Alcom-FT Technical Report Series
ALCOMFT-TR-02-92 (revised November 7, 2002)

A New Approach to Dynamic All Pairs Shortest Paths *

Camil Demetrescu † Giuseppe F. Italiano ‡

Abstract

We study novel combinatorial properties of graphs that allow us to devise a completely new approach to dynamic all pairs shortest paths problems. Our approach yields a fully dynamic algorithm for general directed graphs with non-negative real-valued edge weights that supports any sequence of operations in $\widetilde{O}(n^2)$[1] amortized time per update and unit worst-case time per distance query, where n is the number of vertices. We can also report shortest paths in optimal worst-case time. These bounds improve substantially over previous results and solve a long-standing open problem. Our algorithm is deterministic and uses simple data structures.

1 Introduction

In this paper we present fully dynamic algorithms for maintaining all pairs shortest paths (APSP) in directed graphs with *real-valued* edge weights. A dynamic graph algorithm maintains a given property $\mathcal{P}$ on a graph subject to dynamic changes, such as edge insertions, edge deletions and edge weight updates. Note that edge deletions and edge insertions can be easily formulated as edge weight updates, by setting to $+\infty$ the weight of edges not in the graph. A dynamic graph algorithm should process queries on property $\mathcal{P}$ quickly, and must perform update operations faster than recomputing the solution from scratch, as carried out by the fastest static algorithm. We say that an algorithm is *fully dynamic* if it can handle both edge weight increases and edge weight decreases. A *partially dynamic* algorithm can handle either edge weight increases or decreases, but not both.

The Problem. In the fully dynamic APSP problem we wish to maintain a directed graph with real-valued edge weights under an intermixed sequence of the following operations:

`update`(u, v, w): update the weight of edge (u, v) to the real value w (possibly $w = +\infty$);

*This work has been partially supported by the IST Programme of the EU under contract n. IST-1999-14186 (ALCOM-FT) and by the HPRN Programme of the EU under contract n. HPRN-CT-1999-00104 (AMORE).

†Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, Roma, Italy. Email: `demetres@dis.uniroma1.it`. URL: `http://www.dis.uniroma1.it/~demetres`.

‡Dipartimento di Informatica, Sistemi e Produzione, Università di Roma “Tor Vergata”, Roma, Italy and Centro “Vito Volterra”, Università di Roma “Tor Vergata”, Roma, Italy. Email: `italiano@disp.uniroma2.it`. URL: `http://www.info.uniroma2.it/~italiano`.

[1] Throughout the paper, we use $\widetilde{O}(f(n))$ to denote $O(f(n)\text{polylog}(n))$.

`distance`(x, y):	return the shortest distance from x to y.
`path`(x, y):	report a shortest path from x to y, if any.

Throughout the paper, we denote by n the number of vertices in the graph.

Previous work. The dynamic maintenance of shortest paths has a remarkably long history, as the first papers date back to 35 years ago [15, 16, 19]. After that, many dynamic shortest paths algorithms have been proposed (see e.g., [6, 9, 10, 17, 18, 20]), but their running times in the worst case were comparable to recomputing APSP from scratch.

The first dynamic shortest path algorithms which are provably faster than recomputing APSP from scratch, only worked on graphs with small integer weights. In particular, Ausiello *et al.* [1] proposed a decrease-only shortest path algorithm for directed graphs having positive integer weights less than C: the amortized running time of their algorithm is $O(Cn \log n)$ per edge insertion. Henzinger *et al.* [12] designed a fully dynamic algorithm for APSP on planar graphs with integer weights, with a running time of $O(n^{4/3} \log(nC))$ per operation. This bound has been improved by Fakcharoemphol and Rao in [7], who designed a fully dynamic algorithm for single-source shortest paths in planar directed graphs that supports both queries and edge weight updates in $O(n^{4/5} \log^{13/5} n)$ amortized time per edge operation.

The first big step on general graphs and integer weights was made by King [13], who presented a fully dynamic algorithm for maintaining all pairs shortest paths in directed graphs with positive integer weights less than C: the running time of her algorithm is $O(n^{2.5}\sqrt{C \log n})$ per update. In previous work [3, 4], we have solved fully dynamic APSP on general directed graphs with real weights. In particular, given a direct graph G, subject to dynamic operations, and such that each edge weight can assume at most S different *real* values, we gave a deterministic algorithm that supports each update in $O(n^{2.5}\sqrt{S \log^3 n})$ amortized time and each query in $O(1)$ worst-case time. Other deletions-only algorithms for APSP, in the simpler case of unweighted graphs, are presented in [2].

Our results. We study novel combinatorial properties of graphs that allow us to devise a completely new approach to dynamic all pairs shortest paths. This approach yields a fully dynamic algorithm for APSP on general directed graphs with non-negative real-valued edge weights that achieves the following time bounds: it supports any sequence of operations in $\widetilde{O}(n^2)$ amortized time per update and one look-up in the worst case per distance query; it can also report shortest paths in optimal worst-case time. We remark that our algorithm improves substantially over previous bounds [3, 4, 13]. Furthermore, and differently from all the previous approaches, it solves fully dynamic APSP in its generality. Indeed, it runs on non-negative real weights, and each weight has no limit on the number of different values it can assume. Despite few decades of research on dynamic shortest paths, we remark that this has been an elusive goal. Finally, we note that when the distance matrix has to be maintained explicitly, i.e., distance queries have to be answered with exactly one look-up, as many as $\Omega(n^2)$ entries of the distance matrix can change during each update. Thus, in this model our algorithm is only a polylogarithmic factor away from the best possible bound.

Another interesting feature of our techniques is that both weight increases and weight decreases can be supported with exactly the same code. Surprisingly, the algorithm is rather simple and thus amenable to efficient implementations.

Techniques. Differently from other approaches, which were more based on properties of long paths [2, 4, 11, 13], tree data structures [13] and dynamic reevaluation of products of real-valued matrices [3, 4], our method hinges on the notion of uniform paths: informally, we say that a path is *uniform* if all of its subpaths are shortest paths. Note that as a special case a shortest path is uniform, and thus the set of uniform paths contains all shortest paths. In this paper, we show that uniform paths play a strategic role in the dynamic maintenance of shortest paths. The main idea is to maintain uniform paths in a graph subject to a sequence of updates. We prove that changes in uniform paths can be combinatorially bounded in dynamic graphs, and we show how to update the all-pairs shortest paths efficiently with a simple dynamic modification of Dijkstra's algorithm [5] based on uniform paths.

Preliminaries. Let $G = (V, E, w)$ be a directed graph with non-negative real edge weights. We denote by w_{uv} the weight of edge $(u, v) \in E$, by π_{xy} a path from x to y, and by $w(\pi_{xy})$ the sum of the weights of edges in π_{xy} (weight of π_{xy}). Moreover, we denote by $\ell(\pi_{xy})$ the path π_{xb} such that $\pi_{xy} = \pi_{xb} \cdot (b, y)$, i.e., π_{xy} is the concatenation of path π_{xb} and edge (b, y). Similarly, we denote by $r(\pi_{xy})$ the path π_{ay} such that $\pi_{xy} = (x, a) \cdot \pi_{ay}$.

Let the graph G be subject to a sequence of updates $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$. We denote by t_{σ_i} the time at which update σ_i occurs, with $t_{\sigma_{i+1}} = t_{\sigma_i} + 1$, $1 \leq i \leq k-1$. We also denote by e_σ the edge e affected by update σ: i.e., if $\sigma = \texttt{update}(u, v, w)$, then $e_\sigma = (u, v)$.

Throughout the paper we assume that there is only one shortest path between any pair of vertices (x, y) in G. This is without loss of generality, as ties can be broken consistently. For lack of space, some details are omitted from this extended abstract.

2 Uniform Paths

In this section we introduce the notion of uniform path in a weighted graph G, which is the backbone of our approach to dynamic all pairs shortest paths. We discuss some properties of uniform paths and then use them to prove combinatorial bounds about graphs subject to sequences of updates.

Definition 1 *A path π_{xy} is* UNIFORM *in G if each subpath of π_{xy} is a shortest path in G.*

By the optimal-substructure property of shortest paths, we can check the uniformity of a path π_{xy} by just verifying that $\ell(\pi_{xy})$ and $r(\pi_{xy})$ are shortest paths in G. As a base case, any path formed by a single edge is uniform.

Definition 2 *A path π_{xy} is a* ZOMBIE *in G at time t if it is not a shortest path, but it used to be a shortest path at a time $t' < t$, and none of its edges have been updated in the time interval $(t', t]$.*

Note that each subpath of a zombie is either a zombie or a shortest path.

Definition 3 *A path π_{xy} is* POTENTIALLY UNIFORM *in G at time t if each subpath of π_{xy} is either a shortest path or a zombie at that time.*

Similarly to uniform paths, we can check the potential uniformity of a path π_{xy} by just verifying that $\ell(\pi_{xy})$ and $r(\pi_{xy})$ are either shortest paths or zombies in G.

The next Lemma follows from Definition 1, Definition 2, and Definition 3, and addresses the relationship between shortest paths, uniform paths, zombies, and potentially uniform paths in a graph subject to a sequence of updates.

Lemma 1 *If we denote by SP, UP, Z, and PUP respectively the sets of shortest paths, uniform paths, zombies, and potentially uniform paths in G, then at any time the following inclusions hold: $SP \subseteq UP \subseteq PUP$ and $Z \subseteq PUP$.*

We now show that the number of paths that become potentially uniform at each time during a sequence of updates depends on the number of zombies in the graph.

Theorem 1 *Let G be a graph subject to a sequence Σ of update operations. If at any time throughout the sequence Σ there are at most $O(z)$ zombies between each pair of vertices, then the amortized number of paths that become potentially uniform at each update is $O(zn^2)$.*

Proof (Sketch). For each pair of vertices (x, y) in G, define the *graph of potentially uniform paths H_{xy}* as follows. There is a vertex in H_{xy} for each update operation in Σ, and there is an edge (σ_a, σ_b) in H_{xy} for each path π_{xy} in G having the following three properties:

(i) π_{xy} contains both e_{σ_a} and e_{σ_b};

(ii) No edge in π_{xy} is updated in the time interval $(t_{\sigma_a}, t_{\sigma_b})$;

(iii) π_{xy} becomes potentially uniform at some time $t \in [t_{\sigma_a}, t_{\sigma_b})$.

It is easy to see that at any time at most one path π_{xy} can satisfy Properties (i), (ii) and (iii) above, and that (σ_a, σ_b) in H_{xy} implies $t_{\sigma_a} < t_{\sigma_b}$.

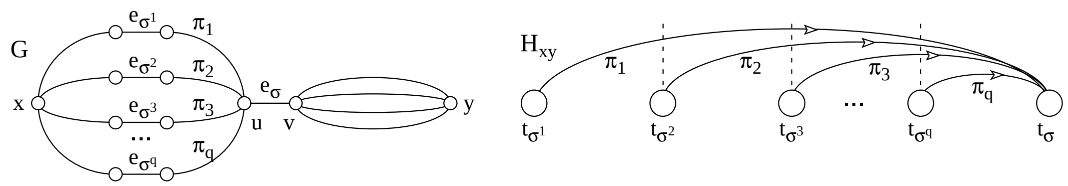

Figure 1: Graph G and corresponding graph of potentially uniform paths H_{xy} for a given sequence of updates. The in-degree of vertex σ in H_{xy} depends on the number of zombies between x and u (or between v and y) in G.

Assume that at any time there are at most $O(z)$ zombies between each pair of vertices in G. We claim that the in-degree of each vertex in H_{xy} must be $O(z)$. Indeed, assume by contradiction that there exists a vertex σ in H_{xy} whose in-degree is $\omega(z)$. By definition of H_{xy}, each edge entering vertex σ of H_{xy} corresponds to a path in G containing edge $e_\sigma = (u, v)$. Denote by P the set of such paths. Let $Q = \{\pi_1, \ldots, \pi_q\}$ be the subset of

P such that path π_i corresponds to edge (σ^i, σ) in H_{xy}, $t_{\sigma^1} < t_{\sigma^2} < \ldots < t_{\sigma^q} < t_\sigma$, and for each $\pi_i \in Q$, e_{σ^i} precedes e_σ in π_i (see Figure 1). Notice that, as a consequence of Property (ii), it must be $e_{\sigma^1} \neq e_{\sigma^2} \neq \ldots \neq e_{\sigma^q}$.

Assume without loss of generality that $q = |Q| \geq |P|/2$, and thus $q = \omega(z)$. By Definition 3, for $1 \leq i \leq q$, subpath $\langle x, \ldots, u\rangle \subseteq \pi_i$ must be either a shortest path or a zombie at some time T_i in $[t_{\sigma^i}, t_\sigma)$. No edge of π_i is touched in (t_{σ^i}, t_σ): consequently, $\langle x, \ldots, u\rangle \subseteq \pi_i$ will still be either a shortest path or a zombie in $[T_i, t_\sigma)$. Since subpaths $\langle x, \ldots, u\rangle \subseteq \pi_i$ cannot be all shortest paths simultaneously, this implies that at time $t_\sigma - 1$ there must be at least $q - 1 = \omega(z)$ zombies between x and u in G, clearly a contradiction.

To prove that the amortized number of paths that become potentially uniform at each update is $O(zn^2)$, we observe that a path π_{xy} corresponding to an edge $(\sigma_a, \sigma_b) \in H_{xy}$ will become potentially uniform only once at a time $t \in [t_{\sigma_a}, t_{\sigma_b})$, and will keep on being potentially uniform in $[t, t_{\sigma_b})$. This is a consequence of the fact that, once $\ell(\pi_{xy})$ and $r(\pi_{xy})$ have been both either shortest paths or zombies (and this happens at time t), they will be either shortest paths or zombies until they are further touched. If we charge the appearance of π_{xy} at time t to the update operation σ_b, since the in-degree of σ_b in H_{xy} is $O(z)$, then σ_b is charged at most $O(z)$ times. Thus, the amortized number of paths between any pair x and y that become potentially uniform at each update in G is $O(z)$, which yields an overall bound of $O(zn^2)$. $\square$

It is not difficult to construct a sequence Σ that produces a graph with as many as $\Omega(n^3)$ zombies. We now show how to keep the number of zombies under control. We define a Dummy Update to be an operation that overwrites the weight of an edge without changing its value.

Definition 4 *Let $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_k\rangle$ be an update sequence and let*

$$F(\Sigma) = \langle \sigma_1, \delta_1^1, \ldots, \delta_1^{q_1}, \sigma_2, \delta_2^1, \ldots, \delta_2^{q_2}, \ldots, \sigma_k, \delta_k^1, \ldots, \delta_k^{q_k}\rangle$$

be another sequence derived from Σ such that each δ_i^j is a dummy update, and such that, if an edge is updated by operation σ_t in Σ, it is touched by dummy update δ_i^j in $F(\Sigma)$ when $i = t+1,\ t+2,\ t+4,\ t+8,\ \ldots,\ t+2^p,\ \ldots,$ for some j. We call $F(\Sigma)$ a Smoothed Sequence *derived from Σ, we refer to $\langle \delta_i^1, \ldots, \delta_i^{q_i}\rangle$ as the* Burst of dummy updates after *σ_i, and we say that dummy update $\delta_{t+2^p}^j$ is* Triggered by the original update *σ_t.*

Note that a smoothed sequence is obtained by squeezing a burst of dummy updates between each pair of consecutive updates in the original sequence. Throughout the paper, we assume that the burst of dummy updates $\langle \delta_i^1, \ldots, \delta_i^{q_i}\rangle$ occurs exactly at the same time as update σ_i, i.e., $t_{\delta_i^j} = t_{\sigma_i}$ for any j. We now show that a smoothed sequence is operationally equivalent to the sequence it is derived from, and produces a small number of zombies.

Theorem 2 *Let $\Sigma = \langle \sigma_1, \sigma_2, \ldots, \sigma_k\rangle$ be an update sequence. Then the smoothed sequence $F(\Sigma) = \langle \sigma_1, \delta_1^1, \ldots, \delta_1^{q_1}, \sigma_2, \delta_2^1, \ldots, \delta_2^{q_2}, \ldots, \sigma_k, \delta_k^1, \ldots, \delta_k^{q_k}\rangle$ has the following properties:*

1. *For $1 \leq i \leq k$, $q_i \leq O(\log k)$;*
2. *For $1 \leq i \leq k$, after performing operation σ_i, the graph produced by Σ and the graph produced by $F(\Sigma)$ are equal;*

3. At each time there are at most $O(\log k)$ zombies between each pair of vertices in the graph produced by $F(\Sigma)$.

Proof (Sketch). Properties 1 and 2 follow directly from Definition 4. To prove Property 3, we first note that a dummy update on an edge (x, y) does not create new shortest paths and does not introduce new zombies: its only effect is to remove zombies containing edge (x, y). Similarly to the proof of Theorem 1, for each pair of vertices (x, y) in G, consider the *graph of shortest paths* H^*_{xy}. There is a vertex in H^*_{xy} for each update operation τ in $F(\Sigma)$ and there is an edge (τ_a, τ_b) in H^*_{xy} for each path π_{xy} in G having the following three properties:

(i) π_{xy} contains both e_{τ_a} and e_{τ_b};

(ii) no edge in π_{xy} is updated in the time interval (t_{τ_a}, t_{τ_b});

(iii) π_{xy} becomes a shortest path at some time $t \in [t_{\tau_a}, t_{\tau_b})$.

It is easy to see that at any time at most one path π_{xy} in G can satisfy Properties (i), (ii) and (iii) above, and that (τ_a, τ_b) in H^*_{xy} implies $t_{\tau_a} < t_{\tau_b}$.

Let $\pi^1, \ldots, \pi^z$ be the zombies between vertices x and y in the graph G obtained right after performing update σ_t in $F(\Sigma)$, i.e., at time t. Let $(\tau^i, \widehat{\tau}^i)$ be the edge in H^*_{xy} corresponding to π^i, $1 \le i \le z$. Since π^i is a zombie at time t, it must be $t_{\tau^i} \le t < t_{\widehat{\tau}^i}$. If τ^i is a dummy update, let σ^i be the original update that triggered τ^i, $t_{\sigma^i} < t_{\tau^i}$. Thus, to each zombie π^i we can associate an original update σ^i, with $t_{\sigma^1} < t_{\sigma^2} < \ldots < t_{\sigma^z} \le t$. Let σ^i and σ^{i+1} be two such updates, and let T be the largest power of 2 smaller than $t - t_{\sigma^i}$, i.e., $T = 2^{\lfloor \log_2 (t - t_{\sigma^i}) \rfloor}$. We claim that it must be $t_{\sigma^i} + T \le t_{\sigma^{i+1}}$.

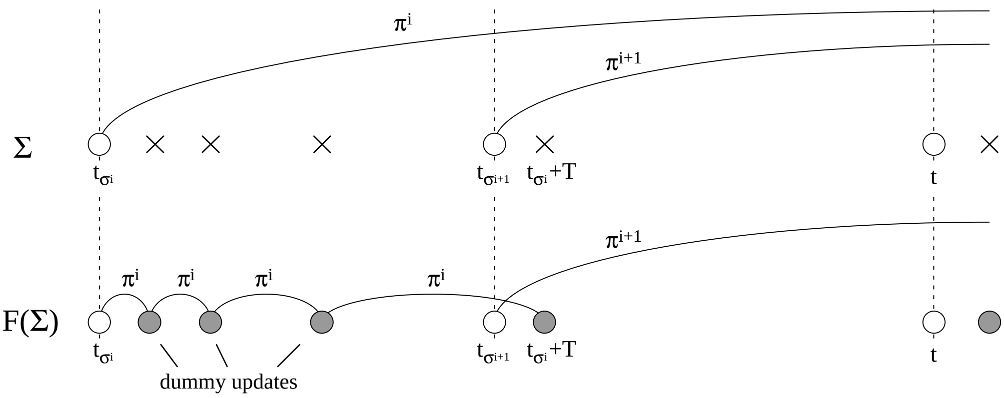

Figure 2: Graphs of shortest paths H^*_{xy} for an update sequence Σ and for the smoothed updated sequence $F(\Sigma)$. If $t_{\sigma^{i+1}} < t_{\sigma^i} + T$, then π^i cannot be a zombie at time t in the graph produced by $F(\Sigma)$.

Indeed, assume by contradiction that $t_{\sigma^{i+1}} < t_{\sigma^i} + T$ (see Figure 2). Since $T = 2^{\lfloor \log_2 (t - t_{\sigma^i}) \rfloor}$ is a power of 2, by Definition 4 there is a dummy update triggered by σ^i that touches an edge of π^i at time $t_{\sigma^i} + T$: thus π^i cannot be a zombie at time $t_{\sigma^i} + T$ by Definition 2. Notice that, since π^{i+1} must be a shortest path between x and y after π^i, and no edge of π^i and π^{i+1} is updated in the time interval $[t_{\sigma^{i+1}}, t]$, π^i cannot become again a shortest path between x and y in the time interval $[t_{\sigma^{i+1}}, t]$. Thus π^i cannot become a zombie again in the time interval $[t_{\sigma^i} + T, t]$, contradicting the assumption that π^i was a zombie at time t.

We have thus shown that if $\pi^1, \ldots, \pi^z$ are zombies at time t, there must be z updates $\sigma^1, \ldots, \sigma^z$ in the original sequence Σ such that $t_{\sigma^i} + 2^{\lfloor \log_2 (t - t_{\sigma^i}) \rfloor} \leq t_{\sigma^{i+1}} \leq t$, for $1 \leq i \leq z-1$. This implies $z = O(\log t) = O(\log k)$. $\square$

We note that $\Theta(\log k)$ zombies between a pair of vertices in G can be actually realized, as illustrated in Figure 3.

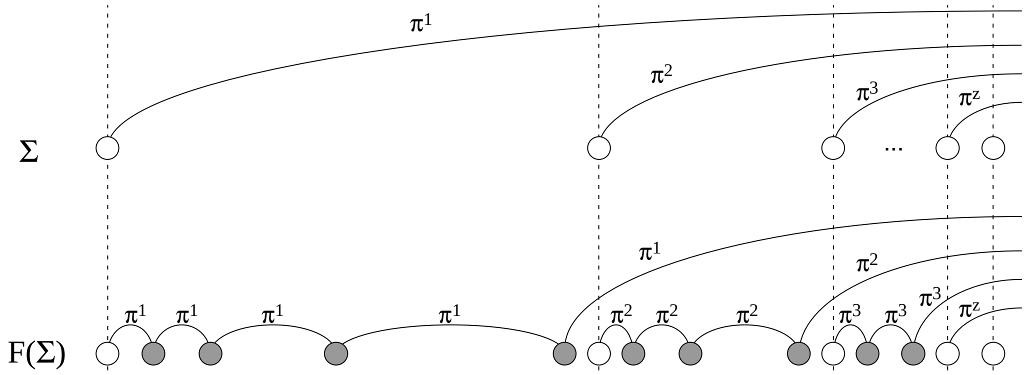

Figure 3: Worst case instance of Σ such that at time k there are $\Theta(\log k)$ zombies between pair x, y in the graph produced by $F(\Sigma)$.

3 Dynamic All Pairs Shortest Paths

In this section we show how to exploit the combinatorial properties of uniform paths presented in Section 2 to design a fully dynamic algorithm for maintaining all pairs shortest paths in a weighted directed graph in $\widetilde{O}(n^2)$ amortized time per update. The main idea is to maintain dynamically the potentially uniform paths of the graph in a data structure. Since by Lemma 1 shortest paths are potentially uniform, this guarantees that we are keeping information about shortest paths as well. To support an `update`(u, v, w) operation, we first remove from our data structure all paths that contain the updated edge, and then we use an all-pairs modification of Dijkstra's algorithm [5] where at each step a shortest path is extracted from a priority queue and it is combined with existing shortest paths and zombies to form new potentially uniform paths.

Data Structure. For each pair of vertices x and y in G we maintain the weight $w_{xy} \geq 0$ of edge (x, y) (or $+\infty$ if no such edge exists), the time t_{xy} of the latest update of edge (x, y), and the following two data structures:

- $P_{xy} = \{\pi_{xy} : \pi_{xy}$ is potentially uniform in G $\}$, maintained as a priority queue where each item $\pi_{xy} \in P_{xy}$ has priority $w(\pi_{xy})$;
- $P^*_{xy} = \{\pi_{xy} : \pi_{xy}$ is a shortest path or a zombie in G $\}$, maintained as a set;

We note that each path π_{xy} in P_{xy} and P^*_{xy} can be represented implicitly using constant space by storing pointers to the subpaths $\ell(\pi_{xy})$ and $r(\pi_{xy})$. Furthermore, for each path π_{xy} stored in P_{xy} we maintain $w(\pi_{xy})$ and the following four lists:

- $L(\pi_{xy}) = \{\pi_{x'y} = (x', x) \cdot \pi_{xy} :\ \pi_{x'y}$ is potentially uniform in G };
- $L^*(\pi_{xy}) = \{\pi_{x'y} = (x', x) \cdot \pi_{xy} :\ \pi_{x'y}$ is a shortest path or a zombie in G };
- $R(\pi_{xy}) = \{\pi_{xy'} = \pi_{xy} \cdot (y, y') :\ \pi_{xy'}$ is potentially uniform in G };
- $R^*(\pi_{xy}) = \{\pi_{xy'} = \pi_{xy} \cdot (y, y') :\ \pi_{xy'}$ is a shortest path or a zombie in G }.

Finally, we keep a counter *time* of the number of original `update` operations performed.

Implementation. Each `distance`(x, y) or `path`(x, y) query can be answered by simply accessing the minimum weight path in P_{xy}. An `update`(u, v, w) can be implemented as follows:

`update`(u, v, w):
    $time \leftarrow time + 1$
    $t_{uv} \leftarrow time$
    `unsmoothed-update`(u, v, w) {original update σ_i}
    for each $(x, y) \in E :\ \ time - t_{xy} = 2^{\lceil \log_2(time - t_{xy}) \rceil}$ **do**   {smoothing}
        `unsmoothed-update`(x, y, w_{xy}) {dummy update δ_i^j}

The `update` procedure realizes the smoothed update sequence described in Definition 4, and calls `unsmoothed-update` to update the data structure for both original and dummy updates. The `unsmoothed-update` procedure is the heart of the update algorithm:

`unsmoothed-update`(u, v, w):
    `delete`(u, v)
    `insert`(u, v, w)

It works in two phases: it first removes the updated edge (u, v) by calling `delete`(u, v) and then reinserts it with the new weight w by calling `insert`(u, v, w). Procedures `delete` and `insert` can be implemented as follows:

- `delete`(u, v): removes any path π_{xy} containing edge (u, v) from P_{xy}, P^*_{xy}, $L(r(\pi_{xy}))$, $L^*(r(\pi_{xy}))$, $R(\ell(\pi_{xy}))$, and $R^*(\ell(\pi_{xy}))$. This can be done iteratively by first removing path $\pi_{uv} = (u, v)$ and then by removing all the paths listed in $L(\pi_{xy})$ and $R(\pi_{xy})$ for each path π_{xy} removed in the previous iterations.
- `insert`(u, v, w): works in three phases as follows.

  Phase 1: Sets the weight of edge (u, v) to the new value w. If $w < +\infty$, then path $\pi_{uv} = (u, v)$ is added to P_{uv}, $L(r(\pi_{uv}))$, and $R(\ell(\pi_{uv}))$;

  Phase 2: Bulk loads a priority queue H with the minimum weight path $\pi_{xy} \in P_{xy}$ for each pair of vertices (x, y);

  Phase 3: Repeatedly extracts paths π_{xy} from H in increasing weight order. The first extracted path for each pair (x, y) is a shortest path between x and y, while paths extracted later on for the same pair are ignored. Whenever a shortest path π_{xy} is extracted, we check whether π_{xy} appears already in P^*_{xy}, i.e., it was already a shortest path or a zombie before the update. If not, we add π_{xy} to P^*_{xy}, $L^*(r(\pi_{xy}))$, and $R^*(\ell(\pi_{xy}))$, and we combine it with existing shortest paths and zombies to form new potentially uniform paths. This can be done by scanning all paths $\pi_{x'b}$ listed in $L^*(\ell(\pi_{xy}))$ and all paths $\pi_{ay'}$ listed in $R^*(r(\pi_{xy}))$, forming the new potentially uniform paths $\pi_{x'y} = (x', x) \cdot \pi_{xy}$ and $\pi_{xy'} = \pi_{xy} \cdot (y, y')$. Each new potentially uniform path π_{ij} is added to P_{ij}, $L(r(\pi_{ij}))$, $R(\ell(\pi_{ij}))$, and H as soon as it is discovered.

Analysis. Since P_{xy} is a subset of the paths in G, the correctness of query operations follows directly from Lemma 1. To prove the correctness of our implementation of `update`, we just need to prove that P_{xy} is maintained correctly for each pair (x, y) in G. Operations on the other data structures are simple bookkeeping operations and can be easily handled. Since `update` hinges on `unsmoothed-update`, it is sufficient to prove the correctness of this procedure. We first discuss an invariant maintained by algorithm `insert`:

Invariant 1 *For each pair of vertices x and y in G, the first path connecting x and y extracted from H in Phase 3 of* `insert` *is a shortest path.*

Proof (Sketch). At the beginning of Phase 3 of `insert`, H contains a shortest path π_{uv} and, for each pair of vertices $(x, y) \neq (u, v)$, a minimum weight potentially uniform path π_{xy} that does not contain the updated edge (u, v). We prove that the first path extracted from H for each pair (x, y) in Phase 3 is a shortest path between x and y. Suppose by contradiction that at some extraction this invariant is violated, and a non-shortest path $\widehat{\pi}_{xy}$ is extracted. Consider the earliest of these events, and let (x, y) be the involved pair. Let π_{xy} be any shortest path between x and y, with $w(\pi_{xy}) < w(\widehat{\pi}_{xy})$. Since $\pi_{xy} \notin H$ at the time of the wrong extraction, then π_{xy} has to be necessarily a new potentially uniform path, otherwise it would have been already inserted in H in Phase 2. Moreover, since (u, v) is added to P_{uv} in Phase 1 of `insert`, π_{xy} cannot coincide with edge (u, v), otherwise it would have been already inserted in H in Phase 2. It follows that either $\ell(\pi_{xy})$ or $r(\pi_{xy})$, which are shortest paths after the update by the suboptimality of π_{xy}, were neither shortest paths nor zombies before the update. Since edge weights are non-negative, then $w(\ell(\pi_{xy})) \leq w(\pi_{xy}) < w(\widehat{\pi}_{xy})$ and $w(r(\pi_{xy})) \leq w(\pi_{xy}) < w(\widehat{\pi}_{xy})$. Since paths are extracted from H in increasing weight order and all the extractions before the wrong extraction are correct, then both $\ell(\pi_{xy})$ and $r(\pi_{xy})$ should have been extracted from H before the wrong extraction. This implies that π_{xy} should have been formed and inserted in H at the extraction time of either one of $\ell(\pi_{xy})$ or $r(\pi_{xy})$. This contradicts the assumption that the invariant is violated. $\square$

Theorem 3 *Algorithm* `unsmoothed-update` *correctly updates P_{xy} for each pair of vertices x and y.*

Proof (Sketch). We first observe that `delete`(u, v) removes from the data structures any potentially uniform paths that contain the updated edge (u, v), and thus any path π_{xy} that is no longer potentially uniform after the update is removed from P_{xy}.

To complete the proof, we show that any path π_{xy} that becomes potentially uniform after the update is added to P_{xy} by `insert`(u, v, w). We recall that a path π_{xy} becomes potentially uniform if both $\ell(\pi_{xy})$ and $r(\pi_{xy})$ are shortest paths or zombies after the update, but at least one of them (say $\ell(\pi_{xy})$) was neither a shortest path nor a zombie before the update. By Invariant 1, the first extracted path for each pair (x, y) in Phase 3 is a shortest path between x and y, and then $\ell(\pi_{xy})$ is certainly extracted at some iteration. A that time, $\ell(\pi_{xy})$ is correctly combined in Phase 3 with $r(\pi_{xy}) \in R^*(r(\ell(\pi_{xy})))$ to form π_{xy}, which is added to P_{xy}. $\square$

Algorithm `unsmoothed-update` maintains correctly the potentially uniform paths, and thus the shortest paths. The reason we are using `update` instead of using directly `unsmoothed-update` is that a bad (unsmoothed) sequence of updates might produce a graph with many zombies, which may yield many new potentially uniform paths per update. To conclude the analysis, we address the running time of our implementation.

Theorem 4 *Our data structure supports each* `update` *operation in* $O(n^2 \log^3 n)$ *amortized time, and each* `distance` *and* `path` *query in optimal time.*

Proof (Sketch). The bounds for queries are immediate, since the minimum weight path in P_{xy} can be retrieved in optimal time.

To bound the running time of each `update`, consider first the time required by an `unsmoothed-update` operation. We can charge the cost of `delete` to previous `insert` operations that created the paths removed by this procedure. Next, Phase 1 of `insert` involves constant time operations plus an insertion into a priority queue P_{uv}, and thus can be implemented in $O(\log n)$ time overall. To bound the time required by Phase 2 of `insert`, we consider that finding the minimum in P_{xy} takes constant time and each insertion in H takes $O(\log n)$ time. This gives a total bound of $O(n^2 \log n)$ for this phase. Finally, we note that a path π_{xy} is a new potentially uniform path if and only if both $\ell(\pi_{xy}) = \langle x, \ldots, b\rangle \in P^*_{xb}$ and $r(\pi_{xy}) = \langle a, \ldots, y\rangle \in P^*_{ay}$ *after* the update, and either $\ell(\pi_{xy}) \notin P^*_{xb}$ or $r(\pi_{xy}) \notin P^*_{ay}$ *before* the update. Thus, since in Phase 3 of `insert` we process a shortest path π_{xy} extracted from H only if it was not already in P^*_{xy} before the update, then we spend time only for the new potentially uniform paths. This implies that Phase 3 requires $O(n^2 \log n)$ time for extracting the $O(n^2)$ pairs which are initially in H, plus $O(p \log n)$ time to insert and extract paths into/from H, where p is the number of paths that become potentially uniform after the update. Since `update` implements the smoothed sequence of Definition 4, by Theorem 2 at any time in a sequence of k updates there are at most $O(\log k)$ zombies between each pair of vertices in G. By Theorem 1, $p = O(n^2 z) = O(n^2 \log k)$ amortized. Thus, each call to `unsmoothed-update` requires $O(\log n + 2n^2 \log n + n^2 \log n \log k) = O(n^2 \log n \log k)$ amortized time.

Since each `update` causes $O(\log k)$ `unsmoothed-update` operations, then the amortized running time of `update` is $O(n^2 \log n \log^2 k) = O(n^2 \log^3 n)$ on any sequence of length k polynomial in n. To get rid of this assumption, we can simply restart our data structure from scratch every $\Theta(n^2)$ operations. $\square$

As a final note, we observe that the update time can be reduced to $O(n^2 \log^2 n)$: we maintain H with Fibonacci Heaps [8], which support `insert-key` and `decrease-key` operations in constant amortized time, and we insert a new potentially uniform path π_{xy} in H only if it is shorter than any previous path $\widehat{\pi}_{xy}$ between x and y already in H. In this case, $\widehat{\pi}_{xy}$ is replaced by π_{xy} with a `decrease-key` operation on H. If no path between x and y is already in H, then π_{xy} is inserted in H with an `insert-key` operation. Doing so, procedure `insert` performs at most $O(n^2)$ extractions of minima from H, which require $O(\log n)$ time each, and an amortized number of at most $O(p) = O(n^2 \log n)$ `insert-key` or `decrease-key` operations on H for inserting the p new potentially uniform paths, which take $O(1)$ amortized time each. This implies an $O(n^2 \log^2 n)$ overall amortized time for the $O(\log k) = O(\log n)$ `unsmoothed-update` operation performed at each original `update` operation.

References

[1] G. Ausiello, G.F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12(4):615–38, 1991.

[2] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for transitive closure and all-pairs shortest paths. In *Proc. 34th ACM Symposium on Theory of Computing (STOC'02)*, pages 117–123, 2002.

[3] C. Demetrescu and G.F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *Proc. of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS'01), Las Vegas, Nevada*, pages 260–267, 2001.

[4] C. Demetrescu and G.F. Italiano. Improved bounds and new trade-offs for dynamic all pairs shortest paths. In *Proc. of the 29-th International Colloquium on Automata, Languages, and Programming (ICALP'02), Malaga, Spain*, 2002.

[5] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] S. Even and H. Gazit. Updating distances in dynamic graphs. *Methods of Operations Research*, 49:371–387, 1985.

[7] J. Fakcharoemphol and S. Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. In *Proc. of the 42nd IEEE Annual Symposium on Foundations of Computer Science (FOCS'01), Las Vegas, Nevada*, pages 232–241, 2001.

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

[9] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Semi-dynamic algorithms for maintaining single source shortest paths trees. *Algorithmica*, 22(3):250–274, 1998.

[10] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34:351–381, 2000.

[11] D. H. Greene and D.E. Knuth. *Mathematics for the analysis of algorithms.* Birkhäuser, 1982.

[12] M.R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, August 1997.

[13] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS'99)*, pages 81–99, 1999.

[14] V. King and M. Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proceedings of the 7th Annual International Computing and Combinatorics Conference (COCOON), LNCS 2108*, pages 268–277, 2001.

[15] P. Loubal. A network evaluation procedure. *Highway Research Record 205*, pages 96–109, 1967.

[16] J. Murchland. The effect of increasing or decreasing the length of a single arc on all shortest distances in a graph. Technical report, LBS-TNT-26, London Business School, Transport Network Theory Unit, London, UK, 1967.

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

[18] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.

[19] V. Rodionov. The parametric problem of shortest distances. *U.S.S.R. Computational Math. and Math. Phys.*, 8(5):336–343, 1968.

[20] H. Rohnert. A dynamization of the all-pairs least cost problem. In *Proc. 2nd Annual Symposium on Theoretical Aspects of Computer Science, (STACS'85), LNCS 182*, pages 279–286, 1985.

Appendix

In this Appendix we provide further details on the implementation of procedures `delete`, `insert`, `distance`, and `path`, that will appear in the full version of the paper. A preliminary C implementation of our data structure can be found at the URL:
`http://www.dis.uniroma1.it/~demetres/experim/LDSP/`.

```
delete(u, v):
    if (u, v) ∈ P_uv then
        Q ← {(u, v)}
        while Q ≠ ∅ do
            extract any π_xy from Q
            remove π_xy from P_xy, L(r(π_xy)), and R(ℓ(π_xy))
            if π_xy ∈ P*_xy then remove π_xy from P*_xy, L*(r(π_xy)), and R*(ℓ(π_xy))
            add paths in L(π_xy) and paths in R(π_xy) to Q
```

```
insert(u, v, w):
    w_uv ← w                                                            {Phase 1}
    if w < +∞ then
        w((u, v)) ← w; ℓ((u, v)) ← π_uu; r((u, v)) ← π_vv
        add (u, v) to P_uv, L(π_vv), and R(π_uu)
    H ← ∅                                                               {Phase 2}
    for each (x, y) do add π_xy ∈ P_xy with minimum w(π_xy) to H
    while H ≠ ∅ do                                                      {Phase 3}
        extract π_xy from H with minimum w(π_xy)
        if π_xy is the first extracted path for pair (x, y) then
            if π_xy ∉ P*_xy then
                add π_xy to P*_xy, L*(r(π_xy)), and R*(ℓ(π_xy))
                for each π_x'b ∈ L*(ℓ(π_xy)) do
                    π_x'y ← (x', x) · π_xy
                    w(π_x'y) ← w_x'x + d_xy
                    ℓ(π_x'y) ← π_x'b; r(π_x'y) ← π_xy
                    add π_x'y to P_x'y, L(π_xy), R(π_x'b), and H
                for each π_ay' ∈ R*(r(π_xy)) do
                    π_xy' ← π_xy · (y, y')
                    w(π_xy') ← d_xy + w_yy'
                    ℓ(π_xy') ← π_xy; r(π_xy') ← π_ay'
                    add π_xy' to P_xy', L(π_ay'), R(π_xy), and H
```

```
distance(x, y):
    if P_xy = ∅ then return +∞
    else return weight of a minimum weight path in P_xy
```

```
path(x, y):
    if P_xy = ∅ then return ∅
    else return a minimum weight path in P_xy
```