Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles

Mikkel Thorup

AT&T Labs—Research, Shannon Laboratory, 180 Park Avenue
Florham Park, NJ 07932, USA.
mthorup@research.att.com

Abstract. We present a solution to the fully-dynamic all pairs shortest path problem for a directed graph with arbitrary weights allowing negative cycles. We support each vertex update in $O(n^2(\log n + \log^2(\bar{m}/n)))$ amortized time. Here, n is the number vertices, m the number of edges and $\bar{m} = n + m$. A vertex update inserts or deletes a vertex with all incident edges, and we update a complete distance matrix accordingly. The algorithm runs on a comparison-addition based pointer-machine.

1 Introduction

Recently Demetrescu and Italiano [1] presented an exciting new approach to the fully-dynamic all-pairs shortest path (APSP) problem with positive weights. Their algorithm supports each vertex update in $O(n^2 \log^3 n)$ amortized time[1]. Here n is the number of vertices. A vertex update inserts or deletes a vertex with all its incident edges. Between updates, a complete distance matrix is maintained. The algorithm also maintains the next hop on a shortest path from any vertex towards any destination. The algorithm runs on a comparison-addition based pointer-machine.

We refer the reader to [1] for the rich history of dynamic shortest path problems which has publications dating back to 1967. Here we just note that before [1], the best amortized update time for APSP in general graphs was $\tilde{O}(n^{2.5})$ with unit weights [6]. The new $O(n^2 \log^3 n)$ amortized update time from [1] is a substantial improvement and allows arbitrary non-negative weights.

1.1 An Even Faster Fully-Dynamic APSP Algorithm

In this paper, we present a different version of the algorithm from [1], maintaining the same type of information, but being easier to analyze and tune, getting tighter bounds, and thus providing a better understanding of the general new approach. Our amortized update time is $O(n^2(\log n + \log^2(\bar{m}/n)))$. We also reduce the space from $O(nm \log n)$ to $O(mn)$. Here m is the number of edges and $\bar{m} = m + n$.

[1] In the final remarks of [1], Demetrescu and Italiano state that their bounds can be improved to $O(n^2 \log^2 n)$ using a Fibonacci heap, but that claim is withdrawn in [2].

T. Hagerup and J. Katajainen (Eds.): SWAT 2004, LNCS 3111, pp. 384–396, 2004.

While the improvement is only by one or two log-factors, depending on the sparsity of the graph, it should be compared with a lower-bound of $\Omega(n^2)$ needed just to update the distance matrix. Thus, we cannot improve by more than log-factors. Also, our algorithm picks distances out of a priority queue, and since we may have $\Theta(n^2)$ distance changes per vertex update, any such algorithm needs $\Omega(n^2 \log n)$ comparisons. It is also interesting to compare our algorithm with the standard static competitor, running Dijkstra's [3] single source algorithm from each source with a Fibonacci heap [4] in $O(n^2 \log n + nm)$ total time. Our $O(n^2(\log n + \log^2(\bar{m}/n)))$ bound is never worse, and it is an improvement whenever $m = \omega(n \log n)$. We note that there is a faster comparison-addition based APSP algorithm [8] running in $O(n^2 \log \log n + nm)$ time, but that algorithm is not based on priority queues.

1.2 Allowing Negative Weights and Cycles

Our new version can be extended to deal with negative weights, allowing negative cycles. For these arbitrary weights, we get the same amortized update time of $O(n^2(\log n + \log^2(\bar{m}/n)))$. To the best of our knowledge, for arbitrary weights, this is the first fully-dynamic APSP algorithm with better amortized updates than a static recomputation from scratch. The extension is non-trivial, but, unfortunately, there is not room for it in this extended abstract.

1.3 Notation

The vertex set of a graph G is denoted $V(G)$ and the edge set is denoted $E(G)$. If $U \subseteq V(G)$, then $G \setminus U$ denotes the subgraph of G where we have removed the vertices from U with their incident edges. As a slight abuse of notation, if v is a single vertex, we define $G \setminus v = G \setminus \{v\}$. If v precedes w in a path P, then $P[v, w]$ denotes the subpath from v to w of P. Also, *first*(P) and *last*(P) denote the first and the last vertex in P. An s-t path is a path P with $s =$ *first*(P) and $t =$ *last*(P). If P and Q are paths with *last*$(P) =$ *first*(Q), then PQ denotes the concatenation of P and Q.

2 The Approach of Demetrescu and Italiano

In this section, we present the new approach of Demetrescu and Italiano to the dynamic APSP problem [2]. The presentation is, however, directed towards our own developments to be presented in the subsequent sections.

We say that two paths are *alternatives* if they start and finish in the same vertices. A path is a *shortest path* if there is no shorter alternative. *We assume that shortest paths are unique*, that is, for a shortest path all alternatives are longer. In [2] is presented an elegant way of achieving this uniqueness even in the deterministic case and without loss of efficiency.

2.1 Selecting Shortest Paths Via Generated Paths

The dynamic APSP algorithm operates on a set of *selected paths*. Generally, selected paths are paths that at some stage have been identified as shortest. After each vertex update, the algorithm will make sure to select all current shortest paths. It may also de-select some selected paths so as to make sure that not too many paths are selected.

Demetrescu and Italiano [2] presented a very interesting approach for selecting shortest paths. A path P is *generated* if we get a selected path no matter which end-point from P we remove. We say that a path P is *improving* if it is strictly shorter than any selected alternative.[2]

Trivial paths consisting of a single vertex form a special case, for if we remove that vertex, we have nothing left. We define *all trivial paths to be generated*.

Lemma 1. *(a) Let Q be a shortest path which is not selected. Then Q has an improving generated subpath R. In particular, if there are no improving generated paths, then all shortest paths are selected.*
(b) Let P have minimum length amongst all generated improving paths. Then P is a shortest path which is not yet selected.

Proof. To prove (a), let R be a minimal subpath of Q that is not selected. Then R is generated and shortest but not selected, so R is improving.

To prove (b), suppose for a contradiction that P is not shortest and consider a shortest alternative Q of P. Since P is improving, we know that Q is not selected. Hence by (a), we have an improving generated subpath R of Q. Now $\mathit{length}(R) \leq \mathit{length}(Q) < \mathit{length}(P)$ contradicting the choice of P. Thus we conclude that P is a shortest path. □

Lemma 1 provides us a *process to select all shortest paths*. As long as there are improving generated paths, we select such a path of minimal length. By Lemma 1, this process selects exactly the shortest paths which were not shortest when we started.

2.2 A Path System with Priority Queues

In order to implement the selection of all shortest paths, [2] presents a *path system* to maintain selected and generated paths. We refer to all paths in the system as *system paths*, noting that the same path may be both selected and generated.

The path system knows the graph, so when a vertex v is inserted, the trivial path (v) is generated immediately. Whenever a path is selected, the system combines it with previously selected paths in new generated paths. We can only select

[2] Our "selected paths" are the "zombies" in [1] and the "historically shortest paths" in [2]. Our "generated paths" are the "potentially uniform paths" in [1] and the "locally historical paths" in [2]. The concept of "improving generated paths" is important in [1,2] but was not named. Our terminology is shorter, and more convenient when we later want to talk about other types of generated paths.

a path if it is a generated path generated by the system. We can ask the system to destroy all system paths containing a given vertex. This happens automatically when a vertex is deleted from the graph. The path system is implemented below in §2.3 in constant time per system path change. The above operations maintain that all selected paths are generated. Recursively this implies that any subpath of a selected (generated) path is selected (generated).

For each start-finish vertex pair (s, t), the we have a *start-finish priority queue* $\mathcal{Q}_{(s,t)}$ with all system paths from s to t, the shorter with higher priority. If the shortest system path in $\mathcal{Q}_{(s,t)}$ is not selected, then P is an improving generated path which participates in a *global priority queue* $\mathcal{Q}_G$. Using classic comparison-based priority queues [11], each operation on a queue is supported n $O(\log n)$ time.

The selection of all shortest paths is now implemented as follows. As long as the global priority queue $\mathcal{Q}_G$ is non-empty, we select the shortest path from $\mathcal{Q}_G$. When $\mathcal{Q}_G$ is empty, for each s and t, the shortest s-t path is found in $\mathcal{Q}_{(s,t)}$.

2.3 Implementing the Path System

We now show how to implement the path system itself. Currently, all system paths are generated paths, and all subpaths of generated paths are generated paths. Every system path is given a unique identifier, from which we can derive information such as end-points, length, and first edge.

If P is non-trivial, we say that P is a *pre-extension* of $P \setminus \mathit{first}(P)$ and *post-extension* of $P \setminus \mathit{last}(P)$. If Q is generated, we store with Q the set of its generated pre-extension and the set of its generated post-extensions. Using these sets, we can identify and destroy all generated paths containing a given vertex in constant time per path.

Together with Q we also store the sets of selected pre- and post-extensions. When a new pre-extension P_1 of Q is selected, we take each currently selected post-extension P_2 of Q, and generate the new path $P_1 \cup P_2$. The case when a new post-extension is selected is symmetric. Thus each new path is generated in constant time.

In the above path system, we pay constant time per path change. This is dominated by the $O(\log n)$ time it takes to modify the start-finish priority queues and the global priority queue in §2.2.

2.4 A Basic Dynamic APSP Algorithm

Using the above path system, we have a basic algorithm for the dynamic APSP problem. If a vertex is inserted, we select all shortest paths. If a vertex is deleted, we first destroy all system paths containing it, and then select all shortest paths.

2.5 The Key to Efficiency

The following lemma is crucial to efficiency:

Lemma 2. *Suppose all selected paths containing v are shortest. If s and t are vertices different from v, there is at most one generated s-t path which contains v. Moreover, there are at most $O(n^2)$ generated paths containing v.*

Proof. Consider a generated s-t path P which contains v. Then $P[s, v]$ is contained in the selected path $P \setminus t$ which contains v. Hence $P[s, v]$ is the shortest s-v path. Symmetrically, $P[t, v]$ is the shortest v-t path, so P is unique. It immediately follows that v is internal to at most n^2 generated paths.

Now, consider a generated path P from v to some vertex t, and let t' be the predecessor of t in P. Then $P[s, t']$ is a selected path, so P is uniquely determined by t' and t. Symmetrically, there are at most n^2 choices of generated paths finishing in v. □

2.6 Efficiency of the Static Case

In the static case, in the process selecting all shortest paths, the path system will first generate all trivial paths. We will then continue to select and generate paths until we have selected the set of all shortest paths.

Lemma 3. *When all selected paths are shortest, the total number of generated paths is $O(n^3)$. In particular, it takes $O(n^3 \log n)$ time to select all shortest paths in the static case.*

Proof. Since we have n vertices, the first part is a direct corollary of Lemma 2. Also, there are at most n^2 shortest paths to select. We spend $O(\log n)$ time per system path, so the total running time is $O(n^3 \log n)$. □

2.7 Efficiency of the Incremental Case

Consider the *incremental* case of the simple algorithm, that is, no deletes are allowed. When a vertex is inserted, all new selected paths are shortest paths containing v. Then Lemma 2 implies that we create at most $O(n^2)$ system paths, and that takes $O(n^2 \log n)$ time. That is,

Lemma 4. *The basic algorithm supports an insert in $O(n^2 \log n)$ time.* □

2.8 Efficiency of the Decremental Case

Now consider the *decremental* case of the basic algorithm, that is, no inserts are allowed. Here, we first run the static algorithm as in §2.6. Each time a vertex is deleted, all system paths containing it are destroyed. Afterwards the basic algorithm selects all new shortest paths. The crucial observation is that selected paths remain shortest until destroyed.

Lemma 5. *Starting with a graph with n vertices, the basic algorithm can support up to n deletions in $O(n^3 \log n)$ total time.*

Proof. By Lemma 3, there can be at most $O(n^3)$ generated paths when the deletions are completed. All other generated paths are destroyed by deletions. By Lemma 2, each deletion destroys $O(n^2)$ generated paths. Consequently, the total number of path changes is $O(n^3)$, so the total running time is $O(n^3 \log n)$. □

2.9 The Fully-Dynamic Case

Unfortunately, the basic algorithm is not efficient in the fully-dynamic case. One can construct a sequence of n vertex inserts followed by n vertex deletes so that each delete makes $\Theta(n^3)$ path changes. The problem is that a shortest path selected after one insert may not be shortest after some future insert. With such non-shortest selected paths, the efficiency of deletions breaks down.

The approach of Demetrescu and Italiano [2] to the fully-dynamic APSP problem is that when a vertex v is inserted, for $I = 0, 1, ...$, we wait for 2^I updates, and then we make an extra dummy update on v. The *dummy update* deletes and inserts v with the same edges. The effect is to de-select paths through v that are no longer shortest. With this grooming, they generalize Lemma 2 to show that there can be at most $O(n^2 \log n)$ generated paths through any vertex. Consequently, no update or dummy update can destroy more than $O(n^2 \log n)$ paths. Since each real update gives rise to $O(\log n)$ dummy updates, they get $O(n^2 \log^2 n)$ path updates per real vertex update, hence an amortized update time of $O(n^2 \log^3 n)$.

3 Our Basic Algorithm

Our algorithm for the fully-dynamic APSP problem follows a general idea of Henzinger and King [5] reducing a fully-dynamic problem to a logarithmic number of decremental problems. Henzinger and King's idea was originally developed for the fully-dynamic minimum spanning tree problem. Here we use the idea in the context of the fully-dynamic APSP problem, essentially exploiting the efficiency of the simple decremental algorithm in §2.8. In our first implementation, we improve the APSP amortized update time to $O(n^2 \log^2 n)$. In the next section, we will tune our implementation for sparse graphs, getting the claimed amortized update time of $O(n^2(\log n + \log^2(\bar{m}/n))$. Finally, we will sketch how to deal with negative edge weights.

3.1 Dividing into Levels

Updates are numbered $t = 1, 2, 3...$ and the *birth date* of a vertex is the number of the update inserting it. The graph is rebuild whenever $t \geq 2n$ with n the current number of vertices. We then set $t = 1$ and rebuild the graph with n reinserts. Asymptotically, this does not affect our amortized time bounds.

We impose a standard type binary hierarchy over the update sequence. We say that *level* I is active after update t if bit i is set in t. Here bit 0 is the least significant bit. Also, t *activates the level* of its least significant set bit, that is, if L is the least significant set bit of t, then level L is inactive before t and active after t. Also, t *deactivates* the active levels lower than L.

If level I is active, we let t_I denote the update that activated level I. Note that if level $J > I$ is also active, then $t_J < t_I$. Hence, among active levels, we sometimes refer to active higher levels as *older levels*.

When we activate a level I, we construct a *level I graph* G_I as a copy of the current graph G. The vertices from G_I are called *level I vertices*. While level I is active, we do not add any vertices to G_I but if a level I vertex is deleted from G, it is also deleted from G_I. Thus G_I is a decremental dynamic graph. We destroy G_I when level I is deactivated. We will often identify an active level I with its decremental level graph G_I.

The vertices in G_I that were not in the previous level graph G_J are called *level I centers*. More formally, we have active levels I and J with $I < J$ and no active levels between I and J. Then the level I centers are the vertices inserted during updates $t \in (t_J, t_I]$. We let C_I denote the set of level I centers. Then $G_J = G_I \setminus C_I$. Clearly each vertex v is center in exactly one level I, and we say that v is *centered* in level I. Consider a path P. Let v be its youngest vertex, centered in some level I. Then G_I is the oldest level graph containing P. We say that P is *centered* in v, in level I, and in G_I. The basic goal of an active level I is to identify the shortest paths in G that are centered in level I.

3.2 The Level Path System

We will have a specialized level system of selected paths. Each selected path P may be *selected for any level I* with $P \subseteq G_I$. Also P may be *selected for the current graph G*. However, we have the requirement that *if P is selected for G, then P has to be selected for all levels I with $P \subseteq G_I$*. Although this is not part of the definition, in our algorithms, a path selected for a level I will always be shortest in G_I.

A path P is *generated by level I* if it satisfies the following two conditions:

- P is centered in level I.
- if P is not a trivial path, then, no matter which end-point we remove from P, we get a path selected for level I.

The first conditions ensures that P can only be generated by a single level. If we do not want to specify this level, we say that P is *level generated*.

The second condition implies that if P is level generated, it is also generated with the original definition from §2. However, the converse is not true, for an originally generated paths may not be generated by any level. Our restriction to levels is our key to efficiency, but before turning to efficiency, we prove that our level path system can be used to generate shortest paths.

We say a path P in the current graph G is *improving* if it is shorter than any alternative selected for G. This redefinition of improving does not take into account paths that are only selected for levels. Analog to Lemma1, we get

Lemma 6. *(a) Let Q be a shortest path in the current graph G which is not selected for G. Then Q has an improving level generated subpath R. In particular, if there are no improving level generated paths, then all shortest paths are selected for G.*

(b) Let P have minimum length amongst all level generated improving paths. Then P is a shortest path in G which is not yet selected for G.

Proof. To prove (a), let R be a minimal subpath of Q that is not selected for G. Then R is improving. Let I be the level that R is centered in. If R is trivial, it is generated by level I. Otherwise, removing either end-point from R, we get a path S which is selected for G, but then S is also selected for level I. Consequently, R is generated by level I. With (a) settled, the proof of (b) is identical to that of Lemma 1(b). □

3.2.1 Implementation. It is straightforward to modify the path system from §2.2–2.3 to deal with levels as described above. More precisely, we make an independent path system for each level based on paths selected for that level. However, in the level I path system, we have the restriction that two level I selected paths may only be combined in a generated path if at least one of them is centered in level I. An efficient implementation requires that level I selected post-extensions P_2 of a path Q are divided depending on whether P_2 is centered in level I. Consider a new level I selected pre-extension P_1 of Q. If P_1 is centered in level I, we generate $P_1 \cup P_2$ for all level I selected post-extensions P_2 of Q; otherwise, we only use those P_2 that are centered in level I.

The start-finish priority queue $\mathcal{Q}_{(s,t)}$ now has all s-t paths that are either level generated or selected for the current graph. If the shortest system path in $\mathcal{Q}_{(s,t)}$ is not selected for the current graph, then P participates in the global priority queue $\mathcal{Q}_G$.

3.3 Fully-Dynamic APSP with the Level Path System

Given the above level path system, we have a simple fully-dynamic APSP algorithm. The system starts with an empty graph, no level graphs, and an empty level path system.

To process an update t, our first action is to de-select all paths from the current graph. The update t activates some level L and deactivates all levels $K < L$. To execute the deactivation, we destroy all system paths through the level K centers, and de-select all other paths from level K. If the update t deletes a vertex v, we also destroy all system paths containing v.

Next we activate level L. If the update t inserts a vertex, it becomes a level L center along with the centers from the deactivated levels. Each trivial path consisting of a level L center is generated immediately by level L. Now, as long as the global priority queue $\mathcal{Q}_G$ is non-empty, we pick its shortest path P. Then we select P for the current graph G and for all levels I with $P \subseteq G_I$. By Lemma 6, the above process generates exactly the shortest paths in the current graph G.

3.4 Analysis

We note that we only select a path for an active level I when P is shortest in the current graph G. Trivially, this implies that P is shortest in the subgraph G_I, and since G_I is decremental, P remains shortest till G_I is deactivated. Thus, we have

Invariant 1 *All paths selected for level I are shortest paths in G_I.* □

With this invariant, Lemma 2 applies to all paths generated by level I. Consequently,

Lemma 7. *On a given level, we have only $O(n^2)$ system paths through any vertex.* □

Lemma 8. *The number of paths generated by level I is always $O(2^I n^2)$. Moreover, $O(2^I n^2)$ bounds the number of changes to paths generated by level I while I is in an active period, including the deactivation at the end.*

Proof. A path is only generated by level I if it goes through one of the at most 2^I centers on level I center. By Lemma 7, there can never be more than $O(n^2)$ generated paths through any such center, so we can have at most $O(2^I n^2)$ level I generated paths. By Lemma 7, we also get that each of the at most 2^I deletions can destroy at most $O(n^2)$ level I generated paths. As in the proof of Lemma 5, we conclude that $O(2^I n^2)$ bounds the total number of changes to paths generated by level I. □

We are now ready to analyze our total cost.

Lemma 9. *The above APSP algorithm supports each vertex update in $O(n^2 \log^2 n)$ amortized time.*

Proof. We analyze the cost as follows.

- At each update, we identify $O(n^2)$ shortest paths P. Each P is selected for the current graph with a priority queue cost of $O(\log n)$. Also, P is selected at constant cost for $O(\log n)$ levels. Thus, the total cost of selecting shortest paths is $O(n^2 \log n)$.
- A level I is active for 2^I vertex updates, and in this period, by Lemma 8, we have $O(2^I n^2)$ changes to paths generated by level I, that is $O(n^2)$ changes per update. Each such change costs $O(\log n)$ in the priority queues, so over the $O(\log n)$ levels I, we have a cost of $O(n^2 \log n)$ per vertex update.

Adding up the above items, we conclude that we spend $O(n^2 \log^2 n)$ amortized time on each vertex update. We note that a level I alternates between being active and inactive periods of 2^I updates, starting with an inactive period. Hence, in case we stop in the beginning of an active period, we can amortize the active work over the preceding 2^I inactive updates. □

3.5 Faster or Better Analysis?

We have now obtained an amortized bound that is a factor $\log n$ better than the one originally provided in [1] (c.f. §2.9). One may ask if this is just a better analysis or if the algorithm really has a better worst-case performance. We believe the latter for the following reason: Our division into levels can be viewed as very similar to the exponentially spaced dummy updates from [1]. When we activate

a level L, we destroy all selected paths through the centers on deactivated levels $K < L$, and this is an analog to dummy update. Thus, we end up with a similar set of selected paths. However, in [1], we can combine arbitrary selected paths in generated paths. Here we only combine paths selected for the same level. Thus it seems plausible that we save a factor of $\log n$ in the number of generated paths.

4 Tuning for Sparse Graphs

We are now going tune the above algorithm to be more efficient for sparse graphs, reducing the running time to $O(n^2(\log n + \log^2(\bar{m}/n)))$. From the preceding analysis of Lemma 9, we know that the only cost that exceeds $O(n^2 \log n)$ is that of the level generated paths. Below, we will first reduce the number of these paths by reducing the number of levels to $O(\log(\bar{m}/n))$. Second, we will reduce the size of most priority queues to $O(\bar{m}/n)$, thereby reducing their cost to $O(\log(\bar{m}/n))$. These two improvements will reduce the overall update cost to $O(n^2(\log n + \log^2(\bar{m}/n)))$. We note that this only improves over our previous $O(n^2 \log^2 n)$ bound if $\bar{m}/n = n^{o(1)}$. Hence we can assume that we are dealing with a sparse graph with $\bar{m}/n = n^{o(1)}$.

4.1 Fewer Levels

In order to benefit from sparseness of a graph, we are going to divide our updates into epochs of length $\Theta(\bar{m}/n)$. More precisely, when the epoch start, we set it to run for $q = \lceil \bar{m}/(2n) \rceil$ vertex updates. During this period, $\bar{m} = m + n$ and n cannot change by more than a factor 2, so we preserve $q = \Theta(\bar{m}/n)$. We also note that $\bar{m} \geq n$, so q is at least 1.

Before the first update t_B of an epoch, we copy the current graph G into a decremental *base graph* G_B. During the epoch, the base graph is treated like an oldest active level graph. However, all vertices in G_B are viewed as centers, so any path in G_B is viewed as centered in G_B. Since an epoch has only $\bar{m}/n$ updates, it will never activate more than $\log_2(\bar{m}/n)$ regular levels.

All our preceding analysis of efficiency relied on each level having no more centers than the number of updates while active. However, G_B may have $\Omega(n)$ centers, and an epoch only lasts for $\bar{m}/n$ updates. For our sparse graphs, $\bar{m}/n = n^{o(1)}$, so we need a different analysis for G_B.

Lemma 10. *The total number of paths generated by the base graph is at most* $n(m+1) \leq n\bar{m}$.

Proof. As in Invariant 1, all paths selected for the base graph are shortest. There are n trivial paths. Consider any non-trivial path P generated by the base. Let (u, v) be the first edge of P and w the last vertex. The segment $P[v, w]$ is selected for the base, hence the unique shortest path from v to w. Consequently there are at most nm non-trivial base generated paths. □

Lemma 11. *During an epoch, we have* $O(\bar{m}n)$ *paths generated by the base, and they cost* $O(\bar{m}n \log n)$ *time in the priority queues.*

Proof. By Lemma 10, when an epoch ends, we have at most $n\bar{m}$ remaining base generated paths to be destroyed. Also, applying Lemma 7 to the base, we get that each of the $O(\bar{m}/n)$ vertex deletions destroys $O(n^2)$ base generated paths. □

Lemma 12. *With the base graph, our fully-dynamic APSP algorithm supports each vertex update in $O(n^2(\log n)(\log(\bar{m}/n)))$ amortized time and in $O(\bar{m}n)$ space.*

Proof. In the analysis for Lemma 9, we showed that each update takes $O(n^2 \log n)$ amortized time on each level. Now that we have only $O(\log(\bar{m}/n))$ levels, this amortized update time is hence reduced to $O(n^2(\log n)(\log(\bar{m}/n))$. By Lemma 11, the work in the base during an epoch takes $O(\bar{m}n \log n)$ time. In case, we stop in the middle of an epoch, we amortize this work over the previous epoch which was completed with $\Theta(\bar{m}/n)$ updates. The first epoch has an empty base, hence no base work to amortize. Thus, in the base, the amortized work per update is $O(n^2 \log n)$. Here the first epoch is a single insert in an empty graph, and it pays for itself in constant time. Adding up, we support each update in $O(n^2(\log n)(\log(\bar{m}/n))$ time.

Apart from the level generated paths, the space used for each level or base graph is $O(n^2)$, adding up to $O(n^2 \log(\bar{m}/n)) = O(mn)$. By Lemma 10, there are $O(mn)$ paths generated by the base graph, and by Lemma 8, there are $\sum_{I=0}^{\lfloor \log_2(\bar{m}/n) \rfloor} O(2^I n^2) = O(\bar{m}n)$ paths generated by the non-base levels. Thus we conclude that the total space is $O(\bar{m}n)$. □

4.2 Reducing the Priority Queue Cost

As suggested in [1], it is straightforward to reduce the total cost of using the global priority queue $\mathcal{Q}_G$ to $O(n^2 \log n)$ per vertex update if we use a Fibonacci heap [4] supporting inserts and decreases in constant time. A minor change to the processing of an update is that we should wait entering paths in the global priority queue till after we have destroyed all the paths through deactivated centers and a deleted vertex.

Our challenge is to reduce the cost from the start-finish priority queues. The trick is to split each $\mathcal{Q}_{(s,t)}$ into a *small* priority queue $Q^{\text{small}}_{(s,t)}$ with many changes, and a *large* priority queue $Q^{\text{large}}_{(s,t)}$ with few changes. Then $\min Q_{(s,t)} = \min\{\min Q^{\text{small}}_{(s,t)}, \min Q^{\text{large}}_{(s,t)}\}$. The small priority queue contains any s-t path generated by a level I where neither s nor t are centers. It may also contain the unique shortest s-t path if it is selected for the current graph. The large priority queue contains all remaining level generated s-t paths. These are either from the base graph, or they are from a level graph G_I where s or t are centers.

Lemma 13. *A small priority queue $Q^{\text{small}}_{(s,t)}$ has at most $\bar{m}/n$ paths. Hence small priority queue changes take $O(\log(\bar{m}/n))$ time.*

Proof. Each path P in $Q^{\text{small}}_{(s,t)}$ is generated by a non-base level I where neither s nor t are centers. Hence P has a level I center v distinct from s and t. This means that $P[s,v]$ and $P[v,t]$ are selected for I. Hence, by Invariant 1, these segments are shortest paths in G_I. Thus P is uniquely determined by a center v from some non-base level I. An epoch creates only $\bar{m}/n$ such centers, so we conclude that the size of $Q^{\text{small}}_{(s,t)}$ is at most $\bar{m}/n$. □

Lemma 14. *The large priority queues have $O(n^2)$ changes per vertex update, hence a cost of $O(n^2 \log n)$.*

Proof. From Lemma 10, we know that we only have $O(n^2)$ changes to paths generated by the base. However, in the large local priority queue $Q^{\text{large}}_{(s,t)}$, we also have paths P generated by a level I where either s or t are centers. By symmetry, we may assume that s is a level I center. Let v be the second to last vertex in P. Then $P[s,v]$ is selected hence unique shortest in G_I, and (v,t) is an edge. Stepping back, we can characterize P by the vertex s, the graph G_I, and the edge (v,t). The vertex s is one of the $\bar{m}/n$ vertices inserted during the epoch, and during the epoch, it becomes center of $O(\log(\bar{m}/n))$ level graphs G_I. Consequently, the number of such paths is $O((\bar{m}/n)m \log(\bar{m}/n)) = O(\bar{m}^2/n \log(\bar{m}/n)) = O(n^{1+o(1)})$ since $\bar{m} = n^{1+o(1)}$. □

Theorem 2. *With the base graph and the split priority queues, we solve the fully-dynamic APSP problem supporting each vertex update in $O(n^2(\log n + \log^2(m/n)))$ amortized time and in $O(mn)$ space. Our algorithm runs on a comparison-addition based pointer machine.*

Proof. We consider the amortized cost of a vertex update. The space is already established in Lemma 12. From the analysis of Lemma 9, we know that it is only changes to level generated paths that can exceed $O(n^2 \log n)$. Their cost in the global priority queue is $O(n^2 \log n)$, and by Lemma 14, the cost of the large priority queues is $O(n^2 \log n)$. All other changes are to small priority queues. By Lemma 13, their priority queue cost is $O(\log(\bar{m}/n))$. From the analysis of Lemma 9, we know that there are $O(n^2)$ changes to paths generated by each of the $O(\log_2(\bar{m}/n))$ levels. Thus our total cost per vertex update is $O(n^2(\log n + \log^2(m/n)))$. □

We note that if the weights are integers, or floating point numbers in standard representation, we can use the priority queue from [9], reducing the delete time to $O(\log \log n)$ while keeping the insert and decrease-key time constant. This improves our amortized time bound to $O(n^2(\log \log n + \log(m/n) \log \log(m/n)))$ per vertex update.

5 Concluding Remarks

We have reduced the amortized update time for the fully-dynamic APSP problem to $O(n^2(\log n + \log^2(\bar{m}/n))$. However, we use $O(mn)$ space and we would like

to reduce this to $O(n^2)$ space. For many previous fully-dynamic APSP and transitive closure algorithms, the space was reduced from $O(mn)$ to $O(n^2)$ in [7]. In particular, this gave $O(n^2)$ space for the King's APSP algorithm [6] which has an update time of $\tilde{O}(n^{2.5})$ with unit weights. Unfortunately, the reduction breaks down for the current approach with generated paths.

Another interesting problem is to get a worst-case times for each individual update. All current approaches are lazy, and have worst-case update times as slow as a static algorithm. Using some of the ideas from this paper, the author has found better worst-case update times for the fully-dynamic APSP problem [10].

Finally, we mention the challenge of getting a fully-dynamic single source shortest path algorithm with sublinear query and update times. This is open even with amortized times bounds and in the case where we just want to maintain the shortest path from a fixed source s to a fixed destination t.

References

1. C. Demetrescu and G. Italiano. A new approach to dynamic all pairs shortest paths. In *Proc. 35th STOC*, pages 159–166, 2003.
2. C. Demetrescu and G. Italiano. A new approach to dynamic all pairs shortest paths, 2004. Full version of [1] available at `http://www.dis.uniroma1.it/~demetres/`.
3. E.W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959.
4. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
5. M. R. Henzinger and V. King. Maintaining minimum spanning forests in dynamic graphs. *SIAM J. Computing*, 31(2):364–374, 2001.
6. V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th FOCS*, pages 81–89, 1999.
7. V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proc. 7th COCOON*, LNCS 2108, pages 268–277, 2001.
8. S. Pettie. A faster all-pairs shortest path algorithm for real-weighted sparse graphs. In *Proc. 29th ICALP*, LNCS 2380, pages 85–97, 2002.
9. M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. In *Proc. 35th STOC*, pages 149–158, 2003.
10. M. Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths, 2004. Submitted.
11. J. W. J. Williams. Algorithm 232. *Comm. ACM*, 7(6):347–348, 1964.