Journal of Algorithms **34,** 251–281 (2000)
doi:10.1006/jagm.1999.1048, available online at http://www.idealibrary.com on IDEAL®

Fully Dynamic Algorithms for Maintaining Shortest Paths Trees[1]

Daniele Frigioni

Dipartimento di Ingegneria Elettrica, Università di L'Aquila, Monteluco di Roio, I-67040 L'Aquila, Italy; Dipartimento di Informatica e Sistemistica, Universita di Roma "La Sapienza," Via Salaria 113, I-00198 Rome, Italy
E-mail: frigioni@infolab.ing.univaq.it, frigioni@dis.uniroma1.it

and

Alberto Marchetti-Spaccamela and Umberto Nanni

Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza," Via Salaria 113, I-00198 Rome, Italy
E-mail: alberto@dis.uniroma1.it, nanni@dis.uniroma1.it

Received February 18, 1997; revised August 4, 1999

We propose fully dynamic algorithms for 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 weight updates of edges. The algorithms require linear space and optimal query time. The cost of the update operations depends on the class of the considered graph and on the number of the output updates, i.e., on the number of vertices that, due to an edge modification, either change the distance from the source or change the parent in the shortest paths tree. We first show that, if we deal only with updates on the weights of edges, then the update procedures require $O(\log n)$ worst case time per output update for several classes of graphs, as in the case of graphs with bounded genus, bounded arboricity, bounded degree, bounded treewidth, and bounded pagenumber. For general graphs with n vertices and m edges the algorithms

[1] Work partially supported by the ESPRIT Long Term Research Project ALCOM-IT under Contract 20244 and by *Progetto Finalizzato Trasporti 2* (*PFT 2*) of the Italian National Research Council (CNR). The work of the first author is supported by Fellowships Program No. 201.15.11 of the Italian National Research Council. Parts of this work were presented at the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'96), January 28–30, 1996, Atlanta, Georgia.

0196-6774/00 $35.00

require $O(\sqrt{m} \log n)$ worst case time per output update. We also show that, if insertions and deletions of edges are allowed, then similar *amortized* bounds hold.

1. INTRODUCTION

A fundamental and well-studied problem in computer science, operations research, and applied mathematics is finding shortest paths in a weighted graph. This problem arises frequently in many application settings, and its solution provides answers to other interesting problems as well (see, e.g., [1] for a wide variety of practical and theoretical applications of the shortest paths problem).

In this paper we study the single source shortest paths version of the problem in a general directed or undirected graph $G = (V, E)$ with positive real-edge weights. The best known static solution for this problem on directed graphs with n vertices and m edges is the $O(m + n \log n)$ implementation of Dijkstra's algorithm [7] that uses Fibonacci heaps [13]. In the case of undirected graphs the best solution is the one proposed by Thorup in [30], requiring $O(m)$ worst case time.

The dynamic version of the problem consists of maintaining the shortest paths information while the graph changes, without recomputing everything from scratch after each update on the graph. In such a framework the most general repertoire of update operations includes insertions and deletions of edges, update operations on the weight of edges, and insertions and deletions of isolated vertices. If arbitrary sequences of the above operations are allowed, then the problem is referred to as the *fully dynamic problem*; if only insertions (deletions) of edges are allowed, then the problem is referred to as the *incremental* (*decremental*) problem.

The study of shortest paths problems in a dynamic model is relevant in many application settings. For example, if the graph represents a communication network the edge updates reflect the real changes on the network as links that go up and down during its lifetime.

Various approaches to dynamic shortest paths problems have been considered in the literature [4, 10, 11, 15, 18, 19, 24, 25, 27]. In the case of planar graphs a fully dynamic solution for maintaining all-pairs shortest paths with unrestricted edge weights is proposed in [18], while in [19] a sublinear approximation scheme for the fully dynamic single-source shortest paths problem is given. Both these solutions use a topological partition of the graph based on recursive applications of the planar separator theorem [20] and the algorithms proposed are complex and far from being practical. In [5] the authors present efficient dynamic solutions for graphs with bounded treewidth when the weights of edges might change, but

without considering insertions and deletions of edges. An efficient solution for the all-pairs incremental problem has been proposed in [4], assuming that edge weights are restricted in the range of integers in $[1..C]$. A decremental solution for the single source version of the problem on general graphs, with n vertices and m edges, is known in the case of integer edge weights in $[1..C]$ [12], working in $O(nC)$ amortized time per deletion. Further results have been proposed in [10, 24, 25, 27].

To the best of our knowledge, neither a fully dynamic solution nor a decremental solution for the single source shortest paths problems that is asymptotically better (both in worst case and in amortized sense) than recomputing the new solution from scratch, if no restriction is assumed on the class of considered graphs, is known in the literature.

To characterize the performance of dynamic algorithms in these very hard dynamic cases, it seems to be very useful to take into account additional constraints, such as the structure of the output. On the other hand, there exist several application settings, such as incremental compilation, dataflow analysis, text editing, and graphical and/or interactive applications, where explicitly maintaining the solution to a given problem might be required. In this case explicit updates of a given data structure must be carried out after each input modification had been specified. For the above motivations, it seems reasonable to measure the performances of dynamic algorithms using the number of output updates required by each input modification, i.e., in terms of *output complexity* [15, 24–26].

1.1. *Output Complexity*

In [15], the authors of this paper propose to measure the output complexity of the single source shortest paths problem in the following way: given graph $G = (V, E)$ with source s, the *output information* consists of: (i) for any $x \in V$, the value of the distance of x from s; (ii) a shortest paths tree rooted at s. Let μ be an edge operation to be performed on G (insertion, deletion, or weight update), and let G' be the new graph after μ has been performed on G. The *set of output updates* $\delta(G, \mu)$, to be performed on the solution of the problem, is given by the set of vertices that either change their distance from the source in G'' or change their parent in the shortest paths tree, due to the input modification μ. The *number of output updates* caused by μ is the cardinality of $\delta(G, \mu)$.

This notion of output complexity has been also extended in [15] to sequences of updates. Namely, given a graph $G = (V, E)$, let $\sigma = \langle \mu_1, \mu_2, \ldots, \mu_h \rangle$ be a sequence of input modifications (insertions, deletions, or weight updates of edges); each input modification $\mu_i \in \sigma$ is performed on graph G_{i-1}, with $G_0 \equiv G$, and gives the new graph G_i. After

each input modification $\mu_i \in \sigma$ is specified, we are required to update the current output information.

Let $\delta(G_i, \mu_i)$ and $\Delta(G, \sigma)$ be the set of output updates caused by μ_i and by the whole sequence σ, respectively. The total number of output updates over the sequence σ is given by $|\Delta(G, \sigma)| = \sum_{\mu_i \in \sigma} |\delta(G_i, \mu_i)|$. Note that no algorithm can process σ performing explicit updates in less than $|\sigma| + |\Delta(G, \sigma)|$ time: this is the cost of an ideal algorithm that carries out each update to the input and output information of the problem in constant time.

If $A(G, \sigma)$ is the time required by algorithm A to process a sequence σ on G, with explicit updates, then algorithm A *requires* α *amortized time per output update* if, for any sequence σ, $A(G, \sigma) \leq \alpha \cdot |\Delta(\sigma)| + |\sigma| +$ constant.

A similar definition of output complexity has been introduced by Ramalingam and Reps [24–26]. They propose to measure the cost of dynamic graph algorithms as follows: Let $G = (V, E)$ be a graph with n vertices and m edges, let $\mathcal{P}$ be a property to be maintained on G, and let ρ be a set of input modifications (insertions, deletions, and weight update of edges) on G; the performance of a dynamic algorithm is measured as a function of the *extended size* of the output updates, denoted as $\| \rho \|$. The value of $\| \rho \|$ is given by the number $| \rho |$ of *affected* vertices, i.e., vertices that change their output value with respect to $\mathcal{P}$ as a consequence of the input modification ρ, *plus* the number of edges having at least one affected endpoint. Note that $\| \rho \|$ might be n times larger than the number of affected vertices. A similar notion has been used by other authors in [2].

1.2. *Previous Output Bounded Results*

Previous results concerning the dynamic single source shortest paths problem, using output complexity models, have been proposed in [15, 25]. In [25] Ramalingam and Reps provide a fully dynamic algorithm for general graphs. The worst case cost of a single edge update ρ is $O(\| \rho \| \log \| \rho \|)$ ($O(\| \rho \| + | \rho | \log | \rho |)$ when the algorithms are implemented with Fibonacci heaps).

In [15] the authors of this paper separately consider the incremental and the decremental problem, proposing output bounded solutions that explicitly maintain the distances and a shortest paths tree of a graph G with n vertices. The decremental solution works only for planar graphs, and each deletion requires $O(\log n)$ amortized time per output update. The incremental solution works for any graph and its complexity depends on the existence of a k-2{bounded accounting function for G. In particular, for any incremental sequence of updates, 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 (x, y) determines either vertex x or vertex y as the *owner* of the edge; A is k-bounded if k is the maximum over all vertices x of the cardinality of the set of edges owned by x.

An analogous notion can be captured by considering the *orientation* of edges in an undirected graph. In [6] an orientation of an undirected graph $G = (V, E)$ is represented as a function ω which replaces each $(x, y) \in E$ by a directed edge $x \to y$ or $y \to x$. If $deg^+_\omega(x)$ is the out-degree of vertex x under the orientation ω, then ω is k-bounded if, for each $x \in V$, $deg^+_\omega(x) \leq k$. The proof of the existence of a 3-bounded orientation for each planar graph and a linear time algorithm for finding such an orientation have been provided in [6]. The same bound was proved in [23], but the corresponding algorithm required more than linear time.

It is immediate that the notions of k-bounded orientation and k-bounded accounting function is an undirected graph coincide. Since we deal with directed and undirected graphs, for the sake of clarity in the final section we use the accounting function terminology.

The semidynamic solutions proposed in [15] are both based on Dijkstra's algorithm [7], and in both cases the results are obtained using the monotonicity of the required solution. The data structures of [15] are updated by using a *lazy approach*, necessary to defer the required updates after an edge modification takes place as much as possible.

1.3. *Results of the Paper*

The algorithms proposed in this paper explicitly maintain the distances and a shortest paths tree from a source vertex s, for a graph G with n vertices, m edges, and positive real edge weights, under arbitrary sequences of edge updates. Queries can be answered in optimal time, that is, in $O(1)$ worst case time to obtain the distance between s and a given vertex and in $O(l)$ worst case time to return a shortest path of l edges. The cost of the update operations is given as a function of the number of output updates, by using the notion of a k-bounded accounting function:

- if we consider only weight updates of edges, each output update requires $O(k \log n)$ worst case time;
- if we consider a sequence σ of edge updates on a graph G, including insertions and deletions, besides weight updates, then each output update requires $O(k \log n)$ amortized time when the graph G_σ, containing the edges of G and those inserted by σ, has a k-bounded accounting function.

The value of the parameter k defined above for any graph G can be bounded in different ways using structural properties of the graph. In particular, following [15] we have

$k = O(\sqrt{m})$	for general graphs with m edges;
$k = O(1 + \sqrt{\gamma})$	for graphs with genus γ;
$k \leq 3$	for planar graphs;
$k \leq a$	for graphs with arboricity a;
$k \leq d$	for graphs with maximum degree d;
$k \leq t$	for graphs with treewidth t;
$k \leq p$	for graphs with pagenumber p.

Note that, in the case of insertions and deletions of edges, the notion of a k-bounded accounting function is useful only to bound the running times of our algorithms, but does not affect their behavior (in fact the algorithms do not need to know the value k or any upper bound on it).

The proposed solutions compare favorably to the results of Ramalingam and Reps provided in [25] and described at the beginning of the previous section. We also improve on the semidynamic results proposed in [15] as follows:

1. we use different data structures and algorithmic techniques and update the whole data structures, avoiding in this way the use of lazy updates;

2. we extend the case of deletions to general graphs (the previous decremental solution worked only for planar graphs);

3. we propose fully dynamic algorithms and data structures for the single source shortest paths problem with the same running times of the previous semidynamic algorithms;

4. we propose a dynamic solution handling only *weight-increase* and *weight-decrease* operations on the edges that works in $O(k \log n)$ *worst case* time per output update.

We remark that the algorithms proposed in this paper are, at the same time, efficient (at least in terms of output complexity) and practical; in fact we use simple data structures (only linked lists and priority queues) that are suitable for a straightforward implementation. The algorithms have been implemented on top of the LEDA library [22], together with the algorithms in [25] (see Section 5 for the details concerning the experimental comparison of these algorithms).

The paper is organized as follows. In Section 2 some preliminaries are given together with a characterization of graphs in terms of k-bounded

accounting functions. In Section 3 our algorithms handling edge weight updates are described, and the worst case bounds for this case are proved. In Section 4 the algorithms are extended to handle also insertions and deletions of edges, with amortized time bounds. Finally, in Section 5 conclusions and open problems are given.

2. PRELIMINARIES

In the following we assume the standard graph terminology as contained, for example, in [17]. We use the following notation: let $G = (V, E)$ be a weighted undirected graph with n vertices and m edges, and let $s \in V$ be a fixed *source*. To each $(x, y) \in E$, a real positive weight $w_{x,y}$ is associated. Let $d: V \to \Re^+$ be a distance function giving, for each $x \in V$, the minimum distance of x from s, let $T(s) = (V_T, E_T)$ be a shortest paths tree of G rooted at s, and, for any $x \in V$, let $T(x)$ be the subtree of $T(s)$ rooted at x. Every $x \in V$ has one parent (except for source s), denoted as *parent*(x), and a set of children, denoted as *children*(x), in $T(s)$. An edge (x, y) is a *tree edge* if $(x, y) \in E_T$; otherwise it is a *nontree edge*.

Given a graph G with positive real edge weights we want to maintain the tree $T(s)$ and the distances of vertices from s in a fully dynamic environment, where an arbitrary sequence of the following operations is performed on G:

(a) *distance*(x): reports the current distance between s and vertex x;

(b) *min-path*(x): reports a shortest path between s and vertex x;

(c) *insert*(x, y, w): inserts edge (x, y) with weight w;

(d) *delete*(x, y): deletes edge (x, y);

(e) *weight-increase*(x, y, ϵ): increases by quantity ϵ the weight of edge (x, y);

(f) *weight-decrease*(x, y, ϵ): decreases by quantity ϵ the weight of edge (x, y).

After each edge modification has been carried out, we are required to compute a new shortest paths tree and the new distance value for any $x \in V$.

Let G' be the graph obtained from G after an edge operation. For each $z \in V$, $d(z)$ and $d'(z)$ denote the values of the distance of z before and after an edge update, respectively. Furthermore, the parent of z and a new shortest paths tree in G are denoted as *parent*(z) and $T'(s)$, respectively.

Note that our data structures allow us to perform insertions and deletions of isolated vertices efficiently also. Here we consider only undirected

graphs, the extension to directed ones being straightforward. The performance of the proposed algorithms is evaluated in the output complexity model proposed in [15] and described in Section 1.1.

2.1. *Graphs Having a k-bounded Accounting Function*

In this section we briefly summarize the formal characterization of graphs in terms of k-bounded accounting functions that has been presented in [15].

DEFINITION 2.1 [15]. Let $G = (V, E)$ be a graph. An accounting function for G is any function $A : E \to V$ such that, for each $(x, y) \in E$, $A(x, y)$ is either x or y; it is called the owner of (x, y). A is k-bounded if, for each $x \in V$, the set $A^{-1}(x) = \{(x, y) \mid A(x, y) = x\}$ of the edges owned by x has cardinality at most k.

It is trivial to define a k-bounded accounting function for graphs with arboricity k and for graphs with maximum degree k. Since the class of graphs with bounded arboricity includes the graphs with bounded treewidth, there exists a k-bounded accounting function for graphs whose treewidth is at most k (see [3] for the definition of treewidth of a graph). The existence of a 3-bounded accounting function for planar graphs is due to the result, provided in [23], that the arboricity of a planar graph is smaller than or equal to 3. This implies that there exists a k-bounded accounting function for graphs with *pagenumber* equal to k. By using the above results and the fact that the pagenumber of a genus γ graph is $O(\sqrt{\gamma})$ [21], it follows that there exists an $O(\sqrt{\gamma})$-bounded accounting function for graphs with genus γ. Furthermore, since the genus of a graph is always less than its number of edges, it follows that a graph with m edges has an $O(\sqrt{m})$-bounded accounting function.

The minimum k such that a graph G has a k-bounded accounting function can be computed in polynomial time using the decomposition theorem of Nash-Williams [23] and the results of Edmonds [8, 9] (see also [28]).

3. UPDATING EDGE WEIGHTS

In this section we focus on the case in which changes to edge weights are the only operations allowed. Without loss of generality, we assume that G is connected.

The proposed procedures are based on Dijkstra's algorithm [7]. Namely, each procedure uses a priority queue containing vertices of G, in which the priority of each vertex z is the length of the shortest path from s to z

found so far. In Dijkstra's algorithm, when a vertex v is permanently labeled and its distance from s has been computed, *all* the neighbors of v are considered for possible improvements in their current shortest paths from s. We show that it is possible to consider only a subset of the neighbors of v, when its new distance from s has been computed after an edge modification.

Let us sketch the strategy used to handle weight-decrease and weight-increase operations.

In the case of a weight-decrease operation the number of output updates is given by the number of vertices that change the distance from s (in fact, if our algorithm changes the parent of a vertex z in the shortest paths tree after a weight-decrease operation, then z surely decreases its distance from s). If decreasing the weight of edge (x, y) decreases the distance of vertex y from s, a global priority queue C is used, as in Dijkstra's algorithm, to find new distances from s in nondecreasing order. Unlike Dijkstra's algorithm, when a vertex z is dequeued from C and its new distance $d'(z)$ is computed, not all edges leaving z are scanned. In particular, suppose that z has a large number of neighbors; we are interested in those edges leaving z that lead to vertices that, by choosing z as a new parent in the shortest paths tree, would improve their distance from s to a smaller value, with respect to the current one. We will show how z can guess the right edges (z, q) to be scanned, to deliver the *good news*: "there exists a shorter path from s to q."

In the case of a weight-increase operation the number of output updates is given by the number of vertices that either change the distance from s or must change the parent in $T(s)$, so that the new shortest path from s is as good as the previous one. The algorithm proposed in this case works in two phases: first it finds all the vertices that need to be updated, and then it computes the new distances and a new shortest paths tree.

Let us consider a vertex z that increases its distance from the source.

(i) We need to find the new shortest path from s to z. In this case we have to choose the neighbor of z that determines the minimum distance from s to z, and that will be its (possibly new) parent in the shortest paths tree.

(ii) We need to find the neighbors of z that also increase their distance from s. Also, in this case we would like to guess the right edges (z, q) to be scanned to propagate the *bad news*: "the distance from s to q increases."

We will show how any vertex z that increases its distance from the source can guess the right edges to be scanned to solve both (i) and (ii) above.

To efficiently deal with all the above problems, we use the following definition.

DEFINITION 3.1. Let $G = (V, E)$ be a weighted graph. The *backward level* (*forward level*) of edge (z, q) and of vertex q, relative to vertex z, is the quantity $b\_level_z(q) = d(q) - w_{z,q}$ ($f\_level_z(q) = d(q) + w_{z,q}$).

The notions of backward_level and forward_level are the same as those of *temperature* and *D-temperature*, respectively, that have been introduced in [15]. The intuition behind Definition 3.1 is that the level of an edge (z, q) provides information about the shortest available path from s to q passing through z. For instance, let us suppose that, during a weight-decrease operation, $d'(z)$ decreases below $b\_level_z(q)$; i.e., there exists an edge (z, q) such that $b\_level_z(q) - d'(z) = d(q) - w_{q,z} - d'(z) > 0$; i.e., $d(q) > d'(z) + w_{q,z}$. This means that we have found a shorter path to q than the current shortest path to q. In this case, scanning the edges (z, q) in nonincreasing order of b_level ensures that only the *right* edges are considered, i.e., edges (z, q) such that also q decreases the distance from s.

In the case of a weight-increase operation, let us suppose that the operation is performed on an edge in the current shortest path from s to a vertex z. Let l be the new length of that path after the weight-increase operation. Suppose now that there exists a neighbor q of z, such that l increases above $f\_level_z(q)$; i.e., $f\_level_z(q) - l = d(q) + w_{q,z} - l < 0$; i.e., $l > d(q) + w_{q,z}$. This means that we can find an alternate shorter path from s to z in G' that passes through q. If we scan the edges (z, q) in nondecreasing order of $f\,level$, then only the edges giving paths from s to z that are shorter than the current ones are considered.

To apply the above strategy, we need to maintain explicitly the information on the b_level and the f_level for all the neighbors of each vertex. This might require the scanning of each edge adjacent to an updated vertex. In the worst case this number might be n times larger than the number of output updates.

To bound the number of edges scanned by our algorithms each time that a vertex is updated, we partition the set of edges adjacent to each vertex into two subsets: any edge (x, y) has an *owner*, denoted as *owner*(x, y), that is either x or y. For each vertex x, *ownership*(x) denotes the set of edges owned by x, and *not-ownership*(x) denotes the set of edges with one endpoint in x, but not owned by x. If G has a k-bounded accounting function then, for each $x \in V$, *ownership*(x) contains at most k edges. Furthermore, the edges in *not-ownership*(x) are stored in two priority queues as follows:

1. B_x is a max-based priority queue; the priority of edge (x, y) (of vertex y) in B_x, denoted as $b_x(y)$, is the computed value of $b\_level_x(y)$.

2. F_x is a min-based priority queue; the priority of edge (x, y) (of vertex y) in F_x, denoted as $f_x(y)$, is the computed value of $f\_level_x(y)$.

Before processing a sequence σ of edge modifications on G, we compute a distance value $d(x)$ for each vertex x in G, a shortest paths tree $T(s)$, and an initial ownership function for G. Then, for each vertex x and for each edge (x, y) not owned by x, the data structures are initialized by computing $b_x(y) = b\_level_x(y)$ and $f_x(y) = f\_level_x(y)$ (we will see that both these conditions are restored after the execution of any proposed procedure).

In the sequel we will assume that, for any $x \in V$, $D(x)$ stores the distance from s to x computed by the proposed algorithms. We introduce this additional notation (beyond $d(x)$ and $d'(x)$) for the sake of clarity. In fact, in the correctness proofs we distinguish between the actual distances and the computed distances of vertices. In particular, we assume that, before the execution of any update procedure, $D(x) = d(x)$ for each $x \in V$, and we will prove that $D(x) = d'(x)$ upon termination of the procedures. Analogously, $P(x)$ stores the parent of x in the current shortest paths tree computed by the proposed algorithms.

3.1. *Decreasing the Weight of an Edge*

Suppose that the weight of edge (x, y) is decreased by a positive quantity ϵ. Without loss of generality, we assume that $d(x) \leq d(y)$. If $d'(y) < d(y)$, then all vertices that belong to $T(y)$ decrease their distance from s. On the other hand, the new subtree $T'(y)$ may include other vertices not contained in $T(y)$. When a vertex z decreases its distance from s, an edge (z, q) leaving z is scanned if and only if one of the following two conditions arises:

(i) z is the owner of edge (z, q): in this case we say that (z, q) is *scanned by ownership*;

(ii) q is the owner of edge (z, q) and $b\_level_z(q) > d'(z)$; in this case we say that (z, q) is *scanned by priority* and that both vertex q and edge (z, q) are *high* for z.

Note that all the edges of $T(y)$ are scanned after a weight-decrease operation on (x, y). In fact, if $(z, q) \in T(y)$, then z decreases its distance from s; since also q decreases the distance from s of the same quantity of z, (z, q) is scanned either by priority or by ownership from z.

To bound the number of edges scanned by priority when the new distance $d'(z) < d(z)$ is computed for z, we use the information stored in heap B_z by selecting only *high* edges for z, i.e., edges whose priority in B_z is greater than $d'(z)$.

We are now ready to present the algorithm `Decrease` whose pseudocode is given in Fig. 1. Without loss of generality assume that the requested update on edge (x, y) improves the distance from s to y. Remember that, at the beginning of the procedure `Decrease`, $D(z) = d(z)$, for each vertex z. The procedure is divided into three steps. The first two steps perform the obvious preprocessing; the last step updates the distances of vertices from the source in a way similar to Dijkstra's algorithm, applied to the subgraph induced by the updated vertices. In what follows we describe in details the three steps above.

Step 1 checks whether the weight of edge (x, y) has become nonpositive (line 2). If the new weight of (x, y) is positive, then the data structures stored at vertices x and y are properly updated, depending on the owner of (x, y), by procedure `Update_Local`. In particular, `Update_Local`(x, y) checks whether $owner(x, y) = x$ and, in that case, updates the priority of ix in B_y and F_y; otherwise, it updates the priority of y in B_x and F_x. Afterward, line 5 checks whether the distance of y from s is decreased. In this case Step 2 is performed; otherwise, the procedure `Decrease` halts.

```
    procedure Decrease(x, y : vertex; ε : positive_real)
1.  begin {suppose wlog that d(x) ≤ d(y)}
    Step 1
2.      if ε ≥ w_{x,y} then return ERROR {the weight of (x, y) becomes non-positive}
3.      w_{x,y} ⟵ w_{x,y} − ε
4.      Update_Local(x, y)
5.      if D(y) ≤ D(x) + w_{x,y} then EXIT {no distance improves}
    Step 2
6.      D(y) ⟵ D(x) + w_{x,y}
7.      P(y) ⟵ x
8.      C ⟵ ∅ {initialize an empty heap C}
9.      Enqueue(C, ⟨y, D(y)⟩)
    Step 3
10.     while Non_Empty(C) do
11.        begin
12.           ⟨z, D(z)⟩ ⟵ Extract_Min(C)
13.           for each (z, v) s.t. (z, v) ∈ ownership(z) or (z, v) ∈ not-ownership(z) and v is high do
14.              begin
15.                 if (z, v) ∈ ownership(z) then update b_v(z) and f_v(z)
16.                 if D(v) > D(z) + w_{z,v} then
17.                    begin
18.                       D(v) ⟵ D(z) + w_{z,v}
19.                       P(v) ⟵ z
20.                       Insert_or_Improve(C, ⟨v, D(z) + w_{z,v}⟩)
21.                    end
22.              end
23. end
```

FIG. 1. Decrease by quantity ϵ the weight of edge (x, y).

Step 2 first computes $D(y)$ and $P(y)$ as $D(x) + w_{x,y}$ and x, respectively; then it inserts vertex y into the (initially empty) heap C with priority given by $D(x) + w_{x,y}$; by calling the procedure `Enqueue`$(C, \langle y, D(x) + w_{x,y} \rangle)$.

Step 3 computes a new shortest paths tree for G' in a way similar to Dijkstra's algorithm, by recomputing only the portion of $T'(s)$ rooted at y. In particular, while heap C is not empty, the vertex z with minimum priority $D(z)$ is extracted from C by procedure `Extract_Min`(C). When z is extracted from C its priority is the shortest distance from s to z in G'. Then the procedure propagates the new distance $D(z) = d'(z)$ along each edge (z, v) such that either $(z, v) \in$ *ownership*(z) or $(z, v) \in$ *not-ownership*(z) and v is high for z (line 13). This is done as follows. The edges $(z, v) \in$ *not-ownership*(z) such that v is high for z are found by repeatedly extracting vertices from B_z by nonincreasing priority until a vertex that is not high is found. Instead, if $(z, v) \in$ *ownership*(z), then the correct priority of z in heaps B_v and F_v is computed to guarantee that the information stored in the data structures is always updated before and after the execution of any weight-decrease operation.

For each edge (z, v) considered in line 13, if the length $D(z) + w_{z,v}$ of the shortest path from s to v passing through z is smaller than $D(v)$ (line 16), then $D(v)$ and $P(v)$ are updated as $D(z) + w_{z,v}$ and z, respectively. Then the procedure `Insert_or_Improve`$(C, \langle v, D(z) + w_{z,v} \rangle)$ is called, and two possible cases may arise: (i) if $v \notin C$, then v is inserted in C with priority equal to $D(z) + w_{z,v}$; (ii) if $v \in C$, then the priority of v in C is updated to the value $D(z) + w_{z,v}$.

3.2. *Increasing the Weight of an Edge*

Suppose that the weight of an edge (x, y) is increased by a positive quantity ϵ. Clearly, if (x, y) is a nontree edge, then no update is required. Let us suppose that (x, y) is a tree edge and assume, without loss of generality, that $d(x) < d(y)$. It is easy to see that: (i) for each vertex $z \notin T(y)$, $d'(z) = d(z)$; (ii) there exists a new shortest paths tree $T'(s)$ such that, for each vertex $z \notin T(y)$, *parent*$'(z) =$ *parent*(z).

We borrow from [15] the idea of coloring the vertices of the graph as follows:

- $q \in V$ is *white* if q changes neither the distance from s nor the parent in $T(s)$;
- $q \in V$ is *red* if q increases the distance from s; i.e., $d'(q) > d(q)$;
- $q \in V$ is *pink* if q preserves its distance from s, but it must replace the old parent in $T(s)$ (i.e., q is pink if $d'(q) = d(q)$, but *parent*$'(q) \neq$ *parent*(q)).

Initially all vertices are white. It is easy to verify that, if a vertex q is pink or red, then either q is a child or a red vertex in $T(s)$ or $q \equiv y$; furthermore, if q is red, then all the children of q in $T(s)$ must be updated and will be either pink or red; finally, if q is pink or white, then all the other vertices in $T(q)$ are white.

After an edge weight increases, the number of output updates is the number of *red* and *pink* vertices. Therefore, to bound the running time of a weight-increase operation as a function of the number of output updates, it is not possible to search the whole subtree $T(y)$. In fact, $T(y)$ may contain a pink vertex z that can choose a new parent not belonging to $T(y)$ that determines in G' the same distance that z had in G: in this case all the vertices in $T(z)$ do not require any update. Furthermore, given a vertex z, we define its *best nonred neighbor* as the normed vertex q adjacent to z, with minimum value of the quantity $flevel_z(q) - d(z)$. Therefore, the best nonred neighbor of z is the vertex that represents the best alternative parent for z among all nonred neighbors of z in G. Remember that, at the beginning of the procedure `Increase`, $D(z) = d(z)$, for each vertex z.

Now we present in detail the procedure `Increase`, given in Fig. 2. It is divided into three main steps: Step 1 updates the local data structures at vertices x and y and checks whether some distance changes; Step 2 colors the vertices of the graph; Step 3 computes the new distances for the *red* vertices. In what follows we describe in detail the three steps above.

In Step 1 procedure `Update_Local`(x, y) properly updates the priority of edge (x, y) in the two priority queues stored either at vertex x or at vertex y, depending on the owner of (x, y), as already described in Section 3.1. Then, if (x, y) is a nontree edge, procedure `Increase` halts; otherwise, y is inser ted in a heap M with priority $D(y)$, i.e., its current distance from the source. M will contain vertices that will be colored either pink or red. Vertices are extracted from M in nondecreasing order of their distance in G.

In Step 2 vertices are colored by repeatedly extracting from M the vertex with minimum priority (line 9). This step is the same as Step 3.a of the procedure `Delete` given in [15]. We report and comment on it for the sake of completeness. In particular, when a vertex z is extracted from M, we check whether there exists a nonred neighbor q of z such that $D(q) + w_{q,z} = D(z)$ (line 10), because this condition guarantees a pink color for z. This is implemented as follows. We first search for such a neighbor by scanning the edges owned by z (we say that these edges are scanned by ownership; if it exists, then q becomes the new parent of z in the shortest paths tree (z is pink). Otherwise, we select the best nonred neighbor not owned by z by repeatedly extracting elements from F_z (the

```
    procedure Increase(x, y : vertex; ε : positive_real)
1.  begin
    Step 1
2.      Update_Local(x, y)
3.      if (x, y) is a non-tree edge then EXIT {no distance increases}
4.      M ⟵ ∅ {initialize empty heap M}
5.      Q ⟵ ∅ {initialize empty heap Q}
6.      Enqueue(M, ⟨y, D(y)⟩)
    Step 2
7.      while Non_Empty(M) do
8.          begin
9.              ⟨z, D(z)⟩ ⟵ Extract_Min(M)
10.             if there is a nonred neighbor q of z such that D(q) + w_{q,z} = D(z)
11.                 then P(z) ⟵ q  {z is pink}
12.                 else begin
13.                         color(z) ⟵ red
14.                         for each v ∈ children(z) do Enqueue(M, ⟨v, D(v)⟩)
15.                     end
16.         end
    Step 3.a
17.     for each red vertex z do
18.         begin
19.             if z has no nonred neighbor
20.                 then begin
21.                         D(z) ⟵ +∞
22.                         P(z) ⟵ Null
23.                     end
24.                 else  begin
25.                         let u be the best nonred neighbor of z
26.                         D(z) ⟵ D(u) + w_{u,z}
27.                         P(z) ⟵ u
28.                     end
29.             Enqueue(Q, ⟨z, D(u) + w_{u,z}⟩)
30.         end
    Step 3.b
31.     while Non_Empty(Q) do
32.         begin
33.             ⟨z, D(z)⟩ ⟵ Extract_Min(Q)
34.             for each edge (z, v) ∈ ownership(z) do update b_v(z) and f_v(z)
35.             for each edge (z, h) such that h is red do
36.                 begin
37.                     if D(z) + w_{z,h} < D(h) then
38.                       begin
39.                         D(h) ⟵ D(z) + w_{z,h}
40.                         P(h) ⟵ z
41.                         Heap_Improve(Q, ⟨h, D(z) + w_{z,h}⟩)
42.                       end
43.                 end
44.         end
45.     restore the original white color for all the red vertices
46. end
```

FIG. 2. Increase by quantity ϵ the weight of edge (x, y).

edges considered in this phase are scanned by priority). If no nonred neighbor is found, then z is colored red. Otherwise, let q be the best nonred neighbor of z; if $D(q) + w_{q,z} = D(z)$, then z does not change its distance from s, but only its parent in $T(s)$ (z is pink); otherwise, it is colored red. If z has been colored red, then all the children of z in $T(s)$ will be either pink or red; hence, each child v of z in $T(s)$ is inserted in M with priority provided by $D(v)$.

If z is *pink* and q satisfies $D(q) + w_{q,z} = D(z)$, then $D(q) < D(z)$. Since vertices are extracted from M in nondecreasing order of their distance in G, then q will not be processed later on; i.e., q's color will not be changed later on, and hence z's color is correct.

The new shortest paths for the *red* vertices are computed in Step 3.b by using a heap Q that is initialized during step 3.a as follows. Each red vertex z is inserted in Q with priority given by the distance of its best nonred neighbor q plus the weight of edge (q, z) if q exists, by $+\infty$ otherwise (line 29). In the former case, the information on a path from s to z in G' passing through q, whose length is $D(q) + w_{q,z}$, is stored into Q. The values $D(z)$ and $P(z)$ for each *red* vertex z, are initialized accordingly (see lines 19–28).

In Step 3.b the new distances of the red vertices are computed by applying a procedure similar to Dijkstra's algorithm. In particular, red vertices are extracted from Q in order of nondecreasing priority, and the priority of each vertex represents the length of the shortest path fond so far. If z is the vertex with minimum priority $D(z)$ in Q, then $D(z)$ represents the length of a shortest path from s to z in G'.

The information on the new distance of z is propagated along the edges $(z, v) \in \mathit{ownership}(z)$ to update the priority of z in B_v and F_v, respectively (line 34). The information of z's new distance might also allow the priorities of vertices in Q to be updated. Namely, for each edge (z, h) such that also h is red, the algorithm checks whether $D(z) + w_{h,z} < D(h)$ (line 37). If this is the case, then a path from s to h that is shorter than the shortest path found so far has been determined. The values $D(h)$ and $P(h)$ are properly updated and the priority of h in Q is given the value $D(z) + w_{z,h}$, by calling procedure `Heap_Improve`$(Q, \langle h, D(z) + w_{z,h}\rangle)$ (line 41).

When Q is empty all distances have been computed. The procedure terminates by restoring the white color for all the red vertices.

3.3. *Correctness Analysis*

We recall that, for each vertex z, $d(z)$ and $d'(z)$ denote the distance of z from s, before and after an edge operation, respectively; moreover, the current value stored in the data structure as the distance of z from s is denoted as $D(z)$.

The following properties hold before the execution of the update procedures:

(P1) For each $q \in V$, the length of the shortest path from s to q is correctly stored; that is, $D(q) = d(q)$.

(P2) $T(s)$ is a shortest path tree rooted at s for G; i.e., for each vertex z in G, the path from s to z in $T(s)$ is a shortest path.

(P3) For each vertex z, and for each edge (z, q) not owned by z, the following is true: $b_z(q) = b\text{–}level_z(q)$ and $f_z(q) = f\text{–}level_z(q)$.

To prove the correctness of the update procedures we will show that Properties (P1)–(P3) hold after the execution of `Decrease` and `Increase`.

LEMMA 3.1. *Let $G = (V, E)$ be a graph; if a weight-decrease operation is performed on $(x, y) \in E$ and Properties* (P1)–(P3) *hold, then Properties* (P1) *and* (P2) *hold after the execution of the procedure* `Decrease`.

Proof. Suppose that the weight-decrease operation on edge (x, y) improves the distance of y from s. The following facts follow straightforwardly by inspecting the procedure `Decrease`:

(a) For each vertex z, $D(z)$ can only decrease; i.e., $D(z) \leq d(z)$.

(b) For each vertex z, $D(z)$ is an upper bound on $d'(z)$; i.e., $D(z) \geq d'(z)$.

Let v be the vertex nearest to s for which a wrong distance might be computed; i.e., we assume that, for each vertex w such that $d'(w) < d'(v)$, $D(w) = d'(w)$. Let us consider now an edge (z, v) that belongs to a shortest path from s to v after the *weight-decrease* operation on edge (x, y). By the above discussion it follows that $D(z) = d'(z)$. This implies that $d'(z) + w_{z,v} = d'(v) < D(v) \leq d(v)$. If we are able to show that $D(v) = d'(v)$, then we have proved that (P1) holds also after the weight-decrease operation. Two cases may arise:

1. $d(z) = d'(z)$. Since distances can only decrease, we have $d'(v) \leq d(v)$. Furthermore, $d'(v) = d'(z) + w_{z,v} = d(z) + w_{z,y} \geq d(v)$, and hence $d'(v) = d(v)$. By hypothesis and by conditions (a) and (b) we derive that $d'(v) \leq D(v) \leq d(v)$. Since $d'(v) = d(v)$, it follows that $d'(v) = D(v)$.

2. $d(z) > d'(z)$. It is sufficient to show that the algorithm updates the priority of v in C to the value $d'(z) + w_{z,v}$. By hypothesis, when z is extracted from C the value $D(z) = d'(z)$ has been correctly computed in line 18. Let us consider now the behavior of the algorithm immediately after the extraction of z from Q (line 12).

The algorithm scans all the edges (z, v) such that either $(z, v) \in$ *ownership*(z) or $(z, v) \in$ *not-ownership*(z) and v is high for z (line 13). The high vertices in *not-ownership*(z) are found by extracting vertices from B_z. If $v \notin$ *ownership*(z), then, by hypothesis, the priority of v in B_z is correct (Property (P3) holds before the update operation), and v is correctly considered high for z.

For each edge (z, v) scanned in line 13, the algorithm checks whether $D(z) + w_{z,v} < D(v)$. If this is the case, then procedure `Insert_or_Improve`$(C, \langle v, D(z) + w_{z,v} \rangle)$ is called that sets the priority of v in C to the value $D(z) + w_{z,v} = d'(z) + w_{z,v}$ (line 20). If $D(z) + w_{z,v} = D(v)$, then the priority of v in C has been previously updated to the value $D(z) + w_{z,v}$; i.e., a shortest path from s to v in G' has been already found by the algorithm. Since the priority of v in C can never assume a value smaller than $d'(v)$, and it is not possible that v has been previously extracted from C, then Property (P1) holds after the *weight-decrease* operation; i.e., $D(v) = d'(v)$.

We have shown that, when a vertex v decreases its distance from s, it is inserted in heap C with the correct priority. This implies that, when vertex v is extracted from C with priority $D(v)$, this parameter represents the new minimum distance of v from s. The value $D(v)$ has been correctly computed as $D(z) + w_{z,v}$ in line 18 before extracting v from C. Since the parent of v in the new shortest paths tree has been computed as z in line 19, Property P2 holds after the execution of procedure `Decrease`. ▮

LEMMA 3.2. *Let $G = (V, E)$ be a graph; if a weight-decrease operation is performed on $(x, y) \in E$ and Properties* (P1)–(P3) *hold, then after the execution of procedure* `Decrease` *Property P*3 *holds.*

Proof. Let z be a vertex that changes its distance from s as a consequence of the weight-decrease operation, and let (z, q) be an edge adjacent to z. By Lemma 3.1, when z is extracted from C in line 12, the correct distance of z from s has been computed. The priority of z has to be updated in heaps B_q and F_q only when z is the owner of edge (z, q). Procedure `Decrease` correctly updates this information in line 15 after the extraction of z from C. ▮

The following lemma states the correctness of the coloring step (Step 2) of the procedure `Increase`. The lemma is analogous to Lemma 5.1 in [15] and hence we omit the proof.

LEMMA 3.3. *Let $G = (V, E)$ be a graph; if a weight-increase operation is performed on $(x, y) \in E$ and properties* (P1)–(P3) *hold, then Step* 2 *of the procedure* `Increase` *correctly colors all the vertices of G.*

To prove the correctness of the computed distances we need the following lemma.

LEMMA 3.4. *Step* 3.*b of the procedure* `Increase` *computes values* $D(z)$ *for the red vertices in nondecreasing order.*

Proof. Assume by contradiction that the lemma is not true; let z' be the first vertex that does not satisfy the nondecreasing order when it is extracted from Q at line 33.

Let z be the vertex extracted from Q immediately before z'; by contradiction $D(z') < D(z)$. When z' is extracted from Q, the value $D(z')$ has been computed at line 39 as follows: $D(z') = D(q) + w_{q,z'}$, where q satisfies $D(z') > D(q) + w_{q,x'}$ and hence $D(q) < D(z') < D(z)$. This implies that q has been extracted from Q before z and that the priority of z' in Q has been correctly updated in line 41 to the value $D(q) + w_{q,x'}$ before the extraction of z from Q. Since by hypothesis $D(z) > D(q) + w_{q,z'}$, this contradicts the fact that z is extracted from Q before z'. ∎

The following lemma proves that distances are correctly computed.

LEMMA 3.5. *Let* $G = (V, E)$ *be a graph, if a weight-increase operation is performed on* $(x, y) \in E$ *and Properties* (P1)–(P3) *hold, then Properties* (P1) *and* (P2) *are satisfied at the end of the procedure* `Increase`.

Proof. Lemma 3.3 proves that the set of vertices colored red and enqueued in Q is exactly the set of vertices that increase their distance from the source. This implies that, for each pink or white vertex z, $D(z) = d(z) = d'(z)$. Observe that, when a pink vertex z is detected (line 10), the new parent of z is correctly computed in line 11. Since a pink vertex does not change its distance from s, Properties (P1) and (P2) trivially hold for *pink* vertices after the execution of the procedure `Increase`.

The rest of the proof is divided into two parts: we first prove that, for each *red* vertex z, the value $D(z)$, computed at line 39, satisfies $D(z) \geq d'(z)$ and then that $D(z) \leq d'(z)$.

To prove that $D(z) \geq d'(z)$ it is sufficient to prove the following statement: during the execution of Step 3.b, if the priority of z in Q is not $+\infty$, then it is equal to the length of a path from s to z in G'.

We first show that the statement is true at the end of Step 3.a. When z is inserted into Q, its priority is $D(q) + w_{q,z} = d(q) + w_{q,z} > D(z)$, where q is the best nonred neighbor of z (see line 29). Since the color of q is correct, $d'(q) = d(q)$, and the priority of z corresponds to the length of a path from s to z also in G'.

Assume now that the statement is not true during the execution of Step 3.b. Let z be the vertex in Q with minimum priority $D(z) = D(v) + w_{v,z}$ such that the statement is not true; i.e., $D(z)$ is not the length of a path from s to z in G'. Since the statement is true for z at the beginning of

Step 3.b, this implies that the priority of z in Q has been changed to the value $D(v) + w_{v,z}$ in Step 3.b at line 41 and that edge (v, z) belongs to G'. By hypothesis, when v is extracted from Q the value $D(v)$ is the length of a path from s to v in G'. Since edge (v, z) belongs to G', $D(z)$ is the length of a path from s to z in G', which contradicts the hypothesis.

To prove that $D(z) \leq d'(z)$ we proceed by contradiction and assume that z is the first vertex such that, when it is extracted from Q, the computed distance $D(z)$ is not optimal; i.e., $D(z) > d'(z)$. Let q^* be a parent of z in an optimal solution (i.e., $d'(z) = d'(q^*) + w_{q^*,z}$); by Lemma 3.4 and by hypothesis $d'(q^*) < d'(z)$. We distinguish two cases.

Case 1. q^* is not colored red by the algorithm. Since q^* is correctly colored, the priority of z in Q at the end of Step 3.a is at most $d(q^*) + w_{q^*,z} = d'(z)$. Since $d(q^*) = d'(q^*)$ and the priority of z in Q cannot increase during Step 3.b, we contradict the fact that $D(z) > d'(z)$.

Case 2. q^* is colored red by the algorithm. Since z is the first vertex for which $D(z) > d'(z)$, $D(q^*) = d'(q^*)$ and $D(z) > D(q^*) + w_{q^*,z} = d'(z)$. This implies that q^* is extracted from Q before z. Moreover, when $D(q^*)$ has been correctly computed, the procedure `Heap_Improve` considers edge (q^*, z) and updates the priority of z in Q to the value $D(q^*) + w_{q^*,z}$ (line 41), contradicting the fact that $D(z)$ is not optimal. ▮

LEMMA 3.6. *Let $G = (V, E)$ be a graph; if a weight-increase operation is performed on $(x, y) \in E$ and Properties* (P1)–(P3) *hold, then Property* (P3) *holds after the execution of procedure* `Increase`.

Proof. Suppose that Properties (P1)–(P3) hold before the execution of the procedure `Increase`. Let z be any vertex that changes the distance from s after the weight-increase operation, and let (z, q) be an edge adjacent to z. By Lemma 3.3 z is correctly colored *red* in line 13 of the procedure `Increase`. By Lemma 3.5, when z is extracted from Q in line 33, the correct distance of z in G' has been computed. The priority of z must be updated in heaps B_q and F_q only when z is the owner of edge (z, q). This is performed in line 34 of the procedure `Increase`, immediately after the extraction of z from Q. ▮

3.4. *Complexity Analysis*

The data structures require $O(|V| + |E|)$ space, while queries may be answered in optimal time: $O(1)$ to report the distance of any vertex from the source, and $O(l)$ to trace a minimum weight path with l edges. To study the output complexity of our algorithms, that work for any graph, we use the notion of *k-bounded accounting function* (see Definition 2.1).

LEMMA 3.7. *Give a graph $G = (V, E)$, let s be a source vertex and let $T(s)$ be a shortest paths tree for G. If G has a k-bounded accounting function, then it is possible to update $T(s)$ and distances of vertices from is after a weight-decrease operation, in $O(k \log n)$ worst case time per output update.*

Proof. When vertex z is extracted from C (i.e., its shortest distance from s in G' has been computed), the procedure `Decrease` traverses edges (z, q) either by priority or by ownership (line 13).

For each edge scanned by ownership a constant number of operations is performed each requiring $O(\log n)$ worst case time (see lines 15–22). Since there are at most k such edges, the total cost of these operations is $O(k \log n)$.

If edge (z, q) is scanned by priority from z, then $(z, q) \in ownership(q)$. We observe that in lines 13–22, the procedure scans by priority from z at most one edge that is not high for `z`. If edge (z, q) is high, then q decreases its distance from the source, and hence (z, q) belongs to the ownership of a vertex that must be updated.

The above discussion implies that, for each vertex that improves its distance from s after a *weight-decrease* operation, at most $k + 1$ edges are scanned by priority by the procedure `Decrease`; the lemma follows by observing that each scan requires $O(\log n)$ time. ∎

LEMMA 3.8. *Given a graph $G = (V, E)$, let s be a source vertex and let $T(s)$ be a shortest paths tree for G. If G has a k-bounded accounting function, then it is possible to update $T(s)$ and distances of vertices from s after a weight-increase operation, in $O(k \log n)$ worst case time per output update.*

Proof. We first observe that the overall cost of lines 10, 19, and 25 is proportional to the number of edges that are scanned while searching for the best nonred neighbor. If edge (z, q) is considered while searching for a best nonred neighbor for z, then z is colored either pink or red. At most k of the considered edges are owned by z. On the other hand, if (z, q) is not owned by z, then this implies that either q is red (and owns (z, q)) or q is not red, but in this case (z, q) is the last considered edge. This implies that for each red or pink vertex z, there is at most one edge that is scanned by priority from z and that is not owned by a red vertex. Since the graph has a k-bounded accounting function, the total number of edges considered in lines 10, 19, and 25 is at most $(k + 1)$ times the number of output updates. The above argument shows that the total cost of Steps 2 and 3.a is bounded by $O(k \log n)$ times the number of red and pink vertices.

To bound the cost of Step 3.b, observe that it is dominated by the cost of executing lines 33, 34, and 41. Clearly the cost of line 33 is $O(\log n)$ per red vertex, while the cost of line 34 is $O(\log n)$ for each edge that belongs to the ownership of a red vertex, that is, $O(k \log n)$ per *red* vertex. Finally,

the cost of line 41 is obtained as follows: for each red vertex z, the set of edges (z, h) such that h is red is computed by scanning all the edges owned by z and checking whether the other endpoint is red. Since each edge traversal has a logarithmic cost, the total cost of line 41 is $O(k \log n)$ per red vertex. It follows that the total cost of Step 3.b is $O(k \log n)$ per red vertex. ▮

Lemmas 3.7 and 3.8 imply the following theorem.

THEOREM 3.9. *Given a graph $G = (V, E)$, let s be a source vertex and let $T(s)$ be a shortest paths tree for G. If G has a k-bounded accounting function, then it is possible to update $T(s)$ and distances of vertices from s after a weight-decrease or a weight-increase operation, in $O(k \log n)$ worst case time per output update.*

4. INSERTIONS AND DELETIONS OF EDGES

In this section we present our algorithms for maintaining a shortest paths tree and the distance function for a graph G, having a k-bounded accounting function, under an arbitrary sequence of insert, delete, weight-decrease, and weight-increase operations on the edges of G. In this case, since the topology of G changes as edges are inserted and deleted, we have to take into account two problems: (i) the initial k-bounded accounting function can change; (ii) deletions and insertions of edges could disconnect and reconnect the graph, respectively.

We show that these problems can be handled efficiently by properly modifying the procedures for the weight-decrease and weight-increase operations.

In the Sections 4.1 and 4.2 we describe the algorithms for handling *insert* and *delete* operations, pointing out the main differences with respect to the procedures `Decrease` and `Increase`, respectively. In Section 4.3 we show how to modify the procedure `Decrease` and `Increase` to handle weight-decrease and weight-increase operations in the case of a dynamic topology. We use the same data structures and the same notions of *b–level* and *f–level* defined in the previous section.

4.1. *Insertion of an Edge*

When G is modified by inserting edge (x, y) we assume, without loss of generality, that $d(x) \leq d(y)$. If $d'(y) < d(y)$, then all the vertices that belong to $T(y)$ change their distance from s as a consequence of the edge insertion. On the other hand, the new subtree $T'(y)$ may include other vertices not contained in $T(y)$. As in the case of a weight-decrease

operation, the number of output updates of an edge insertion is given by the number of vertices that change their distance from the source as a consequence of that insertion.

The algorithm for edge insertions is similar to the algorithm for the weight-decrease operations: any time a vertex z improves its distance from the source, edges (z, q) are scanned either by ownership (i.e., when z is the owner of edge (z, q)) or by priority (i.e., when $b\_level_z(q) - d'(z)$ is positive and, hence, both vertex q and edge (z, q) are high for z).

The pseudocode of the procedure `Insert` is similar to that of `Decrease` and it is not reported. The main differences with respect to procedure `Decrease` are the following:

1. Step 1 of `Insert` arbitrarily chooses between x and y the owner of the inserted edge (x, y); then it properly updates the local data structures stored at vertices x and y, in the same fashion of procedure `Update_Local`(x, y), used in procedure `Decrease`. During the execution of Step 1, the algorithm halts either if $D(y) \leq D(x) + w_{x,y}$ (i.e., the insertion of (x, y) does not improve any distance from s, as in procedure `Decrease`) or if $D(x) = D(y) = +\infty$ (i.e., both x and y are not connected to s (differently from `Decrease`, where we assumed that all vertices are reachable from s)).

Note that if x and y belong to two different connected components C_1 and C_2 of G, respectively, and $s \in C_1$ (i.e., $D(x)$ is finite, while $D(y) = +\infty$), then when y is inserted into Q with priority $D(x) + w_{x,y}$, this quantity represents the length of a path from s to y in G'. In this case, Step 3 performs a shortest paths computation for all the vertices in C_2.

2. A coloring of the vertices of the graph is defined describing the required updates as a consequence of one edge insertion. Initially all vertices are colored white; if q does not change the distance from s, then it remains white, if q decreases its distance from the source, then q is red. 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*.

3. The update of the local data structures associated to the vertices of the graph is not performed after the extraction of a red vertex z from C by scanning the edges owned by z. This is motivated by the fact that the ownership of edges can change during the execution of `Insert` (see Step 4 described below).

4. An additional Step 4 that uses the above-defined coloring is performed by procedure `Insert`. In this step, for each red vertex z, the edges $(z, q) \in ownership(z)$ such that q is not red, are considered, and the owner of (z, q) is changed to q. These operations are performed by calling procedure `Change_Ownership`(z, q), which deletes (z, q) from *owner-*

ship(z), B_q, and F_q are inserts (z, q) in *ownership*(q), B_z, and F_z with updated priorities $b_z(q)$ and $f_z(q)$ equal to the computed values of $b\_level_z(q)$ and $f\_level_z(q)$, respectively. We observe that the change of ownership for such edges is not necessary for the correctness of the algorithms, but only to obtain the claimed amortized time bounds. Furthermore, for each edge (z, q) such that also q is red, the local data structures at vertices q and z are properly updated by procedure `Update_Levels`(z, q), according to the current owner of edge (z, q). Finally, all red vertices are restored to be white.

The correctness proof of `Insert` is similar to the correctness proof of `Decrease`, and it is omitted. The main difference concerns the update of the data structures associated to the vertices of the graph. In particular, when a vertex z improves its distance from s, procedure `Decrease` considers all edges (z, q) owned by z and correctly updates the priority of z in B_q and F_q, respectively (line 15). As pointed out above, the same approach cannot be used in procedure `Insert` because the ownership of some edges changes in Step 4; however, by reasoning similarly to the proof of Lemma 3.2, it is possible to show that the procedure `Change_Ownership` and `Update_Levels` performed in Step 4 guarantee that the information stored in the local data structures is always updated before and after the execution of any edge insertion.

4.2. *Deletion of an Edge*

Let $G' = (V, E - \{(x, y)\})$ be the new graph obtained from G by deleting the tree edge (x, y). Without loss of generality, we assume that $d(x) \leq d(y)$. It is easy to see that: (i) for each vertex $z \notin T(y)$, $d'(z) = d(z)$; (ii) there exists a new shortest paths tree $T'(s)$ in G' such that, for each vertex $z \notin T(y)$, *parent*$'(z)$ = *parent*(z).

As in the case of a weight-increase operation, the number of output updates of an edge deletion is given by the number of vertices that either have to change their parent in $T(s)$ or have to increase their distance from s; by (i) and (ii) above, all these ver tices belong to $T(y)$.

The procedure `Delete` is very similar to the procedure `Increase`, and its pseudocode is not reported. The main differences are:

1. Step 1 first deletes (x, y) from G as follows: if *owner*$(x, y) = x$, then (x, y) is deleted from *ownership*(x), B_y and F_y; otherwise, it is deleted from *ownership*(y), B_x, and F_x. After this, the algorithm halts either if (x, y) is a nontree edge (i.e., the deletion of edge (x, y) does not change any distance from s (as in the procedure `Increase`)), or if $D(x) = D(y) = +\infty$ (i.e., ix and y are not connected to s (differently from `Increase`)).

2. If the priority of all the vertices in Q is $+\infty$ at the beginning of Step 3.b, then this implies that the deletion of the edge (x, y) disconnects the connected component of G containing s. In this case each red vertex in Q receives a distance equal to $+\infty$ and no edge is scanned.

3. The update of the local data structures associated to the vertices of the graph is not performed after a red vertex z is extracted from Q by scanning the edges owned by z (see Step 4 below).

4. Step 4 is added to procedure `Delete`. This step is analogous to Step 4 of procedure `Insert`; namely, for each red vertex z the following operations are performed. For each edge $(z, q) \in ownership(z)$ such that q is not red, the ownership is changed by procedure `Change_Ownership`(z, q); i.e., (z, q) is deleted from $ownership(z)$, B_q, and F_q, and it is inserted in $ownership(q)$ and in B_z and F_z by using as priority the values $b'_z(q) = d'(q) - w_{zq}$ and $f'_z(q) = d'(q) + w_{zq}$, respectively. Then, for each edge (z, q) such that q is red, the local data structures at vertices z and q are properly updated by the procedure `Update_Levels`(z, q), according to the current owner of edge (z, q). Finally the original white color is restored for all vertices.

The correctness of the procedure `Delete` easily follows from the correctness of the procedure `Increase`, and it is omitted. The main difference concerns the update of local data structures associated to the vertices of the graph. When a vertex z increases its distance from the source, the procedure `Increase` considers all edges (z, q) owned by z and correctly updates the priority of z in B_q and F_q, respectively, in line 34. The same approach cannot be used in the procedure `Delete` since the ownership of some edges changes in Step 4; however, by reasoning similarly to the proof of Lemma 3.6, it is possible to show that the `Change_Ownership` and `Update_Levels` operations, performed in Step 4 of the procedure `Delete`, guarantee that the information stored in the local data structures is always updated before and after the execution of any delete operation.

4.3. *Updates of Edge Weights*

There is an obvious way to perform weight-decrease and weight-increase operations when we are able to handle insertions and deletions of edges. This is done by performing first a deletion of the edge and then an insertion of the same edge with the updated weight. Note that the output complexity of those two operations might be larger than the cost of the corresponding weight-decrease and weight-increase.

To avoid the above problem and to handle efficiently weight-increase and weight-decrease operations also in the case of a dynamic topology, it is

sufficient to modify the procedures `Decrease` and `Increase` in a way analogous to those described in the previous sections for the procedures `Insert` and `Delete`. The correctness proofs of these procedures are simple extensions of previous correctness proofs, and they are omitted.

4.4. *Amortized Time Bounds*

In this section we prove the complexity bounds of the proposed algorithms in the case of a dynamic topology. We consider only the time complexity of the procedures `Insert` and `Delete`, even if the same bounds trivially hold also for the procedures `Decrease` and `Increase`.

Recall that, during an edge insertion, for any vertex z that changes its distance from s, an edge (z, q) is traversed either if z is the owner of the edge ((z, q) is scanned by ownership) or if z is not the owner and (z, q) is high for z ((z, q) is scanned by priority). Analogously, during an edge deletion an edge (z, q) is scanned from z (while looking for its best nonred neighbor) either by ownership or by priority. Finally, we say that, during the execution of `Insert` or `Delete`, an edge is red if both its endpoints are red.

When new distances from s are computed, the last step of both procedures `Insert` and `Delete` modifies the ownership of edges as follows: If an edge (z, q) is scanned by ownership, then its owner is changed; if edge (z, q) is scanned by priority, then its owner is not changed. We recall that the above operations are not necessary for the correctness of the algorithms, but only to obtain the claimed amortized bounds.

We consider the following case. Let $G = (V, E)$ be a graph and let $\sigma = \langle \mu_1, \mu_2, \ldots, \mu_h \rangle$ be a sequence of *insert* and *delete* operations: each input modification $\mu_i \in \sigma$ is performed on graph G_{i-1}, with $G_0 \equiv G$ and gives the new graph $G_i = (V, E_i)$, where E_i, $i = 1, 2, \ldots, h$, is the set of edges obtained from E by performing the update operations $\mu_1, \mu_2, \ldots, \mu_i$.

We assume that there exists a k-bounded accounting function A_σ for the graph $G_\sigma = (V, E_\sigma)$, whose edges include both the edges in G and those inserted during σ. Furthermore, we define the *specialization* of A_σ to graph $G_i = (V, E_i)$ as the function A_i: $E_i \to V$ such that $A_i(x, y) = A_\sigma(x, y)$, for each $(x, y) \in E_i$. The *canonical owner* of the edge (x, y) in G_i is $A_i(x, y)$, while the *current owner* of (x, y) with respect to the current state of the data structures.

THEOREM 4.1. *Given a graph $G = (V, E)$, a source vertex s, a shortest paths tree $T_s(s)$ of G, and a sequence $\sigma = \langle \mu_1, \mu_2, \ldots, \mu_h \rangle$ of insert and delete operations, let $\langle G_1, G_2, \ldots, G_h \rangle$ be the sequence of graphs obtained from G by applying σ. If there exists an accounting function A_σ such that, for any $i = 1, 2, \ldots, h$, A_i is k-bounded, then it is possible to maintain $T(s)$ and*

the distances of vertices from s during σ, in $O(|\Delta(G, \sigma)| \cdot k \log n + |\sigma|)$ total time, that is $O(k \log n)$ amortized time per output update.

Proof. The statement of the theorem says that, during the execution of the sequence σ, the specialization of A_σ to the current graph G_i guarantees that each vertex of G_i owns at most k edges.

We observe that the running time of the procedure `Insert` is dominated by the cost of Step 3, while the running time of `Delete` is dominated by the cost of Step 2 (identical to Step 2 of the procedure `Increase`). To prove the complexity bounds of these steps we use a credit technique (see [29]). For each vertex z, let us denote as $Credit(z)$ the current balance of credits. Credits are distributed according to the following policy:

(i) initially, for each vertex x, we set $Credit(x) = 4k + 2$, and each edge is given 1 credit;

(ii) when an edge is inserted/deleted it is given 1 credit;

(iii) when a vertex z becomes either pink or red, it is given $4k + 2$ credits.

Credits are spent while executing the procedures `Insert` and `Delete`; each edge tranversal requires 1 credit, and $O(\log n)$ time. Therefore, to prove the theorem it is sufficient to show that the number of credits of each vertex/edge is never negative.

Edges are traversed either by priority or by ownership. If we consider edges scanned by priority the same considerations of Lemmas 3.7 and 3.8 hold. In particular, in the case of Step 3 of `Insert` we have that:

(a) If z is red, then at most k of the credits allocated to 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, and hence also the canonical owner in the current graph;

(b) If z is red, then the cost of the last nonred edge scanned by priority from z, is paid using one of the two additional credits that have been given to vertex z when it became red. Hence, $k + 1$ credits are sufficient to each red vertex z to pay for all the edges scanned by priority from z during the execution of `Insert`.

In the case of Step 2 of `Delete`, we observe that (a) and (b) above hold; furthermore, we note that, for each red (pink) vertex z, every edge not owned by z can be scanned by priority at most twice (in line 10 and in line 25 of Fig. 2); finally, if z is pink each edge scanned by priority from z belongs to the ownership of a red vertex, except for the last one, for which we pay by using the additional credit given to z when it became pink.

The above discussion implies that $2k + 2$ credits are sufficient to each red or pink vertex z to pay for all the edges scanned by priority from z during the execution of Step 2 of `Delete`.

To bound the number of edges scanned by ownership let us consider any edge (z, q) and the history of its current ownership while processing the sequence σ.

First observe that, when edge (z, q) of the initial graph G_0 is deleted, it has been scanned by ownership at most $2t + 1$ times, where t is the number of times (z, q) has been scanned by ownership starting from the (unknown) canonical owner. This is due to the fact that the current owner of an edge changes every time the edge is scanned by ownership. This implies that $2k$ credits given to the canonical owner of (z, q) each time it becomes either pink or red are sufficient to pay for two scans by ownership of each edge in its ownership. Therefore, $2t$ scans by ownership can be paid by using the above credit policy.

As far as the possibly extra scan is concerned, we use the credit that has been initially assigned to the edge (see (i) above). The analysis in the case of an edge inserted during σ is analogous; the only difference is that we use the credit assigned to the edge when it is inserted (see (ii) above) to pay for the extra scan. ∎

We remark that the proposed algorithms do not need to know any k-bounded accounting function during the execution of σ, but we require the existence of A_σ only to prove the amortized complexity bounds.

5. CONCLUSIONS AND OPEN PROBLEMS

In this paper we have proposed algorithms for the fully dynamic single source shortest paths problem on general graphs with positive real edge weights that are efficient in terms of output complexity and practical. Output complexity is a robust measure of performance for dynamic algorithms and leads to incomparable results with respect to amortized and worst case analysis, as pointed out in [26]. This notion applies when the algorithms operate within a framework where explicit updates are required on a given data structure, e.g., when they are a part of a large software system. In such a framework output complexity represents a technique for evaluating the computational cost of dynamic algorithms in terms of the *intrinsic cost* of the problem at hand, i.e., in terms of the number of output updates needed to solve the problem during an arbitrary sequence of input updates.

Note that, if explicit updates are required on a general graph with positive real edge weights, then no efficient dynamic solution for the single source shortest paths problem is known in the traditional cost models.

The algorithms proposed in this paper and the algorithms in [25] have been implemented on the top of LEDA library [22]. The experimental results proposed in [14], performed on random graphs and random sequences of updates, show that: (i) the output complexity is a useful parameter with which to evaluate the practical efficiency of dynamic algorithms for the single source shortest paths problem; (ii) the algorithms of Ramalingam and Reps [25] have better performances on random graphs, both with respect to the algorithms proposed in this paper (by a factor of around 2) and with respect to the application of the Dijkstra's algorithm after each input modification (by a factor of around 25). This is due to the simpler data structures used in [25] and to the fact that in a random graph the average degree of a vertex is twice the value of k for that graph. Hence, the simpler data structure of [25] overcomes the larger number of edges scanned by these algorithms with respect to the ones proposed in this paper.

In addition, in [14] a *separator* between the algorithms in [25] and those proposed in this paper has been given. A separator, as defined by Goldberg in [16], is a class of instances such that one of the two algorithms outperforms the other one on the given instances. The experiments in [14] have shown that on the instances of this class the ratio between the time spent by the algorithms in [25] and the time spent by our algorithms can be arbitrarily large, as the degree of the updated vertices increases in size.

An interesting open problem is to improve the time bounds proposed in the paper to the *batch* problem, in which each input update is a *set* of edge modifications to the structure of the graph, instead of a single edge modification. Finally, it would be very interesting to study harder problems in the output complexity model, for example, *flow* problems.

ACKNOWLEDGMENTS

We are indebted to Han La Poutré and Pino Italiano for constructive discussions. We thank two anonymous referees for providing us with many suggestions that significantly improve the presentation of the paper.

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. B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck, Incremental evaluation of computational circuits, *in* ''Proceedings ACM-SIAM Symposium on Discrete Algorithms, pp. 32–42, San Francisco, 1990.
3. S. Arnborg, Efficient algorithms for combinatorial problems on graphs with bounded decomposability—A survey, *BIT* **25** (1985), 2–23.

4. G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni, Incremental algorithms for minimal length paths, *J. Algorithms* **12** (1991), 615–638.
5. S. Chaudhuri and C. D. Zaroliagis, Shortest path queries in digraphs and small treewidth, *in* "Proceedings International Colloquium on Automata Languages and Programming, Szeged, Hungary, July 10–14, 1995," Lecture Notes in Computer Science, Vol. 944, pp. 244–255, Springer-Verlag, Berlin.
6. M. Chrobak and D. Eppstein, Planar orientations with low out-degree and compaction of adjacency matrices, *Theoret. Comput. Sci.* **86** (1991), 243–266.
7. E. W. Dijkstra, A note on two problems in connection with graphs, *Numer. Math.* **1** (1959), 269–271.
8. J. Edmonds, Minimum partition of a matroid into independent subsets, *J. Res. Nat. Bur. Standards B* **65** (1965), 67–72.
9. J. Edmonds, Lehman's switching game and a theorem of Nash-Williams, *J. Res. Nat. Bur. Standards B* **65** (1965), 73–77.
10. S. Even and H. Gazit, Updating distances in dynamic graphs, *Methods Oper. Res.* **49** (1985), 371–387.
11. E. Feuerstein and A. Marchetti-Spaccamela, On-line algorithms for shortest paths in planar graphs, *Theoret. Comput. Sci.* **116** (1993), 359–371.
12. P. G. Franciosa, D. Frigioni, and R. Giaccio, Semidynamic shortest paths and breadth-first search on digraphs, *in* "Proceedings Annual Symposium on Theoretical Aspects of Computer Science, Luebeck, Germany, February 27, 1997–March 1, 1997," Lecture Notes in Computer Science, Vol. 1200, pp. 33–46, Springer-Verlag, Berlin.
13. M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their use in improved network optimization algorithms, *J. Assoc. Comput. Mach.* **34** (1987), 596–615.
14. D. Frigioni, M. Ioffreda, U. Nani, and G. Pasqualone, Experimental analysis of dynamic algorithms for the single source shortest paths problem, *ACM J. Exp. Algorithmics* **3** (1998), 5.
15. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, Semidynamic algorithms for maintaining single source shortest paths trees, *Algorithmica* **22**, 3 (1998), 250–274.
16. A. V. Goldberg, Selecting problems for algorithm evaluation, *in* "Proceedings International Workshop on Algorithm Engineering, London, United Kingdom, July 19–21, 1999," Lecture Notes in Computer Science, Vol. 1688, pp. 1–11, Springer-Verlag, Berlin.
17. F. Harary, Graph Theory," Addison–Wesley, Reading, MA, 1969.
18. P. N. Klein, S. Rao, M. Rauch, and S. Subramanian, Faster shortest path algorithms for planar graphs, *in* "Proceedings ACM Symposium on Theory of Computing, Montreal, Quebec, Canada, May 23–25, 1994," pp. 27–37.
19. P. N. Klein and S. Subramanian, Fully dynamic approximation schemes for shortest paths problems in planar graphs, *in* "Proceedings International Workshop on Algorithms and Data Structures, Montreal, Canada, August 11–13, 1993," Lecture Notes in Computer Science, Vol. 709, pp. 443–451, Springer-Verlag, Berlin.
20. R. J. Lipton and R. E. Tarjan, A separator theorem for planar graphs, *SIAM J. Appl. Math.* **36** (1979), 177–189.
21. S. M. Malitz, Genus g graphs have pagenumber $O(\sqrt{g})$, *J. Algorithms* **17** (1994), 85–109.
22. K. Mehlhorn and S. Näher, LEDA: A platform for combinatorial and geometric computing, *Comm. Assoc. Comput. Mach.* **38** (1995), 96–102.
23. C. Nash-Williams, Edge-disjoint spanning trees of finite graphs, *J. London Math. Soc.* **36** (1961), 445–450.
24. G. Ramalingam and T. Reps, An incremental algorithm for a generalization of the shortest paths problem, *J. Algorithms* **21** (1996), 267–305.
25. G. Ramalingam and T. Reps, On the computational complexity of dynamic graph problems, *Theoret. Comput. Sci.* **158** (1996), 233–277.

26. G. Ramalingam, "Bounded Incremental Computation," Lecture Notes in Computer Science, Vol. 1089, Springer-Verlag, Berlin, 1996.
27. H. Rohnert, A dynamization of the all-pairs least cost paths problem, *in* "Proceedings 2nd Annual Symposium on Theoretical Aspects of Computer Science, Saarbrücken, Germany, January 3–5, 1985," Lecture Notes in Comput. Sci., Vol. 182, pp. 279–286, Springer-Verlag, Berlin.
28. J. Roskind and R. E. Tarjan, A note on finding minimum-cost edge-disjoint spanning trees, *Math. Oper. Res.* **10**, 4 (1985), 701–708.
29. R. E. Tarjan, Amortized computational complexity, *SIAM J. Algebraic Discrete Meth.* **6** (1985), 306–318.
30. M. Thorup, Undirected single source shortest path in linear time, *in* "Proceedings IEEE Symposium on Foundations of Computer Science, Miami Beach, Florida, October 20–22, 1997," pp. 12–21.