Fully Dynamic Output Bounded Single Source Shortest Path Problem*

Daniele Frigioni†‡ Alberto Marchetti-Spaccamela‡ Umberto Nanni‡

(Extended Abstract)

Abstract

We consider the problem of maintaining the distances and the shortest paths from a single source in either a directed or an undirected graph with positive real edge weights, handling insertions, deletions and cost updates of edges. We propose fully dynamic algorithms with optimal space requirements and query time. The cost of update operations depends on the class of the considered graph and on the number of vertices that, due to an edge modification, either change their distance from the source or change their parent in the shortest path tree. In the case of graphs with bounded genus (including planar graphs), bounded degree graphs, bounded treewidth graphs and β-near-planar graphs with bounded β, the update procedures require $O(\log n)$ amortized time per vertex update, while for general graphs with n vertices and m edges they require $O(\sqrt{m}\log n)$ amortized time per vertex update. The solution is based on a dynamization of Dijkstra's algorithm [6] and requires simple data structures that are suitable for a practical and straightforward implementation.

1 Introduction

Finding shortest paths is a fundamental problem in computer science, and its solution provides answers to other interesting problems as well. The dynamic version of a shortest path problem consists in dealing with updates on the structure of the graph, while maintaining the possibility to answer queries on shortest paths without recomputing them from scratch. Various approaches have been considered in literature to deal with dynamic shortest path problems both for *single-source* and *all-pairs* versions.

The most general repertoire of update operations includes insertions and deletions of edges, update operations on the weight of edges, insertions and deletions of isolated vertices. When both insertions and deletions are allowed we refer to the *fully dynamic problem*; if we consider only insertions (deletions) then we refer to the *incremental* (*decremental*) problem.

Recent results concerning the dynamic shortest paths problem for planar graphs are provided in [10] and in [11]. Both these solutions use a topological partition of the graph based on recursive application of the planar separator theorem [13], and the algorithms proposed are complex and far from being practical. In [4] the authors consider efficient dynamic solutions for graphs with bounded treewidth when the weight of edges might change, but without considering insertions and deletions of edges. An efficient solution for the incremental problem has been proposed in [3] assuming that edge weights are integers restricted in the range $[1..C]$. Further results concerning the dynamic shortest paths problem for general graphs have been proposed, for example, in [7, 18, 19]. To the best of our knowledge, if insertions and deletions of edges are allowed and there is no restriction on the class of graphs then neither a fully dynamic solution nor a decremental solution for the single source shortest path problem is known that, in the worst case, is asymptotically better than recomputing the new solution from scratch, even in the case of unit edge weights.

The previous considerations state that single source shortest path is a hard problem to solve efficiently in a fully dynamic framework. However, at the same time, it is very important to find dynamic algorithms that are efficient but especially practical. In fact dynamic shortest path problems arise naturally in a number of contexts, such as in the fields of transportation and communication networks, and in many other applications (see, e.g., [1] for a wide variety of practical and theoretical application fields for the shortest paths problem).

The study of shortest path problems in a dynamic model, where we have to maintain efficiently information about shortest paths in a network during insertions and deletions of edges, is very relevant in the fields described above because the edge update operations reflect the real network changes as links that go up and down during the lifetime of the network.

*Work partially supported by EC ESPRIT Long Term Research Project ALCOM-IT under contract no.20244, and by *PROGETTO FINALIZZATO TRASPORTI 2* of the Italian National Research Council.

†Università di L'Aquila, Dipartimento di Matematica Pura ed Applicata, via Vetoio, I–67010 Coppito (AQ), Italy. frigioni@smaq20.univaq.it

‡Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza". Via Salaria 113 - 00198 - Roma, Italy. {alberto,frigioni,nanni}@dis.uniroma1.it

Since no fully dynamic solution that is both efficient and practical there exists in the standard models (*worst case* and *amortized*), we propose to measure the complexity of dynamic shortest paths algorithms as a function of the number of output modifications performed by the update operations.

1.1 Output complexity and previous results. In several applications, such as incremental compilation, dataflow analysis, text editing, graphical applications there might be the requirement of explicitly maintaining the solution to a given problem. In this case explicit updates must be carried out after that each modification has been specified. In these situations it is interesting to measure the performances of a dynamic algorithm using the number of updates that, after an input modification, must be performed on the output in order to restore the correct solution for the problem on hand.

In this paper we use the following output complexity model. Given a graph $G = (V, E)$, let $\sigma = \langle \mu_1, \mu_2, \ldots, \mu_h \rangle$ be a sequence of input modifications; each input modification $\mu_i \in \sigma$ consists of an edge operation (insertion, deletion or cost update) to be performed on graph G_{i-1}, with $G_0 \equiv G$, and gives the new graph G_i. After that each input modification $\mu_i \in \sigma$ is specified we are required to update the current output information. We assume that the *output information* consists of the value of distance from the source s for any vertex $x \in V$ and a single source shortest path tree rooted in s; the set δ_i of *output updates* due to input modification μ_i is formed by the set of vertices that either change the distance from the source or change the parent in the single source shortest path tree as a consequence of the input change μ_i. The *number of output updates* in a sequence σ is the sum of the number of output updates for all input modifications in the sequence, i.e., $|\Delta(\sigma)| = \sum_{\mu_i \in \sigma} |\delta_i|$. Note that no algorithm can process a sequence σ performing explicit updates in less than $|\Delta(\sigma)| + |\sigma|$ time.

Previous results concerning the dynamic single source shortest path problem using this or a similar model to measure the cost of the update operations have been proposed in [18, 8]. In [18] a fully dynamic algorithm for general graphs is evaluated as a function of an *extended size* $||\delta||$ of the output updates, defined as the sum of the number of vertices that change their distance from the source due to a set of edge operations δ, *plus* the number of edges having at least one endpoint in one of such vertices. Note that, in some cases, $||\delta||$ can be n times the cardinality of the set of vertices that must be actually updated (where n is the number of vertices in the graph).

In [8] the authors of this paper separately consider the *incremental* problem and the *decremental* problem proposing solutions that explicitly maintain the distances and a shortest path tree from a single source. For the incremental problem the proposed solution works for any graph. In particular, for any incremental sequence, if the final graph has a k-bounded accounting function, then the complexity of the incremental problem is $O(k \log n)$ amortized time per output update. An accounting function A is a function that for each edge (x, y) determines either vertex x or y as the *owner* of the edge; A is a k-bounded accounting function for G if k is the maximum over all vertices x of the cardinality of the set of edges owned by x. An analogous notion has been previously introduced in [5], where the authors define an *orientation* of a graph $G = (V, E)$ as a function ω which replaces each edge $(x, y) \in E$ by a directed edge $x \to y$ or $y \to x$. We will use the definition as accounting function that is more appropriate in order to evaluate the computational costs of our algorithms.

The value of such parameter k for any graph G can be bounded in different ways by using structural properties of the graph. For example k is immediately bounded by the maximum degree of G. In the case of planar graphs k can be bounded by three [5, 16]. Moreover it is possible to show that $k = O(1 + \sqrt{\gamma})$, where γ is the *genus* of G by observing that the *pagenumber* of a genus γ graph is $O(\sqrt{\gamma})$ [14]; since the genus of a graph is always less than its number of edges, it follows that a graph with m edges has a $O(\sqrt{m})$-bounded accounting function. Furthermore in [8] it has been proved that the parameter k is bounded by the *treewidth* of G. In the full version of this paper it is proved the existence of a $O(1 + \beta)$-bounded accounting function for β-near planar graphs, i.e., graphs that can be drawn on a planar surface with at most $\beta \cdot n$ edge crossings, where n is the number of vertices of the graph.

1.2 Results of the paper. The proposed dynamic algorithms explicitly maintain the distances and the shortest path tree from a source s for a graph G under sequences of insertions/deletions of edges, cost update of edges and insertions/deletions of isolated vertices, with a $O(1)$ worst case query time to obtain the distance between s and any vertex, and $O(l)$ worst case time to return a shortest path of l edges. Each update operation on the output data structures requires $O(k \log n)$ amortized time, if there exists a (possibly changing) k-bounded accounting function after each input modification for the resulting graph. Notice that the notion of k-bounded accounting function is useful only to bound the running time.

The main improvements with respect to the results proposed in [8] are the following:

- we extend the case of deletions from planar to general graphs;
- we propose fully dynamic algorithms and data structures for the single source shortest path problem with the same running time of the previous semidynamic solution;
- we also extend our results to the class of β-near planar graphs by proving the existence of a $O(1+\beta)$-bounded accounting function for any such graph.

Note that the property of having a k-bounded accounting function is a monotonic property of the graph, i.e., if a graph G has a k-bounded accounting function then any subgraph of G has the same property. The solution proposed in [8] for the decremental problem works only for planar graphs, and each deletion requires $O(\log n)$ amortized time per output update. In order to obtain the previous bound, *lazy updates* of the information stored in the data structures are performed, to defer as much as possible the required updates after each edge modification.

Our results improve, at least in amortized sense, those given by Ramalingam and Reps in [18], where a fully-dynamic solution for the single source shortest path problem for general graphs has been proposed which takes $O(||\delta|| \log ||\delta||)$ worst case time to handle a sequence of edge updates, where $||\delta||$ is the parameter described previously.

Our solution requires simple data structures that are really suitable for a practical and straightforward implementation: the basic structures are linked lists and priority queues; a first prototype of the algorithm is currently under development on the top of LEDA library [15].

2 Dynamic maintainance of shortest paths

Let $G = (V, E)$ be a weighted undirected graph with n vertices and m edges, and let $s \in V$ be a fixed *source* vertex. To each edge $(x, y) \in E$, a real positive weight $w_{x,y}$ is associated. Let $d(x)$ be the distance from source s to any node $x \in V$, $T(s)$ be a single source shortest path tree rooted in s and, for any $x \in N$, $T(x)$ be the subtree of $T(s)$ rooted in x. We study the problem of maintaining $T(s)$ and the distance function d for a weighted graph $G = (V, E)$ in a fully dynamic environment where an arbitrary sequence of query and modification operations of the following kinds can be performed on G:

- *distance*(x): report the current distance between s and vertex x;
- *path*(x): report a minimum cost path between s and vertex x;
- *insert*(x, y, w): insert edge (x, y) with weight w;
- *delete*(x, y): delete edge (x, y).

Our data structures allow to perform in constant time insertions and deletions of isolated vertices. Furthermore, with slight changes, our procedures can handle *weight-increase*(x, y, ϵ) and *weight-decrease*(x, y, ϵ) operations, consisting in increasing (or decreasing) by a real quantity ϵ the weight of edge (x, y). Here we will consider only undirected graphs, being straightforward the extension to directed ones.

Within this framework, an instance of the *fully-dynamic single source shortest path problem* with explicit updates is defined by a graph $G = (V, E)$ with real edge weights, a *source* vertex $s \in V$, an initial single source shortest path tree $T(s)$, and by a sequence $\sigma = \langle \mu_1, \mu_2, \ldots, \mu_h \rangle$ of *insert* and *delete* operation to be performed on G. Each modification $\mu_i \in \sigma$ when applied to graph G_{i-1} ($G_0 = G$) gives a new graph G_i. We are required to compute the new single source shortest path tree $T_i(s)$, and the distance value $d(x)$ for any $x \in V$. The set of output updates required by an input modification μ_i will be denoted as δ_i. In the case of an edge insertion the number of *output updates* is given by the number of vertices that change their distance from the source s as a consequence of that insertion. In the case of an edge deletion the number of output updates is given by the number of vertices that, either increase their distance from s or have to change their parent in $T(s)$ as a consequence of that deletion. The set of output updates required by the whole sequence σ will be denoted by $\Delta(\sigma)$.

Theorem 3.1 proves that, for any arbitrary sequence σ of input modifications the update cost is $O(k \log n)$ amortized time per output update, if all the graphs $G_0, G_1, \ldots, G_h$ allow a k-bounded accounting function. The parameter k can be bounded as follows:

- $k = O(\sqrt{m})$ for general graphs with m edges;
- $k = O(1+\sqrt{\gamma})$ for graphs with genus γ (hence $O(1)$ for planar graphs);
- $k = O(1+\beta)$ for β-near planar graphs;
- $k \leq d$ for graphs with maximum degree d;
- $k \leq t$ for graphs with treewidth t

Furthermore distance queries require constant time, while shortest path queries require time proportional to the number of edges in the returned minimum path.

Both procedures `Insert` and `Delete` proposed in the following are based on Dijkstra's algorithm for the single source shortest path problem [6]. In such algorithm, when a vertex y is permanently labeled and its distance from the source has been computed, all the neighbours of y are considered for possible improvements in the current shortest path from the source (incidentally, this is the strategy adopted in [18]). We show that it is possible to consider only a small fraction of neighbours of a node y, when its new distance

from the source has been computed after an edge modification.

In the following, for each vertex $z \in V$, $d(z)$ and $d'(z)$ will denote the values of the distances of z respectively before and after the insertion or the deletion of an edge, while $D(z)$ will denote the distance of vertex z from the source stored in the data structures.

DEFINITION 2.1. *Let $G = (V, E)$ be a weighted graph and $T(s)$ be a single source shortest path tree rooted in s. The* insertion_level *(*deletion_level*) of edge (z, q) (of vertex q) relative to vertex z is the quantity $i_level_z(q) = D(q) - w_{z,q}$ ($d_level_z(q) = D(q) + w_{z,q}$). The* insertion_slope *(*deletion_slope*) of edge (z, q) (of vertex q) relative to vertex z is the quantity $i_slope_z(q) = i_level_z(q) - D(z) = D(q) - D(z) - w_{q,z}$ ($d_slope_z(q) = d_level_z(q) - D(z) = D(q) - D(z) + w_{q,z}$).*

The intuition behind the above definitions is as follows: suppose that while processing an *insert* operation the algorithm has computed the new distance $D(z) = d'(z)$ of vertex z from s and that there exists an edge (z, q) such that $i_slope_z(q)$ is positive: this means that the path from the source to q that passes through vertex z is shorter than the shortest path from s to q in the graph before the insert operation. The case of a delete operation is analogous.

2.1 Data Structures. A single source shortest path tree $T(s)$ is represented explicitly, maintaining a bipartition of the edges in *tree edges* and *non-tree edges*. For any vertex $x \in V$, the following data structures are defined: $D(x)$ stores the computed distance of vertex x from s; *parent*(x) and *children*(x) refer to the current structure of tree $T(s)$; any edge $(x, y) \in E$ has an *owner* that must be either x or y and will be denoted as *owner*(x, y). For each vertex $x \in V$, *ownership*(x) stores the set of edges owned by x, and *not-ownership*(x) denotes the set of edges with one endpoint on x, but not owned by x. The edges in the set *not-ownership*(x) are stored as follows:

1. B_x is a max-based priority queue containing the edges in *not-ownership*(x); the priority (or *B-level*) of vertex y in heap B_x, denoted as $b_x(y)$, will be the computed value of $i_level_x(y)$;
2. F_x is a min-based priority queue containing the edges in *not-ownership*(x); the priority (or *F-level*) of vertex y in heap F_x, denoted as $f_x(y)$, will be the computed value of $d_level_x(y)$.

Before processing the sequence σ of the input modifications, for each vertex x and for each edge (x, y) not owned by x the heap structures are initialized by computing $b_x(y) = i_level_x(y)$ and $f_x(y) = d_level_x(y)$. As we will see these conditions are enforced by both procedures `Insert` and `Delete`.

2.2 Insertion of an edge. If a new edge (x, y) is inserted (we assume wlog that $d(x) \leq d(y)$) the current distances of vertices from the source can only decrease. If $d'(y) < d(y)$ then all vertices that belong to $T(y)$ change their distance from s as a consequence of the edge insertion. On the other side, the new subtree $T'(y)$ may include other vertices not included in $T(y)$.

Let us sketch the strategy of our algorithm to handle edge insertions. If the insertion of edge (x, y) improves the distance of vertex y from s, a global priority queue C is used like in Dijkstra's algorithm in order to find new distances from the source in nondecreasing order. Unlike Dijkstra's algorithm, when a vertex z is dequeued from C and its new distance from the source $D(z) = d'(z)$ is computed, not all edges leaving z are scanned in order to "announce" the new distance $d'(z)$. Any of such edges (z, q) is scanned only if either z is the *owner* of edge (z, q) (in this case edge (z, q) is *scanned by ownership*), or $i_slope(z, q)$ is positive (i.e., $b_z(q) > D(z)$: in this case edge (z, q) is *scanned by priority*, and both vertex q and edge (z, q) are said to be *high* for z).

To check the above conditions, we need to maintain the correct information on the *i_level* for all the neighbours of each vertex; if this information is updated explicitly during each input modification then it might require the scanning of each edge incident to a vertex that has changed its distance from s. In the worst case this number might be n times larger than the number of output updates.

In order to trace out vertices that must be updated, we define a coloring of vertices and edges in the graph describing the required updates as a consequence of one edge insertion: initially all vertices and edges are colored *white*; if q does not change the distance from the source then it is a *white* vertex; if q decreases its distance from the source, then q is a *red* vertex. Note that a *white* vertex does not require any update; if q is *red* then all adjacent vertices that are *high* for q must be updated and will be *red* vertices. Also edges are colored by the algorithm according to the following rule: edge (x, y) is colored *red* if x is *red* and edge (x, y) is *high* for x.

We are now ready to present the `Insert` algorithm whose pseudocode is given in Figure 1.

In Step 1 procedure Set_Ownership(x, y) arbitrarily chooses the owner of the inserted edge (x, y) and it properly updates the local data structures stored at vertices x and y.

In Step 2 procedure Enqueue$(C, \langle y, D(x) + w_{x,y}; x\rangle)$ initializes a global heap C with vertex y with priority $p_C(y) = d'(y) = D(x) + w_{x,y}$ and candidate parent x.

Step 3 computes a new shortest path tree for the current graph G in a way similar to Dijkstra's algorithm. In particular, while heap C is not empty, vertex z

```
    procedure Insert(x, y : vertex; w_{x,y} : positive_real);
    Step 1 {suppose wlog that d(x) ≤ d(y)}
1.      Set_Ownership(x, y);
2.      if D(y) ≤ D(x) + w_{x,y} then EXIT;
    Step 2
3.      C ⟵ ∅;
4.      Enqueue(C, ⟨y, D(x) + w_{x,y}; x⟩);
    Step 3
5.      while non-Empty(C) do
6.          begin
7.              ⟨z, p_C(z); q⟩ = Extract_Min(C);
8.              parent(z) ⟵ q;
9.              D(z) ⟵ p_C(z) = D(q) + w_{q,z};
10.             color(z) ⟵ red;
11.             for each high edge (z, h) ∈ not-ownership(z) do
12.                 begin{(z, h) is scanned by priority}
13.                     Insert-Improve(C, ⟨h, D(z) + w_{h,z}; z⟩);
14.                     color(z, h) ⟵ red;
15.                 end
16.             for each edge (z, h) ∈ ownership(z) do
17.                 if D(h) > D(z) + w_{hz} {i.e., if i_slope_z(h) > 0}
18.                   then begin{(z, h) is scanned by ownership}
19.                         Insert-Improve(C, ⟨h, D(z) + w_{h,z}; z⟩);
20.                         color(z, h) ⟵ red;
21.                   end;
22.         end;
    Step 4
23.     for each red vertex z do
24.         for each nonred edge (z, h) ∈ ownership(z) do
25.             Change_Ownership(z, q);
26.     for each red edge (z, q) do
27.         Update_Levels(z, q);
28.     restore the original color white for all edges and vertices;
```

Figure 1: Insertion of edge $(x, y; w)$

with minimum priority $p_C(z)$ and candidate parent q is extracted from C by procedure Extract_Min(C). When vertex z is extracted then its priority in the heap C is equal to the new distance $D(z) = d'(z)$ of z from s. Hence $D(z)$ is set to be $p_C(z)$, the parent of z in $T(s)$ is set to be q and z is colored *red*. Then the procedure repeatedly extracts by priority vertices from heap B_z until a vertex that is not *high* is found; for each *high* vertex q, if $q \notin C$ then q is inserted in C with priority equal to the length of the shortest path passing through z; otherwise the priority of q in C is updated to the value of the length of the path from s to q passing through z. These operations are performed by calling procedure Insert-Improve($C, \langle q, D(z) + w_{q,z}; z\rangle$). All edges scanned in this phase are colored *red*. The new distance $D(z) = d'(z)$ of vertex z is "announced" also along each edge $(z, q) \in$ *ownership*(z). In particular for each edge $(z, q) \in$ *ownership*(z) if (z, q) is *high* for z (i.e., $i_slope_z(q) > 0$) then procedure Insert-Improve($C, \langle q, D(z) + w_{q,z}; z\rangle$) is called, and edge (z, q) is colored *red*.

In Step 4 the ownership of each nonred edge owned by a *red* vertex, is changed by calling procedure Change_Ownership(x, y). Namely, assuming wlog that x is the owner of edge (x, y), then item (x, y) is deleted from *ownership*(x), B_y and F_y and it is inserted in *ownership*(y), B_x and F_x with updated levels $b_x(y) = i_level_x(y)$ and $f_x(y) = d_level_x(y)$ respectively. The change of ownership is not necessary for the correctness of the algorithms, but only to obtain the claimed amortized time bounds. Furthermore, for each *red* edge (z, q) the relative levels of q in B_z and F_z and of z in B_q and F_q are properly updated by procedure Update_Levels(z, q). The Update_Levels operations performed in this step guarantee that all the information stored in the data structures is always updated during the execution of any update procedure. Finally, all *red* edges and vertices are restored to be *white*.

2.3 Deletion of an edge. If edge (x, y) is deleted from G and $d(x) < d(y)$ then all vertices that must be updated are in the tree $T(y)$. Note that, in order to bound the number of operations as a function of the number of output updates, it is not possible to search the whole subtree $T(y)$, since not all vertices in $T(y)$ must be updated. In fact there could be a vertex w in $T(y)$ such that: 1) in the graph G (before the delete operation) there exists an alternative shortest path P from s to w that does not include edge (x, y) and whose length is equal to $d(w)$; 2) the last edge of P is edge (r, w) and r does not belong to $T(y)$. Hence, in the updated graph, vertex w belongs to the set of updated vertices (it does not change its distance but it changes its parent in the single source shortest path tree); on the contrary all vertices in $T(w)$ change neither the distance nor their parent in the single source shortest path tree.

Procedure `Delete` shown in Figure 2 first finds all vertices to be updated after an edge deletion, and then computes the new distances and the new shortest path tree. In order to determine the updated vertices, we define a coloring of the vertices and of the edges of the graph. We assume that all vertices and edges are initially *white*, then a subset of vertices and edges are colored *red* or *pink* (step 2). After that the new distances and a new shortest path tree are computed (Step 3). At the end the original *white* color is restored both for vertices and edges (Step 4). Vertices are colored as follows: if q does not change neither the distance from s nor the parent in $T(s)$, then q is a *white* vertex; if q increases the distance from the source, i.e., if $d'(q) > d(q)$, then q is a *red* vertex; if q preserves its distance from s, but it must replace the old parent in $T(s)$, then q is a *pink* vertex (i.e., q is *pink* if $d'(q) = d(q)$, and either q is a child of a *red* vertex in $T(s)$, or $q \equiv y$). Note that: if q is a *white* vertex it does not require any

update; if q is *red* then all children of q in $T(s)$ must be updated and will be either *pink* or *red*; if q is *pink* then vertices in $T(q)$ (except q itself) are *white*. Edges are colored as follow: if edge (x, y) is such that both x and y are *red*, then it is colored *red*; all red edges will be scanned by the algorithm, and if a nonred edge is scanned, then it will be colored *pink*; in any other case edge (x, y) is colored *white*.

```
    procedure Delete(x, y : vertex);
    Step 1
1.      Remove(x, y);
2.      if (x, y) is not a tree edge, then EXIT;
3.      M ←− ∅; Q ←− ∅;
4.      Enqueue(M, ⟨y, D(y)⟩);
    Step 2.a
5.      while non-Empty(M) do
6.          begin
7.              ⟨z, d(z)⟩ = Extract_Min(M);
8.              q ←− Mildest_Slope(z);
9.              if q exists and it is an equivalent parent for z
10.                 then begin
11.                         parent(z) ←− q;
12.                         color(z) ←− pink;
13.                        end
14.                 else begin
15.                         color(z) ←− red;
16.                         for each v ∈ children(z) do
17.                             Heap_Insert(M, ⟨v, D(v)⟩);
18.                         end
19.             end
    Step 2.b
20.     color red each edge with both endpoints red;
    Step 3.a
21.     for each red vertex z do
22.         begin
23.             q ←− Mildest_Slope(z);
24.             if q exists
25.                 then Enqueue(Q, ⟨z, D(q) + w_{z,q}; q⟩);
26.                 else Enqueue(Q, ⟨z, ∞; ∅⟩);
27.         end
    Step 3.b
28.     while non-Empty(Q) do
29.         begin
30.             ⟨z, D(z); q⟩ = Extract_Min(Q);
31.             parent(z) ←− q; d(z) ←− D(q) + w_{z,q};
32.             for each red edge (z, h) leaving z do
33.                 Heap_Improve(Q, ⟨h, D(z) + w_{h,z}; z⟩);
34.         end
    Step 4
35.     for each red vertex z do
36.         for each nonred edge (z, h) ∈ ownership(z) do
37.             Change_Ownership(z, q);
38.     for each red edge (z, q) do
39.         Update_Levels(z, q);
40.     restore the original color white for all edges and vertices;
```

Figure 2: Deletion of edge (x, y)

For each vertex $q \in V$ we define its *mildest-slope* neighbour as the nonred vertex adjacent to q with minimum positive *d_slope* with respect to q. Therefore the *mildest-slope* neighbour z of a vertex q that has increased its distance from s due to an edge deletion, is the vertex that represents the best alternative parent for q among all nonred neighbours of q. We require that the *mildest-slope* neighbour must be nonred since the distance of a red vertex increases in the updated graph and the new distance is not known when the procedure that determines the *mildest-slope* neighbour is called.

The *mildest-slope* neighbour z of a vertex q is found by using procedure `Mildest_Slope`; it first searches the best nonred neighbour among the edges owned by q; then the procedure selects the best nonred neighbour not owned by q by repeatedly extracting elements from F_q; finally it chooses the best among the two nonred neighbours found so far.

In a way similar to the case of an edge insertion, during a deletion any vertex z changing its distance from s announces the new distance to a neighbour q only if one of the following conditions arises: *i*) z is the owner of edge (z, q): in this case (z, q) is scanned *by ownership*; *ii*) q is the owner of (z, q) and that edge is scanned from z while looking for the *mildest-slope* neighbour: in this case (z, q) is scanned *by priority*.

Pseudocode of procedure `Delete` is given in Figure 2. Let (x, y) be the deleted edge and assume, without loss of generality, that $d(x) \leq d(y)$.

In Step 1 procedure Remove(x, y) is called. If $owner(x, y) = x$ then item (x, y) is deleted from $ownership(x)$ and from heaps B_y and F_y, otherwise it is deleted from $ownership(y)$, B_x and F_x. Then, if (x, y) is a *non-tree* edge, procedure `Delete` halts; otherwise vertex y is inserted in a global heap M with priority $p_M(y)$ equal to its current distance from the source $d(y)$.

In Step 2 vertices and edges of the graph are colored by performing a search that repeatedly extracts from M the vertex z with minimum priority. Then procedure `Mildest_Slope`(z) is called in order to find the best current alternative path from the source to z. If z does not change its distance from s but only its parent in $T(s)$, then it is colored *pink*; otherwise it is colored *red* and all its children in $T(s)$ are enqueued in M with priority provided by their current distance from the source. When heap M is empty all edges with both endpoints *red* are colored *red* (step 2.b), by using ownerships of *red* edges.

The computation of the shortest path for each *red* vertex q is performed in Step 3 by using two substeps. In Step 3.a the algorithm tries to compute a path (not necessarily the shortest path) in the updated graph by using procedure `Mildest_Slope`. If such a path is found vertex q is inserted in a global heap Q with priority $p_Q(q)$ provided by the *d_level* of its *mildest-slope* neighbour. If the *mildest-slope* neighbour of q does not

exists, vertex q is inserted in Q with infinite priority. In Step 3.b a computation similar to Dijkstra's algorithm is performed on the subgraph induced by *red* vertices, by using heap Q. In particular while Q is not empty vertex q with minimum priority is repeatedly extracted and the value $d'(q)$ is set to the current value of $p_Q(q)$; furthermore each vertex extracted from Q propagates the new information only along *red* edges in order to improve the priority of some other vertex in Q. Note that the coloring of *red* vertices in Step 2 is used in order to initialize Q in Step 3.b.

In Step 4 for each *red* vertex z the following operations are performed with the same behaviour of the analogous operations performed in Step 4 of `Insert`: *a*) for each nonred edge $(z,q) \in$ *ownership*(z), the ownership is changed, i.e., item (z,q) is deleted from *ownership*(z), B_q and F_q, and it is inserted in *ownership*(q) and in heaps B_z and F_z by using as priority the new levels $b'_z(q) = d'(q) - w_{zq}$ and $f'_z(q) = d'(q) + w_{zq}$ respectively; *b*) for each *red* edge (z,q) the levels of q in B_z and F_z and of z in B_q and F_q are properly updated by procedure Update_Levels(z,q). Finally the original *white* color is restored for all vertices and edges.

3 Analysis of the Algorithms

3.1 Correctness analysis. The correctness of our algorithms is based on the following properties that must hold before and after the execution of the update procedures.

P1) for each vertex $q \in V$ the length of the shortest path is correctly stored, that is $D(q) = d(q)$;

P2) $T(s)$ is a single-source shortest path tree rooted in s for the current graph G;

P3) for each vertex z and for each edge (z,q) not owned by z the following is true: $b_z(q) = i_level_z(q)$ and $f_z(q) = d_level_z(q)$.

LEMMA 3.1. *Let $G = (V,E)$ be a rooted graph, if an edge is inserted in (deleted from) G and properties* P1-P3 *hold, then after the execution of procedure* `Insert` *(procedure* `Delete`*) property* P3 *holds.*

Sketch of the Proof. Before processing the sequence σ, for each vertex x and for each edge (x,y) not owned by x heaps B_x and F_x are initialized by computing $b_x(y) = i_level_x(y)$ and $f_x(y) = d_level_x(y)$ respectively. The thesis is a consequence of the execution of Step 4 at the end of both procedures `Insert` and `Delete`. Let z be a *red* vertex, and let (z,q) be adjacent to z. Both for `Insert` and `Delete` procedures there are two possibilities to be considered, depending on whether edge (z,q) is scanned or not. Then if edge (z,q) is scanned there are other two subcases to be considered depending on whether the edge is scanned by ownership or by priority. All these cases will be analyzed in the full paper. □

LEMMA 3.2. *Let $G = (V,E)$ be a rooted graph and G' be the new graph after the insertion of edge (x,y). Let us suppose that properties $P1$, $P2$ and $P3$ hold before the* `Insert` *operation then properties $P1$ and $P2$ hold after the procedure has been executed.*

Sketch of the Proof. Assume that the algorithm does not satisfy property $P1$. Then there must exist two vertices v and z such that: (i) the value $D(z)$ found by the algorithm as the distance from s to z is not correct; (ii) the algorithm correctly computes the distance from s to v; (iii) edge (v,z) is in a shortest path from s to z after the update operation. i)-iii) imply the following inequalities: $d'(v) + w_{v,z} = d'(z) < D(z) \leq d(z)$.

There are two possibilities.

case 1. $d(v) = d'(v)$. In this case we have that $d'(z) = d'(v) + w_{v,z} = d(v) + w_{v,z} \geq d(z)$. Since $d'(z) \leq d(z)$ it follows that $d'(z) = d(z)$. This contradicts the fact that distances have been correctly computed before the considered update operation.

case 2. $d(v) > d'(v)$. It is sufficient to show that the algorithm updates the priority of vertex z in the global heap C to the value $d'(v) + w_{v,z}$. If $(x,y) = (v,z)$ then this operation is performed in step 2. If $x = v$ and $z \neq y$ then by iii) above (x,z) is in the shortest path tree after the update operation; since (x,y) is in the updated shortest path tree it follows that $d'(z) = d(z)$ contradicting the fact that distances have been correctly computed before the considered update operation.

If $x \neq v$ let us consider the behaviour of the algorithm immediately after the step that computes the distance $d'(v)$ of vertex v from the source in the updated graph G'. If $z \in$ *ownership*(v) then edge (v,z) is scanned and the priority of z in C is set to be $d'(v) + w_{v,z}$. Otherwise the algorithm considers the set of all *high* vertices r in the heap B_v. Since $d'(v) < d(z) - w_{v,z}$ vertex z is considered, edge (v,z) is scanned and the priority of z in C is set to be $d'(v) + w_{v,z}$. In any case the priority of z in C is correctly updated and this contradicts the fact that the distance of z from the source is not correctly computed. □

LEMMA 3.3. *Let $G = (V,E)$ be a rooted graph, let G' be the new graph after the deletion of edge (x,y), and let us suppose that properties P1–P3 hold before the execution of procedure* `Delete`*. Procedure* `Delete` *colors correctly all vertices $q \in V$ in Step 2.*

Sketch of the Proof. The proof is by induction on the values of distances of vertices from s and will be given in the full paper. □

LEMMA 3.4. *Let $G = (V, E)$ be a rooted graph and G' be the new graph after the deletion of edge (x, y). If properties* P1–P3 *hold before the deletion, then* P1–P2 *hold at the end of procedure* `Delete`.

Sketch of the Proof. Lemma 3.3 proves that the set of vertices colored *red* (in Step 2) and enqueued in Q (in Step 3.a) is exactly the set of vertices that increase their distance from the source. This implies that, for each *pink* or *white* vertex z, $D(z) = d(z) = d'(z)$. The task of Step 2.b is simply to color *red* all the edges with both endpoints *red*. Now let us focus on Step 3, that computes the new distance from the source for each *red* vertex. Note that for each *red* vertex, the best nonred neighbour must be searched only after that all the vertices have been colored. Therefore each vertex in heap Q is enqueued with a value of priority, either equal to the length of a path passing through a non-red vertex (not necessarily the shortest path), or an infinite value when no nonred neighbour is available.

The proof that properties *P1* and *P2* for red vertices are maintained is very similar to the proof of correctness of Dijkstra's algorithm. Observe that each red vertex q is managed in the following way:

a) in Step 3.a the priority of vertex q in queue Q is computed as the length of the shortest path from q to s such that the first vertex in such a path is not red (if such a path does not exist, the priority is given a conventional infinite value);

b) in Step 3.b the priority of q in Q might decrease if a red neighbour z of q provides a path shorter than the one previously computed: note that when a vertex z is extracted from Q, the procedure determines for each *red* neighbour v of z the length the shortest paths from the source using (z, v) as the last edge. If any vertex v is not connected to the source after the deletion of edge (x, y), then the procedure correctly sets to an infinite value the item $D(v)$ that stores the distance from s.

□

3.2 Complexity analysis. We use the notion of *k-bounded accounting function* to share the computational costs of the algorithms.

DEFINITION 3.1. *Let $G = (V, E)$ be a graph. An* accounting function *for G is a function $A : E \to V$ such that for any edge $(x, y) \in E$, $A(x, y)$ is either x or y, which is called the* owner *of edge (x, y). $A : E \to V$ is a* k-bounded accounting function *for G if for any vertex $x \in V$, the set $A^{-1}(x)$ of the edges owned by x has cardinality at most k.*

THEOREM 3.1. *Given a rooted graph $G = (V, E)$ let $\sigma = \langle \mu_1, \mu_2, \ldots, \mu_h \rangle$ be a sequence of insert and delete operations on G requiring $|\Delta(\sigma)|$ output updates. If G_i, for $i = 1, 2, \ldots, h$, has a k-bounded accounting function, then it is possible to maintain $T(s)$ and the distance function during the sequence σ in total time $O(|\Delta(\sigma)| \cdot k \log |V| + |\sigma|)$, that is $O(k \log |V|)$ amortized time per output update.*

The data structures require $O(|V| + |E|)$ space, while queries may be answered in time: $O(1)$ to report the distance of any vertex from the source, and $O(l)$ to trace a minimum cost path with l edges.

Sketch of the Proof. The space and queries time bounds are immediate. In order to prove the complexity bounds of the update procedures we distribute credits to vertices and edges according to the following policy:

i) initially each vertex is given $4k$ credits and each edge is given 2 credits;

ii) when an *insert* or *delete* operation is performed, 2 additional credits becomes available;

iii) when an edge (x, y) is inserted in G, we set $\mathit{Credit}(x, y) = 2$;

iv) when a vertex z changes its distance from the source, it is given $8k + 2$ credits; if it changes its parent in $T(s)$ but not the distance it is given 2 credits.

For each vertex z and for each edge (x, y), let us denote respectively as $\mathit{Credit}(z)$ and $\mathit{Credit}(x, y)$ the current balance of credits. Credits are spent while executing procedures `Insert` and `Delete`. In both procedures a constant number of operations is done (each requiring at most $O(\log n)$ time) for each red edge and for each red vertex to broadcast the new distances and to revise ownership of edges. Furthermore in procedure `Delete` a search is performed by procedure `Mildest_Slope` called both in steps 2 and 3.a. All edges scanned in a single call to procedure `Mildest_Slope` are connected to a *red* vertex, with the exception of the last vertex, which should be nonred. This means that for any *red* or *pink* vertex z, in each call to this procedure, a single edge can be scanned without being colored red: we charge two of the credits given to the *red* or *pink* vertex z. Note that, while processing an edge deletion, changing the parent of a vertex in order to maintain the old value of distance from the source requires constant amortized time as observed in [12].

In order to evaluate the number of credits necessary to pay the total number of *red* edges scanned we observe that each *red* vertex can scan edges either by priority (and in this case the other endpoint is surely colored red) or by ownership. Both for `Insert` and `Delete`, at most $2k$ of the credits provided to (*red*) vertex z are sufficient to pay for all red edges scanned by priority from z since, for each of such edges, both endpoints are colored *red*.

As far the edges scanned by ownership are concerned we first consider the case in which there exists a single *stable* k-bounded accounting function that does not change while insertions and deletions of edges are performed on the graph. It is a special case of the more general case, but we consider it first as a "warming up".

We assume that there exists a *canonical* k-bounded accounting function A^c_σ for the graph $G^c_\sigma = (V, E^c_\sigma)$, whose edges include both the edges in the initial graph and those inserted during the considered sequence of operations. We define *canonical owner* of each edge the owner provided by function A^c_σ. In contrast we will call *current owner* of an edge (x, y) the vertex that owns (x, y) in the current state of the data structures. Let us consider any edge and the history of its current ownership while processing the sequence of update operations. Any two times that the edge is scanned by ownership (with consequent switch of ownership), the canonical owner must become the current owner, and $2k$ of the credits gained when a vertex x is updated are sufficient to balance the number of scanned edges.

We now consider the general case where a different k-bounded accounting function may exists for each G_i while a sequence of edge updates is processed. Note that the credit policy described in the case of a stable accounting function does not guarantee that all the scans can be paid by using the credits distributed due to output updates. As an example, consider the following unfortunate sequence of events (let (x, y) be an edge in E and suppose that the canonical owner is x):

1. edge (x, y) is scanned by ownership from y, but this does not cause any change in vertex x;
2. due to some operation in the graph, the canonical ownership of edge (x, y) changes to y;
3. edge (x, y) is scanned by ownership from x, but this does not cause any change in vertex y;
4. due to some operation in the graph, the canonical ownership of edge (x, y) changes to x.

The events 1–4 define a cycle such that, when the current owner scans edge (x, y) by ownership, it is not the canonical owner of the same edge, and hence it does not gain any credit to balance the edge to be scanned.

In order to prove the theorem in the general case we assume that for any G_i there exists a canonical ownership function that for each edge determines a *canonical owner*. Let us call *unpaid scan* the event of scanning an edge (x, y) from y by ownership, without changing the distance of vertex x from the source (x is the current canonical owner of (x, y)). In order to bound the number of unpaid scans performed while modifying the structure of a graph $G = (V, E)$, we need to use the credits allocated on the edges. While processing a sequence of edge updates σ the credits on the edges are handled as follows:

i) if an edge insertion occurs and an unpaid scan is performed on edge (x, y), then:

- if $\mathit{Credit}(x, y) > 0$ then this quantity is decremented by 1;
- if $\mathit{Credit}(x, y) = 0$ then all edges (i, j) such that $\mathit{Credit}(i, j) = 0$ are *refunded* taking 2 credits from the canonical owner, and then $\mathit{Credit}(x, y)$ is decremented by 1;

ii) when a vertex x changes its distance from the source, for all edges (x, y) that are in the canonical ownership of x we set $\mathit{Credit}(x, y) = 2$.

We need to prove that the balance of credits on any vertex is never negative. When all edges (i, j) with $\mathit{Credit}(i, j) = 0$ are refunded the graph $G^R = (V, E^R)$, with $E^R = \{(i, j) \mid (i, j) \in E \text{ and } \mathit{Credit}(i, j) = 0\}$, is a subgraph of G, and hence the canonical k-bounded accounting function in the current graph is also a k-bounded accounting function for G^R.

The ownership policy and the credit policy guarantee the following property: if, for any edge (x, y), $\mathit{Credit}(x, y)$ reaches the value 0, then the two unpaid scans have been performed from the two different endpoints x and y. When an edge refund occurs, the balance in any vertex cannot become negative, since between any two consecutive edge refund of a given edge (x, y), both vertices x and y have been updated and hence both have been given $2k$ credits to cover a possible edge refund. □

Theorem 3.1 allow us to bound the update time for specific classes of graphs. We recall that the *genus* of a graph G is the smallest integer γ such that it is possible to draw G on a planar (or spherical) surface with γ *handles* with no edge intersections [9]. The *pagenumber* of a graph G is the minimum number of planar graphs in which G can be embedded [14]. One possible definition of the treewidth of a graph can be found in [2]. Finally a β-near-planar graph, where β is a nonnegative integer, is a graph $G = (V, E)$ that can be drawn on a planar surface with at most $\beta \cdot n$ edge crossings (see e.g. [17]). The following corollary holds.

COROLLARY 3.1. *The fully dynamic single source shortest path problem can be solved in $O(k \log n)$ amortized time per output update, where, for any graph $G = (V, E)$:*

- *$k = O(\sqrt{m})$ where $m = |E|$;*
- *$k = O(1 + \sqrt{\gamma})$ if G has genus γ (hence $k = O(1)$ if G is planar);*
- *$k = O(1 + \beta)$ if G is β-near planar;*
- *$k \leq d$ if G has maximum degree d;*
- *$k \leq t$ if G has treewidth t*

The proof of the existence of a $\sqrt{\gamma}$-bounded accounting function for graphs with genus γ is due to the result, provided in [14], that the *pagenumber* of a genus γ graph is $O(\sqrt{\gamma})$, and to the fact that the arboricity of a planar graph is less equal than three [16]. Since the genus of a graph is always less than its number of edges, it follows that a graph with m edges has a $O(\sqrt{m})$-bounded accounting function. The proof of the existence of a k-bounded accounting function for bounded treewidth graphs is given in [8]. In the full version of this paper it will be proved that there exists an $O(1+\beta)$-bounded accounting function for β-near planar graphs.

4 Conclusions and open problems

In this paper we have proposed fully dynamic data structures and algorithms for maintaining a single source shortest path tree for a graph G with $O(k \log n)$ amortized time per output update. An interesting open problem is to improve the amortized bounds proposed in the paper to worst case bounds. In particular, as observed in [12], if there exists an algorithm for the fully dynamic maintenance of a k-bounded accounting function (or any approximation within a constant factor) of a graph subject to sequences of *insert* and *delete* operations on the edges, this would allow to improve from $O(\log n)$ amortized time to $O(\log n)$ worst case time per output update the performances of our algorithms.

Acknowledgments: We are indebted to Han La Poutré and Pino Italiano for constructive discussions and suggestions. We like also to thank an anonymous referee for providing us useful comments and appropriate references.

References

[1] R. K. Ahuia, T. L. Magnanti and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, NJ (1993).

[2] S. Arnborg, Efficient Algorithms for Combinatorial Problems on Graphs with Bounded Decomposability - A survey, *BIT*, **25** (1985), 2–23.

[3] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela and U. Nanni, Incremental Algorithms for Minimal Length Paths, *J. Algorithms*, **12**, 4 (1991), 615–638.

[4] S. Chaudhuri and C. D. Zaroliagis, Shortest Path Queries in Digraph of Small Treewidth, *Proc. Int. Coll. Aut. Lang. Progr. (ICALP '95)*, 1995; LNCS 944, 244–255.

[5] M. Chrobak and D. Eppstein, Planar orientations with low out-degree and compaction of adjacency matrices, *Th. Comp. Sci.*, **86** (1991), 243–266.

[6] E. W. Dijkstra, A note on two problems in connection with graphs, *Numer. Math.*, **1** (1959), 269–271.

[7] S. Even and H. Gazit, Updating distances in dynamic graphs, *Meth. Oper. Res.*, **49** (1985), 371–387.

[8] D. Frigioni, A. Marchetti-Spaccamela and U. Nanni, Incremental Algorithms for the Single Source Shortest Path Problem, *Proc. 14th Int. Conf. FST&TCS '94*, 1994; LNCS 880, 113–124.

[9] F. Harary, *Graph Theory*, Addison-Wesley (1969).

[10] P. N. Klein, S. Rao, M. Rauch and S. Subramanian, Faster shortest-path algorithms for planar graphs, *Proc. 26th ACM Symp. Theory of Comp. (STOC '94)*, 1994, 27–37.

[11] P. N. Klein and S. Subramanian, Fully Dynamic Approximation Schemes for Shortest Path Problems in Planar Graphs, *Proc. Int. Workshop Alg. Data Struct. (WADS '93)*, 1993; LNCS 709, 443–451.

[12] H. La Poutré, *personal communication*, 1994.

[13] R. J. Lipton and R. E. Tarjan, A Separator Theorem for Planar Graphs, *SIAM J. Appl. Math.*, **36** (1979), 177–189.

[14] S. M. Malitz, Genus g Graphs Have Pagenumber $O(\sqrt{g})$, *J. Algorithms*, **17** (1994), 85–109.

[15] K. Mehlhorn and S. Näher, LEDA, a Platform for Combinatorial and Geometric Computing, *Comm. of the ACM.*, **38** (1995).

[16] C. Nash-Williams, Edge-disjoint spanning trees of finite graphs, *J. London Math. Soc.*, **36** (1961), 445–450.

[17] V. Radhakrishnan, H.B. Hunt III and R. Stearns, Efficient Algorithms for Solving Systems of Linear Equations and Path Problems, *Proc. Annual Symp. Th. Aspects Comp. Sci. (STACS '92)*, 1992; LNCS 577, 109–119.

[18] G. Ramalingam and T. Reps, An Incremental Algorithm for a Generalization of the Shortest Path Problem, *Tech. Rep. 1087*, Comp. Sc. Dept, Un. of Wisconsin, Madison, WI (1992).

[19] H. Rohnert, A dynamization of the all-pairs least cost path problem, *Proc. Annual Symp. Th. Aspects Comp. Sci. (STACS '85)*, 1985; LNCS 182, 279–286.

[20] R. E. Tarjan, Amortized computational complexity, *SIAM J. Alg. Discr. Meth.*, **6** (1985), 306–318.