Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs

Liam Roditty
Department of Computer Science
Bar-Ilan University
Ramat-Gan 52900, Israel
liamr@macs.biu.ac.il

Uri Zwick
School of Computer Science
Tel Aviv University
Tel Aviv 69978, Israel
zwick@cs.tau.ac.il

November 8, 2009

Abstract

We obtain three new dynamic algorithms for the approximate all-pairs shortest paths problem in unweighted undirected graphs:

1. For any fixed $\varepsilon > 0$, a decremental algorithm with an expected *total* running time of $\tilde{O}(mn)$, where m is the number of edges and n is the number of vertices in the initial graph. Each distance query is answered in $O(1)$ worst-case time, and the stretch of the returned distances is at most $1+\varepsilon$. The algorithm uses $\tilde{O}(n^2)$ space.
2. For any fixed integer $k \geq 1$, a decremental algorithm with an expected *total* running time of $\tilde{O}(mn)$. Each query is answered in $O(1)$ worst-case time, and the stretch of the returned distances is at most $2k-1$. This algorithm uses, however, only $O(m + n^{1+1/k})$ space. It is obtained by dynamizing techniques of Thorup and Zwick. In addition to being more space efficient, this algorithm is also one of the building blocks used to obtain the first algorithm.
3. For any fixed $\varepsilon, \delta > 0$ and every $t \leq m^{1/2-\delta}$, a fully dynamic algorithm with an expected amortized update time of $\tilde{O}(mn/t)$ and worst-case query time of $O(t)$. The stretch of the returned distances is at most $1+\varepsilon$.

All algorithms can also be made to work on undirected graphs with small integer edge weights. If the largest edge weight is b, then all bounds on the running times are multiplied by b.

1 Introduction

The objective of a dynamic All-Pairs Shortest Paths (APSP) algorithm is to efficiently process an online sequence of update and query operations. Each update operation inserts or deletes edges from an underlying graph. Each query operation asks for the distance between two specified vertices in the current graph. Recall that, 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 of edges, 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*.

The dynamic APSP problem is a very interesting problem, from both the theoretical and practical perspectives, and it received a lot of attention in recent years. Static approximate versions of the APSP problems were also the focus of a lot of research recently. In this paper we show that techniques from these two areas, together with some new ideas, can be combined to obtain very efficient approximate dynamic APSP algorithms for undirected graphs.

The distance from a vertex u to a vertex v in a graph G is denoted by $\delta(u,v)$. We say that an estimate $\hat{\delta}(u,v)$ of the distance $\delta(u,v)$ is of *stretch* t if and only if $\delta(u,v) \leq \hat{\delta}(u,v) \leq t \cdot \delta(u,v)$. We let t-APSP be the problem of producing, upon request, a stretch t estimate of the distance between any two given vertices of the graph. We are especially interested in obtaining stretch $1+\varepsilon$ estimates, for an arbitrary small $\varepsilon > 0$, as they are, in most cases, as good as exact distances. (In particular, for distances up to $1/\varepsilon$, such estimated distances are exact.)

We present two new partially dynamic algorithms and one new fully-dynamic algorithm for the dynamic approximate APSP problem for undirected graphs. Our algorithms are Monte-Carlo, i.e, they have a one-sided error. Our first algorithm is a decremental $(1+\varepsilon)$-APSP algorithm that has, for any fixed $\varepsilon > 0$, a total expected running time of only $\tilde{O}(mn)$ and it answers each distance query in $O(1)$ worst-case time. The algorithm uses $\tilde{O}(n^2)$ space. A running time of $O(mn)$ is a natural barrier for the decremental APSP problem for two reasons: (i) The fastest combinatorial algorithm for the exact *static* problem runs in $O(mn)$ time; (ii) The best decremental algorithm for the *single-source* shortest paths problem, due to Even and Shiloach [15], also runs in $O(mn)$ time.

The $\tilde{O}(n^2)$ space used by our decremental $(1+\varepsilon)$-APSP algorithm may be prohibitive in many practical applications. (See, e.g., the concluding remarks of [10].) Our second decremental algorithm presents a tradeoff between the amount of space used and the accuracy of the estimates obtained. For any integer $k \geq 1$, it uses $O(m + n^{1+1/k})$ space and produces distance estimates of stretch $2k-1$. The total expected running time of the algorithm is still $\tilde{O}(mn)$. The algorithm is obtained by partially dynamizing the *approximate distance oracles* of Thorup and Zwick [23]. Note that the algorithm can still answer any distance query in $O(1)$ time even though it does not maintain an explicit $n \times n$ matrix of distance estimates.

Our first decremental algorithm is based on a sampling technique first used by Ullman and Yannakakis [24]. This technique was also used by [17, 25, 18]. Our second decremental algorithm, as mentioned, is based on the constructions of [23]. Our two decremental algorithms are inter-dependent. Parts from each one of them are used as building blocks in the other algorithm.

Finally, relying on our first decremental algorithm and using a static approximate APSP algorithm of Elkin [13], we obtain, for every $\varepsilon, \delta > 0$, and for every $t \leq m^{1/2-\delta}$, a fully-dynamic $(1+\varepsilon)$-APSP algorithm with an amortized update time of $\tilde{O}(mn/t)$ and a query time of $O(t)$. In particular, we can get a $(1+\varepsilon)$-APSP algorithm with an amortized update time of $\tilde{O}(m^{1/2+\delta}n)$ and a query time of $O(m^{1/2-\delta})$. (Note that $m^{1/2} \leq n$ and $m^{1/2}n \leq n^2$.)

The rest of this paper is organized as follows. In the next section we discuss the relation of our new algorithms to previously available dynamic and static APSP algorithms. Our first decremental algorithm is then developed in two installments. In Section 3 we present a decremental $(1+\varepsilon)$-APSP algorithm with a total running time of $\tilde{O}(mn)$ but with a non-constant query time of $O(\log\log n)$. In Section 4 we then explain how the query time can be reduced to $O(1)$. Our second decremental algorithm is again developed in two installments. In Section 5 we develop, for every $d \geq 1$, a decremental $(2k-1)$-APSP algorithm with a total running time of $\tilde{O}(dmn^{1/k})$ that can produce stretch $2k-1$ estimates for all distances that are at most d. In Section 6 we combine this with parts taken from our first decremental algorithm to obtain a decremental $(2k-1)$-APSP algorithm, for all distances, that runs in $\tilde{O}(mn)$ time and uses only $O(m + n^{1+1/k})$ space. Our fully-dynamic $(1+\varepsilon)$-APSP algorithm is then presented, in one installement, in Section 7. We end in Section 8 with some concluding remarks and open problems.

2 Related work

Demetrescu and Italiano [10], in a major breakthrough, obtained recently a fully dynamic algorithm for the directed APSP problem with an amortized update time of $\tilde{O}(n^2)$. Each distance query is answered in $O(1)$ worst-case time. (Thorup [22] presents an improvement of this result.) Each update operation inserts, deletes, or changes the weights of a set of edges, all incident on the same vertex of the graph. No better algorithm is known for the undirected version of the problem. An amortized update time of $\tilde{O}(n^2)$ is essentially optimal, if the distance matrix is to be explicitly maintained, as done by the algorithm of [10], since each update operation may change $\Omega(n^2)$ distances in the matrix.

The $\tilde{O}(m^{1/2+\delta}n)$ amortized update time of our fully-dynamic $(1+\varepsilon)$-APSP algorithm beats the $\tilde{O}(n^2)$ amortized update time of [10] whenever $m \leq n^{2(1-\delta)}$. The query time, alas, is much larger. (It should be remembered, of course, that our algorithm is for an easier problem. We consider the unweighted and undirected version of the problem and are willing to settle for approximate distances.)

Ausiello *et al.* [2] obtained an incremental algorithm for the APSP problem for unweighted directed graphs with a total running time of $O(n^3 \log n)$. (An extension of this algorithm for graphs with small integer edge weights is given in [3].) Baswana *et al.* [5] obtained a decremental algorithm for the APSP problem for unweighted directed graphs with a total update time of $O(n^3 \log^2 n)$. Both algorithms answer distance queries in $O(1)$ worst-case time. (The *total* running time of a partially dynamic algorithm is the total number of operations performed by the algorithm as the edges of the graph are inserted, or deleted, one by one.)

Baswana *et al.* [5] obtained a decremental algorithm for the directed $(1+\varepsilon)$-APSP problem with a total running time of $\tilde{O}(m^{1/2}n^2)$. In [6] they consider the same problem considered by us here and obtain decremental algorithms for the undirected 3-APSP, 5-APSP and 7-APSP problems with expected running times of $\tilde{O}(mn^{10/9})$, $\tilde{O}(mn^{14/13})$ and $\tilde{O}(mn^{28/27})$, respectively.

Our decremental $(1+\varepsilon)$-APSP algorithm substantially improves on results of Baswana *et al.* [6]. Our algorithm is faster (total running time of $\tilde{O}(mn)$) and more accurate (stretch $1+\varepsilon$). It can also be turned into a zero-error algorithm. Our decremental $(2k-1)$-APSP algorithm also improves on the results of [6]. It is faster, as accurate and uses less space.

As mentioned, there has also been a lot of work on obtaining approximate solutions of the static APSP problem. For more details, and additional references, see [4, 7, 1, 12, 8, 14, 13, 25, 23]. We mention here only two results that have a direct bearing on the current paper.

Thorup and Zwick [23] show that for any fixed integer $k \geq 1$ it is possible to preprocess a weighted undirected graph in $O(mn^{1/k})$ time and produce a data structure of size $O(n^{1+1/k})$ such that any distance in the graph can be approximated in $O(1)$ time. The stretch of the estimated distances produced is $2k-1$. As mentioned, one of the contributions in this paper is a decremental version of these distance oracles.

Elkin [13], extending results of Elkin and Peleg [14], shows that for any $\varepsilon, \delta > 0$ there exists $\beta = \beta(\varepsilon, \delta)$ such that estimated distances from a set of sources S to all vertices of an unweighted undirected graph can be computed in $O(mn^{\delta} + |S|n^{1+\delta})$ time, where m and n are the number edges and vertices, respectively, in the graph. Each estimated distance $\hat{\delta}(u,v)$ satisfies the following inequality $\delta(u,v) \leq \hat{\delta}(u,v) \leq (1+\varepsilon)\delta(u,v) + \beta$. Our fully-dynamic algorithms uses the algorithm of Elkin. (It should be noted that our fully-dynamic algorithm has a stretch of $1+\varepsilon$, *without* the additive error term present in Elkin's result. In particular, all distances smaller than $1/\varepsilon$ are found exactly by our algorithm.)

Finally, we note that the best known algorithm for the static $(1+\varepsilon)$-APSP problem in sparse undirected graphs is still the trivial algorithm of running a BFS from each vertex of the graph. The running time of this algorithm is $O(mn)$. (Using fast matrix multiplication, the exact problem can be solved in $O(n^{2.38})$ time [16, 20, 21], but for sparse enough graphs the $O(mn)$ algorithm is faster.) Our decremental $(1+\varepsilon)$-APSP algorithm, which solves a harder problem, has an almost matching running time of $\tilde{O}(mn)$.

3 A decremental $(1+\varepsilon)$-APSP algorithm with an $O(\log\log n)$ query time

In this section we describe a simple decremental $(1+\varepsilon)$-APSP algorithm that runs, with high probability, in $\tilde{O}(mn/\varepsilon)$ time and has a query time of $O(\log\log n)$. The following obvious observation is similar to an observation used by Ullman and Yannakakis [24] and by various other shortest paths algorithms:

Lemma 3.1 *Let $G = (V, E)$ be a graph on n vertices and let $1 \leq d \leq n$. Let S be a random subset of vertices obtained by selecting each vertex, independently, with probability $(c \ln n)/d$, for some constant c. (If $(c \ln n)/d \geq 1$, we take $S = V$.) Then, with a probability of at least $1 - n^{-(c-1)}$, for every vertex $v \in V$ contained in a connected component of G of size at least d there is a vertex $w \in S$ such that $\delta(w, v) \leq d$.*

Proof: If v is contained in a connected component of size at least d, then there is a set $N(v)$ of at least d vertices that are at distance at most d from v. The probability that S does not contain any of these vertices is $(1-(c\ln n)/d)^d < n^{-c}$. Multiplying this by the number of vertices, we get that the failure probability is at most $n^{-(c-1)}$. □

As stated, the lemma applies to a fixed graph. However, as the choice of the random set S is independent of the graph, it is clear that the lemma also applies in the dynamic setting. The failure probability should simply be multiplied by the number of different versions of the graph. In the decremental setting, we consider only $m \le n^2$ versions of the graph, so the failure probability is at most $n^{-(c-3)}$.

We are now ready to start the description of our algorithm. Let $I = \{1, 2, \ldots, \log n\}$. For every $i \in I$, let S_i be a random set obtained by sampling each vertex of V, independently, with probability $q_i = \min\{\frac{c \ln n}{\varepsilon 2^i}, 1\}$, where ε is the desired accuracy of the reported distances, and c is a large enough constant that controls the error probability. (For the first $O(\log\log n)$ indices we have $q_i = 1$, so they are not really needed.)

For every $u \in V$ and $i \in I$, we let $p_i(u)$ be a closest vertex to u from S_i. If there is no vertex from S_i in the connected component of u, then $p_i(u)$ is undefined and we let $\delta(u, p_i(u)) = \infty$. By Lemma 3.1, if $\delta(u, p_i(u)) < \infty$, then $\delta(u, p_i(u)) \le \varepsilon 2^i$, with high probability. In the sequel, we assume that this happens for every $u \in V$ and $i \in I$. (If $\varepsilon 2^i < \delta(u, p_i(u)) < \infty$, then the choice of S_i is 'unlucky', and we can replace it. To find the $p_i(u)$'s we add a new vertex s_i to the graph and connect it with edges to all the vertices of S_i. We then maintain, using [15], a decremental shortest paths tree from s_i. The cost of decrementally maintaining a single shortest paths tree up to depth d is $O(md)$. Thus, the total cost of maintaining these $O(\log n)$ trees up to depth n is $O(mn\log n)$. If $u \in V$ is contained in the subtree of w, we set $p_i(u)$ to w. After each edge deletion, we can update the $p_i(u)$'s in $O(n \log n)$ time, so the total time spent on updating these values is also only $O(mn\log n)$.

For every $i \in I$ and every $w \in S_i$, we also decrementally maintain, using the algorithm of [15], the first 2^{i+2} levels of a shortest paths tree from w. As the cost of decrementally maintaining a single shortest paths tree up to depth d is $O(md)$, the total cost of maintaining all the trees is

$$O(\sum_i |S_i| m 2^i) \;=\; O(\sum_i \frac{cn\ln n}{\varepsilon 2^i} m 2^i) \;=\; O(\frac{mn\log^2 n}{\varepsilon})\ .$$

A query asking for the distance from u to v is answered in the following way. If we somehow know that $2^i \le \delta(u,v) < 2^{i+1}$, we can return $\delta(u, p_i(u)) + \delta(p_i(u), v)$. Note that as $\delta(p_i(u), u) \le \varepsilon 2^i$ and $\delta(p_i(u), v) \le \delta(p_i(u), u) + \delta(u, v) < 2^{i+2}$, both distances $\delta(u, p_i(u))$ and $\delta(p_i(u), v)$ can be found in the tree of $p_i(u)$. It is easy to see that we have

$$\begin{aligned}
\hat{\delta}(u,v) &= \delta(u,p_i(u)) + \delta(p_i(u),v) \\
&\le \delta(u,p_i(u)) + (\delta(p_i(u),u) + \delta(u,v)) \\
&= \delta(u,v) + 2\delta(u,p_i(u)) \\
&\le (1+2\varepsilon)\delta(u,v)\ .
\end{aligned}$$

Clearly $\delta(u,v) \le \hat{\delta}(u,v)$. Thus, the stretch of the estimate produced is at most $1+2\varepsilon$.

As we do not know the right i, the obvious approach is to check all values $i \in I$ and return the minimum estimated distance obtained. (Not all values of i yield such an estimate, as u or v may not be contained in the tree of $p_i(u)$. We simply ignore such values of i.) This gives us a decremental $(1+\varepsilon)$-APSP algorithm with a total running time of $O(mn\log^2 n/\varepsilon)$ and a query time of $O(\log n)$.

We can reduce the query time to $O(\log\log n)$ using binary search. Suppose again that $2^i \le \delta(u,v) < 2^{i+1}$. Suppose we try to get an estimate using $j \le i$. If the attempt succeeds, we get an estimate of stretch at most $1+\varepsilon$. If it fails, because v is not contained in the tree of $p_j(u)$, then we know that our choice of j was too small. If we try to get an estimate using a value $j > i$, then the attempt may fail as $p_j(u)$ may not be defined, but then we know that our choice of j is too large. Thus, we can use binary search to find the smallest j for which we do get an estimate, and this estimate will be of stretch at most $1+\varepsilon$.

4 A decremental $(1+\varepsilon)$-APSP algorithm with an $O(1)$ query time

We first explain how the query time can be reduced to $O(1)$ if it is known that $\delta(u,v) \geq n^{1/2}$.

Let $r = \lfloor \frac{1}{2} \log n \rfloor$. We keep a table of estimated distances between any pair of vertices in S_r. The size of the table is $|S_r|^2 = \tilde{O}(n)$. After each update, we recompute the table, by querying the algorithm of Section 3. As each query takes $O(\log \log n)$ time, the total time needed is only $\tilde{O}(n)$.

When asked for an estimate of $\delta(u,v)$, we return $\delta(u, p_r(u)) + \hat{\delta}(p_r(u), p_r(v)) + \delta(p_r(v), v)$. This takes only $O(1)$ time, as $\delta(u, p_r(u))$ is stored in the tree of $p_r(u)$, $\delta(p_r(v), v)$ is stored in the tree of $p_r(v)$, and $\hat{\delta}(p_r(u), p_r(v))$ is stored in the table we prepared. It is not difficult to show, using an argument similar to the one used above, that the stretch of this estimate, if $\delta(u,v) \geq n^{1/2}$, is at most $1+4\varepsilon$.

To handle shorter distances, we use the decremental approximate distance oracles of the next section. Choosing $k=2$, we get a decremental oracle for distances up to $d = n^{1/2}$ whose total running time is $\tilde{O}(dmn^{1/2}) = \tilde{O}(mn)$. Each query is answered in $O(1)$ time with a stretch of at most 3. We can, however, use this crude estimate as a start for our search, from the previous section, for the right value of $i \in I$. The search will now take only $O(1)$ time and produce an estimate of stretch $1+\varepsilon$.

5 A decremental distance oracle for relatively short distances

Thorup and Zwick [23] constructed static distance oracles with the following properties:

Theorem 5.1 ([23]) *Let $G = (V, E)$ be an undirected graph with positive weights attached to its edges. Let $|E| = m$ and $|V| = n$. Let $k \geq 1$ be a fixed integer. Then, it is possible to preprocess G in $O(mn^{1/k})$ expected time, and produce a data structure of size $O(n^{1+1/k})$, such that for any $u, v \in V$ it is possible to produce, in $O(1)$ worst-case time, an estimate $\hat{\delta}(u,v)$ of the distance $\delta(u,v)$ from u to v in G that satisfies $\delta(u,v) \leq \hat{\delta}(u,v) \leq (2k-1) \cdot \delta(u,v)$.*

In this section we obtain the following partially dynamic version of these oracles:

Theorem 5.2 *Let $G = (V, E)$ be an undirected graph with integer weights attached to its edges that undergoes a sequence of edge deletions. Let $|E| = m$ and $|V| = n$. Let $k \geq 1$ be a fixed integer and let $d \geq 1$. It is possible to maintain, in $O(dmn^{1/k})$* total *expected time, a data structure of size $O(m + n^{1+1/k})$, such that after each edge deletion, for every $u, v \in V$ it is possible to produce, in $O(1)$ worst-case time, an estimate $\hat{\delta}(u,v)$ of the distance $\delta(u,v)$ from u to v with the following properties: If $\delta(u,v) \leq d$, then $\delta(u,v) \leq \hat{\delta}(u,v) \leq (2k-1) \cdot \delta(u,v)$. If $\delta(u,v) > d$, then $\delta(u,v) \leq \hat{\delta}(u,v)$.*

Before we present our partially dynamic oracles, we need to review the static construction of [23]. We do that in the next subsection. In Section 5.2 we then present our partially dynamic version.

5.1 The static distance oracle of Thorup and Zwick

The construction starts by defining a hierarchy $A_0 \supseteq A_1 \supseteq A_2 \supseteq \cdots \supseteq A_k$ of subsets of V as follows: We start with $A_0 = V$. For every $1 \leq i < k$, we let A_i be random subset of A_{i-1} obtained by selecting each element of A_{i-1}, independently, with probability $n^{-1/k}$. Finally, we let $A_k = \phi$. The elements of A_i are referred to as *i-centers*. We let $\delta(v, A_i) = \min_{w \in A_i} \delta(w, v)$, for $0 \leq i < k$. As $A_k = \phi$, we let $\delta(v, A_k) = \infty$. For every $v \in V$ and $0 \leq i < k$, we let $p_i(v) \in A_i$ be such that $\delta(p_i(v), v) = \delta(v, A_i)$. (Note that $p_0(v) = v$.)

```
DIST_k(u, v) :
w ← u ; i ← 0
while w ∉ B(v)
    i ← i + 1
    (u, v) ← (v, u)
    w ← p_i(u)
return δ(w, u) + δ(w, v)
```

Figure 1: The query answering algorithm of [23].

Definition 5.3 (Clusters and bunches [23])
For every i-center $w \in A_i - A_{i+1}$, where $0 \le i < k$, we define the cluster $C(w)$ *as follows:*

$$C(w) \;=\; \{\, v \in V \mid \delta(w,v) < \delta(v, A_{i+1}) \,\} \;.$$

For every $v \in V$ we define the bunch $B(v)$ *as follows:*

$$B(v) \;=\; \bigcup_{i=0}^{k-1} B_i(v) \;,$$

where

$$B_i(v) \;=\; \{\, w \in A_i - A_{i+1} \mid \delta(w,v) < \delta(v, A_{i+1}) \,\} \;.$$

Clearly, $v \in C(w)$ if and only if $w \in B(v)$. Clusters have the following important 'connectedness' property:

Lemma 5.4 ([23]) *If $v \in C(w)$ and u is on a shortest path from w to v in G, then $u \in C(w)$.*

Proof: Suppose $w \in A_i - A_{i+1}$. If $u \notin C(w)$, then $\delta(u, A_{i+1}) \le \delta(w,u)$. But then $\delta(v, A_{i+1}) \le \delta(v,u) + \delta(u, A_{i+1}) \le \delta(v,u) + \delta(u,w) = \delta(v,w)$, contradicting the assumption that $v \in C(w)$. □

It follows that the cluster $C(w)$ can be constructed by running a modified version of Dijkstra's algorithm from w. Dijkstra's algorithm maintains for each vertex u, encountered during the search from w, a *tentative distance* $d[u]$. At the start of the algorithm the only encountered vertex is w, and $d[w] = 0$. Each encountered vertex is either *marked* or *unmarked*. All encountered vertices are initially unmarked. The encountered vertices that are still unmarked are held in a priority queue $Q(w)$. The key associated with each encountered vertex u is $d[u]$, its tentative distance from w. In each iteration the algorithm chooses an unmarked vertex u with a smallest tentative distance and marks it. It then *relaxes* all the edges touching u. An edge $(u,v) \in E$ is relaxed as follows. If v was not encountered yet, we set $d[v] \leftarrow d[u] + \ell(u,v)$. (Here $\ell(u,v)$ is the length of the edge (u,v).) If v was already encountered, so $d[v]$ is already defined, we let $d[v] \leftarrow \min\{d[v], d[u] + \ell(u,v)\}$. It is not difficult to show that when a vertex u is marked, $d[u] = \delta(w,u)$. (The proof can be found in any textbook, e.g., [9].) The algorithm halts when all the encountered vertices are marked.

The simple modification required in Dijkstra's algorithm is the following: Relax an edge $(u,v) \in E$ only if $d[u] + \ell(u,v) < \delta(A_{i+1}, v)$. It is not difficult to see that the vertices encountered, and marked, by this modified version of Dijkstra's algorithm are exactly the vertices of $C(w)$. For the straightforward correctness proof, the reader is referred to [23].

The analysis of the construction relies on the following bound on the expected size of the bunches:

Lemma 5.5 ([23]) *For every vertex $v \in V$ and every $0 \le i < k$ we have $E[|B_i(v)|] \le n^{1/k}$.*

Proof: The claim for $i = k-1$ is obvious as $E[|A_{k-1}|] = n^{1/k}$. Suppose, therefore, that $0 \leq i < k-1$. Let $w_1, w_2, \ldots$ be the vertices of A_i in a non-decreasing order of distance from v. Then, $w_j \in B_i(v)$ only if $w_1, w_2, \ldots, w_{j-1} \notin A_{i+1}$. As each element of A_i becomes an element of A_{i+1}, independently, with probability $p = n^{-1/k}$, we get that $\Pr[w_j \in B_i(v)] \leq (1-p)^{j-1}$. Thus, $E[|B_i(v)|] = \sum_{j\geq 1} \Pr[w_j \in B_i(v)] \leq \sum_{j \geq 1}(1-p)^j \leq p^{-1} = n^{1/k}$, as required. $\square$

As an immediate consequence, we get that each vertex $v \in V$ is contained in an expected number of at most $kn^{1/k}$ clusters.

A distance query is answered using the algorithm given in Figure 1. To check the condition $w \notin B(v)$ in constant time, we keep, for every $v \in V$, a hash table containing $B(v)$. Each distance query is therefore answered in $O(k)$ time, which is $O(1)$ time, as k is fixed. It is shown in [23] that the estimate $\hat{\delta}(u, v)$ returned is of stretch at most $2k-1$.

5.2 A decremental version of the distance oracle

In this section we describe a decremental version of the approximate distance oracle of [23]. The challenge is to maintain the bunches and clusters as the graph undergoes a sequence of edge deletions.

As the center hierarchy $A_0 \supseteq A_1 \supseteq A_2 \supseteq \cdots \supseteq A_k$ is picked in an oblivious manner, without even looking at the graph $G = (V, E)$, we can use the same center hierarchy for all the versions of the graphs. We next describe how we update the clusters and bunches following the deletion, in turn, of each edge of G.

To find, for every $v \in V$ and $0 \leq i < k$, the i-center $p_i(v)$ closest to v, we add a dummy source vertex s_i to the graph and connect it with edges of length 0 to all the vertices of A_i. We use the algorithm of [15] to decrementally maintain the first $\bar{d} = (2k-1)d$ levels of a shortest paths tree from s_i as the graph undergoes a sequence of edge deletions. The collection of these k trees is referred to as the *set forest*. The total time required, for each value of i, is $O(dm)$. As k is a constant, the total time required for all values of i is also $O(dm)$. Maintaining $\delta(v, A_i)$ is therefore fairly straightforward. Maintaining closest vertices $p_i(v)$ for which $\delta(v, p_i(v)) = \delta(v, A_i)$, for every $v \in V$ and $0 \leq i < k$, is more subtle. It can be done, however, in $O(dmn^{1/k})$ total time.

For the sake of efficiency, we tune a bit the definition of the closest vertex. Let $v \in V$, $p_i(v)$ is the vertex from A_i that survives to the set A_j, where $j > i$ is maximal and its distance from v is $\delta(A_i, v)$. It follows from this definition that a vertex changes its closest vertex between A_i and A_{i+1} only if $p_i(v) \notin A_{i+1}$. It also follows that whenever such a change occurs then $\delta(A_i, v) < \delta(A_{i+1}, v)$.

Maintaining closest vertices. As explained above we add a dummy source vertex s_i to the graph and connect it with edges of length 0 to all the vertices of A_i. In fact we can connect s_i only to the vertices of $A_i - A_{i+1}$. The value of $\delta(A_i, v)$ can be computed by scanning the trees starting with the tree of s_{k-1} backwards till s_i.

Let (u, v) be an edge which is deleted from the graph and assume it is a tree edge in the tree of s_i. There are two cases. The first case is when $p_i(v) \in A_{i+1}$. If this is the case we simply process the deletion of (u, v) from the tree of s_i as a regular deletion from a decremental shortest paths tree using the algorithm of [15].

The second case is when if $p_i(v) \in A_i - A_{i+1}$. Since (u, v) is a tree edge it must be that the path from $p_i(v)$ to v in the tree uses the edge (u, v). In this case we need to check whether the vertex that serves as $p_i(v)$ before the deletion can serve as $p_i(v)$ also after the deletion. Let $w_1, w_2, \ldots, w_\ell \in A_i - A_{i+1}$ and assume, w.l.o.g, that $p_i(v) = w_1$. As a first step we try to check whether there is a path from v to w_1 of the same length. To do that we search among the edges of v, that were not checked yet as a possible connection to w_1 at the current length, for an edge (u', v) such that $p_i(u') = w_1$ and $\delta(u', w_1) = \delta(u, w_1)$. If such an edge is found we can stop without doing any further change. If such an edge does not exist then we try to connect v to s_i throughout other w_j. By scanning the incoming edges of v, from the beginning of the list now, we find (if exist) an edge (u', v) such that $p_i(u') = w_j$ and $\delta(u', w_j) = \delta(u, w_1)$, for some u' and w_j. If such a replacement is found, then $p_i(v)$ is set to w_j.

If we do not succeed to connect v to the tree with a path of the same length as before the deletion then we increment $\delta(A_i, v)$ by one and try to reconnect it to the tree as in a regular decremental shortest paths tree.

We repeat on the same process recursively for every vertex v' that was hanged on v before the deletion. If its closest vertex $p_i(v')$ is in $A_i - A_{i+1}$ we first try to keep it unchanged and if we do not succeed then we scan the edge list of v' from the beginning for a replacement.

After we are done with the deletion of (u, v) from the trees of $s_0, s_2, \ldots, s_{k-1}$ we need to preform closest vertex update for every vertex v that after the deletion for some i the value $\delta(A_i, v)$ was increased or its closest vertex $p_i(v)$ that was in $A_i \setminus A_{i+1}$ before the deletion is no longer its closest vertex. We can detect all these vertices without increasing the total update cost. We scan the trees from s_{k-1} to s_1 and compute the closest vertex for such vertices according to the tune definition, that is, for every j we search for the maximal j' which is greater than j and satisfies $\delta(A_j, v) = \delta(A_{j'}, v)$. The total cost of that is $O(k)$ and it is done only to vertices that their distance was increased.

The correctness of the algorithm stems from the correctness of the algorithm of [15]. The running time, however, has to be carefully analyzed. Every time the distance to a vertex is increased then we are free to scan its edges as the total time this event occurs is only $\bar{d}$ times and the vertex participates in k trees which results in $O(md)$ total time. The problem is when the distance is not increased and after an edge is deleted we have to verify that the previous $p_i(v)$ is still the closest vertex from $A_i - A_{i+1}$ to v and not some other vertex from $A_i - A_{i+1}$. When v searches in its edge list it does it given a specific distance. It tries to find a replacement edge that keeps the same $p_i(v)$ with the same distance. Every edge is tested exactly once for this purpose. Thus, given a distance and a vertex from $A_i - A_{i+1}$ that serves before the deletion as $p_i(v)$ the edges of v are scanned only once. There are $\bar{d}$ possible distances. The question is for a given distance how many possible vertices there are from $A_i - A_{i+1}$ that can serve as $p_i(v)$. Note that since we are using the tune definition of closest vertex, every such a vertex is also in $B_i(v)$, and from Lemma 5.5 we know that the expected size of $B_i(v)$ is at most $n^{1/k}$. Thus, we conclude that the total expected running time is $O(dmn^{1/k})$.

Maintaining cluster. Another challenging task is keeping track of the changes in the clusters $C(w)$, for every $w \in V$. Recall that $v \in C(w)$, where $w \in A_i - A_{i+1}$, if and only if $\delta(w, v) < \delta(v, A_{i+1})$. As the graph $G = (V, E)$ is only loosing edges, both $\delta(w, v)$ and $\delta(v, A_{i+1})$ can only increase. But, the order relation between $\delta(w, v)$ and $\delta(v, A_{i+1})$ may change several times as the edges of $G = (V, E)$ are deleted, one by one. Thus, vertices may both join and leave $C(w)$.

The modified version of Dijkstra's algorithm described in Section 5.1 constructs, for every $w \in V$, a tree of shortest paths from w to all vertices in $C(w)$. We again use the algorithm of [15] to decrementally maintain this tree, up to level $\bar{d} = (2k - 1)d$. The collection of these n trees is referred to as the *cluster forest*. The basic property of the algorithm of [15] is that an edge touching a vertex v is rescanned only following an increase in the distance from w to v.

For every vertex v in the tree of $C(w)$ whose distance from w increased as a result of the last edge deletion, we check whether v should still belong to $C(w)$. If not, we remove v from $C(w)$. (Note that if v is removed from $C(w)$, then by Lemma 5.4 all vertices in the subtree of v are also removed from $C(w)$.)

Finding the vertices that should join the cluster $C(w)$ is a somewhat more complicated process. After each edge deletion we construct, for every $0 \leq i < k$, a set X_i of all the vertices whose distance to A_i increased as a result of the deletion, but for which this distance is still at most $\bar{d}$. Recall that $v \in C(w)$, where $w \in A_i - A_{i+1}$, if and only $\delta(w, v) < \delta(v, A_{i+1})$. Thus, a vertex v can join $C(w)$ only after an increase in $\delta(v, A_{i+1})$, i.e., only if $v \in X_{i+1}$.

Let $v \in X_{i+1}$. To find out whether v should join a cluster $C(w)$, where $w \in A_i - A_{i+1}$, we should check whether $\delta(w, v) < \delta(v, A_{i+1})$. However, the distance $\delta(w, v)$ may not be known to us at this stage, so we cannot check this condition directly. We thus try, at first, to check whether v should join clusters that contain neighbors of v. Note that a vertex $v \in X_{i+1}$ may potentially join many clusters, and not just one.

For every $v \in X_{i+1}$, every edge $(u, v) \in E$ and every i-center $w \in B_i(u) - B_i(v)$, we check whether $\delta(w, u) +$

```
Dijkstra(w):

while Q(w) ≠ φ
    u ← Extract-Min(Q(w))
    δ(w, u) ← d_w[u]
    B_i(u) ← B_i(u) ∪ {w}
    for each (u, v) ∈ E s.t. w ∉ B_i(v)
        if δ(w, u) + ℓ(u, v) < δ(v, p_i(s))
            Relax(Q(w), u, v)
```

```
Relax(Q(w), u, v):

d' ← δ(w, u) + ℓ(u, v)
if d' ≤ d̄ then
    if v ∉ Q(w) then
        decrease-key(Q(w), v, d')
    else if d_w[v] > d'
        insert(Q(w), v, d')
```

```
Examine(X_{i+1}):

C ← φ
for each v ∈ X_{i+1} do
    for each (u, v) ∈ E do
        for each w ∈ B_i(u)−B_i(v)
            if δ(w, u)+ℓ(u, v) < δ(v, A_{i+1})
                C ← C ∪ {w}
                Relax(Q(w), u, v)
for each w ∈ C do
    Dijkstra(w)
```

Figure 2: Decremental maintenance of bunches and clusters

$\ell(u, v) < \delta(v, A_i)$. If so, then v should clearly join $C(w)$. (Note that v may join $C(w)$ even if $\delta(w, u) + \ell(u, v) \geq \delta(v, A_i)$, as there might be a shorter way of getting from w to v without passing through u. We will detect that later.) If v should join $C(w)$ we add v to a priority queue $Q(w)$ with an associated key $d_w[u] = \delta(w, u) + \ell(u, v)$. If v was already contained in $Q(w)$, we decrease its key, if appropriate.

This initial stage produces, for every i-center $w \in A_i - A_{i+1}$ a priority queue $Q(w)$ containing vertices that should definitely join $C(w)$. Not all vertices that are to join $C(w)$ are necessarily contained in $Q(w)$, but as we shall argue later, if a vertex v should join $C(w)$, then there is a shortest path from w to v that passes through a vertex added to $Q(w)$.

After this initial stage is over, we simply restart, for every i-center w the modified Dijkstra's algorithm from w, with $Q(w)$ serving the role of the priority queue that holds the vertices that were encountered, but not yet marked. We claim that this process will encounter, and subsequently mark, all vertices that should join $C(w)$, and only them.

A pseudo-code describing this two stage process is given in Figure 2. For every $0 \leq i < k$, we issue a call to $Examine(X_{i+1})$. These calls will find all vertices that should join clusters and add them to the appropriate clusters. Procedure $Examine(X_{i+1})$ performs the first stage of the process described above and calls procedure *Dijkstra* to restart the modified Dijkstra's algorithm to complete the construction of $C(w)$.

Theorem 5.2 follows from the following two lemmas:

Lemma 5.6 *The algorithm described above correctly maintains the clusters.*

Proof: The proof is a simple extension of the correctness proof of the modified Dijkstra algorithm. We have to show that the algorithm correctly updates the clusters after an edge deletion. The removal of vertices from clusters is a relatively straightforward process as if a vertex is currently in the cluster its distance to the cluster root is known and a violation of the cluster rule can be detected easily. The more difficult task is identifying vertices that should join clusters as their distance from the cluster root is not known. Let $w \in A_i - A_{i+1}$ and let v be a vertex that before the deletion is not contained in $C(w)$ and after the deletion it should be contained in $C(w)$. Let x be the first vertex on the shortest path from w to v after the deletion that was not in $C(w)$ before the deletion. By the definition of X_{i+1} it follows that $x \in X_{i+1}$. When $Examine(X_{i+1})$ is called then there is an edge from a vertex in $C(w)$ to x which causes x to be added to $Q(w)$ and to w to be added to $\mathcal{C}$. When $Dijkstra(w)$ is called it follows from the correctness proof of the modified $Dijkstra$ that the vertex v will be found. □

Lemma 5.7 *The total expected cost for maintaining the cluster forest is $O(dmn^{1/k})$.*

Proof: As mentioned, the algorithm of [15] rescans the edges of a vertex v, in a shortest paths tree rooted at w, only when the distance from w to v increases. As we only keep the first $\bar{d} = (2k-1)d$ levels of the trees, the edges of each vertex are scanned at most $\bar{d}$ times per tree. As each vertex is contained in an expected number of only $kn^{1/k}$ trees, we would like to claim that the expected number of times that the edges of a vertex v are scanned is at most $\bar{d}kn^{1/k}$. The lemma would then follow. This reasoning is basically correct, but its rigorous proof is quite subtle. The difficulty lies in the fact that vertices may belong to different trees at different times.

Let $w \in A_i \setminus A_{i+1}$. The edges of v are scanned in $C(w)$ once when v joins $C(w)$ and then each time $\delta(v, w)$ changes until v leaves $C(w)$. We first separately analyze the cost of joining new clusters. In the decremental setting, v can only join $C(w)$ if $\delta(v, A_{i+1})$ increases, which can happen at most $\bar{d}$ times. Each time, v joins $O(n^{1/k})$ clusters. Thus, the total number of times the edges of v are scanned because of v joining a cluster is $O(k\bar{d}n^{1/k})$.

We now turn to analyze the case where the distance between v and the cluster center decreases. This will allow us to bound the expected number of times that the edges of a vertex $v \in V$ are rescanned in trees rooted at vertices of A_i. Let $\delta_t(w, v)$ denote the distance from w to v in the graph at *time* t, i.e., after the deletion of the first t edges, and let $C_t(w)$ be the cluster of w at that time. To bound the number of times that the edges of v are scanned, we bound the number of indices t for which $v \in C_t(w)$ and $\delta_t(w, v) < \delta_{t+1}(w, v)$.

In the spirit of the proof of Lemma 5.5, we let $w_{t,1}, w_{t,2}, \ldots$ be the vertices of A_i arranged in non-decreasing order of distance from v after the t-th deletion. In the proof of Lemma 5.5 ties in distances were resolved arbitrarily. Here, we should be slightly more careful. We arrange them in a non-decreasing lexicographic order of $(\delta_t(v, w), \delta_{t+1}(v, w))$. Thus, if w and w' have the same distance from v at time t, and the distance of w' increases as a result of the next edge deletion, but the distance of w does not, then w appears before w' in the ordering. With this ordering, we have the following important property:

Claim 5.8 *For every $v \in V$ and $j \geq 1$, the sequence $\delta_t(v, w_{t,j})$ is non-decreasing. Furthermore, if $\delta_t(v, w_{t,j}) < \delta_{t+1}(v, w_{t,j})$ then also $\delta_t(v, w_{t,j}) < \delta_{t+1}(v, w_{t+1,j})$.*

As in the proof of Lemma 5.5, the probability that $v \in C_t(w_{t,j})$ is at most $(1-p)^{j-1}$, where $p = n^{-1/k}$. Let $I = \{\, (t, j) \mid \delta_t(v, w_{t,j}) < \delta_{t+1}(v, w_{t,j}) \leq \bar{d}\,\}$. Clearly, the expected number of times the edges of v are scanned, in all trees rooted at vertices of A_i, is at most $\sum_{(t,j)\in I} \Pr[v \in C_t(w_{t,j})]$.

By Claim 5.8, for each j, the set I contains at most $\bar{d}$ pairs of the form (t, j). In other words, there are at most $\bar{d}$ times in which the distance to the j-th closest vertex to v increases. Thus $\sum_{(t,j)\in I} \Pr[v \in C_t(w_{t,j})] \leq \bar{d} \sum_{j\geq 1} (1-p)^{j-1} \leq \bar{d}p^{-1} = \bar{d}n^{1/k}$, as required. □

Finally, after reconstructing the clusters, and hence the bunches, we use a dynamic hashing algorithm ([11],[19]) to update the hash table of each bunch.

6 Decremental distance oracles for all distances

We use the decremental $(2k-1)$-APSP algorithm of the previous section with $d = n^{1-1/k}$. The total running time is then $\tilde{O}(dmn^{1/k}) = \tilde{O}(mn)$. To take care of distances that are at least d we use a slightly modified version of the $(1+\varepsilon)$-APSP algorithm of Section 4. Instead of working with $I = \{1, 2, \ldots, \log n\}$ we work with the set $I_d = \{\log d, \ldots, \log n\}$. For each $i \in I_d$ we maintain trees (of depth at most 2^{i+2}) from the $(cn \ln n)/(\varepsilon 2^i)$ vertices of S_i. The amount of space needed for storing all these trees is only $O(n \sum_{i \geq \log d} |S_i|) = O(n \sum_{i \geq \log d}(cn \ln n)/(\varepsilon 2^i)) = \tilde{O}(n^2/d) = \tilde{O}(n^{1+1/k})$. Thus, the total amount of space used by the combination of the two algorithms is only $\tilde{O}(m + n^{1+1/k})$, as promised.

7 A fully-dynamic $(1+\varepsilon)$-APSP algorithm

The fully-dynamic algorithm uses a technique of [17] for converting a decremental algorithm into a fully-dynamic algorithm. This technique was also used by us in the previous part. A few additional ideas are required here, however.

The 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 Section 4. A random subset $S \subseteq V$ of vertices, of size $(cn \ln n)/(\varepsilon d)$, is chosen, where d is a parameter to be chosen later. The static algorithm of Elkin [13] is used to find approximate distances from the vertices of S to all vertices of the graph. For any vertex $v \in V$, we let $p(v) \in S$ be the vertex of S closest to v. The set C is initialized to the empty set.

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 d levels of shortest paths trees $T_{in}(v)$ and $T_{out}(v)$, containing shortest paths to and from v, are constructed. The trees $T_{in}(v)$ and $T_{out}(v)$ are constructed and maintained using the algorithm of [15]. Finally, the algorithm of [13] is rerun to find the new distances from the vertices of S to all vertices of the graph. For any vertex $v \in V$, we let $p(v) \in S$ be the vertex of S closest to v.

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 Section 4. Next, the algorithm of [15] is used to update the shortest paths trees $T_{in}(v)$ and $T_{out}(v)$, for every $v \in C$. Finally, the algorithm of [13] is rerun to find the new distances from the vertices of S to all vertices of the graph, and for every $v \in V$ we again let $p(v) \in S$ be the vertex of S closest to v.

A distance query $Query(u, v)$, asking for an estimate of the distance $\delta(u, v)$ from u to v in the current 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 $\delta(u, v) \leq \ell_1$, 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 $\ell_1 \leq (1+\varepsilon)\delta(u, v)$.

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 check whether $u \in T_{in}(w)$ and $v \in T_{out}(w)$. If so, we compute a bound $\delta(u, w) + \delta(w, v)$ on the distance $\delta(u, v)$. (The distance $\delta(u, w)$ is obtained by querying $T_{in}(w)$ while $\delta(w, v)$ is obtained by querying $T_{out}(v)$.) By taking the minimum of all these bounds we get a second distance estimate that we denote by ℓ_2. (If there is no $w \in C$ for which $u \in T_{in}(w)$ and $v \in T_{out}(w)$, then $\ell_2 = \infty$.) Again, we have $\delta(u, v) \leq \ell_2$. Furthermore, if $\delta(u, v) \leq d$, 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 $\delta(u, v) = \ell_2$. Finally, we let $\ell_3 \leftarrow \delta(u, p(u)) + \delta(p(u), v)$. The final answer returned by the algorithm is $\min\{\ell_1, \ell_2, \ell_3\}$.

As the query time of the decremental algorithm of Section 4 is $O(1)$, the query time here is $O(t)$. To minimize the amortized update time, we set $d = n^{1+\delta}/m^{1/2}$, where $\delta > 0$ is an arbitrary small constant.

Theorem 7.1 *For any fixed $\varepsilon, \delta > 0$ and every $t \le m^{1/2}/n^{\delta}$, the fully dynamic approximate all-pairs shortest paths algorithm has an expected amortized update time of $\tilde{O}(mn/t)$ and worst-case query time of $O(t)$. The stretch of the returned distances is at most $1+\varepsilon$.*

Proof: The correctness of the algorithm follows from the arguments outlined above. As each estimate ℓ_1, ℓ_2 and ℓ_3 obtained is the length of a path in the graph from u to v, we have $\delta(u,v) \le \ell_1, \ell_2, \ell_3$. Thus, the estimate returned by the algorithm can never be too small.

If there is a shortest path from u to v that does not use any edge inserted in the current phase, then $\ell_1 \le (1+\varepsilon)\delta(u,v)$. 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 $\delta(u,v) \le d$, then the exact distance from u to v will be found while examining the trees $T_{in}(w)$ and $T_{out}(w)$.

Finally, suppose that $\delta(u,v) \ge d$. With very high probability, we have $\delta(u,p(u)) \le \frac{1}{2}\varepsilon d$, and therefore $\delta(u,p(u)) + \delta(p(u),v) \le \delta(u,p(u)) + (\delta(p(u),u) + \delta(u,v)) = \delta(u,v) + 2\delta(u,p(u)) \le (1+\varepsilon)\delta(u,v)$.

We next analyze the complexity of the algorithm. The total cost of maintaining the decremental data structure is $\tilde{O}(mn)$. As each phase is composed of at least t update operations, this contributes $\tilde{O}(\frac{mn}{t})$ 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 d. The total cost of maintaining these trees is $O(dm)$. (Note that this also covers the cost of all future operations performed on these trees.) Finally, each insert or delete operation requires the recomputation of approximate distances from S. Using the algorithm of [13], this takes $O(mn^{\delta} + |S|n^{1+\delta})$ time. As $|S| = O((n\log n)/d)$, the running time is $\tilde{O}(mn^{\delta} + \frac{n^{2+\delta}}{d})$. For every $u \in S$ and $v \in V$, we get an estimate $\hat{\delta}(u,v)$ of the distance $\delta(u,v)$ that satisfies $\delta(u,v) \le \hat{\delta}(u,v) \le (1+\frac{\varepsilon}{2})\delta(u,v) + \beta(\delta,\varepsilon)$, where $\beta(\delta,\varepsilon)$ is a constant. If $\delta(u,v) \ge 2\beta/\varepsilon$, then $\delta(u,v) \le \hat{\delta}(u,v) \le (1+\varepsilon)\delta(u,v)$. (This will be satisfied, as our choice of d will be non-constant.) The total amortized cost of each update operation is therefore $\tilde{O}(\frac{mn}{t} + dm + mn^{\delta} + \frac{n^{2+\delta}}{d})$.

Each query is handled by the algorithm in $O(t)$: The estimate ℓ_1 is obtained in $O(1)$ 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 again obtained in $O(1)$ time.

To minimize the amortized update time, we choose $d = n^{1+\delta}/m^{1/2}$. The amortized update time is then $O(\frac{mn}{t} + m^{1/2}n^{1+\delta})$. (Note that $m^{1/2} \le n$.) For $t \le m^{1/2}/n^{\delta}$ we have $m^{1/2}n^{1+\delta} \le \frac{mn}{t}$. Thus, for $t \le m^{1/2}/n^{\delta}$ we get an amortized update time of $\tilde{O}(mn/t)$ and query time $O(t)$. □

8 Concluding remarks and open problems

We obtained two new decremental algorithms and one new fully-dynamic algorithm for the dynamic approximate APSP problem for unweighted undirected. The total running time of our decremental algorithms is $\tilde{O}(mn)$, a bound that will be hard to beat as almost any improvement on it will yield improved results also for the static version of the problem. It is not difficult also to extend our algorithms to work on graphs with small integer weights. The running time is multiplied by b, the largest edge weight.

An interesting open problem is whether similar results can be obtained for the decremental version of the *exact* APSP problem.

The techniques used to obtain our decremental algorithms can also be used to obtain *incremental* algorithms, with the same time bounds, for the approximate APSP problem.

Our fully-dynamic algorithm presents an interesting tradeoff between the amortized update time and the query time. Improving this tradeoff is an interesting open problem.

References

[1] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM Journal on Computing*, 28:1167–1181, 1999.

[2] G. Ausiello, G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *J. Algorithms*, 12(4):615–638, 1991.

[3] G. Ausiello, G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. On-line computation of minimal and maximal length paths. *Theoretical Computer Science*, 95(2):245–261, 1992.

[4] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28:263–277, 1999.

[5] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for transitive closure and all-pairs shortest paths. In *Proc. of 34th STOC*, pages 117–123, 2002.

[6] S. Baswana, R. Hariharan, and S. Sen. Maintaining all-pairs approximate shortest paths under deletion of edges. In *Proc. of 14th SODA*, pages 394–403, 2003.

[7] E. Cohen. Fast algorithms for constructing t-spanners and paths with stretch t. *SIAM Journal on Computing*, 28:210–236, 1999.

[8] E. Cohen and U. Zwick. All-pairs small-stretch paths. *Journal of Algorithms*, 38:335–353, 2001.

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

[10] C. Demetrescu and G.F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. In *Proc. of 15th SODA*, pages 362–371, 2004.

[11] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer Auf Der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23:738–761, 1994.

[12] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. *SIAM Journal on Computing*, 29:1740–1759, 2000.

[13] M. Elkin. Computing almost shortest paths. In *Proc. of 20th PODC*, pages 53–62, 2001.

[14] M. Elkin and D. Peleg. (1+epsilon, beta)-spanner constructions for general graphs. *SIAM J. Comput.*, 33(3):608–631, 2004.

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

[16] Z. Galil and O. Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54:243–254, 1997.

[17] M. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *Proc. of 36th FOCS*, pages 664–672, 1995.

[18] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. of 40th FOCS*, pages 81–91, 1999.

[19] R. Pagh and F.F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

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

[21] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. of 40th FOCS*, pages 605–614, 1999.

[22] M. Thorup. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *SWAT: Scandinavian Workshop on Algorithm Theory*, pages 384–396, 2004.

[23] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.

[24] J.D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20:100–125, 1991.

[25] U. Zwick. All-pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49:289–317, 2002.