Fully Dynamic Shortest Paths and Negative Cycle Detection on Digraphs with Arbitrary Arc Weights*

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

Abstract

We study the problem of maintaining the distances and the shortest paths from a source node in a directed graph with arbitrary arc weights, when weight updates of arcs are performed. We propose algorithms that work for any digraph and have optimal space requirements and query time. If a negative–length cycle is introduced during *weight-decrease* operations it is detected by the algorithms. The proposed algorithms explicitly deal with zero–length cycles. The cost of update operations depends on the class of the considered digraph and on the number of the output updates. We show that, if the digraph has a k-bounded accounting function (as in the case of digraphs with genus, arboricity, degree, treewidth or pagenumber bounded by k) the update procedures require $O(k \cdot n \cdot \log n)$ worst case time. In the case of digraphs with n nodes and m arcs $k = O(\sqrt{m})$, and hence we obtain $O(\sqrt{m} \cdot n \cdot \log n)$ worst case time per operation, which is better for a factor of $O(\sqrt{m}/\log n)$ than recomputing everything from scratch after each input update.

If we perform also insertions and deletions of arcs all the above bounds become amortized.

1 Introduction

We study the dynamic single source shortest path problem in a directed or undirected graph with real arc weights. The best known static algorithm for this problem takes $O(mn)$ time if the graph has n nodes and m arcs; it either detects a negative-length cycle, if one exists, or solves the shortest path problem (see e.g. [1]). If the arc weights are integers in $[-C..C]$ the best static algorithm takes $O(\sqrt{n}m \log C)$ time [12].

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 weight 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, 7, 8, 10, 11, 13, 17, 18]. However, in the general case, neither a fully dynamic solution nor a decremental solution for the single source shortest path problem is known in the literature that is asymptotically better than recomputing the new solution from scratch.

---

[*] Work partially supported by the ESPRIT Long Term Research Project ALCOM-IT under contract no. 20244, and by *Progetto Finalizzato Trasporti 2* of the Italian National Research Council (CNR). The work of the first author was done while he was visiting the Max Planck Institut für Informatik, IM Stadtwald, 66123, Saarbrücken (Germany), supported by the NATO – CNR Advanced Fellowships Program n. 215.29 of the Italian National Research Council.

[†] Max Planck Institut für Informatik, IM Stadtwald, 66123, Saarbrücken (Germany).

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

In the case of graphs with arbitrary arc weights we are not aware of any dynamic solution that is provably better than recomputing everything from scratch. In this paper we propose a solution with a worst case bound of $O(\sqrt{m} \cdot n \cdot \log n)$ when the weight of an arc is changed. This is better for a factor of $O(\sqrt{m}/\log n)$ than recomputing everything from scratch after each input update using the best known static algorithm.

The above bounds hold in the general case; they are even 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 path tree. We now consider the above conditions in more detail.

A) The structural constraints that we consider have been introduced in the framework of dynamic algorithms in [10]. 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. An analogous notion, the *orientation* of arcs in an undirected graph, has been introduced in [5]. An orientation of an undirected graph $G = (V, E)$ is a function ω which replaces each edge $(x, y) \in E$ by a directed arc $x \to y$ or $y \to x$. If $deg^+_\omega(x)$ is the out-degree of node x under the orientation ω then ω is k-bounded if for each node $x \in V$ $deg^+_\omega(x) \leq k$. The notions of k-bounded orientation and k-bounded accounting function in an undirected graph coincide. Since we deal with directed and undirected graphs, for the sake of clarity in the sequel we use the terminology of accounting function.

The value of parameter k defined above for any graph G can be bounded in different ways (see e.g. [10, 11]). For example, k is immediately bounded by the maximum degree of G. Furthermore, the value of parameter k for any graph G is bounded by the arboricity, the pagenumber and the treewidth of G. Moreover, it is possible to show that $k = O(1 + \sqrt{\gamma})$, where γ is the *genus* of G, by observing that the *pagenumber* of a genus γ graph is $O(\sqrt{\gamma})$ [14]; since the genus of a graph is always less than its number of arcs, it follows that a graph with m arcs has a $O(\sqrt{m})$-bounded accounting function.

If we assume that the graph has a k-bounded accounting function then the running times of the algorihms proposed in this paper become $O(n \cdot k \cdot \log n)$. We remark that the notion of k-bounded accounting function is useful only to bound the running times, but does not affect the behavior of our algorithms.

B) The analysis of dynamic algorithms using the *output complexity* model has been introduced by Ramalingam and Reps in [15, 17] and it has been subsequently modified by the authors of this paper in [10]. Other authors used similar concepts in [2]. In [15, 17] Ramalingam and Reps propose also the only dynamic solution for shortest paths on digraphs with arbitrary arc weights known in the literature. In this solution they assume the digraph has no negative-length cycles before and after any input change. In addition, they do not deal with zero-length cycles. In fact, they show that in their model there exist no dynamic algorithm for shortest paths whose performances can be bounded in terms of the number of output updates, if zero-length cycles are allowed. In [16] they propose a dynamic output bounded solution for a generalization of the shortest path problem.

Following the model of [10], in the case of the single source shortest path problem for a digraph G with source node s, the *output information* consists of *(i)* for any node $x \in V$ the value of the distance of x from s, and *(ii)* a shortest path tree rooted in s. Let μ be an arc operation to be performed on G (insertion, deletion, or weight update), and G' be the new graph after that μ has been performed on G. The *set of output updates* $\mathcal{U}(G, \mu)$ to be performed on the solution of the problem is given by the set of nodes in N that either change their distance from the source in G', or change their parent in the shortest path tree, due to μ. The *number of output updates* caused by μ is the cardinality of set $\mathcal{U}(G, \mu)$. This notion of output complexity can be extended to compute *amortized* costs [10]. We also remark that experiments show that the output complexity is a useful parameter to evaluate the practical efficiency of dynamic algorithms for the single source shortest path problem with positive weights [9].

If the digraph has a k-bounded accounting function, then our algorithms require $O(k \log n)$ time per output update in the case of *weight-decrease* operations. When the weight of an arc (x, y) is increased we observe that if (x, y) is an arc belonging to the shortest path 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 path tree rooted at y. In this case the running time of the update procedure is $O(k \log n)$ per output update plus time linear in the size of the subtree of the shortest path tree rooted at y.

Some of the results of the paper exploit techniques developed by the authors in previous papers. Namely, the idea of k-bounded accounting function and the one of associating a potential function to arcs [10, 11]. Roughly speaking the potential function that is associated to each arc allows to run a procedure similar to Dijkstra's algorithm [6] only on the subgraph that is affected by an input change. However the potential function used in previous papers does not allow to deal with negative weights and negative-length cycles. The original contribution of this paper can be summarized as follows.

1. We explicitly deal with negative-length cycles that might be introduced by an operation that decreases the weight of an arc or that adds a new arc to the digraph. Our algorithm first detects if a negative-length cycle has been introduced by the update operation and computes the set of nodes affected by the input update. Then the new shortest path tree is recomputed only on the subgraph induced by the affected nodes.

2. In the case of deletions or of weight increase operations the main difficulty is given by the presence of zero-length cycles. Our algorithm computes the set of nodes affected by the input update and detects the zero-length cycles that include affected nodes. Then the new shortest path tree is recomputed only on the subgraph induced by the affected nodes. Since zero-length cycles have never been handled before, we believe that the correctness proof of this part is interesting on its own, regardless from the other results of the paper.

3. We introduce a new potential function that allows us to compute the new distances after an input change to the graph without using a label correcting algorithm, but running a Dijkstra-like algorithm after that the set of nodes affected by the input change has been determined.

In the sequel of the paper we present only the operations that decrease or increase the weight of an arc, and we assume that the graph has been preprocessed in order to found a k-bounded accounting function.

The extension to the general case in which also insertions and deletions of arcs are allowed is based on techniques developed in [11], and will be presented in the full version of the paper. Let k_i be the minimum k s.t. the graph has a k-bounded accounting function after the i-th update in a sequence, and let $k^* = \max\{k_i\}$. The techniques of [11] do not require to compute k_i after each update operation. It is shown how to compute an upper bound on k_i that, in the worst case, can be larger than k_i, but that is dynamically modified after each update operation in such a way that, on the average, it is at most $2k^*$ on any sequence of update and query operations. In this way the worst case bounds that will be presented in the paper will become amortized if insertions and deletions of arcs are allowed.

The paper is organized as follows. After some preliminaries, given in the next section, our algorithms for the fully-dynamic single source shortest path problem for arc weight updates are described in Sections 3 and 4, and the worst case bounds for this case are proved. In Section 5 we provide conclusions and open problems.

2 Preliminaries

Let $G = (N, A)$ be a weighted directed graph (digraph) with n nodes and m arcs, and let $s \in N$ be a fixed *source* node. For each node $z \in N$, 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 $(x_i, x_{i+1}) \in A$, $\forall i = 1, 2, \ldots, r-1$. 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 $(x_i, x_{i+1}) \in A$, $\forall i = 1, 2, \ldots, r-1$ 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 cycles then we denote as $d : N \to \Re$ the distance function that gives, for each node $x \in N$, the length of the shortest path from s to x in G. Given two nodes x and y we denote as $d(x, y)$ the length of the shortest path between x and y. $T(s) = (N_T, A_T)$ denotes a shortest path tree rooted at s; for any $x \in N$, $x \neq s$, $T(x)$ is the subtree of $T(s)$ rooted in x. We now recall the well known condition that states the optimality of a distance function d on $G = (N, A)$. For each arc $(z, q) \in A$, the following *optimality condition* holds: $d(q) \leq d(z) + w_{z,q}$.

For each $z \in N$, $d(z)$ and $d'(z)$ denote the distance of z before and after an arc modification, respectively. The new shortest path tree in the graph G' obtained from G after an arc operation, is denoted as $T'(s)$.

Definition 2.1 *After an arc update (weight-increase or weight-decrease) in G, we define the quantity $\delta(z) = d'(z) - d(z)$ as the* variation *of distance of node z from the source.*

Definition 2.2 *Given $G = (N, A)$ and $z \in N$, the* backward_level *$b\_level_z(q)$ of arc $(z, q) \in \textsc{out}(z)$ is given by $d(q) - w_{z,q}$; the* forward_level *$f\_level_z(v)$ of arc $(v, z) \in \textsc{in}(z)$ is given by $d(v) + w_{v,z}$. After an arc update (weight-increase or weight-decrease) in G, the* variation *of distance of node z from the source is $\delta(z) = d'(z) - d(z)$.*

In order to bound the number of arcs scanned by our algorithms we assume that the sets $\textsc{in}(z)$ and $\textsc{out}(z)$ are partioned into two subsets as follows. Any arc $(x, y) \in A$ has an *owner* that must be either x or y. For each $x \in N$, $\textsc{in-own}(x)$ denotes the subset of $\textsc{in}(x)$ containing the arcs owned by x, and $\overline{\textsc{in-own}}(x)$ denotes the set of arcs in $\textsc{in}(x)$ not owned by x. Analougously, $\textsc{out-own}(x)$ and $\overline{\textsc{out-own}}(x)$ represent the arcs in $\textsc{out}(x)$ owned and not owned by x, respectively. We say that G has a k-bounded accounting function if both $\textsc{in-own}(x)$ and $\textsc{out-own}(x)$ contain at most k arcs.

In addition to the standard representation of digraph G we use the following data structures. For each node x, $D(x)$ and $P(x)$ store the distance and the parent of x in the shortest path tree, respectively. $D(x)$ satisfies the following properties: *i)* $D(x) = d(x)$ before the execution of any of the algorithms proposed in the paper; *ii)* $D(x) = d'(x)$ after the execution of the algorithms. Furthermore, $\Delta(z)$ stores the computed value of $\delta(z)$; before and after the execution of any update procedure its value is 0 and $\Delta(z) = \delta(z)$. Finally, we use two additional variables $D'(z)$ and $P'(z)$ that stores the temporary value of $D(z)$ and $P(z)$ during the execution of the algorithms.

The arcs in $\textsc{in-own}(x)$ and in $\textsc{out-own}(x)$ are stored in two linked lists, each containing at most k arcs. The arcs in $\overline{\textsc{in-own}}(x)$ and in $\overline{\textsc{out-own}}(x)$ are stored in two priority queues as follows:

1. $\overline{\textsc{in-own}}(x)$ 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 $f\_level_x(y)$.

2. $\overline{\textsc{out-own}}(x)$ is a max-based priority queue where the priority of arc (x, y) (of node y), denoted as $b_x(y)$, is the computed value of $b\_level_x(y)$;

Before processing a sequence of arc modifications on G, we have to compute the shortest distance $d(x)$, for each node x in G, an initial shortest path tree and an initial k-bounded ownership for G. Then, 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 (we will see that both these conditions are restored after the execution of any procedure proposed in the following).

3 Decreasing the weight of an arc

In this section we show how to maintain a shortest path tree of a digraph $G = (N, A)$ with arbitrary arc weights, after decreasing the weight of an arc. We assume that the graph before the execution of the *weight-decrease* operation does not contain negative–length cycles; if the *weight-decrease* operation on arc (x, y) does not introduce a negative–length cycle, Procedure `Decrease`, shown in Figure 1, properly updates the current shortest path 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. On the other hand, if $d(x) + w_{x,y} - \epsilon < d(y)$ then 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 define the subgraph $G_R = (N_R, A_R)$ of G as follows: $N_R \subseteq N$ is the set of *red* nodes; $A_R \subseteq A$ is the set of arcs of G induced by the nodes in N_R plus, for each $z \in N_R$, all arcs in OUT-OWN(z). Let $n_R = |N_R|$. The following facts can be easily proved:

F1) If node y decreases its distance from s after decreasing the weight of (x, y), then the new shortest paths from s to the *red* nodes will contain arc (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 shortest distance from s.

F2) Node x reduces its shortest distance from s after decreasing the weight of arc (x, y) if and only if the *weight-decrease* operation introduces a negative–length cycle; in this case (x, y) belongs to C.

F3) If decreasing the weight of (x, y) introduces a negative–length cycle C in G, then, for each node $z \in C$, the (acyclic) shortest 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 before the *weight-decrease* operation.

Let us consider the nontrivial case where $d'(y) < d(y)$. In order to update the distances of *red* nodes we adopt a strategy similar to that of Dijkstra's algorithm on G_R. In particular, the *red* nodes are inserted in a heap Q. The presence of arcs with negative weights implies that, if we want to use a Dikstra-like algorithm, the priority of a node in Q cannot be the lenght of the path found by the procedure. However we will see that if the priority of z in Q is $\Delta(z)$, that is an estimate of the (negative) variation $\delta(z) = d'(z) - d(z)$, then a Dijkstra's like procedure is sufficient to update the shortest path tree.

Namely, at the beginning of the procedure, all nodes z in $T(y)$, are enqueued with variation $\Delta(z) = \Delta(y)$. Then, in the main **while** loop, the nodes are dequeued from Q and, for each z dequeued with priority $\Delta(z)$, the new distance from s is computed as $D'(z) = D(z) + \Delta(z)$. At this point, both for the arcs (z, h) in OUT-OWN(z), and for the arcs (z, h) in $\overline{\text{OUT-OWN}}(z)$ such that $b_z(h) > D'(z)$, the priority of h and v in Q is possibly updated (i.e., if $D'(z) + w_{z,h} - D(h) < \Delta(h)$), as well as the current parent. If any improvement is determined for node x, then by Fact *F2*, a negative-length cycle is detected.

In the main **while** loop, for each *red* node z, the new distance from the source and the (possibly new) parent in a shortest path tree is computed in the auxiliary variable $D'(z)$. Only after that this computation is carried out successfully (i.e., without trying to update node x), the data structures are actually updated.

```
procedure Decrease($x, y$ : node; $\epsilon$ : positive_real)
begin
$w_{x,y} \longleftarrow w_{x,y} - \epsilon$
$\Delta(y) \longleftarrow D(x) + w_{x,y} - D(y)$
if $\Delta(y) < 0$ then
   begin
      $P'(y) \longleftarrow x$
      $Q \longleftarrow \emptyset$ {initialize an empty heap $Q$}
      for each node $z \in T(y)$ do
         begin
            if $z = x$ then a negative cycle has been detected
            color $z$ red
            $\Delta(z) \longleftarrow \Delta(y)$ {uniform variation within $T(y)$}
            Enqueue($Q, \langle z, \Delta(z) \rangle$)
         end
      while Non_Empty($Q$) do
         begin
            $\langle z, \Delta(z) \rangle \longleftarrow$ Extract_Min($Q$)
            $D'(z) \longleftarrow D(z) + \Delta(z)$
            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
              if $D'(z) + w_{z,h} - D(h) < \Delta(h)$ then
                    begin
                       if $h = x$ then a negative cycle has been detected
                       color $h$ red
                       $P'(h) \longleftarrow z$
                       $\Delta(h) \longleftarrow D'(z) + w_{z,h} - D(h)$
                       Heap_Insert_or_Improve($Q, \langle h, \Delta(h) \rangle$)
                    end
         end
      for each red node $z$ do
         begin
            uncolor $z$
            $D(z) \longleftarrow D'(z)$
            $P(z) \longleftarrow P'(z)$
            $\Delta(z) \longleftarrow 0$
            for each arc $(v,z) \in$ IN-OWN($z$) do $b_v(z) \longleftarrow D(z) - w_{v,z}$
            for each arc $(z,v) \in$ OUT-OWN($z$) do $f_v(z) \longleftarrow D(z) + w_{z,v}$
         end
   end
end
```

Figure 1: Decrease by quantity ϵ the weight of arc (x, y)

In the following we prove the correctness of Procedure `Decrease`. It is based on the following lemma. We assume that before the *weight-decrease* operation the data structures store the correct values, i.e., the array of parents induces a shortest path tree rooted in s and, for each $z \in N$, $D(z) = d(z)$.

Lemma 3.1 *Let z be any node of G with variation $\delta(z) = d'(z) - d(z) < 0$ after a weight-decrease operation on arc (x, y). If $\langle y = z_0, z_1, \ldots, z_p = z \rangle$ is the fragment of any shortest path from s to z in G' starting from y, then for $i = 1, 2, \ldots, p$, we have: $\delta(z_{i-1}) \leq \delta(z_i)$.*

Proof. By contradiction, let us suppose that, in the hypotheses of the lemma, there exist a node $z \in N$ and an index i such that $\delta(z_{i-1}) > \delta(z_i)$. Since arc (z_{i-1}, z_i) belongs to a shortest path in G' from s to z_i, then $d'(z_{i-1}) + w_{z_{i-1},z_i} = d'(z_i)$. By combining the two relationships above, we obtain:

$$d'(z_{i-1}) - d(z_{i-1}) = \delta(z_{i-1}) > \delta(z_i) = d'(z_{i-1}) + w_{z_{i-1},z_i} - d(z_i)$$

and hence $d(z_i) > d(z_{i-1}) + w_{z_{i-1},z_i}$, which contradicts the optimality conditions on arc (z_{i-1}, z_i) before the update. □

Theorem 3.2 *Let $G = (N, A)$ be a digraph with arbitrary arc weights, if the weight of arc (x, y) is decreased by quantity ϵ, then Procedure* `Decrease`$(x, y; \epsilon)$ *is correct, i.e., either it detects the introduction of a negative–length cycle, or after its execution*
a) for each node $z \in N$, $D(z) = d'(z)$, and
b) the parent array induces shortest path tree rooted in s.

Proof. A crucial point of the algorithm is the order in which nodes are dequeued from Q in line 17, and the updates that are performed on the priorities of nodes in Q. To prove the theorem we need to state some preliminary properties.

P1) *All the red nodes are extracted from Q in nondecreasing order of priority.*
When a node z is extracted from Q with priority $\Delta(z) < 0$, any node h such that $(z, h) \in$ OUT(z) may decrease its priority in Q to a value $\Delta(h)$ not smaller than $\Delta(z)$. In fact, from lines 25–26, we have:

$$\Delta(h) = D'(z) + w_{z,h} - D(h) = \Delta(z) + D(z) + w_{z,h} - D(h) = \Delta(z) + d(z) + w_{z,h} - d(h) \geq \Delta(z)$$

where the last inequality is due to the optimality condition on arc (z, h) before the arc update.

P2) *If $z \in Q$, then $P'(z)$ locates a possible path (non necessarily optimal) to the source.*
In fact, as can be easily proved by induction, one of two possibilities arises: *i)* $z \equiv y$ and $P'(z) \equiv x$; *ii)* $P'(z)$ has been already dequeued from Q and, in turn, it has found a path to the source through $P'(P'(z))$. Note that this property, actually weaker than statement (b) of the theorem, shows that all the *red* nodes will be possibly appended to some tree rooted in s.

P3) *If a node z has a priority $\Delta(z)$ in the queue, then this is an upper bound to the actual variation of its distance from the source, i.e., $\Delta(z) \geq \delta(z)$.*
This can be easily proved by induction on the number of changes made on the priorities in Q and on the basis of property *P2*.

P4) *If z is dequeued with priority $\Delta(z)$, for each neighbor h of z still in the queue, its priority is subject to the constraint: $\Delta(h) \leq D'(z) + w_{z,h} - d(h)$.*
It is sufficient to check that, as soon as a node z is dequeued, all the arcs (z, h) leaving z that produce possible improvements are scanned (see lines 19–27).

Now we prove the theorem by contradiction. Two possible mistakes may arise:

a) a node z is enqueued, but its distance from the source is not correctly computed;

b) a node z changes its distance from the source but it is not enqueued in Q.

Case a) Let z be the first dequeued node whose computed distance from s is wrong, i.e.,

$$d'(z) < D'(z) = d(z) + \Delta(z) \tag{1}$$

in fact, as a consequence of property *P3*, $D'(z)$ must be an upper bound of the actual distance of z from s.

Let us consider any shortest path from z to the source, and let p be the parent of z in this optimal path. Two possibilities arise, according on whether p is still in the queue or not when z is dequeued.

1. p is still in the queue when z is dequeued. Let us consider the optimal path from s to z. In particular, since (x,y) belongs to any shortest path from the source to a *red* node, we consider only the fragment of that path between x and z: $\langle x = z_0, z_1, \ldots, z_{h-1}, z_h, \ldots, z\rangle$ where z_h, possibly coincident with p, is the first node in such a path which is still in the queue. In turn, its parent z_{h-1} is either a nonred node, or has already been dequeued: in both cases its distance from the source has been correctly computed, i.e., $D'(z_{h-1}) = d'(z_{h-1})$. Therefore, by property *P4*, when z is dequeued, the priority of z_h in Q is subject to the constraint

   $$\Delta(z_h) \leq d'(z_{h-1}) + w_{z_{h-1},z_h} - d(z_h) = \delta(z_h).$$

   Since $\Delta(z_h) \geq \delta(z_h)$ by property *P3*, we have that $\Delta(z_h) = \delta(z_h)$. By inequality (1), we know that $\delta(z) = d'(z) - d(z) < \Delta(z)$. On the other side z is dequeued from Q before z_h, and then, by Property *P1*: $\delta(z) < \Delta(z) \leq \Delta(z_h) = \delta(z_h)$. This contradicts Lemma 3.1, that states that the values of δ must be monotone nondecreasing along any optimal path from the source to node z.

2. p has already been dequeued or, as a special case, it is coincident with node x. In this case the distance of p from s has been correctly computed, i.e., $D'(p) = d'(p)$. Hence, by property *P4*, $\Delta(z) \leq d'(p) + w_{p,z} - d(z)$. By combining this inequality with inequality (1) above, we obtain: $d'(z) < d(z) + \Delta(z) \leq d'(p) + w_{p,z}$, which contradicts the fact that p is an optimal parent of z.

Case b) Let z be the closest node to the source (in terms of number of arcs) that should be enqueued, but it is not. Hence the parent p of z in such a path is either x, or it was enqueued and, as proved above, its distance from the source has been correctly computed.

On the other side, when node p was dequeued, all the arcs (z,h) leaving z that may produce possible improvements to the priority of h have been scanned (see lines 19–27), in the inductive hypothesis that the data structures store correct information before the updates take place. □

The following theorem gives the output complexity bounds of Procedure `Decrease`.

Theorem 3.3 *Let $G = (N,A)$ be a digraph with arbitrary arc weights. If G has a k-bounded accounting function, then it is possible to update $T(s)$ and the distances of nodes from s, or to detect the introduction of a negative–length cycle in G after the execution of a weight-decrease operation, in $O(n_R \cdot k \cdot \log n)$ time.*

Proof. Each red node z is enqueued in Q exactly once, as it can be easily shown using the monotonicity of the priorities in Q. When z is dequeued, an arc $(z,h) \in \textsc{out}(z)$ is scanned (see line 19) only if: *i)* $(z,h) \in \textsc{out-own}(z)$: in this case the arc is scanned *by ownership*; *ii)* $(z,h) \in \overline{\textsc{out-own}}(z)$ and $b_z(h) > D'(z)$: in this case the arc is scanned *by priority.*

At most $k \cdot n_R$ arcs are scanned by ownership. In order to find out the arcs to be scanned by priority, for each *red* node z Procedure `Decrease` traverses only the arcs (z,v) in $\overline{\textsc{out-own}}(z)$ such that $b_z(v) > D'(z)$, i.e., such that also v is *red*, plus the first arc in $\overline{\textsc{out-own}}(z)$ not satisfying this property. Therefore, the algorithm will scan by priority only arcs between *red* nodes, plus n_R arcs that do not satisfy this property. Since the subgraph induced by the *red* nodes has a k-bounded accounting function, then at most $(k+1) \cdot n_R$ arcs are scanned by priority by the algorithm.

In the worst case, each arc scanned by priority requires $O(\log n)$ time to be selected in its owner's local priority list and it requires a possible insertion or decrease-priority operation in the global priority queue Q (line 26. In this queue there will be exactly n_R node insertions and at most $k \cdot n_R$ decrease-priority operations. These operations require at most $O(k \cdot n_R \cdot \log n)$ worst case time.

In the last phase (lines 29–37), the updates to the local priority queues $\overline{\text{IN-OWN}}$ and $\overline{\text{OUT-OWN}}$ of all the nodes (each requiring $O(\log n)$ time) are performed by scanning again the lists IN-OWN and OUT-OWN of the *red* nodes (lines 35–36), leading to 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 *F1* $(x, y) \in C$, and by Fact *F3* all nodes in C are *red.* Hence, the above bounds hold also when a negative-length cycle is detected by the algorithm. □

Corollary 3.4 *In a general digraph* $k = O(\sqrt{m})$, *and hence it is possible to update* $T(s)$ *and the distances of nodes from* s *after the execution of a weight-decrease operation, in* $O(n \cdot \sqrt{m} \cdot \log n)$ *worst case time.*

Note that, in a fully dynamic sequence of operations, it is not possible to amortize the decrease operations in the local priority queues $\overline{\text{OUT-OWN}}$, since these are interleaved with *weight-increase* operations. Though, to deal with a monotone sequence of *weight-decrease* operation, by using Fibonacci heaps both in the global queue Q and in the local priority queues, the bound can be improved to $O(n_R(k + \log n))$ worst case time.

4 Increasing the weight of an arc

In the following we assume that before the *weight-increase* operation the data structures store the correct values, i.e., the array P induces a shortest path tree rooted in s and, for each $z \in N$, $D(z) = d(z)$. If the weight of an arc $(x, y) \in A$ is increased, no negative–length cycle can be introduced in G as a consequence of that operation. On the other hand 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 path tree $T'(s)$ such that, for each $z \notin T(y)$ the old parent in $T(s)$ is preserved.

We define a coloring of the nodes of G, depending on the algorithm, in order to distinguish how nodes are affected by the execution of a *weight-increase* operation, as follows:

- $q \in N$ is *white* if and only if q changes neither the distance from s nor the parent in $T(s)$;
- $q \in N$ is *red* if and only if q increases the distance from the source, i.e., $d'(q) > d(q)$;
- $q \in N$ is *pink* if and only if q maintains its distance from s, but changes the parent in $T(s)$.
- $q \in N$ is *blue* if q belongs to a zero–length cycle that is detected by the algorithm.

It is easy to verify that, if q is *red* then all the children of q in $T(s)$ must be updated and will be either *pink* or *red*; furthermore, if q is *pink* or *white* then all nodes in $T(q)$ are *white.* Observe that a node is colored *pink* depending on the possibility of finding alternative paths with the same lenght 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 before the *weight-increase* operation is not a shortest path after the update operation; it is sufficient that there exists one shortest path from s to q of the same length of the shortest path from s to q before the *weight-increase* operation, where node q has the same parent as before. We also observe that a node colored *blue* will be later colored again with a different color; on the other side when a node is colored *white*, *pink* or *red* the color will not be changed anymore.

In this case the output complexity is given by nodes colored red and pink. We remark that in the case of a dynamic update μ that increases the weight of an arc the cardinality of $\mathcal{U}(G,\mu)$ 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 that is found by the algorithm. Hence it is possible to have different sizes for the set of output updates, due to the input modification μ.

Algorithm `Increase` (Figure 4), works in two phases. In the first phase it uses Procedure `Color`, shown in Figure 2, 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 path tree to each node which is colored *pink*. This is done by inserting the nodes in a heap M, extracting them in non-decreasing order of their distance from the source, and searching an alternative shortest path from the source. Since Procedure `Color` can also modify the shortest path tree, we denote as $T_c(s)$ the tree after the execution of `Color`. In the second phase Procedure `Increase` properly updates the information on the shortest paths from s for all the nodes that have been colored *red*, 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 arc (q,p) belongs to a shortest path from s to p in G (i.e., $d(p)=d(q)+w_{q,p}$). Node q is an* equivalent parent *for p if it is a candidate parent of p and $d'(q)=d(q)$.*

```
procedure Color($y$ : node)
begin
    $M \longleftarrow \emptyset$ {$M$ is an heap}
    Enqueue($M, \langle y, D(y)\rangle$)
    while Non_Empty($M$) do
        begin
            $\langle z, D(z)\rangle$ = Extract_Min($M$)
            if $z$ is uncolored then
                Search_Equivalent_Path($z, y$)
        end
    color red each arc with both endpoints red
end
```

Figure 2: Color the nodes in $T(y)$ after increasing the weight of arc (x,y)

Procedure `Color` uses Procedure `Search_Equivalent_Path`, shown in Figure 3, which searches for a path from s to a node z in G' whose length is equal to $D(z)$. In order to achieve this goal Procedure `Search_Equivalent_Path` uses a stack Q which is initialized with node 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 $i=2,3,\ldots,k$. The nodes in Q are either uncolored or *blue*, and hence they represent nodes that are waiting that their current candidate parent is correctly colored.

Let us define the *best nonred neighbor* of a node z as the node q such that $(q,z) \in \text{IN}(z)$, q is nonred, and the shortest path from s to z passing through q is the minimum among those passing through the nonred neighbors of z. After having initialized Q with z, `Search_Equivalent_Path` searches the best nonred neighbor of z in IN-OWN(z) and in $\overline{\text{IN-OWN}}(z)$, respectively; finally, it chooses node q as the best between the two neighbors found. If q belongs to $T(z)$ or to the subtree of $T(y)$ rooted at some node currently in the stack, then a zero–length cycle has been detected; then the procedure colors *blue* the nodes in that cycle (this is done in order to avoid multiple visits of the same node) and considers another candidate parent of z.

A number of cases may arise once we have found the best nonred neighbor q of z. If q is not a candidate parent of z, and z is not *blue*, then there is no alternative path from s to z in G' with length $D(z)$ and, therefore, z is colored *red*. Then all the children of z in $T(y)$ are inserted in the heap M and they will be colored later either *red* or *pink*. On the other hand, if q is not a candidate parent of z and z is *blue*, then z belongs to a zero–length cycle. In this case we cannot give a final color to z; hence z is deleted from Q, and it will be given a final color later.

If q is a candidate parent of z and q does not belong to $T(y)$ or it has been already colored either *pink* or *white* then it is an equivalent parent for z. In fact, in this case we have found a shortest path from s to z passing through q whose length is $D(z)$: z must be colored either *white* or *pink*, depending on whether or not node q was the parent of z in $T(s)$ before the *weight-increase* operation.

If q is *blue* or $q \in Q$ then a zero–length cycle has been detected, and therefore all nodes currently in Q are colored *blue*. Finally, if q is uncolored, then q represents a candidate parent of z, but the algorithm is not able to determine whether q is an equivalent parent of z, because it could change its distance later. In this case the search continues, node q is pushed in Q, and the algorithm looks for an equivalent parent of q, that now is the top node in the stack.

Observe that a *blue* node will be colored again; in fact, when the node on the top of Q is colored either *pink* or *white*, then all the nodes in the stack and all the *blue* nodes have found a path from s of the same length of the previous one, and hence they are colored either *pink* or *white*, depending on whether or not they change their parent in $T(s)$. Moreover the last step of the procedure colors *red* all remaining *blue* nodes.

In the sequel we show the correctness of Procedures `Color` and `Search_Equivalent_Path`. It is based on the following two lemmas.

Lemma 4.1 *Let $G = (N, A)$ be a digraph with arbitrary arc weights, and assume that a weight increase operation is performed on arc (x, y). If p and q are two nodes in $T(y)$ such that p and q belong to a zero–length cycle, then p and q are colored either both red or both nonred by Procedure* `Search_Equivalent_Path`.

Proof. First observe that if an arc (a, b) belongs to a zero–length cycle, then a is a candidate parent of b.

Let p and q be two nodes in a zero–cycle, and assume that the lemma is not true. Without loss of generality, we can assume that there exists arc (q, p), and that q is nonred and p is *red*. Node p is colored *red* either at line 12 or at line 37 of Procedure `Search_Equivalent_Path`. Note that if p is *red* then the algorithm has surely considered node q as a candidate parent for p before coloring p *red*. Since the *red* color given to p is not changed, then three cases may arise depending on the color of q at the time it was considered as a candidate parent for p.

1. q was either *pink* or *white*: in this case the algorithm colors p nonred (either *white* or *pink*) either in line 23 or in line 25. This contradicts the hypothesis that p is colored *red*.
2. q was *blue*: in this case p is colored *blue* at line 34. Since both q and p are now colored *blue*, and q is a candidate parent for p, then they will be colored later, either both nonred at line 31 or both *red* at line 37, contradicting the hypothesis that they have different colors.
3. q was uncolored: two subcases may arise. *i*) If $q \in Q$,then a zero–length cycle has been detected containing both q and p, both the nodes are colored *blue* by the algorithm at line 34, and the same reasoning of case 2 above can be applied. *ii*) If $q \notin Q$ then q is pushed on the stack Q after p (line 35). When q is colored nonred by the algorithm, the loop at lines 20–30 colors either *white* or *pink* all the nodes in the stack and all the *blue* nodes, and therefore also p; this contradicts the hypothesis that p is colored *red*.

Since in any case we have derived a contradiction, then the lemma follows. □

```
    procedure Search_Equivalent_Path($z, y$ : node)
1.  begin
2.    $Q \longleftarrow \emptyset$ {$Q$ is a stack}
3.    Push($Q, z$)
4.    repeat
5.      $p$ := Top($Q$)
6.      let $q$ be the best nonred neighbor of $p$
7.      if $q$ does not exist or $q$ is not a candidate parent for $p$
8.        then begin
9.              Pop($Q$)
10.             if $p$ is not blue then
11.               begin
12.                 $color(p) \longleftarrow red$
13.                 for each $v \in children(p)$ do Heap_Insert($M, \langle v, D(v)\rangle$)
14.               end
15.           end
16.       else if $q \notin T(y)$ or $q$ is pink or $q$ is white
17.             then
18.               begin
19.                 $w := q$
20.                 repeat
21.                   $v$ := Top($Q$)
22.                   Pop($Q$)
23.                   if $(w, v) \in T(s)$ then $color(v) \longleftarrow white$
24.                     else begin
25.                            $color(v) \longleftarrow pink$
26.                            $P(v) := w$
27.                          end
28.                   color white all nodes in $T(v)$
29.                   $w := v$
30.                 until $Q$ becomes empty
31.                 color white or pink all remaining blue nodes
32.               end
33.             else if $q \in Q$ or $q$ is blue {a zero–length cycle has been detected}
34.                   then color blue all nodes in $Q$
35.                   else Push($Q, q$) {$q \in T(y)$ and $q$ is uncolored}
36.   until $Q$ becomes empty
37.   color red all blue nodes
38. end
```

Figure 3: Search an alternative parent for node z in T(s)

Lemma 4.2 *Let $G = (N, A)$ be a digraph with arbitrary arc weights, if a weight-increase operation is performed on arc (x, y) and node z is colored red, then all the nodes $q \in T(y)$ such that $D(q) + d(q, z) = D(z)$ are colored red as well.*

Proof. Let $q \in T(y)$ such that $D(q) + d(q, z) = D(z)$, and $q = q_0, q_1, q_2, \ldots, q_{k-1}, q_k = z$ be the path between q and z whose length is $d(q, z)$. The proof is by induction on k.

Base step. $k = 1$: in this case there exists arc (q, z), $d(q, z) = w_{q,z}$, and $D(q) + w_{q,z} = D(z)$. If q and z belong to a zero–length cycle then by Lemma 4.1 they receive the same color, and hence, since z is *red*, then they are both colored *red*. Otherwise, assume that the thesis is not true, i.e., that q is not *red*. Three cases may arise. Either q is *white* or q is *pink* or it is uncolored. If q is uncolored then Procedure `Search_Equivalent_Path` finds node q as a candidate parent of z in line6 and pushes it on the stack Q in line 35. Note that when q is colored either *pink* or *white* all nodes in Q are colored either *pink* or *white*; therefore, z cannot be *red*. On the other hand, if q is either *pink* or *white* then Procedure `Search_Equivalent_Path` finds node q as a candidate parent of z in line 6 and colors z either *pink* or *white* in lines 20–30. In both cases this contradicts the

hypothesis that z is colored *red*.
Inductive step. $k > 1$: by inductive hypothesis assume that all the nodes $q_1, q_2, \ldots, q_{k-1}$ are colored *red*. Furthermore, since node q_1 is on a shortest path from s to z passing through node q, then $D(q_1) = D(q) + w_{q,q_1}$. Now, if we suppose that the color of q is not *red*, then we can use the same reasoning used in the base step to derive a contradiction. □

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 if and only if $d'(z) > d(z)$.*

Proof. We prove the lemma by contradiction. Let z be a node that receives a wrong color during the execution of Procedure `Color` such that: *(i)* all nodes that belong to the path in $T_c(s)$ between s and z are correctly colored; *(ii)* z is the first one to receive a wrong color among the nodes that satisfy *(i)* above. If the lemma is not true then such a node exists. We distinguish three cases depending on the color given by the algorithm to z.

1. z is colored *white*. If the color of z is *white* this implies that it has the same parent w in $T(s)$ and in $T_c(s)$ and that w is *white* or *pink*; therefore, if the color of z is wrong, the color of w is wrong as well, and should be *red*, contradicting condition *(i)* above.
2. z is colored *pink*. In this case it is sufficient to show that z cannot be colored *red* by the algorithm. Let w be the new parent of z in $T_c(s)$. If both w and z belong to a zero–length cycle then, by Lemma 4.1, z and w are colored either both *red* or both non red. If the color of z is wrong, i.e., z should be *red*, then also the color of w is wrong, contradicting (i) above.
   If both w and z do not belong to a zero–length cycle, let $v_1, v_2, \ldots, v_k = z$, $k \geq 1$, be the nodes extracted from Q in the loop at lines 20–30 of `Search_Equivalent_Path`, before coloring z *pink*. Let v_0 be the neighbor of v_1 selected by the procedure as the parent of v_1 in $T_c(s)$, then either v_0 does not belong to $T(y)$ or it has been previously colored *pink* or *white*. By *(i)* above v_0 is correctly colored and hence $d'(v_1) = d(v_1)$. Now observe that the arcs of the path from s to v_0 stored in $T_c(s)$, together with arcs $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, z)$ determine a path P from s to z.
   We distinguish two cases. **a)** If $k = 1$ and z is colored *pink* then $d(v_0) + w_{v_0,z} = d(z)$; since $d(v_0) = d'(v_0)$ then P is a shortest path from s to z of length $d(z)$; hence z cannot be *red*. **b)** If $k > 1$ then observe that all the nodes $v_1, v_2, \ldots, v_{k-1}$ are colored either *pink* or *white*, and that the subpath of P from s to v_{k-1} is a shortest path in G' with length $d'(v_{k-1}) = d(v_{k-1})$. Furthermore, since z is colored *pink*, we have $d(v_{k-1}) + w_{v_{k-1},z} = d(z)$. If z was *red* then by Lemma 4.2 v_{k-1} has to be *red*. Therefore the color of v_{k-1} is wrong, and this contradicts conditions *(i)* and *(ii)* above.
3. z is colored *red*. Let q and w be the parent of z in $T(s)$ and $T_c(s)$, respectively. First observe that, before coloring z *red*, q has been surely colored *red* (in fact z is pushed in the stack after q has been colored *red*); by condition *(ii)* above the color of q is correct and, therefore, z cannot be *white*. It remains to show that z cannot be *pink*. We distinguish three cases.
   a) If $w \in T(y)$ and both z and w belong to a zero–length cycle then, since the algorithm has colored z *red*, by Lemma 4.1 also w is colored *red*. If the color of z is wrong, i.e., z is *pink*, then the color of w is wrong contradicting condition (i) above.
   b) If $w \in T(y)$ and z does not belong to a zero–length cycle then, since the algorithm has colored z *red*, by Lemma 4.2 all nodes $v \in T(y)$ such that $d(v) + d(v, z) = d(z)$ have already been colored *red*. By condition *(ii)* above the color of these nodes is correct. This implies that w is correctly colored *red*, and hence that z cannot be *pink*.
   c) If $w \notin T(y)$ then note that z is deleted from stack Q after that all its possible candidate parent have been considered. It follows that the algorithm colors z *red* after that all the possible

candidate parents of z in $T(y)$ have been correctly colored *red* (by condition (*ii*) above), and there exists no candidate parent for z outside $T(y)$. Therefore, w is not a candidate parent for z, z cannot be colored *pink*, and it is correctly colored *red*. □

We now present 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 respect to the standard Dijkstra's algorithm is the priority given to each *red* node z in H, which is $\Delta(z)$ instead of $D(z)$. Then the procedure repeatedly extracts node z with minimum priority from H and updates its distance label. In this case, for each red arc $(z, h) \in \text{OUT}(z)$, Procedure `Heap_Improve` updates the priority associated to h in H (if required), in order to restore the optimality condition.

```
procedure Increase($x, y$ : node; $\epsilon$ : positive_real)
begin
   $w_{x,y} \longleftarrow w_{x,y} + \epsilon$
   if $(x, y)$ is not a tree arc
      then update either $b_x(y)$ or $f_y(x)$ (depending on the owner of $(x, y)$) and EXIT
   Color($y$)
   $H \longleftarrow \emptyset$ {initialize an empty heap $H$}
   for each red node $z$ do
      begin
         let $p$ be the best nonred neighbor of $z$
         if $p \neq$ Null
            then begin
                     $P'(z) \longleftarrow p$
                     $D'(p) \longleftarrow D(p)$
                     $\Delta(z) \longleftarrow D'(p) + w_{p,z} - D(z)$
                 end;
            else $\Delta(z) \longleftarrow +\infty$
         Enqueue($H, \langle z, \Delta(z) \rangle$)
      end
   while Non_Empty($H$) do
      begin
         $\langle z, \Delta(z) \rangle \longleftarrow$ Extract_Min($H$)
         $D'(z) \longleftarrow D(z) + \Delta(z)$
         for each red arc $(z, h)$ leaving $z$ do
            if $D'(z) + w_{z,h} - D(h) < \Delta(h)$
                 then begin
                         $P'(h) \longleftarrow z$
                         $\Delta(h) \longleftarrow D'(z) + w_{z,h} - D(h)$
                         Heap_Improve($H, \langle h, \Delta(h) \rangle$)
                      end
         uncolor all the red arcs $(q, z)$ entering $z$
      end
   for each red node $z$ do
      begin
         uncolor $z$
         $D(z) \longleftarrow D'(z)$
         $P(z) \longleftarrow P'(z)$
         $\Delta(z) \longleftarrow 0$
         for each arc $(v, z) \in$ IN-OWN($z$) do $b_v(z) := D(z) - w_{v,z}$
         for each arc $(z, v) \in$ OUT-OWN($z$) do $f_v(z) := D(z) + w_{z,v}$
      end
end
```

Figure 4: Increase by quantity ϵ the weight of arc (x, y)

In order to prove the correctness of procedure `Increase` we need the following lemma, analogous to the one we have shown for a *weight-decrease* operation.

Lemma 4.4 *Let z be any node of G with variation $\delta(z) = d'(z) - d(z) > 0$ after that a weight increase operation is performed on G. If $\langle s = z_0, z_1, \ldots, z_p = z\rangle$ is a shortest path from s to z in G', then for $i = 1, 2, \ldots, p$, we have: $\delta(z_{i-1}) \leq \delta(z_i)$.*

Proof. By contradiction, let us suppose that, there exists a node $z \in N$ and an index i such that $\delta(z_{i-1}) > \delta(z_i)$. Note that arc (z_{i-1}, z_i) belongs to a shortest path from s to z_i in G', and therefore $d'(z_{i-1}) + w_{z_{i-1},z_i} = d'(z_i)$. By combining the two relationships above, we obtain:

$$d'(z_{i-1}) - d(z_{i-1}) = \delta(z_{i-1}) > \delta(z_i) = d'(z_{i-1}) + w_{z_{i-1},z_i} - d(z_i)$$

and hence $d(z_i) > d(z_{i-1}) + w_{z_{i-1},z_i}$, which contradicts the optimality condition on arc (z_{i-1}, z_i) before the arc update. □

Theorem 4.5 *Procedure* `Increase`$(x, y; \epsilon)$ *is correct, i.e., after its execution*
a) for each node $z \in N$, $D'(z) = d'(z)$, and
b) the parent array induces a shortest path tree rooted in s.

Proof. After coloring all the nodes, the *red* arcs are colored, too. Then each *red* node z is enqueued in H (lines 7–18) with priority given by the difference between the length of the shortest path passing through a non-red neighbor of z and $d(z)$. The nodes that will get an initial priority in H smaller than $+\infty$ have at least a nonred neighbor.

We first prove that $D(z) \geq d'(z)$ and that the parent array induces a single source path tree rooted in s. In fact, if during the execution of the procedure a node z has a non-null pointer $P'(z) \equiv p$ and priority $\Delta(z)$ then there is a path form s to z passing through p of length $d(z) + \Delta(z)$. In fact, as can be easily proved by induction, either p is nonred, or it has been already dequeued from H and, therefore, there is a path to the source by using its pointer $P'(p)$. In both cases it follows that $\Delta(z) \geq \delta(z)$ and, therefore, $D'(z) \geq d'(z)$. This also implies that $\Delta(z) \geq \delta(z)$.

In order to prove the optimality of the distances computed by the algorithm we need the following property.
Claim. *If z is either a nonred node (with $\Delta(z) = \delta(z) = 0$) or a node that has been already dequeued with priority $\Delta(z)$, for each neighbor h of z still in the queue, its priority satisfies: $\Delta(h) \leq D'(z) + w_{z,h} - d(h)$.*
To prove the claim it is sufficient to check that, as soon as a node z is dequeued, all the remaining red arcs (z, h) leaving z are scanned for possible improvements of the red neighbors still in the queue (see lines 23–29). After that, the priority of nodes in the heap can only decrease.

We now show that the computed distances are optimal by contradiction; namely, let z be the first dequeued node whose computed distance from the source is wrong, i.e.,

$$d'(z) < D'(z) = d(z) + \Delta(z). \tag{2}$$

Let us consider a shortest path from s to z, and let p be the parent of z in this optimal path. Two possibilities arise.

1. p is still in the queue when z is dequeued. Let us consider the optimal path from s to z: $\langle s = z_0, z_1, \ldots, z_{h-1}, z_h, \ldots, z\rangle$ where z_h, possibly coincident with p, is the first node in such a path which is still in the queue. In turn, its parent z_{h-1} is either a nonred node, or has already been dequeued: in both cases its distance from the source has been correctly computed, i.e., $D'(z_{h-1}) = d'(z_{h-1})$. Therefore, by the above claim, when z is dequeued, the priority of z_h in H is subject to the constraint

$$\Delta(z_h) \leq d'(z_{h-1}) + w_{z_{h-1},z_h} - d(z_h) = \delta(z_h).$$

Since $\Delta(z_h) \geq \delta(z_h)$, we have that $\Delta(z_h) = \delta(z_h)$. By inequality (2) above, we know that $\delta(z) = d'(z) - d(z) < \Delta(z)$. We will now show that $\Delta(z) \leq \Delta(z_h)$ and, hence, $\delta(z) < \Delta(z) \leq \Delta(z_h) = \delta(z_h)$. This contradicts Lemma 4.4, that states that the values of δ must be monotone nondecreasing along any optimal path from the source to node z.
In order to prove that $\Delta(z) \leq \Delta(z_h)$ it is sufficient to prove that all *red* nodes are extracted from H in nondecreasing order of priority.
In fact after that all red nodes have been inserted in the queue, and node z is extracted from Q with priority $\Delta(z) > 0$, a neighbor h of z may improve its priority in H to a value $\Delta(h)$ not smaller than $\Delta(z)$. In fact, from lines 27–28, we have:

$$\Delta(h) = D'(z) + w_{z,h} - D(h) = \Delta(z) + D(z) + w_{z,h} - D(h) = \Delta(z) + d(z) + w_{z,h} - d(h) \geq \Delta(z)$$

where the last inequality is due to the optimality condition on (z, h) before the update.

2. p has already been dequeued or is nonred. In this case $D'(p)$ has been correctly computed, i.e., $D'(p) = d'(p)$. Hence, by the claim, $\Delta(z) \leq d'(p) + w_{p,z} - d(z)$. By combining this inequality with inequality (2) above, we obtain:

$$d'(z) < d(z) + \Delta(z) \leq d'(p) + w_{p,z}$$

which contradicts the fact that p is parent of z in an optimal path. □

Let n_R, n_P and n_W be the number of nodes that have been colored *red*, *pink*, and *white*, respectively, at the end of Procedure `Color`, and let m_R, $m_R \leq k \cdot n_R$, be the number of *red* arcs, i.e., arcs whose both endpoints are *red*. In the following, we first evaluate the complexity of `Color` in terms of parameters n_R, n_P, and n_W; then we bound the running time of `Increase` as a function of the same parameters.

Lemma 4.6 *Let $G = (N, A)$ be a digraph with arbitrary arc weights. If G has a k-bounded accounting function and a weight-increase operation is performed on arc (x, y), then it is possible to color nodes in $T(y)$ in $O((n_R + n_P)k \log n + n_W)$ total time.*

Proof. First we observe that, every time an uncolored node z on the top of Q is considered during the execution of Procedure `Search_Equivalent_Path`, we first select the old parent q of z as a possible candidate parent for z. This is done in order to avoid to traverse the arcs in $\textsc{in}(z)$ when z is a node that will be colored *white*. If q is uncolored then we push it on the stack and find a candidate parent for it.

Observe now that a node is inserted in M at most once and only *red* and *pink* nodes are inserted in M. Hence, the total cost of `Color` (without the cost of `Search_Equivalent_Path`), is $O((n_R + n_P)k \log n)$.

In order to bound the cost of all calls to Procedure `Search_Equivalent_Path`, we observe that a node is inserted at most once in Q; since all nodes inserted in Q are colored it follows that the total cost of stack operations is $O(n_R + n_P + n_W)$. Now observe that the total cost of executing lines 20– 30 (for all calls to the procedure) is bounded by $O(n_P + n_W)$ (in fact at each iteration of the loop a node is colored either *white* or *pink*). Analogously the total cost of line 37 is $O(n_R)$.

The total cost of the remaining part of the code can be bounded as follows; let c_p and $\overline{c}_p$ be the total number of 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 for p when p is on top of Q. It is immediate to see that, with the exclusion of line 6, lines 20– 30 and line 37, the total cost of all calls to the procedure is $O(\sum_{\forall p}(c_p + \overline{c}_p + 1))$. Now we observe that if p is *white* then $c_p + \overline{c}_p = 1$. If p is either *pink* or *red* then $c_p \leq k$ (in fact we assume that the graph admits a k-bounded accounting function). It follows that $\sum_{\forall p}(c_p + \overline{c}_p + 1) = O(n_W + k(n_P + n_R)) + \sum_{\forall p} \overline{c}_p$.

In order to bound the last term of the right hand side, observe that $\overline{c}_p$ is bounded by one plus the number of arcs in $\overline{\text{IN-OWN}}(p)$ to a *pink* or a *red* node; since the head node that owns the arc is colored, then the hypothesis that the graph has a k–bounded function implies that $\sum_{\forall p} \overline{c}_p = O((k+1)(n_P + n_R))$. □

Theorem 4.7 *Let $G = (N, A)$ be a digraph with arbitrary arc weights. If G has a k-bounded accounting function, then it is possible to update $T(s)$ and the distances of nodes from s after the execution of a weight-increase operation, in $O((n_R + n_P)k \log n + n_W)$ time.*

Proof. By Lemma 4.6, the running time of `Color` and `Search_Equivalent_Path` is $O((n_R + n_P)k \log n + n_W)$. The *red* arcs can be colored in $O(k \cdot n_R)$ time; the same time is required to uncolor such arcs (line 30).

All the red nodes are enqueued in H (lines 7–18) 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 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.

In the main **while** loop (lines 19–31), n_R *red* nodes are dequeued. Observe that each time that a *red* node z is extracted from H, and therefore it improves its distance from the source, Procedure `Increase` traverses all *red* arcs in OUT(z). This means that $m_R = k \cdot n_R$ *red* arcs are scanned, and hence, the total cost of the calls to `Heap_Improve` is $O(k \cdot n_R \cdot \log n)$. The above discussion implies that the worst case cost of the **while** loop is equal to $O(k \cdot n_R \cdot \log n)$.

In the last phase (lines 32–40), the updates to the local priority queues $\overline{\text{IN-OWN}}$ and $\overline{\text{OUT-OWN}}$ of all the *red* nodes (each requiring $O(\log n)$ time) are performed by scanning again the lists IN-OWN and OUT-OWN of the *red* nodes (lines 38–39), leading to an overall worst case time bound of $O(k \cdot n_R \cdot \log n))$ for this phase. □

Corollary 4.8 *It is possible to update $T(s)$ and the distances of nodes from s in a general digraph after the execution of a weight-increase operation, in $O(n \cdot \sqrt{m} \cdot \log n)$ worst case time.*

5 Conclusions and open problems

We have proposed algorithms for the fully dynamic single source shortest path problem on general graphs with positive and negative arc weights that are better by a factor of $O(\sqrt{m}/\log n)$ than recomputing the new solution from scratch. We are currently implementing the algorithms and we believe that they might be of practical interest; this is partly based on the experiments performed in the case of positive arc weights [9].

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 graph with arbitrary arc weights.

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. *Proc. 1st ACM-SIAM Symposium on Discrete Algorithms*, 32–42, 1990.

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

[4] S. Chaudhuri and C. D. Zaroliagis. Shortest path queries in digraphs of small treewidth. *Proc. Int. Coll. on Automata Languages and Programming. Lecture Notes in Computer Science* 944, 244–255, 1995.

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

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

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

[8] P. G. Franciosa, D. Frigioni, R. Giaccio. Semi dynamic shortest paths and breadth-first search on digraphs. *Proc. Annual Symposium on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science* 1200, 26–40, 1997.

[9] D. Frigioni, M. Ioffreda, U. Nanni, G. Pasqualone. Experimental analysis of dynamic algorithms for the single source shortest path problem. *Proc. 1st Work. on Algorithm Engineering*, pp. 54–63, 1997.

[10] D. Frigioni, A. Marchetti-Spaccamela and U. Nanni. Semi dynamic algorithms for maintaining single source shortest path trees. *Algorithmica* - Special Issue on Dynamic Graph Algorithms, to appear.

[11] D. Frigioni, A. Marchetti-Spaccamela, U. Nanni. Fully dynamic output bounded single source shortest path problem. *Proc. ACM-SIAM Symposium on Discrete Algorithms*, 212–221, 1996.

[12] A. V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM J. Comp.*, **24** (1995), 494–504.

[13] P. N. Klein, S. Rao, M. Rauch and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Proc. ACM Symposium on Theory of Computing*, 27–37, 1994.

[14] S. M. Malitz. Genus g graphs have pagenumber $O(\sqrt{g})$. *Journal of Algorithms*, **17** (1994), 85–109.

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

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

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

[18] H. Rohnert. A dynamization of the all-pairs least cost path problem. *Proc. 2nd Annual Symposium on Theoretical Aspects of Computer Science. Lecture Notes in Computer Science* 182, 279–286.