Computing Shortest Paths with Comparisons and Additions

Seth Pettie[*†] Vijaya Ramachandran[*]

Abstract

We present an undirected all-pairs shortest paths (APSP) algorithm which runs on a pointer machine in time $O(mn\alpha(m,n))$ while making $O(mn \log \alpha(m,n))$ comparisons and additions, where m and n are the number of edges and vertices, respectively, and $\alpha(m,n)$ is Tarjan's inverse-Ackermann function. This improves upon all previous comparison & addition-based APSP algorithms when the graph is sparse, i.e., when $m = o(n \log n)$.

At the heart of our APSP algorithm is a new single-source shortest paths algorithm which runs in time $O(m\alpha(m,n) + n \log\log r)$ on a pointer machine, where r is the ratio of the maximum-to-minimum edge length. So long as $r < 2^{n^{o(1)}}$ this algorithm is faster than *any* implementation of Dijkstra's classical algorithm in the comparison-addition model.

For *directed* graphs we give an $O(m + n \log r)$-time comparison & addition-based SSSP algorithm on a pointer machine. Similar algorithms assuming integer weights or the RAM model were given earlier.

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 surprising then, that for real-weighted graphs the classic textbook SSSP algorithms of Bellman-Ford [CLR90] and Dijkstra [Dij59, FT87] (for positive weights) remain unimproved, and for sparse graphs, solving APSP with n applications of Dijkstra's SSSP algorithm remains unbeaten. Nearly all progress on the SSSP and APSP problems has come through using scaling methods, integer matrix multiplication, or RAM/integer-based techniques.

We return to the problem of solving shortest paths on real-weighted graphs. We assume a simple model: a pointer machine, which is weaker than the RAM, and only two operations to act on reals: comparisons and additions. This model is strong enough to implement the textbook shortest path algorithms of Dijkstra, Bellman-Ford, Floyd-Warshall, and 'min-plus' matrix multiplication.

Let us review what is known about 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, Goldberg [G95] (improving [G85b, GT89]) gave an SSSP algorithm running in time $O(m\sqrt{n} \log N)$, where all edge weights are greater than $-N$. For reasonable N, 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.

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 word size is polylogarithmic or if the maximum edge weight is small [Tho96, Ram96, Ram97, CGS97, Hag00].

For the APSP problem with arbitrary positive edge weights, n applications of Dijkstra's algorithm solves APSP in $O(mn + n^2 \log n)$ time. A potential improvement to this algorithm is given by Karger et al. [KKP93] which runs in time $O(m^*n + n^2 \log n)$, where m^* is the number of distinct edges appearing in any 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

[*]This work was supported in part by Texas Advanced Research Program Grant 003658-0029-1999 and NSF Grant CCR-9988160. Address: Dept. of Computer Sciences, The Univ. of Texas at Austin, Austin, TX 78712. Email: {seth,vlr}@cs.utexas.edu

[†]Seth Pettie also supported by an MCD Graduate Fellowship

restrict the range of edge weights to relatively small integers (see [S95, GM97, SZ99] for undirected graphs and [AGM97, Z98] for directed graphs).

We have focussed on shortest path algorithms for general graphs. There are several faster algorithms for restricted inputs, e.g. [Fre91, KS93, HK+97, FR01, Tho01] for planar graphs and [SR94, EH97, EH99, HS99, CKT00] for certain geometric inputs.

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] showed that by assuming integral edge weights and a random access machine, undirected SSSP could be solved in linear time. Thorup noted that his algorithm can be thought of as a recursive version of Dinic's [Din78] bucketing strategy. Thorup's algorithm, based on traversing the graph's *component hierarchy* (C.H.), is a departure from Dijkstra's 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 adapt the component hierarchy approach to the comparison-addition & pointer machine model of computation. We give two (related) methods for constructing a component hierarchy. Once the C.H. is built, using either method, we show how to compute SSSP from *any* source in $O(m\alpha(m,n))$ time. Our methods for constructing the C.H. differ from [Tho99, Hag00] in that they depend heavily on knowing the structure and edge lengths of the graph's minimum spanning tree. Our main results are the following:

THEOREM 1.1. *Let* $CH = \min\{n\log n, n\log\log r\}$, *where r is the ratio of the maximum to minimum edge length. The undirected s-sources shortest paths problem is solved on a pointer machine in $O(sm\alpha(m,n) + CH)$ time using $O(sm\log\alpha(m,n) + MST(m,n) + CH)$ comparisons and additions, where MST(m,n) is the comparison complexity of the minimum spanning tree problem.*

COROLLARY 1.1. *The undirected all-pairs shortest path problem is solved on a pointer machine in $O(mn\alpha(m,n))$ time using $O(mn\log\alpha(m,n))$ comparisons and additions.*

COROLLARY 1.2. *Undirected single source shortest paths are computed on a pointer machine in $O(m\alpha(m,n) + n\log\log r)$ time using $O(m\log\alpha(m,n) + n\log\log r + MST(m,n))$ comparisons and additions.*

In Theorem 1.1, the CH term becomes negligible when the number of sources $s > \log n$. Corollary 1.1 improves Dijkstra's APSP algorithm and Karger et al. [KKP93] when $m = o(n\log n)$, and Corollary 1.2 is superior to Dijkstra's SSSP algorithm when $r < 2^{n^{o(1)}}$. Our recent experimental work [PR01a] indicates that our algorithm easily outperforms Dijkstra's algorithm for APSP, and even when computing SSSP from a few dozen sources in sparse graphs.

In Section 6 we observe that a bucketing scheme we use gives a pointer machine-based SSSP algorithm for *directed* graphs running in time $O(m + n\log r)$, an improvement over Dijkstra's algorithm if $\log r = o(\log n)$. Similar observations were made in [AM+90, G01] for the RAM model or with integer edge weights.

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 stored in the variables $v_1, \dots, v_m$. Each variable v_i, $0 < i < \infty$, holds one real number and may only be manipulated 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, and therefore define 'comparison-addition complexity' to be the number of such operations. For other non-numerical tasks, such as managing data structures, we assume only a *pointer machine* [Tar79]. The pointer machine makes few assumptions while still being able to simulate functional programming languages such as pure Lisp or ML. Unlike in the RAM model, pointers are their own data type, subject

[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 becomes NP-hard.

[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, word shifts, 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.

only to comparison for equality and dereferencing (following the pointer). There is never more than a logarithmic factor difference between pointer machine complexity and RAM complexity, however this small separation can be very important in problems with near-linear complexity (such as SSSP). Using the pointer machine ensures that we are not sweeping a logarithmic factor under the rug.

Although the comparison-addition 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 $q_1 <^? q_2 \equiv a_1 + b_2 <^? a_2 + b_1$, 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.

Division is, of course, not a primitive operation and it cannot be simulated exactly. For our algorithms we are content to have a decent approximation of division. Suppose that v_1 and v_m are known to be the smallest and largest input variables, resp. We can approximate the ratios $\{\frac{v_i}{v_1} : 1 \le i \le m\}$ 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 as follows. 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 $a <^? v_i$ where a is the sum of at most $\log(\frac{1}{\epsilon})$ elements from $\mathcal{D}$, or of the form $b \cdot v_1/c <^? v_i$, 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: $b \cdot v_1 <^? c \cdot v_i$. There are $\frac{1}{\epsilon}$ possible terms which can appear on the left side of the $<^?$ 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.

Our assumption of real-valued edge lengths is particularly relevant in the context of recent progress on shortest paths problems. Nearly all new techniques rely heavily on the assumption of integral edge lengths. No adaptations of 'scaling' algorithms [G95, G85b, GT89] to the reals or the comparison-addition model are known; the RAM-based priority queue algorithms [AM+90, Tho96, Ram96, Ram97, CGS97, Hag00] require word-sized integers, as do the matrix multiplication-based algorithms in [S95, GM97, SZ99, AGM97, Z98].

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 – see [PR00]. 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.

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 an SSSP algorithm may be reduced to an addition-free MST algorithm.

2.2 Notation

The input is a weighted undirected graph $G = (V, E, \ell)$ and a distinguished source vertex $s \in V$. Here $\ell : E \to \mathcal{R}^+$ 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.

2.3 Dijkstra's Algorithm

Dijkstra's algorithm [Dij59] maintains a *tentative* distance $D(v) \geq d(v)$ 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 vertices $v \in S$, $D(v) = d(v)$, 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 $d(v)$. This modified algorithm is not *inherently* as difficult as sorting yet 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 Non-greedy Approach

Thorup's SSSP 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. *For a subgraph H and real interval I, let $H_I = \{v \in H : d(v) \in I\}$. A subgraph H is* **safe** *over interval I w.r.t. vertex set X if for all $v \in H_I$, $d_{X \cup H_I}(v) = d(v)$.*

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

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. The procedure `Visit` given below, takes a subgraph H and an interval I with the guarantee that H is safe over I w.r.t. the set of visited vertices S. After `Visit` is finished all vertices in H_I will be visited and added to S. Initially $S = \emptyset$, and `Visit`$(G, [0, \infty))$ is called on input graph G. Invariant 0 is maintained at all times.

```
Visit($H$, $[a, b)$)
 1. If $b \leq a$ or $V(H) \subseteq S$ return.
 2. If $H$ is a vertex and $D(H) \in [a, b)$,
       $S := S \cup \{H\}$, update $D(\cdot)$ and return.
 3. Choose $t \leq b - a$, let $\mathcal{H}^t$ be set of
    connected components of $H$ restricted
    to edges shorter than $t$.
 4. For each subgraph $h \in \mathcal{H}^t$,
       Visit($h, [a, a + t)$)
 5. Visit($H, [a + t, b)$)
```

In the case of Thorup's algorithm [Tho99], edge lengths are assumed to be integers and t is always chosen to be the largest power of two such that $\mathcal{H}^t$ contains at least two subgraphs. Given this system for choosing the t parameter, Thorup precomputes a *component hierarchy* to assist the recursive invocations of his algorithm.

Making this skeletal version of Thorup's algorithm efficient amounts to solving three interrelated problems: building the component hierarchy, so that $\mathcal{H}^t$ is easily found, updating D-values in order to maintain Invariant 0, and making only those essential recursive calls. Any recursive call in line 4 or 5 which does not cause a vertex to be visited is inessential.

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, a complete rethinking of Thorup's algorithm is needed to bring the component hierarchy approach to the comparison-addition model. In Section 3 we describe properties of a component hierarchy which are 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 Section 4. It takes $O(MST(m,n) + n \log\log r + \log r)$ time to construct our hierarchy, where $MST(m,n)$ is the comparison complexity of the minimum spanning tree problem. In Section 5 we describe our algorithm, deferring its analysis, and that of a "lazy" bucketing scheme to Section 6. Once our component hierarchy is constructed, SSSP can be computed *from any source* in $O(m \log\alpha(m,n))$ comparisons and additions. Finally, in Section 8 we give a faster undirected APSP algorithm with no dependence on r.

3 A Refined Component Hierarchy

Our algorithm uses a component hierarchy $\mathcal{CH}$ which is a refinement of a cruder hierarchy dubbed $\mathcal{CH}^1$. $\mathcal{CH}^1$ is a rooted tree sectioned into *levels*, where the leaves are at level 0. The level j nodes of $\mathcal{CH}^1$ correspond to connected components of the graph restricted to edges with length $< \ell_{\mathbf{min}} \cdot 2^j$. Since by definition no edge has weight less than $\ell_{\mathbf{min}}$, level 0 nodes correspond to vertices in the graph; we do not differentiate between the two in our notation. To keep the size of $\mathcal{CH}^1$ linear we splice out all one-child nodes.

To derive $\mathcal{CH}$ we replace each internal $\mathcal{CH}^1$ node x having children $\{x_i\}$ with a tree H_x rooted at x with leaf set $\{x_i\}$. The internal nodes of H_x, called *clusters*, represent their descendant leaves in $\mathcal{CH}$ and naturally correspond to a subgraph of G induced by those leaves. Even in $\mathcal{CH}$ we will continue to refer to x and the $\{x_i\}$ as $\mathcal{CH}^1$ nodes. H_x nodes will have several properties, the most important of which is that a cluster's size (sum of edge lengths in its minimum spanning tree) will not be too much smaller than the diameter of its parent cluster. The effect this property has on the running time of our algorithm will become clear in Sections 5 and 6. Before we describe the properties of H_x we must introduce some additional definitions and structures.

Let M be the minimum spanning tree (MST) of G, which is computed optimally using the algorithm of Pettie & Ramachandran [PR00]. For a $\mathcal{CH}^1$ node x let M_x be the subgraph of M corresponding to x. If we extend this notation to cluster nodes, letting M_z be the graph induced by the descendants of a cluster node z, it turns out that M_z is not necessarily connected. In Section 4 we ensure connectedness by adding *dummy nodes* to H_x, each of which represents a little piece of the MST M.

We let the *mass* of a subgraph H be defined as $\sum_{e\in H} \ell(e)$ and its *diameter* be the largest shortest path distance between two vertices. We assign *ranks* to all $\mathcal{CH}$ nodes so that their mass and diameter satisfy the bounds given in Definition 3.1 and Properties 3.1 & 3.2. Briefly, a rank i cluster in H_x, for x a level j $\mathcal{CH}^1$ node, will have diameter no more than $4\tau_i^j$ and ideally has mass at least τ_i^j, $\tau_\cdot^\cdot$ defined below.

Let $\lambda_0 = 0$, $\lambda_1 = 10$, $\lambda_{i+1} = 2^{\lambda_i/2^i}$, and let
$$\tau_i^j = \ell_{\mathbf{min}} \cdot 2^{j-1} \cdot \lambda_i.$$

Below, x is a level j $\mathcal{CH}^1$ node and z an arbitrary rank i node of H_x.

Definition 3.1. *If $mass(M_z) < \tau_i^j$ then z is* **stunted**, *otherwise it is* **unstunted**.

Property 3.1. *At most half the children of a rank $i+1$ internal node are stunted. The unstunted children have rank i, the stunted rank no more than i.*

Property 3.2. *If z is an internal node of H_x then $diam(M_z) \le 4 \cdot \tau_i^j$.*

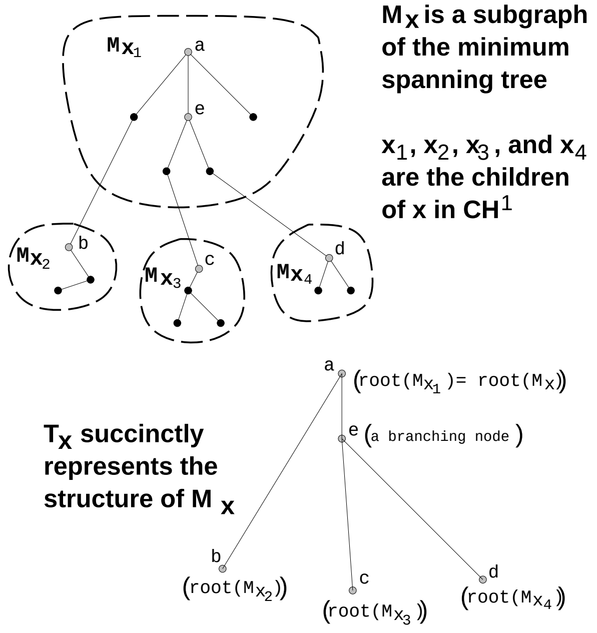
Our method for constructing H_x is described in Section 4, though an understanding of its construction is not necessary to understand the algorithms presented in Section 5 (SSSP) and Section 8 (APSP). The main result of Section 4 is given below.

Lemma 3.1. *The refined component hierarchy $\mathcal{CH}$ can be computed from scratch in $O(MST(m,n) + n\log\log r + \log r)$ time, where $MST(m,n)$ is the decision-tree complexity of MST and r bounds the ratio of any two edge weights in the MST.*

4 Construction of H_x

Recall that H_x is a rooted tree constructed to replace the $\mathcal{CH}^1$ node x, where the leaf set of H_x is the set of children of x. The internal nodes of H_x represent subgraphs of the MST and must satisfy Properties 3.1 & 3.2 governing the mass and diameter of these subgraphs (see Section 3). In principle, H_x could be constructed directly from M_x, though this would be inefficient. What we actually do is compute, for each $x \in \mathcal{CH}^1$, a succinct representation of M_x, called T_x, which contains enough information to compute H_x according to specs. T_x is defined below, however we refer the reader to [PR01] for a linear-time algorithm to construct all the T_x.

For the purpose of defining an ancestor relationship, we root the MST M at an arbitrary vertex. With this orientation let $root(M_x)$ be the root vertex of M_x and let $\{x_i\}$ be the children of x in $\mathcal{CH}^1$. By definition T_x is the subtree of M_x induced by the roots $\{root(M_{x_i})\}$ and those *branching vertices* of M_x which, if removed, divide $\{root(M_{x_i})\}$ into three or more groups – see Figure 1. The length of an edge in T_x, corresponding to a path in M_x, is then the length of the path.



M_{x_1}
a
e
M_X is a subgraph of the minimum spanning tree
x_1, x_2, x_3, and x_4 are the children of x in CH^1
M_{x_2}
b
M_{x_3}
c
M_{x_4}
d
a
(root(M_{x_1})= root(M_X))
e (a branching node)
T_X succinctly represents the structure of M_X
b
(root(M_{x_2}))
c
(root(M_{x_3}))
d
(root(M_{x_4}))

Figure 1: Example M_x and associated T_x

The τ_i^j values were defined in Section 3, however the only properties required of τ_i^j here are that $\tau_0^j = 0$, $\tau_1^j \geq \ell_{\mathbf{min}} \cdot 2^{j-1}$, and $\tau_{i+1}^j \geq 2\tau_i^j$

H_x is constructed from the bottom up while traversing T_x in a post-order fashion. A vertex $v \in T_x$ with children $\{v_k\}$ is only processed after the subtrees rooted at the $\{v_k\}$ have already been processed. After processing a child v_k we assume there is a sequence of sets $(S_0^k, S_1^k, \ldots)$ associated with v_k, where elements of S_i^k are rank i nodes: *roots* of previously constructed subtrees of H_x. The sets eventually associated with v will be named $(S_0, S_1, \ldots)$. All H_x nodes constructed so far satisfy Properties 3.1 & 3.2; in addition, two other properties are satisfied by the $\{S_i^k\}$.

Property 4.1. *For all i, k, $\sum_{z \in S_i^k} mass(M_z) < \tau_{i+1}^j$.*

Property 4.2. *For all i, k, $(\bigcup_{i' \leq i,\, z \in S_{i'}^k} M_z)$ is either empty or a connected subgraph containing v.*

Initially we let $S_i := \bigcup_k S_i^k$. Processing v amounts to restoring Property 4.1 (w.r.t. the $\{S_i\}$) and restoring Property 4.2 w.r.t. the parent of v, rather than v; this is achieved by *promoting* elements of the $\{S_i\}$. By 'Promote S_i to $S_{i'}$', we mean that all elements of S_i are removed from S_i and made the children of a newly created rank $i+1$ node, which is then added to the set $S_{i'}$. Typically $S_{i'}$ would be S_{i+1}. Property 4.1 basically says that after the subtree rooted at v_k is processed, no elements are ready to be promoted. Properties 4.1 and 4.2 together imply that for any $z \in S_i^k$ and a vertex w drawn from M_z, the distance from w to v is $\leq 2\tau_{i+1}^j$.

Processing v is done in several steps. The first two, described below, deal with merging the subtrees of the children of v. Later steps deal with the T_x edge linking v to its parent.

(1) If v is the root of M_y, for a child y of x in $\mathcal{CH}^1$, then
  (a) Find ℓ s.t. $\tau_\ell^j \leq mass(M_y) < \tau_{\ell+1}^j$.
  (b) Create a rank ℓ leaf node y; let $S_\ell := S_\ell \cup \{y\}$.
(2) Let i be maximal s.t. $\sum_{z \in S_i} mass(M_z) \geq \tau_{i+1}^j$.
  (a) Prematurely promote items in $S_{i'}$, for all $i' < i$.
  (b) Promote items in S_i to S_{i+1}.

After we initially set $S_i := \bigcup_k S_i^k$, some S_i may be too massive to satisfy Property 4.1; this is resolved in Step (2). Step (1) is for the case when v corresponds to $root(M_y)$ for some child y of x in $\mathcal{CH}^1$. Note that not all vertices of T_x are such roots. Step (2) only applies if there is such an S_i which violates Property 4.1. In Step (2a), by 'prematurely promote' we mean this: the first non-empty set S_{i_1} is promoted to the second non-empty set S_{i_2}, which is then promoted to the third non-empty set, and so on. Any node added to a set by premature promotion could be stunted because it is not massive enough. Property 3.1 is satisfied since any set which receives a stunted node is immediately promoted, and can therefore receive no more. Step (2b) promotes S_i to S_{i+1}, which may cause S_{i+1} to be promoted to S_{i+2}, etc. So long as Property 4.1 is not satisfied the promotions continue. The remaining question is whether the new H_x nodes created are consistent with Properties 3.1 & 3.2.

Notice that Properties 3.1 & 3.2 only apply to cluster nodes, and that leaf nodes created in Step (1) clearly satisfy the mass requirements of Definition 3.1. All the new cluster nodes created in Step (2a) are stunted. However, they must satisfy the diameter requirements of Property 3.2, along with the cluster nodes created in Step (2b). Let z be a new rank i' node created in Step (2a) or (2b). By Property 4.2, M_z is connected and contains the vertex v. By Property 4.1 and the fact that $2\tau_i^j \leq \tau_{i+1}^j$, the distance from every vertex in M_z to v is bounded by $2\tau_{i'}^j$, hence the diameter of M_z is bounded by $4\tau_{i'}^j$, satisfying Property 3.2.

Before the parent of v in T_x, $p(v)$, is processed, we must restore Property 4.2, changing references of

v to $p(v)$. We achieve this by representing the MST path from v to $p(v)$ as one or more dummy nodes, then including those dummy nodes in the sets $\{S_i\}$.

Without loss of generality, assume the edge $(v, p(v))$ has length $< \tau_1^j$. If the edge is longer it can simply be broken into pieces. Property 4.2 is restored as follows.

(3) Create a rank zero dummy leaf z representing $(v, p(v))$ and let $S_0 := S_0 \cup \{z\}$.

(4) While the first non-empty set S_i is too massive to satisfy Property 4.1, promote S_i to S_{i+1}.

LEMMA 4.1. *Given a $\mathcal{CH}^1$ node x at level j and T_x, an H_x satisfying Properties 3.1 & 3.2 can be constructed in $O(mass(M_x)/\ell_{\mathbf{min}}2^j)$ time.*

Proof Sketch: Let v have children $\{v_1, v_2, \ldots, v_\nu\}$, where v_ν represents the child with the most massive subtree. In processing v it is straightforward to show that Steps (1)-(4) take amortized time proportional to $\sum_{k=1}^{\nu-1} \log \frac{|v_k|}{\ell_{\mathbf{min}} \cdot 2^{j-1}}$, where $|v_k|$ represents the mass of the subtree rooted at v_k. Summing over all vertices in T_x the total time can be bounded by $O(mass(M_x)/\ell_{\mathbf{min}} \cdot 2^j)$. □

We are now ready to prove Lemma 3.1 on the time taken to construct $\mathcal{CH}$.

Proof. (Lemma 3.1) First observe that the $\mathcal{CH}^1$ hierarchies of a graph and its MST are identical. We compute the MST optimally (in time proportional to the decision-tree complexity of MST) using the algorithm [PR00]. We then determine $\ell_{\mathbf{min}}$, the minimum edge length in the MST, and for each MST edge e determine the minimum j s.t. $\ell(e) < \ell_{\mathbf{min}} \cdot 2^j$ in $O(\log r + n \log\log r)$ time (see Section 2.1). Once this information is known, $\mathcal{CH}^1$ can be constructed using Union-Find in linear time on a pointer machine [B+98]. All T_x are constructed in linear time and the cost of constructing H_x from T_x is, by Lemma 4.1, $O(mass(M_x)/(\ell_{\mathbf{min}} \cdot 2^j))$. Let us examine the cost contributed by some MST edge e to building all H_x. Suppose $\ell_{\mathbf{min}} \cdot 2^{j-1} \le \ell(e) < \ell_{\mathbf{min}} \cdot 2^j$. Then e contributes $\ell(e)$ to $mass(M_x)$ for at most one x at each level $\ge j$. Thus the maximum cost (in time) attributed to e is bounded by $\sum_{k=j}^{\infty} O(\ell(e)/(\ell_{\mathbf{min}} \cdot 2^k)) = O(1)$. Hence the total cost of building all H_x is $O(n)$ and the total cost for building $\mathcal{CH}$ is $O(MST(m,n) + n \log\log r + \log r)$. By Lemma 2.1 the $MST(m,n)$ term will never be a bottleneck on an SSSP algorithm.

5 Our Algorithm

For each $\mathcal{CH}^1$ node x appearing at level j, we associate a sequence of at least $\lceil \frac{diam(M_x)}{\ell_{\mathbf{min}} 2^{j-1}} \rceil + 1$ buckets; these buckets together will represent an interval which contains $d(v)$, for all $v \in M_x$. Although we know the width of each bucket interval, $\ell_{\mathbf{min}} 2^{j-1}$, we do not know their endpoints in advance. The zeroth bucket interval begins at d_0, which is determined the moment x is first `Visited`. It follows that the i^{th} bucket interval begins at $d_0 + i \cdot \ell_{\mathbf{min}} \cdot 2^{j-1}$. Only nodes of H_x will appear in the buckets and at any given time the leaves of H_x will either appear in some bucket or be represented by a cluster in some bucket. We extend the D-value notation to $\mathcal{CH}$ nodes by letting $D(x)$ be the minimum D-value over all descendants of x. Logically an H_x node z lies in bucket i iff $D(z) \in [d_0 + i \cdot \ell_{\mathbf{min}} \cdot 2^{j-1},\ d_0 + (i+1) \cdot \ell_{\mathbf{min}} \cdot 2^{j-1})$; however a strict adherence to this rule becomes too expensive. In Section 6 we describe a "lazy" bucketing technique which simulates our algorithm's bucketing regimen in linear time.

`Visit` takes a $\mathcal{CH}$ node x, a safe interval I, and visits all $v \in M_x$ s.t. $d(v) \in I$. Initially we let $S := \emptyset$, $D(s) = 0$ and $D(v) = \infty$ for $v \neq s$, and call `Visit`(root($\mathcal{CH}$), $[0, \infty)$) on the root of $\mathcal{CH}$– see Figure 2. D-values are maintained to satisfy Invariant 0.

Lines marked with a letter involve bucketing, or finding and updating D-values. Line (a) causes several **decrease-key**s, one for each incident edge on x. Lines (b) and (d) correspond to a $\mathcal{CH}$ node being processed for the first time, causing the node's children to be bucketed according to their D-values. Line (e) is easily implemented without specialized data structures. Line (c) removes clusters from the first bucket, one by one, until only $\mathcal{CH}^1$ nodes remain. In the next section we analyze the cost of bucketing, which is shown to be linear overall. In Section 7 we describe the split-findmin structure and exactly how it is used in lines (b) and (d).

6 Lazy Bucketing

Consider this contrived specification for a data structure. It assumes a sequence of B buckets (in, say, a linked list), which are labeled in increasing order.

pop	:	Returns an item from the first undeleted bucket, or deletes it if empty.
insert(x, δ, Δ)	:	Insert x into bucket δ. It is guaranteed: eventually x will be rebucketed to one of the earliest Δ buckets.
rebucket(x, δ)	:	Move x to an earlier bucket δ.

LEMMA 6.1. *Consider the bucketing operations* **pop**, **insert**(x, δ, Δ) *and* **rebucket**(x, δ). *There is a comparison-based, pointer machine data structure which implements* **rebucket** *and* **pop** *in constant amortized time and* **insert** *in $O(\log \Delta)$ amortized time.*

A bucketing structure like this would typically be able to enumerate the contents of any bucket without undue work. The problem is that keeping items in their

```
Visit($x$, $[a,b)$ )
   (a)   If $x$ is a leaf, let $S := S \cup \{x\}$, update $D$-values, and return.  endif
         If Visit($x,\cdot$) is being called for the first time, then
               Determine $d_0 \in [a,b)$   (uses split-findmin structure, Section 7)
               $a := d_0$
   (b)         Initialize $x$'s bucket array; bucket $x$'s children.  endif
         Let $t := \ell_{\mathbf{min}} \cdot 2^{level(x)-1}$
         While $a < b$ and $V(M_x) \not\subseteq S$ do
   (c)         While Bucket $[a, a+t)$ contains a cluster, say $z$, do
                     Remove $z$, determine its children $\{z_i\}$
   (d)               Bucket each node in $\{z_i\}$.  end while
               For each $\mathcal{CH}^1$ node $y$ in Bucket $[a, a+t)$, do
                     Visit($y$, $[a, a+t)$ )
   (e)               If $V(M_y) \not\subseteq S$ rebucket $y$ in $[a+t, a+2t)$.  end for
               $a := a + t$    end while
         Return.
```

Figure 2: Visit traverses our component hierarchy $\mathcal{CH}$.

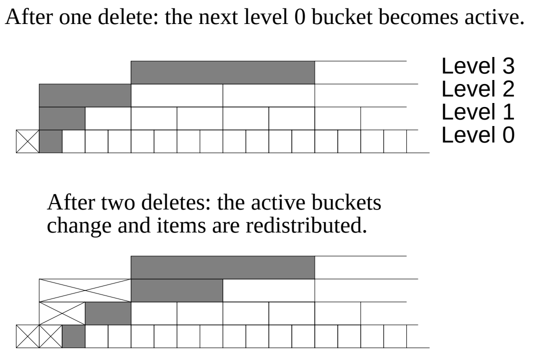

After one delete: the next level 0 bucket becomes active.
Level 3
Level 2
Level 1
Level 0
After two deletes: the active buckets change and items are redistributed.

Figure 3: Ahierarchical bucketing structure. Only active (grayed) buckets contain items.

correct bucket requires $\Theta(\log B)$ comparisons per insert or rebucket to locate the bucket labeled δ. On the other hand, if we do not inspect any buckets but demand that **pop**s perform correctly, **rebucket** and **pop** can be performed in $O(1)$ time and **insert**s in $O(\log\Delta)$ time (all amortized) as follows. Maintain levels of buckets, one *active* bucket per level, where an i^{th} level bucket presides over 2^i original buckets. An item which logically resides in bucket δ must either reside in an active bucket presiding over bucket δ, or in the level $\log\Delta$ active bucket (each item has an associated Δ). An **insert** puts the item in the lowest level eligible bucket; a **rebucket** operation moves the item to a lower level bucket if necessary. A **pop** operation which deletes the first bucket causes any higher level buckets presiding over it to be deleted — see Figure 3. Deleting any bucket causes its successor on the same level to become active and its contents to be redistributed to lower level buckets. Ahuja et al. [AM+90] used a similar bucketing scheme in their SSSP algorithm, however their data structure is tailored to the RAM model and is slower than our data structure for handling the bucketing tasks of Visit.

Using the lazy bucketing scheme described above, let us analyze the bucketing costs of our algorithm. Any node in $\mathcal{CH}$ is inserted just once, in line (b) or (d), and extracted from the first undeleted bucket in line (c). Let a node be called a level j rank i node if it has rank i and appears in H_x for some level j $\mathcal{CH}^1$ node x. Let $M_j = \{e : e \in M, \ell(e) < \ell_{\mathbf{min}} \cdot 2^j\}$. By Definition 3.1 and Property 3.1 the total number of children of level j rank $i+1$ nodes is $\leq 2 \cdot mass(M_j)/\tau_i^j$. Such a node x is inserted only when its parent $p(x)$ is popped from the first bucket. Because the input graph is undirected, the largest separation in $d(\,)$-values between two nodes in $M_{p(x)}$ is no more than $diam(M_{p(x)})$. This means that regardless of which bucket x is inserted into (the δ argument), x will eventually be moved to one of the first $\Delta = diam(M_{p(x)})/\ell_{\mathbf{min}} 2^{j-1} + 1$ buckets. Notice that $diam(M_{p(x)})$ is bounded by Property 3.2, so first inserting x requires $O(\log(4\,\tau_{i+1}^j/\ell_{\mathbf{min}} \cdot 2^{j-1}))$ time, which, by the definition of τ_{i+1}^j, is $O(\lambda_i/2^i) = O(\frac{\tau_i^j}{\ell_{\mathbf{min}} \cdot 2^{i+j-1}})$. Hence the initial insertion costs for all level j rank i nodes is $O(mass(M_j)/(\ell_{\mathbf{min}} \cdot 2^{i+j}))$. Summing over all i,j the total cost contributed by each MST edge is constant, hence the total cost of first time insertions is $O(n)$. The number of reinsertions at line (d) is $O(mass(M_j)/(\ell_{\mathbf{min}} \cdot 2^{j-1}))$ for level j nodes, and is also linear overall.

Using a normalized bucket width of $\ell_{\mathbf{min}}$, Dijkstra's SSSP algorithm can be implemented in $O(m + n\log r)$ time on a pointer machine, which is faster than using

Fibonacci heaps [FT87] if $\log r = o(\log n)$. Goldberg [G01] claimed a similar algorithm for the RAM model.

THEOREM 6.1. *The single source shortest path problem on directed graphs can be solved in $O(m + n \log r)$ time in the comparison-addition/pointer machine model.*

7 The Split-Findmin Data Structure

Given an initial sequence (or *segment*) of n elements, each with an associated key, Gabow [G85] gave an $O((n+m)\alpha(m,n))$ time implementation to execute m operations of the following data structure on a pointer machine.

split(x)	:	Split x's segment into two: the part up to x and the part after x.
decrease-key(x, κ)	:	Set $key(x) := \min\{key(x), \kappa\}$.
findmin(x)	:	Return the element with minimum key in x's segment.

Gabow's implementation is optimal for pointer machines by a result of La Poutre [L96]. We observe that Gabow's algorithm can be modified to run in the same time bound using asymptotically fewer comparisons. In this data structure a **decrease-key** operation must update a sequence of $\alpha(m,n)$ variables having non-increasing values. Once these variables are found, in $O(\alpha(m,n))$ time using no comparisons, they can be updated using a binary search with $O(\log \alpha(m,n))$ comparisons. This improvement is admittedly small, but it is an instance of a rare phenomenon: a separation between data structural complexity and decision-tree complexity.

LEMMA 7.1. *There is a pointer machine algorithm which executes m split-findmin operations in $O(n + m\alpha(m,n))$ time using $O(n + m \log \alpha(m,n))$ comparisons.*

In terms of our SSSP algorithm of Section 5, the sequence of n elements is a left-to-right listing of the leaves of $\mathcal{CH}$ (there are several orderings consistent with $\mathcal{CH}$). The segments of this sequence correspond to the descendants of certain internal nodes of $\mathcal{CH}$, in particular, those as yet unprocessed nodes which appear in some bucket. In lines (b) and (d) of Visit, when such a node z is first processed, **split**s are performed on z's segment so that the resulting segments correspond to z's children. Whenever a node's current D-value is needed for bucketing or rebucketing (lines (a), (b) or (d)) it may be retrieved with a single **findmin** operation.

8 Undirected All-Pairs Shortest Paths

We solve undirected APSP with n passes of our SSSP algorithm, each of which uses $O(m \log \alpha(m,n))$ comparisons and additions. If the time to construct $\mathcal{CH}$, $O(MST(m,n) + n \log\log r + \log r)$, is $O(mn \log \alpha(m,n))$ then we are done. Below we give an alternate method for constructing a component hierarchy which does not depend on r.

As before the component hierarchy is based on the lengths of minimum spanning tree edges. The difference is that instead of normalizing all edge lengths by $\ell_{\mathbf{min}}$, the minimum edge length, we may choose many MST edges to be "normalizing" edges. We sort the MST edges by length, identifying successive lengths which are separated by a large factor; this takes $O(n \log n)$ time. If such a pair of edges exists, we can break the shortest path problem into two *almost* independent shortest paths problems. Specifically, let $(\ell_1, \ell_2, \ldots, \ell_{n-1})$ be the MST edge lengths in increasing order and $\{\ell_j : \ell_{j-1} \cdot n < \ell_j\} \cup \{\ell_1\}$ be the set of normalizing edge lengths. We will refer to the k^{th} smallest normalizing edge length as ℓ_{i_k}. Our component hierarchy is composed of layered *strata*. The k^{th} stratum represents connected components of the graph derived by contracting edges with length $< \ell_{i_k}$ and removing edges with length $\geq \ell_{i_{k+1}}$. Each stratum, having say n' MST edges, may be further decomposed into a component hierarchy as described in Sections 3 and 4, where r is bounded by $n^{n'}$. Only small changes to the Visit procedure are needed. In the line 'Let $t := \ell_{\mathbf{min}} \cdot 2^{level(x)-1}$', where x is a stratum k node, $\ell_{\mathbf{min}}$ should be replaced by ℓ_{i_k} and $level(x)$ should be interpreted as the level of x within the stratum. The comparison-addition cost of finding the MST, sorting its edges, and finding all normalizing edges is $O(m + n \log n)$. So in general, the s-sources shortest paths problem is solved in $O(sm\alpha(m,n) + \min\{n \log n, n \log\log r\})$ time.

References

[AHU74] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesly, 1974.

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

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

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

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

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

[CGS97] B. V. Cherkassky, A. V. Goldberg, C. Silverstein. Buckets, heaps, lists, and monotone priority queues. In *Proc. SODA* 1997, 83–92.

[CLR90] T. Cormen, C. Leiserson, R. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

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

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

[EH97] D. Eppstein, D. Hart. An efficient algorithm for shortest paths in vertical and horizontal segments. In *Proc. WADS 1997*, LNCS 1272, p. 234.

[EH99] D. Eppstein, D. Hart. Shortest paths in an arrangement with k-line orientations. In *Proc. SODA 1999*, 310–316.

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

[Fre91] G. N. Frederickson. Planar graph decomposition and all pairs shortest paths. *J. Assoc. Comput. Mach.* 38 (1991), no. 1, 162–204.

[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 *Proc. FOCS 1985*, 90–99.

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

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

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

[G01] A. Goldberg. A simple shortest path algorithm with linear average time. InterTrust Technical Report STAR-TR-01-03, March 2001.

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

[G+80] R. L. Graham, A. C. Yao, F. F. Yao. Information bounds are weak in the shortest distance problem. *J. Assoc. Comput. Mach.* 27 (1980), no. 3, 428–444.

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

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

[HS99] J. Hershberger, S. Suri. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM J. on Comput.* 28, no. 6, 2215–2256.

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

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

[KS93] P. N. Klein, S. Sairam. A fully dynamic approximation scheme for all-pairs shortest paths in planar graphs. In *Proc. WADS 1993*, LNCS 709, 442–451.

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

[PR00] S. Pettie, V. Ramachandran. An optimal minimum spanning tree algorithm. In *Proc. ICALP 2000*, LNCS volume 1853, 49–60.

[PR01] S. Pettie, V. Ramachandran. Computing undirected shortest paths using comparisons and additions. Tech. report TR-01-12, Dept. of Comp. Sci., UT-Austin, 2001.

[PR01a] S. Pettie, V. Ramachandran. Experimental evaluation of a new shortest path algorithm. Tech. report TR-01-37, Dept. of Comp. Sci., UT-Austin, 2001.

[Ram96] R. Raman. Priority queues: small monotone, and trans-dichotomous. In *Proc. Euro. Symp. on Alg.* 1996, 121–137.

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

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

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

[SP73] P. M. Spira, A. Pan. On finding and updating shortest paths and spanning trees. In *Proc. Symp. Switching and Automata Theory* 1973, 82–84.

[SR94] J. A. Storer, J. H. Reif. Shortest paths in the plane with polygonal obstacles. *J. Assoc. Comput. Mach.* 41 (1994), no. 5, 982–1012.

[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.

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

[Tho96] M. Thorup. On RAM priority queues. In *Proc. SODA* 1996, 59–67.

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

[Tho01] M. Thorup. Compact oracles for reachability and approximate distances in planar digraphs. In *Proc. FOCS 2001.*

[Z98] U. Zwick. All pairs shortest paths in weighted directed graphs – exact and almost exact algorithms. *Proc. FOCS'98* (1998), 310-319.