On dynamic shortest paths problems

Liam Roditty * *Uri Zwick* *

Abstract

We obtain the following results related to dynamic versions of the shortest-paths problem:

(i) Reductions that show that, for weighted directed or undirected graphs, the incremental and decremental *single-source* shortest-paths problems are, in a strong sense, at least as hard as the static *all-pairs* shortest-paths problem. We also obtain slightly weaker results for the unweighted incremental or decremental single-source shortest-paths problem.

(ii) A randomized fully-dynamic algorithm for the all-pairs shortest-paths problem in directed unweighted graphs. The amortized update time of the algorithm is $\tilde{O}(m\sqrt{n})$ and the worst case query time is $O(n^{3/4})$.

(iii) A deterministic $O(n^2 \log n)$ time algorithm for constructing a $O(\log n)$-spanner with $O(n)$ edges for any weighted undirected graph on n vertices. The algorithm uses a simple algorithm for incrementally maintaining single-source shortest-paths tree up to a given distance.

1 Introduction

The objective of a dynamic shortest path algorithm is to efficiently process an online sequence of update and query operations. Each update operation inserts or deletes edges from an underlying dynamic graph. Each query operation asks for the distance between two specified vertices in the current graph. A dynamic algorithm is said to be *fully dynamic* if it can handle both insertions and deletions. An *incremental* algorithm is an algorithm that can handle insertions, but not deletions, and a *decremental* algorithm is an algorithm that can handle deletions, but not insertions. Incremental and decremental algorithms are sometimes referred to as being *partially dynamic*. An *all-pairs* shortest paths (APSP) algorithm is an algorithm that can report distances between any two vertices of the graph. A *single-source* shortest paths (SSSP) algorithm can only report distances from a given source vertex.

In this paper we present three results related to dynamic shortest paths problems. We first present simple reductions that show that the innocent looking incremental and decremental SSSP problems are, in a strong sense, at least as hard as the static APSP problem. This may explain the lack of progress on these seemingly simple problems, and shows why it will be difficult to improve classical algorithms for these problems, such as the decremental algorithm of Even and Shiloach [ES81].

We then present a new fully dynamic APSP algorithm for unweighted directed graphs. The amortized update time of the algorithm is $\tilde{O}(m\sqrt{n})$ and the worst-case query time is $O(n^{3/4})$. The algorithm is randomized. The results returned by the algorithm are correct with very high probability. The new algorithm should be compared with a recent algorithm of Demetrescu and Italiano [DI03] and its slight improvement by Thorup [Tho04]. Their algorithm, that works for *weighted* directed graphs, has an amortized update time of $\tilde{O}(n^2)$ and a query time of $O(1)$. For sparse enough graphs our new algorithm has a faster update time. The query cost, alas, is much larger.

*School of Computer Science, Tel-Aviv University, Tel-Aviv, 69978, Israel. E-Mail: {liamr,zwick}@cs.tau.ac.il

The new algorithm can also be compared to fully dynamic *reachability* algorithms for directed graphs obtained by the authors in [RZ02] and [RZ04]. A reachability algorithm is only required to determine, given two vertices u and v, whether there is a directed path from u to v in the graph. The reachability problem, also referred to as the *transitive closure* problem, is, of course, easier than the APSP problem. A fully dynamic reachability algorithm with an amortized update time of $\tilde{O}(m\sqrt{n})$ and a worst-case query time of $O(\sqrt{n})$ is presented in [RZ02]. A fully dynamic reachability algorithm with an amortized update time of $O(m + n \log n)$ and a worst-case query time of $O(n)$ is presented in [RZ04].

Finally, we present a simple application of incremental SSSP algorithms, showing that they can be used to speed up the operation of the greedy algorithm for constructing spanners. In particular, we obtain an $O(n^2 \log n)$ time algorithm for constructing an $O(\log n)$-spanner with $O(n)$ edges for any weighted undirected graph on n vertices. The previously fastest algorithm for constructing such spanners runs in $O(mn)$ time.

The rest of this paper is organized as follows. In the next section we describe the hardness results for incremental and decremental SSSP. We also discuss the implications of these results. In Section 3 we then present our new fully dynamic APSP algorithm. In Section 4 we present our improved spanner construction algorithm. We end in Section 5 with some concluding remarks and open problems.

2 Hardness of incremental or decremental single-source shortest-paths

We start with two simple reductions that show that the incremental and decremental *weighted* SSSP problems are at least as hard as the static weighted APSP problem. We then present two similar reductions that show that the incremental and decremental *unweighted* SSSP problems are at least as hard as several natural static graph problems such as Boolean matrix multiplication and the problem of finding all edges of a graph that are contained in triangles.

Let $\mathcal{A}$ be an incremental (decremental) algorithm for the weighted (unweighted) directed (undirected) SSSP problem. We let $init_{\mathcal{A}}(m, n)$ be the time needed to initialize $\mathcal{A}$ on a graph with m edges and n vertices. We let $update_{\mathcal{A}}(m, n)$ be the amortized edge insertion (deletion) time of $\mathcal{A}$, and $query_{\mathcal{A}}(m, n)$ be the amortized query time of $\mathcal{A}$, where m and n are the number of edges and vertices in the graph at the time of the operation. We assume that the functions $init_{\mathcal{A}}(m, n)$, $update_{\mathcal{A}}(m, n)$ and $query_{\mathcal{A}}(m, n)$ are monotone in m and n.

Theorem 2.1 *Let $\mathcal{A}$ be an incremental (decremental) algorithm for the* weighted *directed (undirected) SSSP problem. Then, there is an algorithm for the static APSP problem for weighted graphs that runs in* $O(\mathrm{init}_{\mathcal{A}}(m+n, n+1) + n\cdot\mathrm{update}_{\mathcal{A}}(m+n, n+1) + n^2\cdot\mathrm{query}_{\mathcal{A}}(m+n, n+1))$ *time.*

Proof: Let $G = (V, E)$ be a graph, with $|V| = n$ and $|E| = m$, and let $w : E \to \mathbb{R}^+$ be an assignment of non-negative edge weights to its edges. The proof works for both directed and undirected graphs. We assume, without loss of generality, that $V = \{1, 2, \ldots, n\}$. Let $W = \max_{e \in E} w(e)$ be the maximum edge weight in the graph.

Assume, at first, that $\mathcal{A}$ is a decremental algorithm. We construct a new graph $G_0 = (V \cup \{0\}, E \cup \{0\} \times V)$, where 0 is a new source vertex. A new edge $(0, j)$, where $1 \le j \le n$ is assigned the weight $j \cdot nW$. (See Figure 1(a).) The graph G_0, composed of $n+1$ vertices and $m+n$ edges, is passed as the initial graph to the decremental algorithm $\mathcal{A}$. The source is naturally set to be 0. After $\mathcal{A}$ is initialized, we perform the n queries $query(j)$, for $1 \le j \le n$. Each query $query(j)$ returns $\delta_{G_0}(0, j)$, the distance from 0 to j in G_0. As the weight of the edge $(0, 1)$ is substantially smaller than the weight of all other edges emanating from the source, it is easy to see that $\delta_G(1, j) = \delta_{G_0}(0, j) - nW$, for every $1 \le j \le n$. We now delete the edge $(0, 1)$ from G_0 and perform again the n queries $query(j)$, for $1 \le j \le n$. We now have $\delta_G(2, j) = \delta_{G_0}(0, j) - 2nW$,

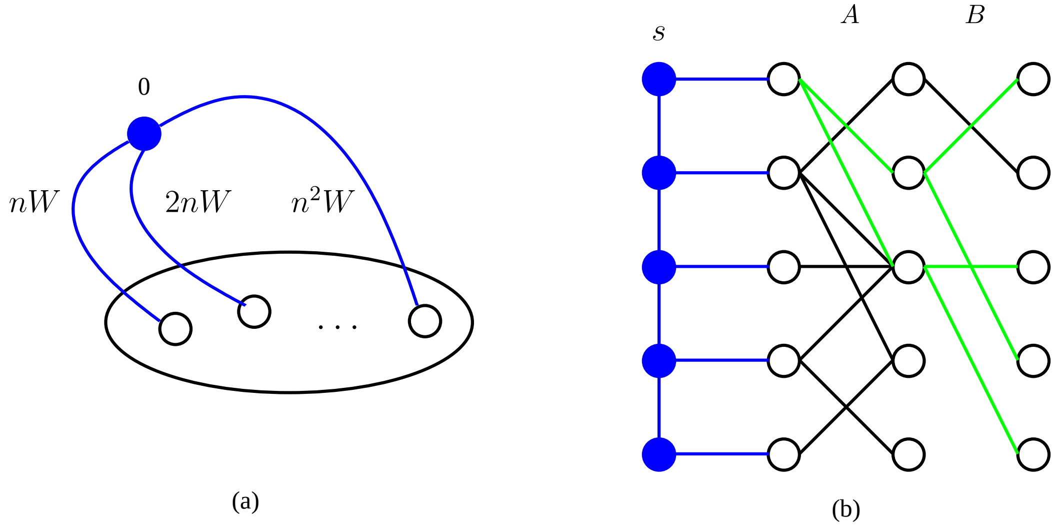

Figure 1: Reductions of static problems to incremental or decremental SSSP problems

for every $1 \le j \le n$. Repeating this process $n-2$ more times we obtain all distances in the original graph by performing only n edge deletions and n^2 queries.

The proof when $\mathcal{A}$ is an incremental algorithm is analogous. The only difference is that we now insert the edges $(0, j)$ one by one, in reverse order. We first insert the edge $(0, n)$, with weight n^2W, then the edge $(0, n-1)$ with weight $(n-1)nW$, and so on. □

We note that the simple reduction just described works for undirected, directed, as well as acyclic directed graphs (DAGs). We next move to unweighted versions of the problem.

Theorem 2.2 *Let $\mathcal{A}$ be an incremental (decremental) algorithm for the* unweighted *directed (undirected) SSSP problem. Then, there is an algorithm that multiplies two Boolean $n \times n$ matrices, with a total number of m 1's, in $O(\mathrm{init}_{\mathcal{A}}(m+2n, 4n) + n \cdot \mathrm{update}_{\mathcal{A}}(m+2n, 4n) + n^2 \cdot \mathrm{query}_{\mathcal{A}}(m+2n, 4n))$ time.*

Proof: Let A and B be two Boolean $n \times n$ matrices. Let $C = AB$ be their Boolean product. Construct a graph $G = (V, E)$ as follows: $V = \{s_i, u_i, v_i, w_i \mid 1 \le i \le n\}$, and $E = \{(s_i, s_{i+1}) \mid 1 \le i < n\} \cup \{(s_i, u_i) \mid 1 \le i \le n\} \cup \{(u_i, v_j) \mid a_{ij} = 1, 1 \le i, j \le n\} \cup \{(v_i, w_j) \mid b_{ij} = 1, 1 \le i, j \le n\}$. (See Figure 1(b).) The graph G is composed of $4n$ vertices and $m + 2n - 1$ edges. Let $s = s_1$. It is easy to see that $\delta_G(s, w_j) = 3$ if and only if $c_{1j} = 1$. We now delete the edge (s_1, u_1). Now, $\delta_G(s, w_j) = 4$ if and only if $c_{2j} = 1$. We then delete the edge (s_2, u_2), and so on. Again we use only n delete operations and n^2 queries. The incremental case is handled in a similar manner. □

Discussion

The best known algorithms for the static APSP problems in weighted directed or undirected graphs run in $\Omega(mn)$ time. A running time of $O(mn + n^2 \log n)$ is obtained by running Dijkstra's algorithm from each vertex. (See [FT87] and Chapter 24 of [CLRS01]). Slightly faster algorithms are available, in various settings. For the best available results see [FT87], [Tho99], [Hag00], [PR02], [Pet02]. Karger *et al.* [KKP93] show that any *path-comparison* algorithm for the problem must have a running time of $\Omega(mn)$. For completeness we note that a non path-comparison algorithm with a complexity of $O(n^3(\log\log n/\log n)^{1/2})$ was obtained by Takaoka [Tak98], using a technique of Fredman [Fre76]. Obtaining a static APSP algorithm for weighted graph whose running time is $O((mn)^{1-\epsilon} + n^2)$, or $O(n^{3-\epsilon})$, for some $\epsilon > 0$, is a major open problem.

The reduction of Theorem 2.1 shows that if there is an incremental or decremental SSSP algorithm that can handle n update operations and n^2 query operations in $o(mn)$ time, then there is also an $o(mn)$ time algorithm for the static APSP problem. We note that the trivial 'dynamic' SSSP algorithm that simply constructs a shortest paths tree from scratch after each update operation handles n update operations in $\tilde{O}(mn)$ time. Almost any improvement of this trivial algorithm, even with much increased query times, will yield improved results for the static APSP problem.

An interesting open problem is whether it is possible to construct incremental or decremental SSSP algorithm for *weighted* graphs that can handle m updates and n^2 queries in $\tilde{O}(mn)$ time.

We next consider *unweighted* versions of the SSSP problem. A classical result in this area is the following:

Theorem 2.3 ([ES81],[HK95]) *There is a decremental algorithm for maintaining the first k levels of a single-source shortest-paths tree, in a directed or undirected unweighted graph, whose total running time, over all deletions, is $O(km)$, where m is the initial number of edges in the graph. Each query can be answered in $O(1)$ time.*

It is easy to obtain an incremental variant of this algorithm. Such a variant is described, for completeness, in Section 4, where it is also used.

How efficient is the algorithm of Even and Shiloach [ES81], and what are the prospects of improving it? If k, the number of levels required is small, then the running time of the algorithm is close to be optimal, as $\Omega(m)$ is an obvious lower bound. But, if a complete shortest paths tree is to be maintained, i.e., $k = n-1$, the running time of the algorithm becomes $O(mn)$. How hard will it be to improve this result?

Our reductions for the unweighted problems are slightly weaker than the ones we have for the weighted problems. We cannot reduce the static APSP problems to the partially dynamic SSSP problems, but we can still reduce the Boolean matrix multiplication problem to them. The APSP problem for undirected unweighted graphs can be reduced to the Boolean matrix multiplication problem (see [GM97],[Sei95],[SZ99]), but this reduction does not preserve sparsity.

The fastest known combinatorial algorithm for computing the Boolean product of two $n \times n$ matrices that contain a total of m 1's runs in $O(mn)$ time. By a combinatorial algorithm here we refer to an algorithm that does not rely on fast algebraic matrix multiplication techniques. Using such algebraic techniques it is possible to multiply the matrices in $O(n^{2.38})$ time (see [CW90]), and also in $O(m^{0.7}n^{1.2} + n^2)$ (see [YZ04]). Obtaining a combinatorial Boolean matrix multiplication algorithm whose running time is $O((mn)^{1-\epsilon}+n^2)$, or $O(n^{3-\epsilon})$, for some $\epsilon > 0$, is a major open problem.

The reduction of Theorem 2.2 shows that reducing the total running time of the algorithm of [ES81] to $o(mn)$, using only combinatorial means, is at least as hard as obtaining an improved combinatorial Boolean matrix multiplication algorithm. Also, via the reduction of the static APSP problem to Boolean matrix multiplication, we get that an incremental or decremental SSSP algorithm with a total running time of $O(n^{3-\epsilon})$, and a query time of $O(n^{1-\epsilon})$, for some $\epsilon > 0$, will yield a combinatorial algorithm for the static APSP problem with a running time of $O(n^{3-\epsilon})$. We believe that this provides strong evidence that improving the algorithm of [ES81] will be very hard.

Chan [Cha02] describes a simple reduction from the *rectangular* Boolean matrix multiplication problem to the *fully dynamic subgraph connectivity* problem. It is similar in spirit to our reduction. The details, and the problems involved, are different, however.

As a final remark we note that we have reduced the APSP problem and the Boolean matrix multiplication problem to *offline* versions of incremental or decremental SSSP problem. It will thus be difficult to obtain improved algorithms for partially dynamic SSSP problems even if all the update and query operations are given in advance.

3 Fully dynamic all-pairs shortest paths

In this section we obtain a new fully dynamic algorithm for the all-pairs shortest paths problem. The algorithm relies on ideas of [HK95] and [RZ02]. We rely on following result of [HK95] and a simple observation of [UY91]:

Theorem 3.1 ([HK95]) *There is a randomized* decremental *all-pairs shortest-paths algorithm for directed unweighted graphs whose total running time, over all deletions, is $O(\frac{mn^2 \log n}{t} + mn \log^2 n)$ and whose query time is $O(t)$, where m and n are the number of edges and vertices in the initial graph, and $t \geq 1$ is a parameter. (In particular, for $t \leq n/\log n$, the total running time is $O(\frac{mn^2 \log n}{t})$.) Every result returned by the algorithm is correct with a probability of at least $1 - n^{-c}$, where c is a preset parameter.*

Lemma 3.2 ([UY91]) *Let $G = (V, E)$ be a directed graph on n vertices. Let $1 \leq k \leq n$, and let S be a random subset of V obtained by selecting each vertex, independently, with probability $p = (c \ln n)/k$. (If $p \geq 1$, we let S be V.) If p is a path in G of length at least k, then with a probability of $1 - n^{-c}$, at least one of the vertices on p belongs to S.*

The new algorithm works in *phases* as follows. In the beginning of each phase, the current graph $G = (V, E)$ is passed to the decremental algorithm of [HK95] (Theorem 3.1). A random subset S of the vertices, of size $(cn \ln n)/k$, is chosen, where k is a parameter to be chosen later. The standard BFS algorithm is then used to build shortest paths trees to and from all the vertices of S. If $w \in V$, we let $T_{in}(w)$ be a tree of shortest paths *to* w, and $T_{out}(w)$ be a tree of shortest paths *from* w. The set C is initially empty.

An insertion of a set E' of edges, all touching a vertex $v \in V$, said to be the *center* of the insertion, is handled as follows. First if $|C| \geq t$, where t is a second parameter to be chosen later, then the current phase is declared over, and all the data structures are reinitialized. Next, the center v is added to the set C, and the first k levels of shortest paths trees $\hat{T}_{in}(v)$ and $\hat{T}_{out}(v)$, containing shortest paths to and from v, are constructed. The trees $\hat{T}_{in}(v)$ and $\hat{T}_{out}(v)$ are constructed and maintained using the algorithm of [ES81] (Theorem 2.3). Finally, shortest paths trees $T_{in}(w)$ and $T_{out}(w)$, for every $w \in S$, are constructed from scratch. (Note that we use $\hat{T}_{in}(v)$ and $\hat{T}_{out}(v)$ to denote the trees associated with a vertex $v \in C$, and $T_{in}(w)$ and $T_{out}(w)$, without the hats, to denote the trees of a vertex $w \in S$. The former are decrementally maintained, up to depth k, while the later are rebuilt from scratch following each update operation.)

A deletion of an arbitrary set E' of edges is handled as follows. First, the edges of E' are removed from the decremental data structure initialized at the beginning of the current phase, using the algorithm of [HK95] (Theorem 3.1). Next, the algorithm of [ES81] (Theorem 2.3) is used to update the shortest paths trees $\hat{T}_{in}(v)$ and $\hat{T}_{out}(v)$, for every $v \in C$. Finally, the trees $T_{in}(w)$ and $T_{out}(w)$, for every $w \in S$, are again rebuilt from scratch.

A distance query *Query*(u, v), asking for the distance $d(u, v)$ from u to v in the current version of the graph, is handled using the following three stage process. First, we query the decremental data structure, that keeps track of all delete operations performed in the current phase, but ignores all insert operations, and get an answer ℓ_1. We clearly have $d(u, v) \leq \ell_1$, as all edges in the decrementally maintained graph are also edges of the current graph. Furthermore, if there is a shortest path from u to v in the current graph that does not use any edge that was inserted during the current phase, then $d(u, v) = \ell_1$.

Next, we try to find a shortest path from u to v that passes through one of the insertion centers contained in C. For every $w \in C$, we query $T_{in}(w)$ for the distance from u to w and $T_{out}(w)$ for the distance from w to v, and add these two numbers. (If $d(u, w) > k$, then u is not contained in $T_{in}(w)$ and the distance from w to u, in the present context, is taken to be ∞. The case $d(w, v) > k$ is handled similarly.) By taking the minimum of all these numbers we get a second answer that we denote by ℓ_2. Again, we have $d(u, v) \leq \ell_2$.

***Init*(G, k, t)**:

1. *Init-Dec*(G, t)
2. $C \leftarrow \phi$
3. $S \leftarrow$ *Random*$(V, (cn \ln n)/k)$
4. *Build-Trees*(S)

***Delete*(E')**:

1. $E \leftarrow E - E'$
2. *Delete-Dec*(E')
3. for every $v \in C$
4.   *Delete-Tree*$(\hat{T}_{in}(v), E', k)$
5.   *Delete-Tree*$(\hat{T}_{out}(v), E', k)$
6. *Build-Trees*(S)

***Insert*(E', v)**:

1. $E \leftarrow E \cup E'$
2. if $|C| \geq t$ then *Init*(G, k, t)
3. $C \leftarrow C \cup \{v\}$
4. *Init-Tree*$(\hat{T}_{in}(v), E, k)$
5. *Init-Tree*$(\hat{T}_{out}(v), E, k)$
6. *Build-Trees*(S)

***Build-Trees*(S)**:

1. for every $w \in S$
2.   *BFS*$(T_{in}(w), E)$
3.   *BFS*$(T_{out}(w), E)$

***Query*(u, v)**:

1. $\ell_1 \leftarrow$ *Query-Dec*(u, v)
2. $\ell_2 \leftarrow \min_{w \in C}$ *Query-Tree*$(\hat{T}_{in}(w), u)$ + *Query-Tree*$(\hat{T}_{out}(w), v)$
3. $\ell_3 \leftarrow \min_{w \in S}$ *Query-Tree*$(T_{in}(w), u)$ + *Query-Tree*$(T_{out}(w), v)$
4. return $\min\{\ell_1, \ell_2, \ell_3\}$

Figure 2: The new fully dynamic all-pairs shortest paths algorithm.

Furthermore, if $d(u, v) \leq k$, and there is a shortest path from u to v in the current graph that passes through a vertex that was an insertion center in the current phase of the algorithm, then $d(u, v) = \ell_2$.

Finally, we look for a shortest path from u to v that passes through a vertex of S. This is done in a similar manner by examining the trees associated with the vertices of S. The answer obtained using this process is denoted by ℓ_3. (If there is no path from u to v that passes through a vertex of S, then $\ell_3 = \infty$.) The final answer returned by the algorithm is $\min\{\ell_1, \ell_2, \ell_3\}$.

A formal description of the new algorithm is given in Figure 2. The algorithm is initialized by a call *Init*(G, k, t), where $G = (V, E)$ is the initial graph and k and t are parameters to be chosen later. Such a call is also made at the beginning of each phase. A set E' of edges, centered at v, is added to the graph by a call *Insert*(E', v). A set E' of edges is deleted by a call *Delete*(E'). A query is answered by calling *Query*(u, v). A call *Build-Trees*(S) is used to (re)build shortest paths trees to and from the vertices of S.

The call *Init-Dec*(G, t), in line 1 of *Init*, initializes the decremental algorithm of [HK95]. The call *Random*$(V, (cn \ln n)/k)$, in line 3, chooses the random sample S. The call *Build-Trees*(S), in line 4, construct the shortest paths trees $T_{in}(w)$ and $T_{out}(w)$, for every $w \in S$. A call *Init-Tree*$(\hat{T}_{in}(v), E, k)$ (line 4 of *Insert*) is used to initialize the decremental maintenance of the first k levels of a shortest paths tree $T_{in}(v)$ to v. Such a tree is updated, following a deletion of a set E' of edges, using a call *Delete-Tree*$(\hat{T}_{in}(v), E')$ (line 4 of *Delete*). A query *Query-Tree*$(\hat{T}_{in}(w), u)$ (line 2 of *Query*) is used to find the distance from u to w in the tree $\hat{T}_{in}(w)$. If u is not in $\hat{T}_{in}(w)$, the value returned is ∞. Such a tree-distance query is easily handled in $O(1)$ time. The out-trees $T_{out}(v)$ are handled similarly. Finally a call *BFS*$(T_{in}(w), E)$ (line 2 of *Build-Trees*) is used to construct a standard, static, shortest paths tree to w. Distances in such trees are again found by calling *Query-Tree*$(T_{in}(w), u)$ (line 3 of *Query*).

Theorem 3.3 *The fully dynamic all-pairs shortest paths algorithm of Figure 2 handles each insert or delete operation in* $O(\frac{mn^2 \log n}{t^2} + km + \frac{mn \log n}{k})$ *amortized time, and answers each distance query in* $O(t + \frac{n \log n}{k})$ *worst-case time. Each result returned by the algorithm is correct with a probability of at least* $1 - 2n^{-c}$*. By choosing* $k = (n \log n)^{1/2}$ *and* $(n \log n)^{1/2} \le t \le n^{3/4} (\log n)^{1/4}$ *we get an amortized update time of* $O(\frac{mn^2 \log n}{t^2})$ *and a worst-case query time of* $O(t)$*.*

Proof: The correctness proof follows from the arguments outlined along side the description of the algorithm. As each estimate ℓ_1, ℓ_2 and ℓ_3 obtained while answering a distance query $Query(u,v)$ is equal to the length of a path in the graph from u to v, we have $d(u,v) \le \ell_1, \ell_2, \ell_3$. We show that at least one of these estimates is equal, with very high probability, to $d(u,v)$.

If there is a shortest path from u to v that does not use any edge inserted in the current phase, then $d(u,v) = \ell_1$, assuming that the estimate ℓ_1 returned by the decremental data structure is correct. The probability of error here is only n^{-c}.

Suppose therefore that there is a shortest path p from u to v that uses at least one edge that was inserted during the current phase. Let w be the *latest* vertex on p to serve as an insertion center. If $d(u,v) \le k$, then the correct distance from u to v will be found while examining the trees $\hat{T}_{in}(w)$ and $\hat{T}_{out}(w)$.

Finally, suppose that $d(u,v) \ge k$. Let p be a shortest path from u to v in the current graph. By Lemma 3.2, with a probability of at least $1 - n^{-c}$ the path p passes through a vertex of w of S, and the correct distance will be found while examining the trees $T_{in}(w)$ and $T_{out}(w)$.

We next analyze the complexity of the algorithm. By Theorem 3.1, the total cost of maintaining the decremental data structure is $O(\frac{mn \log^2 n}{t})$. As each phase is composed of at least t update operations, this contributes $O(\frac{mn \log^2 n}{t^2})$ to the amortized cost of each update operation. Each insert operation triggers the creation (or recreation) of two decremental shortest paths trees that are maintained only up to depth k. By Theorem 2.3 the total cost of maintaining these trees is only $O(km)$. (Note that this also covers the cost of all future operations performed on these trees.) Finally, each insert or delete operations requires the rebuilding of $(cn \ln n)/k$ shortest paths trees at a total cost of $O(\frac{mn \log n}{k})$. The total amortized cost of each update operation is therefore $O(\frac{mn^2 \log n}{t^2} + km + \frac{mn \log n}{k})$, as claimed. Each query is handled by the algorithm in $O(t + \frac{n \log n}{k})$: The estimate ℓ_1 is obtained in $O(t)$ time by querying the decremental data structure. The estimate ℓ_2 is obtained in $O(t)$ by considering all the trees associated with C. Finally the estimate ℓ_3 is obtained in $O(\frac{n \log n}{k})$ time by examining all the trees associated with S.

By examining these bounds it is obvious that $k = (n \log n)^{1/2}$ is the optimal choice for k. By choosing t in the range $(n \log n)^{1/2} \le t \le n^{3/4} (\log n)^{1/4}$, we get a tradeoff between the update and query times. The fastest update time of $O(m(n \log n)^{1/2})$ is obtained by choosing $t = n^{3/4} (\log n)^{1/4}$. □

4 A simple incremental SSSP algorithm and greedy spanners

A simple algorithm for incrementally maintaining a single-source shortest-paths tree from a source vertex s up to distance k is given in Figure 3. The edge weights are assumed to be non-negative integers. The algorithm may be seen as an incremental variant of the algorithm of [ES81]. It is also similar to an algorithm of Ramalingam and Reps [RR96]. The algorithm is brought here for completeness.

For each vertex $v \in V$, $d[v]$ is the current distance from s to v, $p[v]$ is the parent of v in the shortest paths tree, and $N[v]$ are the vertices that can be reached from v by following an outgoing edge. The integer weight of an edge (u,v) is denoted by $wt(u,v)$. As described, the algorithm works on directed graphs. It is easy to adapt it to work on undirected graphs. (We simply need to scan each edge in both directions.)

Init:

1. for every $v \in V$,
2. $\quad d[v] \leftarrow \infty$; $p[v] \leftarrow \textit{null}$; $N[v] \leftarrow \phi$
3. $d[s] \leftarrow 0$

***Insert*(u, v)**:

1. $N[u] \leftarrow N[u] \cup \{v\}$
2. $\textit{Scan}(u, v)$

***Scan*(u, v)**:

1. $d' \leftarrow d[u] + wt(u, v)$
2. if $d' < d[v]$ and $d' \leq k$ then
3. $\quad d[v] \leftarrow d'$; $p[v] \leftarrow u$
4. $\quad$ for every $w \in N[v]$,
5. $\quad\quad \textit{Scan}(v, w)$

Figure 3: A simple incremental SSSP algorithm.

***Greedy-Spanner*(G, k)**:

1. $E' \leftarrow \emptyset$
2. **for each** edge $(u, v) \in E$, in non-decreasing order of weight, **do**
3. $\quad$ **if** $\delta_{E'}(u, v) > (2k - 1) \cdot wt(u, v)$ **then**

3'. $\quad$ [**if** $d_{E'}(u, v) > (2k - 1)$ **then**]

4. $\quad\quad E' \leftarrow E' \cup \{(u, v)\}$
5. return $G' \leftarrow (V, E')$

Figure 4: A greedy algorithm for constructing spanners.

Theorem 4.1 *The algorithm of Figure 3 incrementally maintains a shortest-paths tree from a source vertex s up to distance k in a directed unweighted graph using a total number of $O(km)$ operations, where m is the number of edges in the final graph. Each distance query is answered in $O(1)$ time.*

Proof: (Sketch) It is easy to see that the algorithm correctly maintains the distances of the vertices from s. The total complexity is $O(km)$ as each edge (u, v) is rescanned only when the distance of u from s decreases, and this can happen at most k times. $\square$

We next define the notion of *spanners*.

Definition 4.2 (Spanners [PS89]) *Let $G = (V, E)$ be a weighted undirected graph, and let $t \geq 1$. A subgraph $G' = (V, E')$ is said to be a t-spanner of G if and only if for every $u, v \in V$ we have $\delta_{G'}(u, v) \leq t \cdot \delta_G(u, v)$.*

The greedy algorithm of Althöfer et al. [ADD+93] for constructing sparse spanners of weighted undirected graphs is given in Figure 4. For every integer $k \geq 2$, it constructs a $(2k - 1)$-spanner with at most $n^{1+1/k}$

edges. This is an essentially optimal tradeoff between stretch and size. The algorithm of Althöfer *et al.* is reminiscent of Kruskal's algorithm for the construction of a minimum spanning tree algorithm. A naive implementation of this algorithm requires $O(mn^{1+1/k})$ time.

We consider a variant of the algorithm in which line 3 is replaced by line 3'. For every edge $(u, v) \in E$, the original algorithm checks whether $\delta_{E'}(u, v) > (2k - 1)wt(u, v)$, i.e., whether the *weighted* distance from u to v in the subgraph composed of the edges already selected to the spanner is at most $2k - 1$ times the weight $wt(u, v)$ of the edge. The modified version of the algorithm asks, instead, whether $d_{E'}(u, v) > 2k-1$, i.e., whether the *unweighted* distance between u and v in the subgraph (V, E') is greater than $2k - 1$. We now claim:

Theorem 4.3 *The modified version of the greedy spanner algorithm still produces a $(2k-1)$-spanner with at most $n^{1+1/k}$ edges for any weighted graph on n vertices.*

Proof: The claim follows from a simple modification of the correctness proof of the greedy algorithm. If an edge (u, v) is not selected by the modified algorithm, then $d_{E'}(u, v) \leq 2k - 1$. As the edges are scanned in an increasing order of weight, all the edges on the shortest path connecting u and v in (V, E') are of weight at most $wt(u, v)$, and therefore $\delta_{E'}(u, v) \leq (2k-1)\cdot wt(u, v)$. Thus, the edge (u, v) is also not selected by the original algorithm. The edge set returned by the modified algorithm is therefore a superset of the edge set returned by the original algorithm, and is therefore a $(2k - 1)$-spanner of G.

The proof that the set of edges E' returned by the original algorithm is of size at most $n^{1+1/k}$ relies only on the fact that the *girth* of $G' = (V, E')$ is at least $2k + 1$. This also holds for the set E' constructed by the modified algorithm, as we never add to E' an edge that would form a cycle of size at most $2k$. Hence, the size of the set E' returned by the modified algorithm is also at most $n^{1+1/k}$. □

Theorem 4.4 *The modified greedy algorithm can be implemented to run in $O(kn^{2+1/k})$.*

Proof: We use the algorithm of Figure 3 to maintain a tree of shortest-paths, up to distance $2k-1$, from each vertex of the graph. As the spanner contains at most $n^{1+1/k}$ edges, the total cost of maintaining each one of these trees is only $O(kn^{1+1/k})$, and the total cost of the algorithm is $O(kn^{2+1/k})$, as claimed. □

Discussion

There are several other algorithms for constructing sparse spanners of weighted graphs. In particular, a randomized algorithm of Baswana and Sen [BS03] constructs a $(2k - 1)$-spanner with $O(kn^{1+1/k})$ edges in $O(m)$ expected time. A randomized $O(mn^{1/k})$ algorithm for constructing such spanners is described in [TZ01]. Why then insist on a faster implementation of the greedy algorithm? The answer is that the greedy algorithm constructs slightly sparser spanners. It produces $(2k - 1)$-spanners with at most $n^{1+1/k}$ edges (no big-O is needed here). When k is non-constant, this is significant. When we let $k = \log n$, the greedy algorithm produces an $O(\log n)$-spanner containing only $O(n)$ edges. All other algorithms produce spanners with $\Omega(n \log n)$ edges. It is, of course, an interesting open problem whether such spanners can be constructed even faster.

Another interesting property of the (original) greedy algorithm, shown by [CDNS95], is that the total weight of the edges in the $(2k-1)$-spanner that it constructs is at most $O(n^{(1+\epsilon)/k}\cdot wt(MST(G)))$, for any $\epsilon > 0$, where $wt(MST(G))$ is the weight of the minimum spanning tree of G. Unfortunately, this property no longer holds for the modified greedy algorithm. Again, it is an interesting open problem to obtain an efficient spanner construction algorithm that does have this property.

An efficient implementation of a different variant of the greedy algorithm, in the setting of geometric graphs, is described in [GLN02].

5 Concluding remarks and open problems

We presented a simple reduction from the static APSP problem for weighted graphs to offline partially dynamic SSSP problem for weighted graphs, and a simple reduction from the Boolean matrix multiplication problem to the offline partially dynamic SSSP problem for unweighted graphs.

An interesting issue to explore is whether faster partially dynamic SSSP algorithms may be obtained if *approximate* answers are allowed. (For a step in this direction, but for the decremental approximate APSP problem, see [BHS02, BHS03].)

We presented a new fully dynamic algorithm for the APSP problem for unweighted directed graphs. The amortized update time of the algorithm is $\tilde{O}(m\sqrt{n})$ and the worst-case query time is $O(n^{3/4})$. More generally, for any $n^{1/2} \le t \le n^{3/4}$ we get an amortized update time of $\tilde{O}(m(n/t)^2)$ and a query time of $\tilde{O}(t)$. Obtaining other tradeoffs between the update and query times, especially ones with smaller query times, is an interesting open problem.

Finally, we presented a simple application of incremental APSP algorithms, obtaining an improved implementation of a slight modification of the spanner construction algorithm of [ADD+93]. An interesting open problem here is whether there is a (deterministic) $O(m)$ algorithm that can construct for any $k \ge 1$, not necessarily a constant, $(2k-1)$-spanners with $O(n^{1+1/k})$ edges.

References

[ADD+93] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares. On sparse spanners of weighted graphs. *Discrete & Computational Geometry*, 9:81–100, 1993.

[BHS02] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for transitive closure and all-pairs shortest paths. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing, Montréal, Canada*, pages 117–123, 2002.

[BHS03] S. Baswana, R. Hariharan, and S. Sen. Maintaining all-pairs approximate shortest paths under deletion of edges. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms, Baltimore, Maryland*, pages 394–403, 2003.

[BS03] S. Baswana and S. Sen. A simple linear time algorithm for computing $(2k-1)$-spanner of $O(n^{1+1/k})$ size for weighted graphs. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming, Eindhoven, The Netherlands*, pages 384–296, 2003.

[CDNS95] B. Chandra, G. Das, G. Narasimhan, and J. Soares. New sparseness results on graph spanners. *International Journal of Computational Geometry and Applications*, 5:125–144, 1995.

[Cha02] T. Chan. Dynamic subgraph connectivity with geometric applications. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing, Montréal, Canada*, pages 7–13, 2002.

[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, second edition, 2001.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[DI03] C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, San Diego, California*, 2003.

[ES81] S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, 1981.

[Fre76] M.L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5:49–60, 1976.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[GLN02] J. Gudmundsson, C. Levcopoulos, and G. Narasimhan. Fast greedy algorithm for constructing sparse geometric spanners. *SIAM Journal on Computing*, 31:1479–1500, 2002.

[GM97] Z. Galil and O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Information and Computation*, 134:103–139, 1997.

[Hag00] T. Hagerup. Improved shortest paths on the word RAM. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming, Geneva, Switzerland*, pages 61–72, 2000.

[HK95] M. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science, Milwaukee, Wisconsin*, pages 664–672, 1995.

[KKP93] D.R. Karger, D. Koller, and S.J. Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. *SIAM Journal on Computing*, 22:1199–1217, 1993.

[Pet02] S. Pettie. A faster all-pairs shortest path algorithm for real-weighted sparse graphs. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming, Malaga, Spain*, pages 85–97, 2002.

[PR02] S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms, San Francisco, California*, pages 267–276, 2002.

[PS89] D. Peleg and A.A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13:99–116, 1989.

[RR96] G. Ramalingam and T.W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, 1996.

[RZ02] L. Roditty and U. Zwick. Improved dynamic reachability algorithms for directed graphs. In *Proceedings of the 43th Annual IEEE Symposium on Foundations of Computer Science, Vancouver, Canada*, pages 679–688, 2002.

[RZ04] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. 2004. To appear.

[Sei95] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51:400–403, 1995.

[SZ99] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, New York, New York*, pages 605–614, 1999.

[Tak98] T. Takaoka. Subcubic cost algorithms for the all pairs shortest path problem. *Algorithmica*, 20:309–318, 1998.

[Tho99] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46:362–394, 1999.

[Tho04] M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory, Humlebæk, Denmark*, 2004. To appear.

[TZ01] M. Thorup and U. Zwick. Approximate distance oracles. In *Proceedings of the 33th Annual ACM Symposium on Theory of Computing, Crete, Greece*, pages 183–192, 2001.

[UY91] J. Ullman and M. Yannakakis. High probability parallel transitive closure algorithms. *SIAM Journal on Computing*, 20, 1991.

[YZ04] R. Yuster and U. Zwick. Fast sparse matrix multiplication. 2004. Submitted to ESA'04.