more fundamental value.

In this paper we demonstrate that the component hierarchy approach can indeed be adapted to the comparison-addition model; in particular we give the following two results:

- An undirected SSSP algorithm which makes in the vicinity of $m \log \alpha(m,n) + n \log\log r$ comparisons and additions, where r bounds the ratio of any two edge weights. The algorithm runs on a pointer machine in $O(m\alpha(m,n) + n\log\log r)$ time. This gives a better time bound than Dijkstra's if $r \le 2^{n^{o(1)}}$; in particular, if $r \le n^{(\log n)^{O(1)}}$, the running time is $O(m + n\log\log n)$.
- An undirected APSP algorithms that performs $O(mn \log \alpha(m,n))$ comparisons and additions and runs in $O(mn\alpha(m,n))$ time on a pointer machine. This result improves on Dijkstra's algorithm and Karger at al. [KKP93] when $m = o(n \log n)$.

The main results we prove are stated below.

Theorem 1.1 *The undirected single source shortest path problem can be solved using $O(SSSP(m,n)+ m\log\alpha(m,n) + n\log\log r)$ comparisons and additions, where m and n are the number of edges and vertices, α the inverse-Ackermann function, $SSSP(m,n)$ the comparison-addition complexity of undirected SSSP, and r the ratio of the maximum to minimum edge length.*

Corollary 1.1 *Undirected SSSP can be solved in $O(m\alpha(m,n) + n\log\log r)$ time on a pointer machine using the same number of comparisons & additions claimed in Theorem 1.1.*

Theorem 1.2 *The undirected all pairs shortest path problem can be solved with $O(mn\log\alpha(m,n))$ comparisons and additions in $O(mn\alpha(m,n))$ time.*

Our method for constructing the C.H. differs from [Tho99, Hag00] in that it depends heavily on knowing the structure and edge lengths of the graph's minimum spanning tree. Additionally, we show that the comparison-addition complexity of SSSP is at least as large as the comparisons-only complexity of MST.

2 Preliminaries

2.1 The Model

When the solution to a numerical problem is defined by a set of linear inequalities (e.g. all shortest path problems), the comparison-addition model presents itself as the most fundamental computational model. In this model the input consists, among other things, of m real numbers, initially

[2]Thorup's algorithm seems to require RAM-based priority queues to achieve linear time. Thorup noted that a simplified algorithm runs in $O(\log U + m\alpha(m,n))$ time where U is the largest edge weight, using the following operations: comparisons, additions, most significant bit, and bucketing. In terms of information gained, one bucketing operation amounts to a branch with up to $\max\{n, \log U\}$ outcomes; it is therefore not surprising that an SSSP algorithm faster than Dijkstra's could be developed using these operations.

stored in the variables $v_1, \ldots, v_m$. Each variable v_i, $0 < i < \infty$, can hold one real number and may only be manipulated[3] by *additions*, of the form $v_i := v_j + v_k$, and comparisons, of the form $v_i \overset{?}{<} v_j$. An algorithm then chooses which operations to perform based on the outcome of previous comparisons. We are primarily interested in the number of numerical operations. Any overhead required to examine the non-numerical input or decide which comparisons and additions to make is ignored for simplicity.

Although this model does not include subtraction as a primitive operation it may be simulated with a constant factor loss in efficiency. We may represent a real $q_1 = a_1 - b_1$ as two reals (kept in separate variables). An addition, $q_1 + q_2 = (a_1 + a_2) - (b_1 + b_2)$, or a subtraction, $q_1 - q_2 = (a_1 + b_2) - (a_2 + b_1)$, may be accomplished with two actual additions. A comparison, say $q_1 \overset{?}{<} q_2 \;\equiv\; a_1 + b_2 \overset{?}{<} a_2 + b_1$, may be accomplished with two actual additions and one comparison. The power of these simple algebraic transformations was explored by Fredman [F76] who demonstrated that all-pairs shortest paths could be solved with $O(n^{2.5})$ comparisons and additions, though the required overhead of Fredman's algorithm is considerably larger. We use this simulation of subtraction in Section 4.

In our algorithms it will also be necessary to approximate the ratio of certain input variables. Suppose that v_1 and v_m are known to be the smallest and largest input variables. Then the ratios $\{\frac{v_i}{v_1} \;:\; 1 \le i \le m\}$ can be approximated to within a $1+\epsilon$ factor using $O(\frac{1}{\epsilon} + \log(\frac{v_m}{v_1}) + m(\log\log\frac{v_m}{v_1} + \log\frac{1}{\epsilon}))$ comparisons and additions. Consider first the case $\epsilon = 1$. We generate the set $\mathcal{D} = \{v_1, 2v_1, 4v_1, \ldots, 2^{\lceil \log(\frac{v_m}{v_1})\rceil} v_1\}$ using addition for simple doubling, then perform a binary search over $\mathcal{D}$ for each variable v_i, thus approximating the desired ratios to within a factor of two. For smaller ϵ we simply continue the binary search until the lower and upper bounds on v_i are sufficiently close. Each comparison in this binary search is either of the form $a \cdot v_1 \overset{?}{<} v_i$ where a is the sum of at most $\log(\frac{1}{\epsilon})$ elements from $\mathcal{D}$, or of the form $b \cdot v_1/c \overset{?}{<} v_i$, where $1 \le b, c \le \frac{1}{\epsilon}$ and c is a power of two. The first kind of comparison is simple to handle. The second kind becomes simpler if reduced to a comparison without division: $b \cdot v_1 \overset{?}{<} c \cdot v_i$. There are $\frac{1}{\epsilon}$ possible terms which can appear on the left side of the $\overset{?}{<}$ and $m\log(\frac{1}{\epsilon})$ terms on the right side, all of which can be easily computed in advance.

There are some rather weak lower bounds in the comparison-addition model for shortest paths problems. Spira and Pan [SP73] showed that regardless of additions, $\Omega(n^2)$ comparisons are necessary to solve SSSP on the complete graph. Karger et al. [KKP93] proved that all-pairs shortest paths requires $\Omega(mn)$ comparisons if all summations correspond to paths in the graph. In straight-line computation (using operations of the form $v_i := \min\{v_j, v_k\}$ in lieu of comparisons) Kerr [K70] proved a lower bound of $\Omega(n^3)$ on the all-pairs shortest path problem. Graham et al. [G+80] showed that for the all pairs shortest distance problem, any information-theoretic argument (in the comparison-addition model) could yield only $\Omega(n^2)$ bounds on the number of comparisons. Similarly, no information-theoretic superlinear lower bound on SSSP can be obtained as this is tantamount to computing the log of the number of spanning trees.

One interesting aspect of our SSSP algorithm is that for certain ranges of m, n, and r (e.g., $m = O(n)$, $r = 2^{2^{o(\alpha(m,n))}}$) its comparison-addition complexity is unknown: it depends upon the decision-tree complexity of the minimum spanning tree problem. To date the best upper bound on MST, due to Chazelle [Chaz00], is $O(m\alpha(m,n))$. The following Lemma says that computing the MST will never be a bottleneck for an SSSP algorithm.

[3]Depending on the problem at hand one may also wish to allow addition and multiplication of constants.

Lemma 2.1 *The comparison-only complexity of MST is no more than the comparison-addition complexity of SSSP.*

Proof: Consider only undirected graphs with edge weights of the form $2^{i \cdot cn^2}$ for *distinct* i. It is easily shown that the MST and shortest paths tree for these graphs are identical, regardless of the source. Moreover, in k steps no SSSP algorithm can generate a number which is the sum of more than 2^k edge lengths. Since no SSSP (or MST) algorithm takes more than cn^2 steps, every comparison made by the SSSP algorithm is immediately reducible to a comparison between two edge lengths; hence a SSSP algorithm may be reduced to an addition-free MST algorithm. $\Box$

2.2 Notation

The input is a weighted undirected graph $G = (V, E, \ell)$ and a distinguished source vertex $s \in V$. Here $\ell : E \rightarrow \mathcal{R}^+$ assigns a positive *length* to each edge and the length of a path $< v_0, v_1, \ldots, v_k >$ is defined as $\sum_{i=0}^{k-1} \ell(v_i, v_{i+1})$. We let $\ell_{\mathbf{min}}$ denote the minimum length edge. For $R \subseteq V$ let $d_R(v)$ be the length of a shortest path from s to v in the subgraph induced by $R \cup \{v\}$, and let $d(v) = d_V(v)$. Using this notation the SSSP problem is to find $d(v)$ for all $v \in V$. By convention $|V| = n$ and $|E| = m$. We use the term *vertex* to mean an element of V and *node* to mean a vertex of any other graph we construct during the course of our algorithm. All data structures required of our algorithm may be implemented in the stated time bounds using just a pointer machine [Tar79].

2.3 Dijkstra's Algorithm.

A *tentative* distance $D(v) \geq d(v)$ is maintained for all v, as well as a set S of *visited* vertices whose distance from s has been determined. One invariant is maintained.

Invariant 0. *For all $v \in S$, $D(v) = d(v)$, and for all $v \notin S$, $D(v) = d_S(v)$.*

Initially $S = \emptyset$, $D(s) = 0$, and $D(v) = \infty$ for $v \neq s$. In each step of Dijkstra's algorithm an unvisited vertex with minimum tentative distance is visited, set S is augmented with this vertex and tentative distances updated appropriately. It is simple to prove that Invariant 0 is maintained. Eventually $S = V$, and therefore $D(v) = d(v)$ for all v. An unfortunate aspect of Dijkstra's algorithm is that its complexity is inherently as bad as sorting the d-values of all vertices. Notice that Dijkstra's algorithm remains perfectly correct if *any* vertex v is visited for which $D(v)$ is provably equal to $d(v)$. Although this modified algorithm is not *inherently* as difficult as sorting, it does not suggest an efficient implementation.

The recent algorithms of Thorup [Tho99] (for undirected graphs) and Hagerup (for directed graphs) [Hag00] are implementations of this non-greedy version of Dijkstra's algorithm. Each uses the notion of a *component hierarchy* – a structure based on classifying edges by length – to decide when vertices are ready to be visited.

2.4 Thorup's Algorithm.

Thorup's algorithm [Tho99] maintains Invariant 0 but might not visit vertices greedily. It works by simulating Dijkstra's algorithm in a piecemeal fashion, identifying parts of the problem which can be solved independently of one another.

Definition 2.1 *A subgraph H is* **safe** *over interval I w.r.t. vertex set X if for all $v \in H$ with $d(v) \in I$, $d_{X \cup H}(v) = d(v)$.*

In other words, if $X = S$ is the set of visited vertices, a subgraph is safe over I if one can correctly solve the shortest path problem for $v \in H$ s.t. $d(v) \in I$, without looking at parts of the graph outside of $H \cup S$.

The following Lemmas are not difficult to prove. See [Tho99] for proofs of similar claims.

Lemma 2.2 *Suppose H is safe over $[a, b)$ w.r.t. X, H^t is the subgraph of H induced by edges of length less than t, and $\mathcal{H}^t$ is the set of connected components of H^t. Then all members of $\mathcal{H}^t$ are safe over $[a, \min\{a + t, b\})$ w.r.t. X.*

Lemma 2.3 *If H is safe over $[a, b)$ w.r.t. X, then for $a + t < b$ and $X' \supseteq X \cup \{v \in H : d(v) \in [a, a + t)\}$, H is also safe over $[a + t, b)$ w.r.t. X'.*

Lemma 2.2 says that a subgraph which is safe over some interval can be decomposed into smaller subgraphs, each of which is safe over a shorter interval. Lemma 2.3 says that if a subgraph H is safe over $[a, b)$, then once all vertices in H whose d-values lie in $[a, a + t)$ are visited, H will be safe over $[a + t, b)$. Used together, Lemmas 2.2 and 2.3 form the basis of a recursive SSSP algorithm which is parameterized only by choices of t. In the case of Thorup's algorithm, edge lengths are assumed to be integers and t is always chosen to be the largest power of two which disconnects H. Given this system for choosing the t parameter, Thorup precomputes a *component hierarchy* to assist the recursive invocations of his algorithm. A node x at level j of the component hierarchy represents a maximal connected subgraph C_x induced by edges with length less than 2^j. Its children are those nodes $\{x_i\}$ whose associated $\{C_{x_i}\}$ are maximal connected subgraphs of C_x restricted to edges with length $< 2^{j-1}$. Leaves of the hierarchy correspond to vertices of the graph so we do not differentiate between the two in our notation. Below we give the main recursive procedure of Thorup's algorithm, called `Visit`. It takes a component hierarchy node x and an interval I with the guarantee that C_x is safe over I w.r.t. the current set of visited vertices S. If x is a leaf (i.e. a vertex of G), it follows that $D(x) = d(x)$. Extending the D-value notation to component hierarchy nodes, we let $D(x)$ denote the minimum D-value over all descendant leaves of x. Initially $S := \emptyset$ and `Visit`($root$, $[0, \infty)$) is called on the root of the component hierarchy.

```
Visit($x, [a, b)$ )
  1.   If $b \le a$ or $C_x \subseteq S$ return.
  2.   If $x$ is a leaf, set $S := S \cup \{x\}$, update $D$-values and return.
  3.   Let $t := 2^{level(x)-1}$
  4.   For each child $y$ of $x$ s.t.  $D(y) \in [a, a + t)$,
  5.      Visit($y, [a, a + t)$ )
  6.   Visit($x, [a + t, b)$ )
```

Making Thorup's algorithm efficient amounts to solving three interrelated problems: building the component hierarchy, updating D-values for component hierarchy nodes when leaves are visited (line 2), and quickly deciding which recursive calls to make (line 4).

To build the component hierarchy and update D-values Thorup resorts to RAM priority queues; deciding which recursive calls to make is accomplished by word shifts and bucketing. In other words, new techniques are needed to adapt the component hierarchy approach to the comparison-addition model. In Section 3 we describe the properties of a component hierarchy which is particularly suited to the comparison-addition model. The construction of this hierarchy, which is closely linked to the structure and weighting of the graph's minimum spanning tree, is described in Sections 4 and 5. It takes $O(MST(m, n) + n \log \log r)$ time to construct our hierarchy, where $MST(m, n)$ is the comparison complexity of MST. In Section 6 we describe our algorithm, deferring its analysis,