Fully Dynamic Shortest Paths in Digraphs with Arbitrary Arc Weights [⋆]

Daniele Frigioni [a,∗] Alberto Marchetti-Spaccamela [b]
Umberto Nanni [b]

[a] *Dipartimento di Ingegneria Elettrica, Università dell'Aquila, Monteluco di Roio, I-67040 L'Aquila, Italy*

[b] *Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", via Salaria 113, 00198 Roma, Italy*

Abstract

We propose a new solution for the fully dynamic single source shortest paths problem in a directed graph $G = (N, A)$ with arbitrary arc weights, that works for any digraph and has optimal space requirements and query time. If a negative-length cycle is introduced in the subgraph of G reachable from the source during an update operation, then it is detected by the algorithm. Zero-length cycles are explicitly handled. We evaluate the cost of the update operations as a function of a structural property of G and of the number of the output updates. We show that, if G has a k-bounded accounting function (as in the case of digraphs with genus, arboricity, degree, treewidth or pagenumber bounded by k), then the update procedures require $O(\min\{m, k \cdot n_A\} \cdot \log n)$ worst case time for *weight-decrease* operations, and $O(\min\{m \cdot \log n, k \cdot (n_A + n_B) \cdot \log n + n\})$ worst case time for *weight-increase* operations. Here, $n = |N|$, $m = |A|$, n_A is the number of nodes *affected* by the input update, that is the nodes changing either the distance or the parent in the shortest paths tree, and n_B is the number of non affected nodes considered by the algorithm that also belong to zero-length cycles. If zero-length cycles are not allowed, then n_B is zero and the bound for *weight-increase* operations is $O(\min\{m \cdot \log n, k \cdot n_A \cdot \log n + n\})$. Similar amortized bounds hold if we perform also insertions and deletions of arcs.

Key words: Shortest paths, dynamic algorithms, output complexity, zero-length cycles

---

[⋆] Work partially supported by the ESPRIT Long Term Research Project ALCOM-IT under contract no. 20244, and by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

[∗] Corresponding author
Email addresses: frigioni@ing.univaq.it (Daniele Frigioni), alberto@dis.uniroma1.it (Alberto Marchetti-Spaccamela), nanni@dis.uniroma1.it (Umberto Nanni).

1 Introduction

We study the single source shortest paths problem in a directed or undirected graph $G = (N, A)$ with real arc weights. The best known static algorithm for this problem is the classical Bellman-Ford method, running in $O(m \cdot n)$ worst case time on a graph having n nodes and m arcs; it either detects a negative-length cycle, if one exists, or computes a shortest paths tree (see e.g. [1]).

The dynamic version of the problem consists of maintaining shortest paths while changes in the graph are performed, without recomputing them from scratch. In such a framework the most general repertoire of update operations includes insertions and deletions of arcs, and update operations on the weights of arcs. When arbitrary sequences of the above operations are allowed we refer to the *fully dynamic problem*; if we consider only insertions (deletions) of arcs then we refer to the *incremental* (*decremental*) problem.

In the case of positive arc weights there is a number of papers that propose different solutions to deal with dynamic shortest paths problems [3,4,9,10,13–15,17–19]. However, in the general case, neither a fully dynamic solution nor a decremental solution for the single source shortest paths problem is known in the literature that is asymptotically better than recomputing the new solution from scratch.

In the case of arbitrary arc weights we are aware only of the output bounded solution proposed by Ramalingam and Reps in [16,17]. The main idea of this solution is to use a technique of Edmonds and Karp [8], which allows to transform the weight of each arc in the digraph to a non-negative real without changing the shortest paths. This is done as follows: after an input update, for each $(z, v) \in A$ such that at least one of the endpoints is affected, replace $w_{z,v}$ with the *reduced weight* $d(z) + w_{z,v} - d(v)$, and apply an adaptation of Dijkstra's algorithm to the modified graph. The computed distances represent changes to the distances since the update. The actual distances of nodes after the update can be easily recovered from the reduced weights. The assumption is that all the cycles in the digraph before and after any input update have positive length. In fact, the dag of the shortest paths is maintained, and in the presence of zero-length cycles all of these cycles that are reachable from the source will belong to the digraph of the shortest paths, which will no longer be a dag. The running time is $O(||\delta|| + |\delta| \log |\delta|)$ per update, where $|\delta|$ is the number of nodes affected by the input change δ, and $||\delta||$ is the number of affected nodes plus the number of arcs having at least one affected endpoint. This gives $O(m + n \log n)$ time in the worst case.

In this paper we propose a new fully dynamic solution for the problem of updating the shortest paths from a given source of a digraph with arbitrary arc weights. Differently from the solution in [16,17], zero-length cycles are explicitly handled. The proposed algorithm runs in $O(m \cdot \log n)$ worst case time per operation in the general case, but its complexity is better if one of (or both) the following conditions holds: A) the graph satisfies some structural property; B) the update operation introduces a "small" change in the shortest paths tree. Let us consider these conditions in more detail.

A) The structural constraints that we consider have been introduced in the framework of dynamic algorithms in [13]. An *accounting function* A for G is a function that for each arc (x, y) determines either node x or node y as the *owner* of that arc; A is k-bounded if k is the maximum over all nodes x of the cardinality of the set of arcs owned by x. The value of parameter k defined above can be bounded in different ways (see e.g. [13,14]). In general, a graph with m arcs has a $O(\sqrt{m})$-bounded accounting function. An analogous notion, the *orientation* of edges in an undirected graph, has been introduced in [5].

If we assume that the graph has a k-bounded accounting function (k-baf from now on), then the worst case running time of the proposed algorithm becomes $O(n \cdot k \cdot \log n)$. We remark that the notion of k-baf is useful only to bound the running times, but does not affect the behavior of our algorithms.

Computing the minimum k such that G has a k-baf can be hard; however, a simple greedy algorithm (see Section 2.1) allows us to compute a 2-approximation in $O(m + n)$ time. Using this 2-approximation the cost of our algorithm increases at most of a factor of 2.

B) The analysis of dynamic algorithms using the *output complexity* model has been introduced by Ramalingam and Reps in [16,17] and it has been subsequently modified by the authors of this paper in [13]. Other authors used similar concepts in [2].

In this paper, given a digraph G, a node of G is said to be *affected* if it changes its distance from the source or its parent in the shortest paths tree as a consequence of a dynamic update to G (insertion, deletion, or weight update of an arc). Let us suppose that G has a k-baf and that n_A is the number of nodes of G *affected* by the dynamic update. In the case of *weight-decrease* operations, the proposed algorithm requires $O(k \cdot n_A \cdot \log n)$ worst case time. When the weight of an arc (x, y) is increased we observe that if (x, y) belongs to the shortest paths tree, then the set of output updates depends on the existence of alternative paths; in fact, it does not necessarily include all nodes belonging to the subtree of the shortest paths tree rooted at y. In this case the running time of the proposed procedure is $O(k \cdot (n_A + n_B) \cdot \log n + n)$, where n_B is the number of non affected nodes considered by the algorithm that also belong to zero-length cycles. If zero-length cycles are not allowed, then n_B

is zero and the bound for *weight-increase* operations is $O(k \cdot n_A \cdot \log n + n)$. The term n in the stated bound is the time needed for a preliminary visit of the subtree of the shortest paths tree rooted at y. The stated bounds hold also if we allow a static topology, and hence deletions of arcs and re-insertions (of the deleted arcs). The bounds become amortized if generic insertions and deletions of arcs are allowed.

The main contribution of this paper is a new algorithm for handling arc deletions and weight increases that explicitly deals with zero-length cycles. The algorithm consists of two phases: first it finds the affected nodes and then computes their new distances. As pointed out by Ramalingam and Reps in [16,17] the main difficulty in finding affected nodes in output bounded sense is the presence of zero-length cycles. In fact, if we consider the subgraph of G induced by the arcs belonging to the shortest paths of G, then the distance of a node changes after the input update if and only if the node is no longer reachable from the source in this graph. They also show that, if zero-length cycles are allowed in G, then no output bounded algorithm exists for this problem. The first phase of our algorithm is able to detect the affected nodes in the presence of zero-length cycles in $O(k \cdot (n_A + n_B) \cdot \log n + n)$ worst case time. If $n_A + n_B = o(m/k \log n)$, then this is better than recomputing from scratch the reachability information in the digraph of shortest paths. This bound is obtained by a cost analysis that, besides the notion of k-baf, exploits non-trivial properties of depth first search trees. We believe that the correctness and complexity proofs of this part are of independent interest. The second phase of the algorithm runs within the same time bound as above and is essentially a Dijkstra's computation on the subgraph induced by the affected nodes. This computation is based on the use of the technique of Edmonds and Karp as in [16,17], and on the notion of k-baf.

Another contribution of the paper is an algorithm for handling arc insertions and weight decreases that explicitly deal with negative-length cycles that might be introduced in the subgraph reachable from the source by an operation of this kind. The algorithm either detects if a negative-length cycle has been introduced by the update operation (and in this case discards the update operation), or computes the new shortest paths tree on the subgraph induced by the affected nodes. As for the second phase of the algorithm for arc deletions and weight increases, this algorithm is analogous to that given in [16,17], but the exploitation of the notion of k-baf allows us to obtain improved bounds.

The paper is organized as follows. After some preliminaries, given in the next section, we describe in Section 3 the simple case of a *weight-decrease* operation and discuss the relationships between our solution and that proposed in [16,17]. The main contribution of the paper is presented in Section 4, where we propose a new algorithm for identifying the affected nodes in the presence of zero-length cycles after a *weight-increase* operation. In Section 5 the algorithms of Sections 3 and 4 are extended to handle insertions and deletions of arcs, respectively. Finally, in Section 6 we provide conclusions and open problems.

2 Preliminaries

Let $G = (N, A)$ be a weighted digraph with n nodes and m arcs, and let $s \in N$ be a fixed *source* node. For each $z \in N$, we denote as $\textsc{in}(z)$ and $\textsc{out}(z)$, the arcs of A incoming and outgoing z, respectively. To each arc $(x, y) \in A$, a real weight $w_{x,y}$ is associated.

A *path* in G is a sequence of nodes $\langle x_1, x_2, \ldots, x_r\rangle$ such that $(x_i, x_{i+1}) \in A$, for $i = 1, 2, \ldots, r-1$. The length of a path is the sum of the weights of the arcs in the path. A cycle is a path $\langle x_1, x_2, \ldots, x_r\rangle$ such that $(x_i, x_{i+1}) \in A$, for $i = 1, 2, \ldots, r-1$, and $(x_r, x_1) \in A$. A negative-length (zero-length) cycle is a cycle C such that the sum of the weights of the arcs in C is negative (zero). If the graph does not contain negative-length cycles then we denote as $d : N \to \Re$ the distance function that gives, for each $x \in N$, the length of the shortest path from s to x in G. $T(s)$ denotes a shortest paths tree rooted at s; for any $x \in N$, $x \neq s$, $T(x)$ is the subtree of $T(s)$ rooted at x. The well known *optimality condition* of a distance function d on $G = (N, A)$ says that, for each $(z, q) \in A$, $d(q) \leq d(z) + w_{z,q}$ (see, e.g., [1]).

We consider the problem of maintaining a shortest paths tree in a digraph with arbitrary arc weights, subject to the following operations: (a) *distance*(x), that reports the current distance between s and node x; (b) *min-path*(x), that reports a shortest path between s and node x; (c) *insert*(x, y, w), that inserts arc (x, y) with weight w; (d) *delete*(x, y), that deletes arc (x, y); (e) *weight-increase*(x, y, ϵ), that increases by quantity ϵ the weight of arc (x, y); (f) *weight-decrease*(x, y, ϵ), that decreases by quantity ϵ the weight of arc (x, y). After that each arc modification has been carried out, if a negative-length cycle has been introduced in the subgraph of G reachable from the source, then we have to detect it and to discard the modification; otherwise, we are required to compute a new shortest paths tree.

As in [14], in addition to the standard representation of G, we use the following data structures to bound the number of arcs scanned by our algorithms. For each node z, we partition both $\textsc{in}(z)$ and $\textsc{out}(z)$ in two subsets as follows. Any arc (x, y) has an *owner* that is either x or y. For each $z \in N$, $\textsc{in-own}(z)$ ($\overline{\textsc{in-own}}(z)$) denotes the subset of $\textsc{in}(z)$ containing the arcs owned (not owned) by z. Analogously, $\textsc{out-own}(z)$ and $\overline{\textsc{out-own}}(z)$ represent the arcs in $\textsc{out}(z)$ owned and not owned by z, respectively. We say that G has a k-

baf if $|\text{IN-OWN}(z) \cup \text{OUT-OWN}(z)| \leq k$, for each $z \in N$. The arcs in $\text{IN-OWN}(z)$ and in $\text{OUT-OWN}(z)$ are stored in two linked lists, each containing at most k arcs. The arcs in $\overline{\text{IN-OWN}}(z)$ and in $\overline{\text{OUT-OWN}}(z)$ are stored in two priority queues according to the notions of *backward_level* and *forward_level*, as follows. The *backward_level* of arc $(z, q) \in \text{OUT}(z)$ is $b\_level_z(q) = d(q) - w_{z,q}$; the *forward_level* of arc $(v, z) \in \text{IN}(z)$ is $f\_level_z(v) = d(v) + w_{v,z}$. $\overline{\text{IN-OWN}}(x)$ is a min-based priority queue where the priority of arc (y, x) (of node y), denoted as $f_x(y)$, is the computed value of $f\_level_x(y)$. $\overline{\text{OUT-OWN}}(x)$ is a max-based priority queue where the priority $b_x(y)$ of arc (x, y) (of node y) is the computed value of $b\_level_x(y)$;

Let G' be the digraph obtained from G after an arc operation. The new shortest paths tree in G' is denoted as $T'(s)$. For each $z \in N$, $d'(z)$ denotes the distance of z in G', while the *variation* of distance of z after an arc update in G is the quantity $\delta(z) = d'(z) - d(z)$. The parameters $D(z)$ and $P(z)$ store the distance of z and the parent of z in the shortest paths tree, respectively, while $\Delta(z)$ stores the computed value of $\delta(z)$. Before (after) the execution of any procedure $D(z) = d(z)$ ($D(z) = d'(z)$) and $\Delta(z) = \delta(z) = 0$. During the execution of the update procedures, $\Delta(z)$ is an upper bound on $\delta(z)$; in the case of *weight-decrease/insert* operations $\Delta(z)$ assumes negative values, while in the case of *weight-increase/delete* operations it is positive. We use two additional variables $D'(z)$ and $P'(z)$ that store the temporary values of $D(z)$ and $P(z)$, respectively, during the execution of the algorithms. The actual values are stored in the variables $D(z)$ and $P(z)$ only if no negative-length cycle is detected.

Before processing a sequence of arc modifications on G, we have to compute the distance $d(x)$, for each node x, an initial shortest paths tree and an initial k-bounded ownership for G. To compute an initial shortest paths tree using the Bellman-Ford method (see, e.g., [1]) we need $O(m \cdot n)$ time. In the following subsection we show how it is possible to compute an initial ownership for G in $O(m + n)$ time. For each node x, the data structures are initialized by computing, for each arc $(x, y) \in \overline{\text{OUT-OWN}}(x)$ and for each arc $(v, x) \in \overline{\text{IN-OWN}}(x)$, $b_x(y) = b\_level_x(y)$ and $f_x(v) = f\_level_x(v)$, respectively. This can be done in $O(m \cdot \log n)$ time. It follows that the worst case preprocessing time is $O(m \cdot n)$.

2.1 Computing a k-bounded accounting function

Let $G = (N, A)$ be a digraph and $\text{DEG}(x) = |\text{IN}(x)| + |\text{OUT}(x)|$, for each $x \in N$. The heuristic of Figure 1 computes for each $x \in N$ a set, denoted as $\mathcal{F}_a(x)$, containing the arcs of $\text{IN}(x)$ and $\text{OUT}(x)$ that are owned by x.

```
    Algorithm 2-Approx
1.  begin
2.      put every node $x$ of $G$ in a priority queue $\mathcal{Q}$ using $\textsc{deg}(x)$ as priority
3.      while $\mathcal{Q}$ is not empty do
4.          begin
5.              delete from $\mathcal{Q}$ the node $x$ having minimum priority
6.              put in $\mathcal{F}_a(x)$ all the arcs in $\textsc{in}(x)$ and $\textsc{out}(x)$
7.              for each neighbor $z$ of $x$ do
8.                  begin
9.                      delete arc $(x,z)$ (or $(z,x)$) from all the data structures
10.                     update $\textsc{deg}(z)$ and the priority of $z$ in $\mathcal{Q}$, accordingly
11.                 end
12.         end
13. end
```

Fig. 1. Computing a 2-approximated accounting function

Theorem 2.1 *Let $G = (N, A)$ be any digraph admitting a k-baf. For each $x \in N$, the set $\mathcal{F}_a(x)$, computed by Algorithm* `2-Approx`*, contains at most $2k$ arcs.*

Proof. Let $\langle x_1, x_2, \ldots, x_n \rangle$ be the ordering of the nodes in N induced by Algorithm `2-Approx`. We will prove that $|\mathcal{F}_a(x_i)| \leq 2k$, for all $i = 1, 2, \ldots, n$.

Let us consider $x_i \in N$, $1 \leq i \leq n$, and let $G_i = (N_i, A_i)$ be the subgraph of G induced by the nodes in $\{x_i, x_{i+1}, \ldots, x_n\}$. For each $x \in N_i$, $\textsc{deg}_i(x)$ denotes the degree of x in G_i.

Since G admits a k-baf it follows that $k \geq |A|/|N|$. Since G_i is a subgraph of G then $k \geq |A_i|/|N_i|$, for each $i = 1, 2, \ldots, n$. Furthermore, $|A_i| \geq \textsc{deg}(x_i) \cdot |N_i|/2$, and hence $k \geq \textsc{deg}(x_i)/2 \geq \textsc{deg}_i(x_i)/2$. Since $|\mathcal{F}_a(x_i)| = \textsc{deg}_i(x_i)$, the lemma follows. $\square$

We observe that, in the case of a complete digraph, the bound of Theorem 2.1 is tight.

Algorithm `2-Approx` requires $O(m + n)$ worst case time as follows. The queue $\mathcal{Q}$ is implemented as an array of size n such that, for each *level* $1 \leq i \leq n$, there is a pointer to a double-linked list storing the nodes of G having degree equal to i. The `DeleteMin` operation at line 5 decreases the degree of the next minimum at most with one. Every time the node x with minimum priority is deleted, the degree of each neighbor z of x decreases by one, determining a move of z from its current level i to level $i - 1$ into $\mathcal{Q}$. This move costs $O(1)$, and there is a move each time an arc of the graph is deleted. This implies that, by simply scanning the array, $O(m + n)$ worst case time overall is needed to perform the decrease-key operations and to find and delete the n consecutive minima in $\mathcal{Q}$.

3 Decreasing the weight of an arc

In this section we show how to maintain a shortest paths tree of a digraph $G = (N, A)$ with arbitrary arc weights, after decreasing the weight of an arc. We assume that all the nodes are reachable from s, and that G does not contain negative-length cycles. If the *weight-decrease* operation on arc (x, y) does not introduce a negative-length cycle, then Procedure `Decrease`, shown in Figure 2, properly updates the current shortest paths tree, otherwise it detects the negative-length cycle introduced and halts.

Assume that the weight of arc (x, y) is decreased by a positive quantity ϵ. It is easy to see that if $d(x) + w_{x,y} - \epsilon \geq d(y)$ then no node of G changes its distance from s. Therefore, we focus on the case $d(x) + w_{x,y} - \epsilon < d(y)$. This implies that all the nodes in $T(y)$ decrease their distance from s of the same quantity of y. In addition, $T'(y)$ may include other nodes not contained in $T(y)$. Each of these nodes decreases its distance from s of a quantity which is at most the reduction of y's distance. We denote as *red* the nodes that decrease their distance from s after a *weight-decrease* operation, and as n_R the number of *red* nodes. The following facts can be easily proved:

F1) If node y decreases its distance from s, then the new shortest paths from s to the *red* nodes contain (x, y); if y does not decrease its distance from s then no negative-length cycle is added to G, and all the nodes preserve their distance from s.

F2) Node x reduces its distance from s if and only if the operation introduces a negative-length cycle C; in this case (x, y) belongs to C.

Recall that, before the execution of `Decrease`, for each node z, $\Delta(z) = 0$, $D(z) = d(z)$ and $P(z)$ stores the parent of z in the shortest paths tree. To update the distances of *red* nodes, we adopt a strategy similar to that of Dijkstra's algorithm on the subgraph of G induced by the *red* nodes. By Fact *F1* the Dijkstra-like strategy starts from y.

In detail, each *red* node is inserted in a heap Q with priority $\Delta(z)$, that is an estimate of the (negative) variation $\delta(z) = d'(z) - d(z)$. Then the *red* nodes are processed as in Dijkstra's algorithm. Namely, at the beginning of the procedure, node y is enqueued with priority $\Delta(y) = D(x) + w_{x,y} - \epsilon - D(y)$. In the main **while** loop, the nodes are dequeued from Q; for each node z dequeued with priority $\Delta(z)$, the new distance from s is computed as $D'(z) = D(z) + \Delta(z) = d(z) + \Delta(z)$. Then, Procedure `Heap_Insert_or_Improve`$(Q, \langle h, \Delta(h), z\rangle)$ is called for all the arcs (z, h) in $\text{OUT-OWN}(z)$, and for the arcs (z, h) in $\overline{\text{OUT-OWN}}(z)$ such that $b_z(h) > D'(z)$, and does the following: if $h \notin Q$, then h is inserted in Q with priority $\Delta(h) = D'(z) + w_{z,h} - D(h)$; if $h \in Q$ and $D'(z) + w_{z,h} - D(h) < \Delta(h)$, then the priority of h in Q is updated to the value

```
    procedure Decrease($x, y$ : node; $\epsilon$ : positive_real)
1.  begin
2.  $w_{x,y} \longleftarrow w_{x,y} - \epsilon$
3.  if $D(x) + w_{x,y} - D(y) < 0$ then
4.      begin
5.          $Q \longleftarrow \emptyset$ {initialize an empty heap $Q$}
6.          $\Delta(y) \longleftarrow D(x) + w_{x,y} - D(y)$
7.          Enqueue($Q, \langle y, \Delta(y), x\rangle$)
8.          color $y$ red
9.          while Non_Empty($Q$) or a negative-length cycle is detected do
10.             begin
11.                 $\langle z, \Delta(z), q\rangle \longleftarrow$ Extract_Min($Q$)
12.                 $D'(z) \longleftarrow D(z) + \Delta(z)$
13.                 $P'(z) \longleftarrow q$
14.                 for each $(z, h) \in$ OUT-OWN($z$) and
                        for each $(z, h) \in \overline{\text{OUT-OWN}}(z)$ s.t. $b_z(h) > D'(z)$ do
15.                     if $D'(z) + w_{z,h} - D(h) < \Delta(h)$ then
16.                         begin
17.                             color $h$ red
18.                             if $h = x$ then a negative-length cycle is detected
19.                             else begin
20.                                     $\Delta(h) \longleftarrow D'(z) + w_{z,h} - D(h)$
21.                                     Heap_Insert_or_Improve($Q, \langle h, \Delta(h), z\rangle$)
22.                                 end
23.                         end
24.             end
25.         for each red node $z$ do
26.             begin
27.                 uncolor $z$
28.                 $\Delta(z) \longleftarrow 0$
29.                 if no negative-length cycle has been detected then
30.                     begin
31.                         for each $(v, z) \in$ IN-OWN($z$) do $b_v(z) \longleftarrow D'(z) - w_{v,z}$
32.                         for each $(z, v) \in$ OUT-OWN($z$) do $f_v(z) \longleftarrow D'(z) + w_{z,v}$
33.                         $D(z) \longleftarrow D'(z)$
34.                         $P(z) \longleftarrow P'(z)$
35.                     end
36.             end
37.     end
38. end
```

Fig. 2. Decrease by quantity ϵ the weight of arc (x, y)

$D'(z) + w_{z,h} - D(h)$. If any improvement is determined for node x, then by Fact *F2*, a negative-length cycle has been detected. By the above discussion, it follows that, if no negative-length cycle has been introduced, then when a node z is extracted from Q, $\Delta(z) = d'(y) + l(P) - d(z)$, where P is a shortest path from y to z in G' and $l(P)$ is the length of P.

In the main **while** loop, for each *red* node z, the new distance from the source and the new parent in the shortest paths tree are computed in the auxiliary variables $D'(z)$ and $P'(z)$, respectively. Only after that this computation has been carried out successfully (i.e., without trying to update node x), the data structures are actually updated (see lines 25–36).

The use of the estimate of the (negative) variation of a *red* node z as the priority of z in the algorithm is analogous to the technique outlined by Ramalingam and Reps in [16,17]. This solution is based on the idea of Edmonds and Karp [8] that, if the arc weights of a digraph G are properly rescaled to positive reals, then the shortest paths of G do not change. In detail, if the weight of each arc (a, b) in G is replaced by $f(a) + w_{a,b} - f(b)$, where $f : N \rightarrow \Re$, then a graph G_r is obtained such that the shortest paths of G and G_r are the same, though the shortest distances are different. The reason is that if P is a path from node a to node b, then $l_r(P) = f(a) + l(P) - f(b)$, where $l(P)$ and $l_r(P)$ denote the length of P in G and in G_r, respectively. As a consequence $d_r(a, b) = f(a) + d(a, b) - f(b)$, where $d(a, b)$ and $d_r(a, b)$ denote the length of the shortest path from a to b in G and in G_r, respectively. Hence, if there exists f such that $f(a) + w_{a,b} - f(b) \geq 0$, for every arc (a, b), then the shortest paths in G_r can be computed by Dijkstra's algorithm, and the shortest distances in G can be easily recovered from the rescaled distances.

Ramalingam and Reps noticed that a function f is available for the problem of dynamically updating shortest paths in digraphs with arbitrary arc weights, and in particular $f \equiv d$, i.e., f coincides with the distance function in G. The algorithm they outline consists of applying their solution for positive arc weights to G_r, and to recover the distances in G from the distances in G_r. In detail, node y is inserted in a heap with priority $d(s, x) + w_{x,y} - \epsilon - d(s, y)$ and $d_r(s, y)$ is set to the same value. At this point, for each arc (y, z) in $\textsc{out}(y)$ the reduced weight is computed as $d(s, y) + w_{y,z} - d(s, z)$, and if $d_r(s, y) + d(s, y) + w_{y,z} - d(s, z) = d(s, x) + w_{x,y} - \epsilon + w_{y,z} - d(s, z) < 0$, then z is affected and it is put in the heap with priority $d(s, x) + w_{x,y} - \epsilon + w_{y,z} - d(s, z)$ (or its priority is possibly updated to the same value); this value coincides with $\Delta(z)$. This discussion can be iterated and extended to every affected node. Hence, when an affected node z is extracted from the priority queue, its priority is the same as that of Procedure `Decrease`.

By the above discussion, the correctness of Procedure `Decrease` follows from the correctness of the algorithm of Ramalingam and Reps and of the algorithm in [14]. We report Procedure `Decrease` for the following reasons: *i*) the algorithm of Ramalingam and Reps is only sketched in [16,17] (for example, they do not say how to deal with negative-length cycles that could be introduced in G as a consequence of a *weight-decrease* operation); *ii*) Procedure `Decrease` exploits the notion of k-baf to bound the number of traversed arcs. This allows us to improve over the result in [16,17] as shown in the next theorem.

Theorem 3.1 *Let $G = (N, A)$ be a digraph with arbitrary arc weights. If G has a k-baf, then it is possible to update $T(s)$ and the distances of nodes from s, or to detect a negative-length cycle in G after the execution of a weight-decrease operation, in $O(\min\{m,\ n_R \cdot k\} \cdot \log n)$ worst case time.*

Proof. The cost of the procedure is dominated by the overall cost of lines 11, 21, and 31–32.

First observe that, each red node is enqueued in Q exactly once during the execution of `Decrease`; this can be shown using the monotonicity of the priorities of nodes in Q.

We first prove the $O(m \cdot \log n)$ bound. By the above observation, the overall cost of line 11 is $O(n \cdot \log n)$. When a node z is extracted from Q, all the arcs in $\text{OUT}(z)$ are traversed in the worst case, and each traversal has cost $O(\log n)$; therefore, the overall cost of line 21 is $O(m \cdot \log n)$. Analogously, we can show that the cost of lines 31–32 is $O(m \cdot \log n)$.

We now prove the $O(n_R \cdot k \cdot \log n)$ bound. When z is dequeued, an arc $(z, h) \in \text{OUT}(z)$ is scanned (see line 14) only if: *i)* $(z, h) \in \text{OUT-OWN}(z)$ (the arc is scanned *by ownership*); *ii)* $(z, h) \in \overline{\text{OUT-OWN}}(z)$ and $b_z(h) > D'(z)$ (the arc is scanned *by priority*).

At most $k \cdot n_R$ arcs are scanned by ownership. Concerning the arcs scanned by priority, observe that, for each *red* node z, Procedure `Decrease` traverses only the arcs (z, v) in $\overline{\text{OUT-OWN}}(z)$ such that $b_z(v) > D'(z)$, plus the first arc (z, v) in $\overline{\text{OUT-OWN}}(z)$ such that $b_z(v) \le D'(z)$. Note that, if $b_z(v) > D'(z)$ then v is *red* and, therefore, the algorithm scans by priority only arcs between *red* nodes, plus n_R additional arcs. Since the subgraph induced by the *red* nodes has a k-baf, then at most $(k + 1) \cdot n_R$ arcs are scanned by priority. Each of these arc traversal requires $O(\log n)$ time and determines a possible Insert or a Decrease-key in Q (line 21). Exactly n_R Insert and at most $k \cdot n_R$ Decrease-key are performed on Q, thus requiring $O(k \cdot n_R \cdot \log n)$ worst case time.

In the last phase (lines 25–36), for each *red* node z, the values $b_v(z)$ and $f_v(z)$ in $\overline{\text{IN-OWN}}(v)$ and $\overline{\text{OUT-OWN}}(v)$, respectively, are updated by scanning again the lists $\text{IN-OWN}(z)$ and $\text{OUT-OWN}(z)$ (lines 31–32). Since each of these operations requires $O(\log n)$ time, this gives an overall worst case time of $O(k \cdot n_R \cdot \log n)$ for this phase.

Note that, if decreasing the weight of arc (x, y) introduces a negative-length cycle C, then by Fact *F2*, $(x, y) \in C$. Furthermore, all the nodes in C are *red*; in fact, it is easy to show that, for each node $z \in C$, the (acyclic) path from s to z, passing through (x, y) and the arcs of C connecting y to z, is shorter than the shortest path from s to z in G. Hence, the above bounds hold also when a negative-length cycle is detected by the algorithm. $\square$

4 Increasing the weight of an arc

In this section we show how to maintain a shortest paths tree of a digraph $G = (N, A)$ with arbitrary arc weights, after increasing the weight of an arc. We assume that all the nodes are reachable from s, and that G does not contain negative-length cycles. If the weight of $(x, y) \in A$ is increased, then no negative-length cycle can be introduced in G by that operation. We allow the presence of zero-length cycles and deal explicitly with them. Furthermore, it is easy to see that: *i*) for each node $z \notin T(y)$, $d'(z) = d(z)$; *ii*) there exists a new shortest paths tree $T'(s)$ such that, for each $z \notin T(y)$, the old parent of z in $T(s)$ is preserved.

As argued by Ramalingam and Reps in [16,17], the difficulty of updating the shortest paths after a *weight-increase* operation in a bounded fashion is that of finding the affected nodes when zero-length cycles are allowed. They show that no output bounded algorithm exists in this case and hence they work under the assumption that no zero-length cycle is present in G. In what follows, we eliminate this assumption and show how to explicitly deal with zero-length cycles. In particular, we give an algorithm whose complexity depends on the number of affected nodes and on the number of non-affected nodes considered by the algorithm that also belong to zero-length cycles.

We define a coloring of the nodes of $T(y)$, in order to distinguish how nodes are affected by the execution of a *weight-increase* operation. The *permanent* colors are defined as follows:

- $q \in T(y)$ is *white* if q changes neither the distance from s nor the parent in $T(s)$;
- $q \in T(y)$ is *red* if q increases the distance from s, i.e., $d'(q) > d(q)$;
- $q \in T(y)$ is *pink* if q maintains its distance from s, but changes the parent in $T(s)$.

Nodes belonging to $T(y)$ may also receive a *temporary blue* color before the final permanent color is determined. The *blue* color is used to mark nodes that are visited during the search of alternative paths after the weight increase operation is performed.

It is easy to verify that, if q is *red* then all the children of q in $T(s)$ will be either *pink* or *red*. Observe that a node is colored *pink* depending on the possibility of finding alternative paths of the same length of the shortest path before the *weight-increase* operation; it follows that a node q can be colored *white* even if the shortest path from s to q in G is not a shortest path in G'; it is sufficient that there exists one shortest path from s to q in G' of the same length of the shortest path from s to q in G, where q has the same parent as before.

Finally, observe that if a node q is colored *pink* or *white* and the digraph does not admit zero-length cycles, then all nodes in $T(q)$ are *white*. On the other hand, if zero-length cycles are allowed and a node q is colored *pink*, then a child u of q in $T(q)$ can be colored *pink*. This happens when the new shortest path for q passes through u. In this case, u needs a new parent and will be colored *pink*. It is easy to verify that this can happen only if both q and u belong to a zero-length cycle.

The set of output updates determined by a *weight-increase* operation is given by the set of *red* and *pink* nodes. We remark that, if there exist several shortest paths trees, the size of this set depends on the solution found by the algorithm. In fact, a nonred node might be colored either pink or white depending on the specific shortest path found by the algorithm. Hence, it is possible to have different sizes for the set of output updates.

In the following we assume that, before the *weight-increase* operation, the data structures store the correct values, i.e., the array P of parent pointers induces a shortest paths tree rooted at s and, for each $z \in N$, $D(z) = d(z)$ and $\Delta(z) = 0$.

Algorithm `Increase` works in two phases. In the first phase it uses Procedure `Color`, shown in Figure 3, that colors the nodes in $T(y)$ after increasing the weight of arc (x, y), according to the above described rules, and gives a new parent in the shortest paths tree to each node which is colored *pink*. In the second phase Procedure `Increase` properly updates the information on the shortest paths from s to all the *red* nodes, by performing a computation analogous to Dijkstra's algorithm.

Definition 4.1 *Let $G = (N, A)$ be a digraph in which the weight of arc (x, y) has been increased. A node q is a* candidate parent *of a node p if $d(p) = d(q) + w_{q,p}$. Node q is an* equivalent parent *of p if it is a candidate parent of p and $d'(q) = d(q)$.*

Let us define the *best nonred neighbor* of a node p as the node q such that $(q, p) \in \text{IN}(p)$, q is nonred, and the shortest path from s to p passing through q is the minimum among those passing through the nonred neighbors of p. Note that, if the best nonred neighbor of p is not a candidate parent, then p increases its distance from s and should be colored *red*.

To color a node z, Procedure `Color` looks for an equivalent parent q of z. However, the information stored at node z allows to determine only whether q is a candidate parent of z. Node q is an equivalent parent of z if the distance of q in G' is equal to that in G. To this aim, a search of G' is performed, using $T(y)$. Namely, `Color` calls Procedure `Search_Equivalent_Path`(z, y), shown in Figure 4, which searches for a path from s to z in G', whose length is equal to $d(z)$.

```
    procedure Color($y$ : node)
1.  begin
2.  $M \longleftarrow \emptyset$      $\{M$ is a set$\}$
3.  Add($M, y$)
4.  while Non_Empty($M$) do
5.      begin
6.          $z \longleftarrow$ Extract($M$)
7.          if $z$ is not permanently colored then Search_Equivalent_Path($z, y$)
8.      end
9.  color red each arc with both endpoints red
10. end
```

Fig. 3. Color the nodes in $T(y)$ after increasing the weight of arc (x, y)

Procedure `Search_Equivalent_Path`(z, y) performs a depth-first search starting from z and looking for equivalent parents of the visited nodes, by traversing the arcs entering these nodes backward. This is done by using a stack Q which is initialized with z. During the execution of the procedure, Q contains a set of nodes $q_1, q_2, \ldots, q_k$ ($q_k = Top(Q)$) such that q_i is a candidate parent of q_{i-1}, for $i = 2, 3, \ldots, k$. Nodes in Q are colored *blue* since their permanent color is not yet determined. `Search_Equivalent_Path` repeatedly extracts the top node p from the stack and chooses the *next* best nonred neighbor of p in IN-OWN(p) and in $\overline{\text{IN-OWN}}(p)$, respectively; finally, it chooses node q as the best between the two neighbors found. We remark that each time that line 7 is executed a new node is returned, and that in the search of the next best nonred neighbor of a node the procedure first checks the old parent of the node in the shortest paths tree, as an equivalent parent.

When the next best nonred neighbor q of p is found, a number of cases may arise: if q does not exist or it is not an equivalent parent of p, then p is deleted from the stack (line 9) and will be later colored *red*. In all other cases, q is a candidate parent of p; therefore, if the algorithm will discover later that q does not increase its distance from s in G', then it follows that there is a path from s to p in G', that uses arc (q, p) and has length equal to $d(p)$; this implies that p will not increase its distance from s. The fact that q is a candidate parent of p is recorded by coloring arc (q, p) *blue* (line 12).

If q is a candidate parent of p we distinguish the following possibilities:

- If q is *blue*, then nothing is done (except for coloring *blue* arc (q, p)).
- If q is in $T(y)$ but it is still uncolored (line 27), then q is a candidate parent of p, but the algorithm is not able to determine whether q is an equivalent parent of p or not. In this case node q is colored *blue* and pushed in Q (line 29); therefore, q is the top node in the stack and the search of an alternative shortest path for z will continue from q.
- If q is a candidate parent of p, and q does not belong to $T(y)$ or it has been already colored either *pink* or *white* (line 13), then q is an equivalent parent

```
    procedure Search_Equivalent_Path($z, y$ : node)
1.  begin
2.     $Q \longleftarrow \emptyset$      {$Q$ is a stack}
3.     Push($Q, z$)
4.     $color(z) \longleftarrow blue$
5.     repeat
6.        $p \longleftarrow$ Top($Q$)
7.        let $q$ be the next best nonred neighbor of $p$ (first check the parent
          of $p$ in $T(s)$)
8.        if $q$ does not exist or $q$ is not a candidate parent of $p$
9.           then Pop($Q$)
10.          else
11.             begin
12.                color $blue$ arc $(q, p)$
13.                if $q \notin T(y)$ or $q$ is $pink$ or $q$ is $white$
14.                  then
15.                     begin     {an equivalent parent $q$ for $p$ is found}
16.                        Let $S$ be a search tree of the $blue$ nodes reachable
                           from $q$ via $blue$ arcs
17.                        for each node $v$ in $S$ not permanently colored do
18.                           begin
19.                              let $(w, v)$ be the $blue$ arc of $S$ entering $v$
20.                              if $(w, v)$ belongs to $T(s)$
21.                                 then $color(v) \longleftarrow white$
22.                                 else $color(v) \longleftarrow pink$; $P(v) \longleftarrow w$
23.                              color $white$ all nodes in $T(v)$ not
                                 permanently colored
24.                           end
25.                        $Q \longleftarrow \emptyset$
26.                     end
27.                  else     {$q \in T(y)$ and $q$ is $blue$ or uncolored}
28.                     if $q$ is not $blue$
29.                        then Push($Q, q$); $color(q) \longleftarrow blue$
30.             end
31.    until $Q$ becomes empty
32.    for each remaining $blue$ node $p$ do
33.       begin
34.          $color(p) \longleftarrow red$
35.          for each uncolored node $v \in children(p)$ do Add($M, v$)
36.       end
37. end
```

Fig. 4. Search an equivalent parent of node z in T(s)

of p, and p will be colored either *white* or *pink*, depending on whether q is the parent of p in $T(s)$ or not.

Note that, in this case, other nodes might have found an equivalent parent and, therefore, should be colored either *pink* or *white*. In fact, all *blue* nodes that are reachable from q by means of *blue* arcs have found an equivalent parent, and will be permanently colored either *pink* or *white* (lines 21 and 22). Moreover, if a *blue* node v has found an equivalent parent, then all nodes belonging to $T(v)$ that are *blue* or uncolored will be colored white (line 23); in fact, for each of these nodes, the parent in $T(s)$ is an equivalent parent in G'.

Observe that when a node is colored *pink*, Procedure `Search_Equivalent_Path` modifies the shortest paths tree. Let $T_c(s)$ be the subgraph of G whose node set is given by N, and whose $(n-1)$ arcs are the parent pointers of the nodes different from s after the execution of `Color`. Clearly, $T_c(s)$ is not necessarily a shortest paths tree.

The last step of `Search_Equivalent_Path` colors *red* all remaining *blue* nodes (lines 32-36). These are the nodes visited by the algorithm but not reachable by means of blue arcs from the equivalent parent that has been determined. For each *red* node, all its uncolored children in $T(y)$ are inserted in M and will be permanently colored in subsequent calls to `Search_Equivalent_Path`. Note that, if `Search_Equivalent_Path` does not find an equivalent parent, then it follows that all *blue* nodes will be colored *red*.

In Figure 5 we depict an example of execution of `Search_Equivalent_Path` after increasing the weight of an arc (x, y) of a graph G. Figure 5-(a) shows the arcs of G induced by the nodes in $T(y)$; solid arcs belong to $T(y)$, while dashed arcs do not; (x, y) is the unique arc entering y in G, while (q, z_7) is the unique arc of G going from a node outside $T(y)$ to a node inside $T(y)$. We assume that, before the execution of the *weight-increase* operation, all the arcs depicted in Figure 5-(a) have weight equal to zero. It follows that $d(y) = d(z_i)$, $1 \leq i \leq 7$, and $d(z_7) = d(q) + w_{q,z_7}$.

We consider the execution of `Search_Equivalent_Path`(z_1, y). We assume that, during the previous execution of `Search_Equivalent_Path`(y, y), node y has been already colored *red*, and nodes z_1 and z_5 have been put in M. Figure 5-(b) shows the graph of *blue* arcs visited by `Search_Equivalent_Path`(z_1, y); the numbers associated to the *blue* arcs represent the order in which they are visited during the search of alternative paths. Remember that, when a node is considered by Procedure `Search_Equivalent_Path`, its parent in $T(s)$ is always visited first; in the example, the arcs (y, z_1) and (y, z_5) are not colored *blue* since y is *red*. The dotted arcs in Figure 5-(b) represent the arcs belonging to the search tree S computed at line 16 of `Search_Equivalent_Path`(z_1, y). At the termination of the procedure, nodes z_1, z_4 and z_7 are colored *pink*, nodes z_2 and z_3 are colored *white*, while nodes z_5 and z_6 are colored *red*.

We now present the second phase of Procedure `Increase`, that, roughly speaking is Dijkstra's algorithm applied to the subgraph of G' induced by *red* arcs. Initially, the *red* nodes are inserted in a heap H. The main difference with

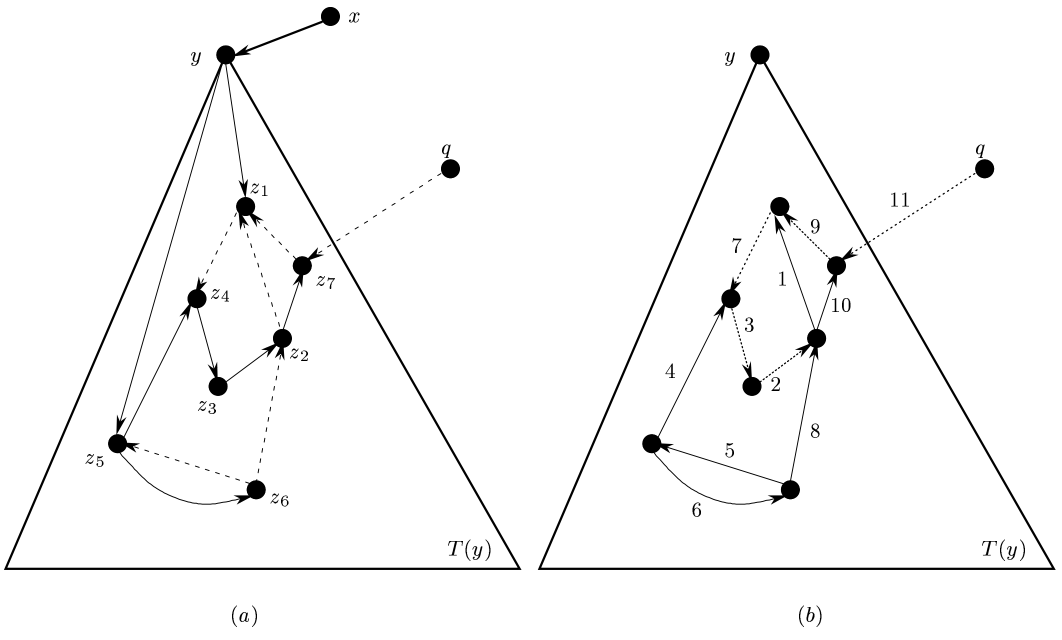

Fig. 5. Example of execution of Procedure `Search_Equivalent_Path`.

respect to the standard Dijkstra's algorithm is the priority given to each *red* node z in H, which is $\Delta(z)$ instead of $D(z)$; the priority $\Delta(z)$ is computed as $D(p) + w_{p,z} - D(z)$, where p is the best nonred neighbor of z in G' (lines 6–13). Then, the procedure repeatedly extracts node z with minimum priority from H and updates its distance label similarly to Dijkstra's algorithm (lines 14–26). Namely, when node z is extracted from H, the new distance of z has been determined and it is stored in $D'(z)$. After this, the procedure looks for shortest paths passing through z. To bound the running time, the algorithm consider only the paths from s to nodes h, such that (z, h) is *red*. Note that, $D(h) + \Delta(h)$ is the length of the current best known path from s to h in G'; if $D'(z) + w_{z,h} < D(h) + \Delta(h)$, then a shorter path from s to h has been found, and the priority of h in H is updated to the value $D'(z) + w_{z,h} - D(h)$ by Procedure `Heap_Improve`. This is done to restore the optimality condition on arc (z, h) in G'.

Concerning the second phase of `Increase`, considerations analogous to those given in the case of Procedure `Decrease` hold with respect to the solution proposed by Ramalingam and Reps in [16,17] and regarding the priorities of nodes in the heap. Hence, the correctness of `Increase` follows directly by the correctness of Procedure `Color`, and by the correctness of the solution given in [16,17]. In the next section we prove the correctness of `Color`.

```
    procedure Increase($x, y$ : node; $\epsilon$ : positive_real)
1.  begin
2.    $w_{x,y} \longleftarrow w_{x,y} + \epsilon$
3.    if $(x, y) \notin T(s)$ then update either $b_x(y)$ or $f_y(x)$ (depending on
      $owner(x, y)$) and EXIT
4.    Color($y$)
5.    $H \longleftarrow \emptyset$      {initialize an empty heap $H$}
6.    for each red node $z$ do
7.      begin
8.        let $p$ be the best nonred neighbor of $z$
9.        if $p \neq$ Null
10.         then $\Delta(z) \longleftarrow D(p) + w_{p,z} - D(z)$
11.         else $\Delta(z) \longleftarrow +\infty$
12.       Enqueue($H, \langle z, \Delta(z), p \rangle$)
13.     end
14.   while Non_Empty($H$) do
15.     begin
16.       $\langle z, \Delta(z), q \rangle \longleftarrow$ Extract_Min($H$)
17.       $D'(z) \longleftarrow D(z) + \Delta(z)$
18.       $P'(z) \longleftarrow q$
19.       for each red arc $(z, h)$ leaving $z$ do
20.         if $D'(z) + w_{z,h} - D(h) < \Delta(h)$
21.             then begin
22.                     $\Delta(h) \longleftarrow D'(z) + w_{z,h} - D(h)$
23.                     Heap_Improve($H, \langle h, \Delta(h), z \rangle$)
24.                  end
25.       uncolor all the red arcs $(q, z)$ entering $z$
26.     end
27.   for each red node $z$ do
28.     begin
29.       uncolor $z$
30.       for each arc $(v, z) \in$ IN-OWN($z$) do $b_v(z) \longleftarrow D'(z) - w_{v,z}$
31.       for each arc $(z, v) \in$ OUT-OWN($z$) do $f_v(z) \longleftarrow D'(z) + w_{z,v}$
32.       $D(z) \longleftarrow D'(z)$
33.       $P(z) \longleftarrow P'(z)$
34.       $\Delta(z) \longleftarrow 0$
35.     end
36. end
```

Fig. 6. Increase by quantity ϵ the weight of arc (x, y)

4.1 Correctness analysis

The correctness of `Color` is based on the properties stated in the following two lemmas.

Lemma 4.1 *A node q is permanently colored by Procedure* `Color` *if and only if $q \in T(y)$.*

Proof. First observe that, all the nodes that are inserted in the set M are colored *blue* in line 4 of Procedure `Search_Equivalent_Path`.

To show that $q \notin T(y)$ implies that q is uncolored, it is sufficient to observe that, initially q is uncolored, and if `Search_Equivalent_Path` detects q as best nonred neighbor, then q is not colored and the search is halted (see line 13).

We now prove that $q \in T(y)$ implies that q is colored. The proof is by induction on the number of nodes of the path from y to q in $T(y)$. The base case is given by node y that is colored *blue* in line 3 of the first execution of `Search_Equivalent_Path`. Let us suppose that a node $q \in T(y)$ is colored. If q is colored *red* (see line 34 of `Search_Equivalent_Path`), then the uncolored children of q in $T(y)$ are enqueued in M, while the others are already colored. If q is colored *white* or *pink*, then the uncolored nodes in $T(q)$ are colored *white* (see line 23 of `Search_Equivalent_Path`), while the remaining nodes in $T(q)$ are already colored. In the remaining case q is colored *blue*. The claim that $q \in T(y)$ implies that q is *permanently* colored follows from the fact that, upon termination of `Color`, all the *blue* nodes are colored either *white*, *pink* or *red* in lines 21, 22 or 34 of `Search_Equivalent_Path`, respectively. □

Lemma 4.2 *Upon termination of* `Color`*, $T_c(s)$ is a tree with root s such that, for each nonred and uncolored node v, the path from s to v in $T_c(s)$ does not pass through red nodes.*

Proof. Let T_i be the graph, with node set N and arc set given by the parent pointers of nodes, upon termination of i-th call to `Search_Equivalent_Path`. To prove the theorem, we first prove by induction that, for all i, T_i is a tree with root s, and that the subgraph of T_i induced by the *white* and *pink* nodes and the nodes not in $T(y)$ is a tree rooted in s.

The case $i = 0$ is trivial: in fact, $T_0 = T(s)$ and all the nodes of T_0 are uncolored. Since $T(s)$ is a tree the claim follows.

Assume that the claim is true for $i - 1$. By Lemma 4.1 all the nodes that do not belong to $T(y)$ are not colored, and the parent pointer of each of these nodes is not modified by Procedure `Search_Equivalent_Path`; therefore, to prove the claim for T_i it is sufficient to show that, for each $v \in T(y)$, there exists a path from v to s in T_i and that, if v is either *white* or *pink*, then along this path there are not red nodes.

We start with the case of nodes in $T(y)$ that are either *pink* or *white*. Note that, for each node that has been permanently colored in previous calls to `Search_Equivalent_Path`, the parent pointer is not changed, and that nodes permanently colored are not recolored. As a consequence, we will prove the claim for the nodes in $T(y)$ that are colored either *pink* or *white* during the i-th execution of `Search_Equivalent_Path`. During this computation T_{i-1} is modified if and only if an equivalent parent q for a blue node p, $p \in T(y)$, is found in line 13, and q satisfies the following properties: q is different from the parent of p in T_{i-1}; $q \notin T(y)$ or q has been colored *white* or *pink* in a previous call to `Search_Equivalent_Path`. In this case p is colored *pink* and q becomes the new parent of p in T_i. Since q is not *red* it follows, by the inductive hypothesis, that there exists a path from p to s that does not pass through *red* nodes.

In this phase, other nodes are colored either *white* or *pink* (lines 21, 22 and 23 of `Search_Equivalent_Path`). Note that, p and the nodes colored in lines 21, 22 and 23 belong to a graph built as follows. The nodes colored in lines 21 and 22 belong to the tree S of *blue* arcs rooted at q that is computed at line 16. Every time a node z of S is permanently colored *white* or *pink* the tree $T_{i-1}(z)$, properly pruned of the nodes already permanently colored, is appended to S. Let $\mathcal{T} = (N(\mathcal{T}), A(\mathcal{T}))$ be the graph obtained by expanding S as above at the end of the i-th call to `Search_Equivalent_Path`. We now show that $\mathcal{T}$ is a tree, that is: *a*) $|A(\mathcal{T})| = |N(\mathcal{T})| - 1$; *b*) $\mathcal{T}$ has no cycle.
a) Every node v in $\mathcal{T}$ has a single entering arc. In fact, if $v \notin S$ this arc is the one coming from the parent of v in T_{i-1}; if $v \in S$ we distinguish two cases: the arc is the single arc entering v in S, or it is the one coming from the parent of v in T_{i-1}, when z has been permanently colored in line 23 before being visited in line 17.
b) Let us assume that there exists a cycle $C = (z_1, z_2, \dots, z_k = z_1)$ whose arcs belong to $\mathcal{T}$. Among the arcs of C there exists one arc, say (z_j, z_{j+1}), such that: $(z_j, z_{j+1}) \notin S$ (S is a tree), $z_j \in T_{i-1}(v)$, for some $v \in S$, and $z_{j+1} \in S$. It follows that $(z_j, z_{j+1}) \in T_{i-1}(v)$, and hence that $z_{j+1} \in T_{i-1}(v)$. If z_{j+1} is an ancestor of v in S, then z_{j+1} has been permanently colored *white* or *pink* in lines 21 or 22, respectively, before permanently coloring v. This contradicts the fact that $z_{j+1} \in T_{i-1}(v)$ at the end of the i-th call to `Search_Equivalent_Path`. In fact, when $T_{i-1}(v)$ is colored *white* in line 23, it is properly pruned of the nodes already permanently colored. If v is an ancestor of z_{j+1} in S, then z_{j+1} has at least two entering arcs in $\mathcal{T}$, contradicting part (*a*) above.

To complete the proof of the lemma, we need to show that if p, $p \in T(y)$, is colored *red* or it is uncolored, then there exists a path from p to s in T_i. This easily follows by induction and by the fact that the parent pointer of p is not modified by `Search_Equivalent_Path`. □

Theorem 4.3 *Let $G = (N, A)$ be a digraph with arbitrary arc weights, if a weight-increase operation is performed on arc (x, y), then Procedure* `Color` *colors a node $z \in T(y)$ red (nonred) if and only if $d'(z) > d(z)$ ($d'(z) = d(z)$).*

Proof. By Lemma 4.1 all nodes in $T(y)$ are colored; we prove by contradiction that the colors are correct. We say that a node z receives a *wrong* color if, either $d'(z) > d(z)$ and z is nonred, or $d'(z) = d(z)$ and z is *red*. Let z be a node that receives a wrong color during the execution of Procedure `Color`, and let us suppose that: *(i)* all nodes that belong to the path in $T_c(s)$ between s and z are correctly colored. If the lemma is not true then such a node exists. We distinguish two cases depending on the color given to z.

(1) z is colored *white* or *pink* (nonred). In this case we show that $d'(z) = d(z)$, and hence that z cannot be colored *red* by the algorithm. Let v be the parent of z in $T_c(s)$. If the algorithm has chosen v as the parent of z in $T_c(s)$, then $d(z) = d(v) + w_{v,z} = d'(v) + w_{v,z}$ and, by Lemma 4.2, v has been colored nonred. Therefore, if the color of z is wrong, then the color of v is wrong as well, and should be *red*, contradicting condition *(i)* above.

(2) z is colored *red*. In this case we show that $d'(z) > d(z)$. Let q be the parent of z in $T(s)$ and $T_c(s)$ (if z is red, then the parent of z does not change in the coloring phase). Observe that, since z is *red* then also q is *red*. By condition *(i)* above the color of q is correct; therefore, if $d'(z) = d(z)$ then the shortest path from s to z in G' cannot include q, and z cannot be *white*.

It remains to show that z cannot be *pink*. Assume by contradiction that z should be *pink* because there is a path from s to z passing through nodes $s = w_0, w_1, \dots, w_k = z$, whose length in G' is equal to $d(z)$; it follows that $d'(w_i) = d(w_i)$, $i = 0, 1, \dots k$, and none of these nodes should be colored *red*. Let w_r, $0 < r \leq k$, be the first node in this path that is wrongly colored *red*; clearly w_{r-1} is not colored *red* by the algorithm. Let us focus on the call to `Search_Equivalent_Path` that colors w_r *red*. During this call w_r is pushed in the stack Q and a candidate parents of w_r is searched. One of two possible cases may arise, according on whether w_{r-1} has been considered or not as a candidate parent of w_r.

- If w_{r-1} has not been considered as a candidate parent of w_r, then the test of line 13 succeeds. Let v be the last node that has been considered as candidate parent of w_r. It follows that arc (v, w_r) is colored *blue* and that w_r is in the stack Q when the test of line 13 succeeds. Note that, there exists a path of blue arcs from the node that is on top of the stack and w_r. This implies that w_r is colored nonred.
- If w_{r-1} has been considered as candidate parent of w_r, then arc (w_{r-1}, w_r) is colored *blue*. If w_{r-1} does not belong to $T(y)$ or it has been already colored nonred then w_r is colored *pink* (line 22); if w_{r-1} belongs to $T(y)$ and it is not permanently colored, it follows that w_{r-1} will be colored nonred during this call of `Search_Equivalent_Path`, and also w_r will be colored nonred.

In both cases we obtain a contradiction; this implies that there is no path from s to z in G' such that, for each node w_i in this path, $d'(w_i) = d(w_i)$ and hence $d'(z) > d(z)$. □

Let us partition the *white* nodes in *good-white* and *bad-white*. The *bad-white* nodes are the nodes colored *white* by the algorithm for which more than one entering arc is visited in line 7 of `Search_Equivalent_Path`. As an example, node z_2 in Figure 5 is *bad-white*, while z_3 is *good-white*; in fact, Procedure `Search_Equivalent_Path`(z_1, y) has visited a unique arc entering z_3, that is arc (z_4, z_3), while it has visited two arcs entering z_2, that is (z_3, z_2) and (z_6, z_2). In what follows we show how to bound the number of *bad-white* nodes.

Procedure `Search_Equivalent_Path` when called on node z, performs a DFS-search of the subgraph of G' that reaches z by traversing the arcs of G backward starting from z. For each visited node v, the arcs (w, v) are traversed (and colored *blue*) such that w is a candidate parent for v. The graph induced by the *blue* arcs is denoted as H. The result of the search is a tree, denoted as $T_{\mathrm{DFS}}(z)$, such that z is reachable from any other node of the tree. Given a node v in $T_{\mathrm{DFS}}(z)$, we denote as $T_{\mathrm{DFS}}(v)$ the subtree of $T_{\mathrm{DFS}}(z)$ rooted at v. In what follows we show the properties of $T_{\mathrm{DFS}}(z)$ that allow us to prove that each *bad-white* node belongs to a zero-length cycle of G, i.e., a cycle in H.

According to the standard terminology (see, e.g., [6]), and taking into account that each arc of $T_{\mathrm{DFS}}(z)$ is directed from a child to its parent, the arcs of $T_{\mathrm{DFS}}(z)$ are partitioned as follows:

- a TREE arc is from a child to its parent in $T_{\mathrm{DFS}}(z)$;
- a FORWARD arc is from a descendant to an ancestor in $T_{\mathrm{DFS}}(z)$;
- a BACKWARD arc is from an ancestor to a descendant in $T_{\mathrm{DFS}}(z)$;
- a CROSS arc connects two nodes of $T_{\mathrm{DFS}}(z)$ that are not ancestors of each other.

Recall that, the nodes visited during `Search_Equivalent_Path` are stored in a stack Q. We assume that stack operations are performed at integral time instants, and denote as E_v the time when node v is enqueued in Q and as D_v the time when node v is dequeued from Q. The dequeuing time of the nodes (which can be obtained by reversing the postorder numbering of the DFS-search) defines an ordering of the visited nodes, as described in [6].

We remark that a node a is ancestor of b in $T_{\mathrm{DFS}}(z)$ if and only if $E_a < E_b < D_b < D_a$. On the other side, if a and b are not ancestors of each other, then they can not be in Q at the same time, i.e., either $E_a < D_a < E_b < D_b$, or $E_b < D_b < E_a < D_a$. In particular, for each cross arc (a, b), we have $D_a < D_b$.

Lemma 4.4 *Given a node b of H, let a be the last node in the ordering defined by Procedure* `Search_Equivalent_Path`*(z, y), such that b is reachable from a in H. Then $b \in T_{\mathrm{DFS}}(a)$.*

Proof. We assume that $D_b < D_a$, because if $a \equiv b$, then the lemma trivially follows. Let $\pi(a,b)$ be a nonempty path from a to b in H. By contradiction, let (u,v) be the first arc in $\pi(a,b)$ such that $u \in T_{\mathrm{DFS}}(a)$ and $v \notin T_{\mathrm{DFS}}(a)$. We distinguish the following cases:

(1) (u,v) is a TREE or a FORWARD arc: this means $u \in T_{\mathrm{DFS}}(v)$. Since $u \in T_{\mathrm{DFS}}(a)$, then a and v have a common descendant in $T_{\mathrm{DFS}}(z)$, and since $v \notin T_{\mathrm{DFS}}(a)$, then $a \in T_{\mathrm{DFS}}(v)$. In this case $D_a < D_v$ and, since there exists a path from v to b, this contradicts the hypothesis that a is the last node in the ordering that reaches b.

(2) (u,v) is a BACKWARD arc: this implies $v \in T_{\mathrm{DFS}}(u)$ and hence $v \in T_{\mathrm{DFS}}(a)$, contradicting the hypothesis that $v \notin T_{\mathrm{DFS}}(a)$.

(3) (u,v) is a CROSS arc: in this case $D_u < D_v$. Since b is reachable from v and (by contradiction) $v \notin T_{\mathrm{DFS}}(a)$, then $D_v < D_a$ (by hypothesis a is the last node in the ordering that reaches b). This implies that $D_u < D_v < D_a$. Since $u \in T_{\mathrm{DFS}}(a)$, then $E_a < E_u < D_u < D_a$. It follows that $E_a < E_u < D_u < D_v < D_a$. Due to the stack policy, $E_a < D_v < D_a$ implies $E_a < E_v < D_v < D_a$ and then $v \in T_{\mathrm{DFS}}(a)$, contradicting the hypothesis that $v \notin T_{\mathrm{DFS}}(a)$. □

Lemma 4.5 *If v is a bad-white node, then v belongs to a zero-length cycle.*

Proof. Note that, if some node is colored *white*, then there is a node z such that the test at line 13 of `Search_Equivalent_Path`(z,y) is satisfied. Let $\overline{q}$ be the node satisfying the condition of line 13, and let v be a node reachable from $\overline{q}$ in H. We prove the lemma by showing that:

(1) if v is an ancestor of $\overline{q}$ in $T_{\mathrm{DFS}}(z)$, then v cannot be *bad-white*;

(2) if v is not an ancestor of $\overline{q}$ in $T_{\mathrm{DFS}}(z)$, then v belongs to a zero-length cycle of *blue* arcs.

(Case 1) v is an ancestor of $\overline{q}$ in $T_{\mathrm{DFS}}(z)$. Let v' be the last node in the path from $\overline{q}$ to v in $T_{\mathrm{DFS}}(z)$ (possibly $v' = \overline{q}$). If v is *white*, then the arc (v', v) belongs to $T(s)$ (i.e., the shortest paths tree before the *weight-increase* operation). Since the old parent in $T(s)$ is visited first by `Search_Equivalent_Path`(z,y), then v' is visited as the first candidate parent of v. Since we perform a DFS-search, then $\overline{q}$ is found before that v is again on the top of the stack. It follows that v cannot be *bad-white*.

(Case 2) v is not an ancestor of $\overline{q}$ in $T_{\mathrm{DFS}}(z)$, that is $D_v < D_{\overline{q}}$. Let us consider the last node a (in the ordering defined by `Search_Equivalent_Path`(z,y)) such that v is reachable from a in H. Since the DFS-search is halted when $\overline{q}$ is found, then the nodes dequeued after $\overline{q}$ are all ancestors of $\overline{q}$ in $T_{\mathrm{DFS}}(z)$. On the other side, v is reachable from $\overline{q}$. Hence the last node a in the order such that v is reachable from $\overline{q}$ must be an ancestor of $\overline{q}$ (possibly $\overline{q}$ itself),

that is $D_{\overline{q}} \leq D_a$. Since $D_v < D_{\overline{q}} \leq D_a$, it follows that $a \not\equiv v$. By Lemma 4.4 $v \in T_{\mathrm{DFS}}(a)$, i.e., there exists a path of tree (*blue*) arcs from v to a. This path along with the path of *blue* arcs from a to v in H form a zero-length cycle of *blue* arcs. □

According to Lemma 4.5, the *bad-white* node z_2 of Figure 5 belongs to a zero-length cycle of *blue* arcs (as an example, the one given by the arcs (z_1, z_4), (z_4, z_3), (z_3, z_2) and (z_2, z_1)).

Let n_R, n_P and n_{BW} be the number of nodes that have been colored *red*, *pink*, and *bad-white*, respectively, at the end of Procedure `Color`.

Lemma 4.6 *Let $G = (N, A)$ be a digraph with arbitrary arc weights. If G has a k-baf and a weight-increase operation is performed on arc (x, y), then nodes in $T(y)$ are colored in $O(\min\{m \log n, (n_{BW} + n_R + n_P)k \log n + n\})$ worst case time.*

Proof. The term n in the stated complexity is the cost of a preliminary visit of $T(y)$, needed to answer in constant time to the test at line 13 of `Search_Equivalent_Path`.

A node is inserted in M at most once and only *red*, *pink* and *bad-white* nodes are inserted in M. Since the owner of each arc that is colored *red* is a *red* node, it follows that the total cost of `Color` without the cost of `Search_Equivalent_Path`, is $O(n_P + n_{BW} + n_R \cdot k)$.

To bound the cost of all calls to `Search_Equivalent_Path`, we first observe that a node is inserted in Q at most once and that all nodes in Q are permanently colored. Let us compute the number of arcs scanned during all calls to `Search_Equivalent_Path` to color nodes permanently. Concerning *good-white* nodes, this number is clearly bounded by n. For each *red* node z, the arcs scanned belong either to $\textsc{in-own}(z)$ or to the ownership of another *red* node (see line 7), except for the last scanned arc. Since each of these arcs is scanned at most a constant number of times, and the graph admits a k-baf, then the total number of arcs scanned to color *red* nodes during all calls to `Search_Equivalent_Path` is bounded by $O(n_R(k + 1))$.

The cost of all calls to Procedure `Search_Equivalent_Path` for the *pink* and *bad-white* nodes is the cost of traversing the arcs colored *blue* and entering these nodes (see lines 11–30). Let B be the number of these arcs, and PBW be the set of all *pink* and *bad-white* nodes. The value of B can be bounded as follows; let b_p and $\overline{b}_p$ be the total number of blue arcs (q, p) in $\textsc{in-own}(p)$ and $\overline{\textsc{in-own}}(p)$, respectively, such that node q has been considered as candidate parent of p when p is on top of Q. The total number of blue arcs is:

$$B = \sum_{\forall p \in PBW} (b_p + \overline{b}_p).$$

Since G admits a k-baf, then for each $p \in PBW$, $b_p \leq k$. It follows that:

$$\sum_{\forall p \in PBW} b_p \leq k(n_P + n_{BW}).$$

To bound the sum of the value $\overline{b}_p$ over all nodes in PBW, observe that $\overline{b}_p$ is bounded by one plus the number of *blue* arcs (q, p), such that $(q, p) \in \overline{\text{IN-OWN}}(p)$ and q is either *bad-white*, *pink* or *red*. Since $(q, p) \in \text{OUT-OWN}(q)$ and the graph has a k-baf, if q is colored either *pink* or *bad-white*, then it follows that

$$\sum_{\forall p \in PBW} \overline{b}_p \leq (k+1)(n_{BW} + n_P + n_R).$$

Each *blue* arc is traversed at most a constant number of times (in lines 7, 12 and 16 of `Search_Equivalent_Path`) and hence $B = O(k(n_{BW} + n_P + n_R))$. Since each of these arc traversal has logarithmic cost, then the cost of all calls to `Search_Equivalent_Path` for the nodes in PBW is $O(k(n_{BW} + n_P + n_R) \log n)$.

The above discussion implies that the worst case cost of `Color` is $O((n_{BW} + n_R + n_P)k \log n + n)$. To prove the bound $O(m \log n)$ it is sufficient to note that every time an arc (v, w) is traversed by `Search_Equivalent_Path`, both v and w are permanently colored during the same call to `Search_Equivalent_Path`. This implies that arc (v, w) will never be traversed again during the execution of `Color`, and the claim follows. □

Theorem 4.7 *Let $G = (N, A)$ be a digraph with arbitrary arc weights. If G has a k-baf, then it is possible to update $T(s)$ and the distances of nodes from s after a weight-increase operation, in $O(\min\{m \log n, (n_{BW} + n_R + n_P)k \log n + n\})$ worst case time.*

Proof. By Lemma 4.6, the cost of `Color` is $O(\min\{m \log n, (n_{BW} + n_R + n_P)k \log n + n\})$. The *red* arcs can be colored in $O(k \cdot n_R)$ time; the same time is required to uncolor such arcs (line 25 of `Increase`).

The red nodes are enqueued in H (lines 6–13) with priority given by the difference between the length of the shortest path passing through a nonred neighbor of z and $d(z)$. In order to find the best nonred neighbor of each *red* node the algorithm scans in the worst case all the $k \cdot n_R$ *red* arcs plus, for each *red* node z, k arcs in $\text{IN-OWN}(z)$ and the first nonred arc in $\overline{\text{IN-OWN}}(z)$. This requires $O(k \cdot n_R \log n)$ worst case time.

We now show that the cost of the **while** loop (lines 14–26) is equal to $O(k \cdot n_R \cdot \log n)$. In the loop n_R *red* nodes are dequeued, because each *red* node is inserted in H exactly once, as can be easily shown using the monotonicity of the priorities of nodes in H. Observe that, when a *red* node z is extracted from H, and therefore it improves its distance from the source, Procedure `Increase` traverses all *red* arcs in $\textsc{out}(z)$. This means that $k \cdot n_R$ *red* arcs are scanned, and hence, the total cost of the calls to `Heap_Improve` is $O(k \cdot n_R \cdot \log n)$.

In the last phase (lines 27–35), for each *red* node z, the values $b_v(z)$ and $f_v(z)$ in $\overline{\textsc{in-own}}(v)$ and $\overline{\textsc{out-own}}(v)$, respectively, are updated by scanning again the lists $\textsc{in-own}(z)$ and $\textsc{out-own}(z)$ (lines 30–31). Since each of these operations requires $O(\log n)$ time, this gives an overall worst case time of $O(k \cdot n_R \cdot \log n)$ for this phase.

Again, to show the bound $O(m \log n)$ it is sufficient to note that each *red* arc is traversed at most a constant number of times during the execution of `Increase`. □

Corollary 4.8 *Let $G = (N, A)$ be a digraph with arbitrary arc weights and no zero-length cycles. If G has a k-baf, then it is possible to update $T(s)$ and the distances of nodes from s after a weight-increase operation, in* $O(\min\{m \log n, (n_R + n_P)k \log n + n\})$ *worst case time.*

5 Insertions and deletions of arcs

The algorithms described for the case of increments and decrements of arc weights rely on a preliminary computation to decide a k-baf for a digraph $G = (N, A)$, that does not change throughout the sequence of updates. If insertions and deletions of arcs are allowed, it is necessary to decide dynamically the ownership of new arcs in order to bound the computational cost of the updates to the data structures.

Again, we assume that before any insertion or deletion of arc there is no negative-length cycle in G. We first consider the case that each node is reachable from the source during the updates and prove the bounds related to this case. In particular, we first provide a solution giving the same worst case bounds of Sections 3 and 4, and then provide a solution whose output bounded cost is amortized over a sequence of arc insertions and deletions. Then, we discuss the modifications needed to the algorithms and to the complexity bounds when we allow the graph to become disconnected during the updates.

We do not provide any pseudocode for Procedures `Insert` and `Delete` and simply point out the main differences with respect to `Decrease` and `Increase`. The main idea is to compute a new 2-approximated k-baf for the digraph obtained after each change. This can be done in $O(m+n)$ time by using Algorithm `2-Approx` (see Section 2.1). Then, for each node x, the data structures $\textsc{in-own}(x)$, $\overline{\textsc{in-own}}(x)$, $\textsc{out-own}(x)$ and $\overline{\textsc{out-own}}(x)$ are updated in $O(m \cdot \log n)$ worst case time. Note that this cost is dominated by the worst case cost of both `Decrease` and `Increase`, that is $O(m \cdot \log n)$.

In what follows, we provide a solution whose output bounded cost is amortized over a sequence of arc insertions and deletions. We introduce the following additional changes. A new k-baf is computed from scratch for the current graph every $k/2$ operations that either insert or delete an arc. When an arc is inserted between two successive recomputation of the accounting function, one of its endpoints is arbitrarily chosen as the owner of the arc. This guarantees that, while processing a sequence of updates, the current ownership of the arcs is within a constant factor of the optimal one.

In the following we denote as $\overline{\textsc{in-own}}(x)$ and $\overline{\textsc{out-own}}(x)$ the *sets* of arcs not contained in the current ownership of node x. In addition, for each node x, we store two priority queues containing all the arcs in $\textsc{in}(x)$ and $\textsc{out}(x)$, denoted as $\textsc{in-adj}(x)$ and $\textsc{out-adj}(x)$, respectively, and defined as follows:

(1) $\textsc{in-adj}(x)$ is a min-based priority queue where the priority of arc (y, x) (of node y), denoted as $f_x(y)$, has one of two possible meanings:
- if $(y, x) \in \overline{\textsc{in-own}}(x)$, then $f_x(y)$ is the computed value of $f\_level_x(y)$;
- if $(y, x) \in \textsc{in-own}(x)$, then $f_x(y)$ is the value of $f\_level_x(y)$ the last time that arc (y, x) was in $\overline{\textsc{in-own}}(x)$;

(2) $\textsc{out-adj}(x)$ is a max-based priority queue where the priority of arc (x, y) (of node y), denoted as $b_x(y)$, has one of two possible meanings:
- if $(x, y) \in \overline{\textsc{out-own}}(x)$, then $b_x(y)$ is the computed value of $b\_level_x(y)$;
- if $(x, y) \in \textsc{out-own}(x)$, then $b_x(y)$ is the value of $b\_level_x(y)$ the last time that arc (x, y) was in $\overline{\textsc{out-own}}(x)$.

Note that, if an arc $(y, x) \in \textsc{in}(x)$ ($(x, y) \in \textsc{out}(x)$) actually belongs to $\textsc{in-own}(x)$ ($\textsc{out-own}(x)$), then its priority in $\textsc{in-adj}(x)$ ($\textsc{out-adj}(x)$) might be obsolete; in fact, this priority is not updated when the distance of node y (x) from the source is updated. Therefore, when arcs are extracted from $\textsc{in-adj}(x)$ or $\textsc{out-adj}(x)$ during the update procedures in order to find the “right” neighbors of x (see the description of Procedures `Decrease` and `Increase`), at most $O(k)$ of these arcs actually do not belong to $\overline{\textsc{in-own}}(x)$ or $\overline{\textsc{out-own}}(x)$, respectively.

Let us consider now a sequence $\sigma = \langle \mu_1, \mu_2, \ldots, \mu_h \rangle$ of arc updates performed starting from a graph G_0, consisting of insertions and deletions of arcs, as well as weight updates. For $i = 1, 2, \ldots, h$, each μ_i is performed on the graph G_{i-1} and gives a new graph G_i; the corresponding update procedure colors $n_R(\mu_i)$ *red* nodes, $n_P(\mu_i)$ *pink* nodes, and $n_{BW}(\mu_i)$ *bad-white* nodes ($n_P(\mu_i)$ and $n_{BW}(\mu_i)$ being 0 when μ_i is either a *weight-decrease* or an *insert* operation).

Theorem 5.1 *Let $G_0 = (N, A)$ be a digraph with arbitrary arc weights, and let $\sigma = \langle \mu_1, \mu_2, \ldots, \mu_h \rangle$ be an arbitrary sequence of arc updates (insertions, deletions, and weight updates) to be performed on G_0. Let us suppose that G_0 and the graph G_i resulting from any arc operation $\mu_i \in \sigma$, have a k-baf and no negative-length cycle. It is possible to update a shortest paths tree and the distances of nodes from a given source s while processing σ, in $O(n + (n_R(\mu_i) + n_P(\mu_i) + n_{BW}(\mu_i)) \cdot k \cdot \log n)$ amortized time, for each arc update μ_i.*

Proof. We limit our attention to sequences σ of only insertions and deletions of arcs; in fact, the cost of these operations dominates the cost of `Decrease` and `Increase`, respectively. Let us fragment σ in subsequences of $k/2$ operations. For each subsequence σ_s of σ, the quantity N_R denotes the sum of the number of nodes that are colored *red* while processing σ_s, i.e., $N_R = \sum_{\mu_i \in \sigma_s} n_R(\mu_i)$. Each subsequence σ_s is processed as follows:

(1) After σ_s compute a new 2-approximated accounting function for the current graph. This requires $O(m + n)$ worst case time using Algorithm `2-Approx`.

(2) After σ_s restore the current ownership for arcs. In particular, for each arc (x, y) changing its owner, say, from x to y, the following operations have to be carried out:

   a. set the new owner of (x, y) to y;

   b. move (x, y) from OUT-OWN(x) to IN-OWN(y);

   c. "refresh" the priority $b_x(y)$ in OUT-ADJ(x): this is necessary since (x, y) is back in the set $\overline{\text{OUT-OWN}}(x)$.

   At most $O(m)$ operations of kind a or b are performed, each requiring constant time. Each refresh operation requires $O(\log n)$ worst case time only if y, the new owner of (x, y), has been colored *red* since (x, y) has been owned by x, and this may happen only for $O(k \cdot N_R)$ arcs throughout the execution of σ_s. Otherwise, i.e., if y has not been colored *red*, the refresh operation requires constant time. The above discussion implies that the total worst case cost of this step is $O(m + k \cdot N_R \cdot \log n)$.

(3) After each arc operation $\mu_i \in \sigma_s$, compute the new shortest paths tree, or detect the introduction of a negative-length cycle: as shown in Theorems 3.1 and 4.7, this can be done in $O(k \cdot n_R(\mu_i) \cdot \log n)$ worst case time for an arc insertion, and $O(n + k \cdot (n_R(\mu_i) + n_P(\mu_i) + n_{BW}(\mu_i)) \cdot \log n)$ for an arc deletion.

By 1, 2, and 3 above, the total worst case time needed to process σ_s is:

$$O(m + n + k \cdot N_R \cdot \log n + n \cdot k + \sum_{\mu_i \in \sigma_s} (n_R(\mu_i) + n_P(\mu_i) + n_{BW}(\mu_i)) \cdot k \cdot \log n)$$

Since $m \leq n \cdot k$ and $\sum_{\mu_i \in \sigma_s} (n_R(\mu_i) + n_P(\mu_i) + n_{BW}(\mu_i)) \geq N_R$, then the total cost becomes:

$$O(n \cdot k + \sum_{\mu_i \in \sigma_s} (n_R(\mu_i) + n_P(\mu_i) + n_{BW}(\mu_i)) \cdot k \cdot \log n)$$

For each arc operation μ_i, this leads to the following amortized bound:

$$O(n + (n_R(\mu_i) + n_P(\mu_i) + n_{BW}(\mu_i)) \cdot k \cdot \log n).$$

□

Corollary 5.2 *If zero-length cycles are not allowed, then it is possible to update a shortest paths tree and the distances of nodes from s, or to detect the introduction of a negative-length cycle, in $O(n + (n_R(\mu_i) + n_P(\mu_i)) \cdot k \cdot \log n)$ amortized time per arc update.*

5.2 Disconnected case

In this section we extend our algorithms to the case that while performing a sequence of update operations the current graph can be disconnected. We assume that before the execution of each operation no negative-length cycle is present in G. First, we observe that the deletion of an arc might disconnect the graph. This implies that the new distance of a node z, belonging to the newly created component, must be $+\infty$. Note that, Procedure `Color` colors z *red* and z is inserted in the priority queue H of `Increase` (see line 16) with priority $+\infty$. Since there are no paths from s to z, then the priority of z in H is not decreased. This implies that z is extracted from H with priority $+\infty$ and $D(z)$ is updated accordingly.

The difficult case is when we insert an arc (x, y) such that $d(x)$ is finite, while $d(y) = +\infty$. In fact, in this case the problem of updating the shortest paths is equivalent to that of computing shortest paths from scratch in the subgraph reachable from y. For this case, Ramalingam and Reps [16,17] propose a solution running in $O(||\delta|| \cdot |\delta|)$ worst case time, where δ is the considered insertion, $|\delta|$ is the number of affected nodes, and $||\delta||$ is the number of affected nodes plus the number of arcs having at least one affected endpoint.

Let $N_U = \{z \mid d(z) = +\infty$ and z is reachable from $y\}$ and $G_U = (N_U, A_U)$ be the subgraph of G induced by the nodes of N_U. The nodes in N_U are affected by the insertion of (x, y), and hence $|N_U| \leq n_R$. Since G admits a k-baf and G_U is a subgraph of G, then G_U admits a k-baf and therefore $|A_U| \leq k|N_U| \leq kn_R$. On the other side, also arcs in the boundary of G_U are to be scanned. In particular, if an arc (z, v) is scanned with $z \in N_U$ and $v \notin N_U$, then either (z, v) is owned by z or it is scanned by priority and hence v is *red*. This implies that the scanned arcs in the boundary of G_U are at most kn_R.

The proposed algorithm is summarized below:

(1) Compute G_U: this is done by visiting G from y as follows. Every time a node z such that $d(z) = +\infty$ is visited, traverse the arcs $(z, v) \in$ OUT-ADJ(z) such that $b_z(v) = d(v) - w_{z,v} = +\infty$, which are stored in the highest positions of OUT-ADJ(z), plus the arcs in OUT-OWN(z). This costs $O(kn_R \log n)$ in the worst case.
(2) Apply Bellman-Ford's algorithm to G_U to compute, for each $z \in N_U$, the distance $d(y, z)$ from y to z in G_U. This costs $O(|N_U||A_U|) = O(kn_R^2)$ in the worst case.
(3) There is no arc from a node $z \in N \setminus N_U$ to a node $v \in N_U$ such that $d(z)$ is finite, while $d(v) = +\infty$, except for (x, y). This implies that the shortest path from s to x in G plus arc (x, y) form a shortest path from s to y in G' whose length is $d(x) + w_{x,y}$. For the same reason, for each $z \in N_U$, $d'(z)$ is computed as $d(x) + w_{x,y} + d(y, z)$. This costs $O(|N_U|) = O(n_R)$ in the worst case.
(4) For each $(u, v) \in A_U$, update $b_u(v)$ in OUT-ADJ(u) according to the computed distances of the nodes in N_U. This costs $O(kn_R \log n)$ in the worst case.
(5) For each $z \in N_U$, consider the arcs $(z, v) \in$ OUT(z) such that $(z, v) \in$ OUT-OWN(z) or $(z, v) \in$ OUT-ADJ(z) and $b_z(v) > d'(z)$; compute $\Delta(v)$ as $d'(z) + w_{z,v} - d(z)$ and put v into a priority queue Q with priority $\Delta(v)$, or update its priority to the value $\Delta(v)$ if v already belongs to Q. This costs $O(kn_R \log n)$ in the worst case.
(6) Update the shortest paths, and the local data structures at the nodes, by applying Procedure `Decrease` starting from the nodes in Q. This costs $O(kn_R \log n + n)$ amortized over a sequence of $k/2$ operations.

Summing up the costs at points 1–6 above we obtain a cost of $O(kn_R^2 + kn_R \log n + n)$ amortized per insertion over a sequence of $k/2$ operations.

6 Conclusions and open problems

We have proposed a new fully dynamic solution for the single source shortest paths problem on digraphs with arbitrary arc weights. It is possible to modify the algorithms proposed in this paper in order to match the worst case bound of the solution in [16,17], that is $O(m + n \log n)$ time per operation. Unfortunately, in this case the proposed algorithms are no longer efficient in terms of output complexity (see [7] for the details).

An interesting problem is to extend the bounds proposed in the paper to the *batch* problem, in which an input update is a *set* of arc modifications, instead of a single arc modification. Another problem is to extend the technique to maintain the all pairs shortest paths in a digraph with arbitrary arc weights. It would be also interesting to investigate whether our technique to handle zero-length cycles might provide new insights with respect to the dynamic reachability problem. In fact, finding a fully dynamic solution which is better in the worst case than recomputing reachability from scratch in a general digraph is a longstanding open problem. Furthermore, as Ramalingam and Reps noted in [16,17], under a certain model of computation there exists no efficient output bounded solution for the fully dynamic reachability problem on digraphs.

Output complexity has been experimentally shown a useful parameter to evaluate the practical efficiency of dynamic algorithms for the single source shortest paths problem in the case of positive weights [12]. The experiments in [7] show that on randomly generated test sets, dynamic algorithms based on the technique of Edmonds and Karp are experimentally faster by several orders of magnitude than the recomputation from scratch. The paper also shows that the algorithm of [16,17] is always faster than that presented in this paper when zero-length cycles are not present. On the other hand, if several zero-length cycles are present, then the algorithm of this paper and its variants are experimentally preferable.

Acknowledgments

We are very grateful to two anonymous referees for their careful revision of the paper, that allowed us to significantly improve the quality of the paper. We specially thank one of the referees for pointing out an error in a preliminary version 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. Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms (SODA90), pp. 32–42, 1990.

[3] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, vol. 12, n. 4 (1991), 615–638.

[4] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewidth. Part I: Sequential algorithms. *Algorithmica*, vol. 27, n. 3 (2000), 212–226.

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

[6] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*, MIT Press, Cambridge, MA (1990).

[7] C. Demetrescu, D. Frigioni, A. Marchetti-Spaccamela, U. Nanni. Maintaining Shortest Paths in Digraphs with Arbitrary Arc Weights: An Experimental Study. Proceedings of the 3rd Workshop on Algorithm Engineering (WAE2000). *Lecture Notes in Computer Science*, vol. 1982, pp. 218–229, Springer, 2000.

[8] J. Edmonds, R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, vol. 19 (1972), 248–264.

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

[10] P. G. Franciosa, D. Frigioni, R. Giaccio. Semi dynamic breadth-first search in digraphs. *Theoretical Computer Science*, vol. 250 (2000), 201–217.

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

[12] D. Frigioni, M. Ioffreda, U. Nanni, G. Pasqualone. Experimental analysis of dynamic algorithms for the single source shortest paths problem. *ACM Journal on Experimental Algorithmics*, vol. 3 (1998), Article 5.

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

[14] D. Frigioni, A. Marchetti-Spaccamela, U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, vol. 34, n. 2 (2000), 251–281.

[15] P. N. Klein, S. Rao, M. Rauch and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Science*, vol. 55, n. 1 (1997), 3–23.

[16] G. Ramalingam. Bounded incremental computation. *Lecture Notes in Computer Science*, vol. 1089, 1996.

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

[18] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest paths problem. *Journal of Algorithms*, vol. 21 (1996), 267–305.

[19] H. Rohnert. A dynamization of the all-pairs least cost paths problem. Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS85). *Lecture Notes in Computer Science*, vol. 182, 279–286, 1985.