Alcom-FT Technical Report Series
ALCOMFT-TR-01-32

A Fully Dynamic Algorithm for Distributed Shortest Paths*

SERAFINO CICERONE GABRIELE DI STEFANO

Dipartimento di Ingegneria Elettrica
Università dell'Aquila, I-67040 Monteluco di Roio - L'Aquila, Italy
E-mail: cicerone@ing.univaq.it, gabriele@gs.ing.univaq.it

DANIELE FRIGIONI UMBERTO NANNI

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

March 20, 2001

Abstract

We propose a fully-dynamic distributed algorithm for the *all-pairs* shortest paths problem on general networks with positive real edge weights. If Δ_σ is the number of pairs of nodes changing the distance after a single edge modification σ (*insert*, *delete*, *weight decrease*, or *weight increase*) then the message complexity of the proposed algorithm is $O(n\Delta_\sigma)$ in the worst case, where n is the number of nodes of the network. If $\Delta_\sigma = o(n^2)$, this is better than recomputing everything from scratch after each edge modification. Up to now only a result of Ramarao and Venkatesan was known, stating that the problem of updating shortest paths in a dynamic distributed environment is as hard as that of computing shortest paths.

1 Introduction

The importance of finding shortest paths in graphs is motivated by the numerous theoretical and practical applications known in various fields as, for instance, in combinatorial optimization and in communication networks (e.g., see [1, 13]). We consider the distributed version of the *all-pairs shortest paths* problem. Finding efficient solutions for this problem is crucial when processors in a network need to route messages with the minimum cost.

Actually, in many practical applications it is required to manage networks that dynamically change over the time, in the sense that communication links can go up and down during the lifetime of the network. For this reason, the problem of *updating* shortest paths in a dynamic distributed environment arises naturally in practical applications. For instance, the *OSPF* protocol, widely used in the Internet (e.g., see [11, 16]), basically updates the routing tables of the nodes after a change to the network by using a distributed version of Dijkstra's algorithm. In this and many other crucial applications the worst case complexity of the adopted

*This work was partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

protocols is never better than recomputing the shortest paths from scratch. Therefore, it is important to find distributed algorithms for shortest paths that do not recompute everything from scratch after each change to the network, because this could result very expensive in practice.

If the topology of a network is represented as a graph, where nodes represent processors and edges represent links between processors, then the typical update operations on a dynamic network can be modeled as insertions and deletions of edges and update operations on the weights of edges. When arbitrary sequences of the above operations are allowed, we refer to the *fully dynamic problem*; if only insert and weight decrease (delete and weight increase) operations are allowed, then we refer to the *incremental* (*decremental*) problem.

Previous works and motivations. Many solutions have been proposed in the literature to find and update shortest paths in the sequential case on graphs with non-negative real edge weights (e.g., see [1, 13] for a wide variety). The state of the art is that no efficient fully dynamic solution is known for general graphs that is faster than recomputing everything from scratch after each update, both for single-source and all-pairs shortest paths. Actually, only *output bounded* fully dynamic solutions are known on general graphs [6, 14].

Some attempts have been made also in the distributed case [4, 7, 9, 12, 15]. In this field the efficiency of an algorithm is evaluated in terms of *message, time* and *space* complexity as follows. The *message complexity* of a distributed algorithm is the total number of messages sent over the edges. We assume that each message contains $O(\log n + R)$ bits, where R is the number of bits available to represent a real edge weight, and n is the number of nodes in the network. In practical applications messages of this kind are considered of "constant" size. The *time complexity* is the total (normalized) time elapsed from a change. The *space complexity* is the space usage per node.

In [7], an algorithm is given for computing all-pairs shortest paths requiring $O(n^2)$ messages, each of size n. In [9], an efficient incremental solution has been proposed for the distributed all-pairs shortest paths problem, requiring $O(n \log(nW))$ amortized number of messages over a sequence of edge insertions and edge weight decreases. Here, W is the largest positive *integer* edge weight. In [4], Awerbuch *et al.* propose a general technique that allows to update the all-pairs shortest paths in a distributed network in $\Theta(n)$ amortized number of messages and $O(n)$ time, by using $O(n^2)$ space per node. In [15], Ramarao and Venkatesan propose algorithms for both finding and updating shortest paths distributively. They propose a distributed algorithm for finding a shortest paths tree of a network with positive real edge weights requiring $\Theta(n^2)$ messages, $O(n^2)$ time, and $O(n)$ space per node. Using this algorithm as a subroutine, they give a solution for the all-pairs shortest paths problem that requires $O(n^3)$ messages and time. Furthermore, when only insertions of edges and edge weight decreases are allowed, they propose a distributed algorithm requiring $O(n^2)$ messages and time for updating all-pairs shortest paths. Finally, they give algorithms that update single-source (all-pairs) shortest paths in a fully dynamic setting in $\Theta(n^2)$ ($O(n^3)$) messages and time, and show that, in the worst case, the problem of updating shortest paths is as difficult as that of computing shortest paths.

The results of Ramarao and Venkatesan have a remarkable consequence. They suggest that two main directions should be investigated in order to devise efficient fully dynamic algorithms for updating all-pairs shortest paths in a distributed network: *i*) to study the trade-off between the message, time and space complexity for each kind of dynamic change;

ii) to devise algorithms that are efficient in different complexity models (with respect to worst case and amortized analyses).

Concerning the first direction, in [9] an efficient incremental solution has been provided, and the difficulty of dealing with edge deletions has been addressed. This difficulty arises also in the sequential case (see for example [3]).

In this paper, the second direction is investigated. Concerning the choice of a different complexity model, we observed that the *output complexity* [5, 6, 13] was a good candidate. In fact, the output complexity has been shown to be a robust measure of performance for dynamic algorithms in the sequential case [2, 5, 6, 13, 14]. This notion applies when the algorithms operate within a framework where explicit updates are required on a given data structure. In such a framework, output complexity allows to evaluate the cost of dynamic algorithms in terms of the *intrinsic cost* of the problem on hand, i.e., in terms of the number of updates to the output information of the problem that are needed after any input update. Here we show the merits of this model also in the field of distributed computation, and show that in several cases it is possible to improve over the results of Ramarao and Venkatesan [15].

Results of the paper. The novelty of this paper consists of a new efficient and practical solution for the fully dynamic distributed all-pairs shortest paths problem. To the best of our knowledge, the proposed algorithm represents the *first* solution for this problem, whose message complexity in several cases compares favorably with respect to recomputing everything from scratch after each edge modification. This result is achieved by explicitly devising an algorithm whose main purpose is to minimize the cost of each output update determined by an input modification.

We use the following complexity model. Given an input change σ and a source node s, let $\delta_{\sigma,s}$ be the set of nodes changing either the distance or the parent in the shortest paths tree rooted at s as a consequence of σ. Furthermore, let $\delta_\sigma = \cup_{s\in V}\delta_{\sigma,s}$ and $\Delta_\sigma = \sum_{s\in V}|\delta_{\sigma,s}|$. We evaluate the message and time complexity of our algorithm as a function of Δ_σ. Intuitively, this parameter represents a lower bound to the number of messages of constant size to be sent over the network after the input change σ. In fact, if the distance from u to v changes due to σ, then at least u and v have to be informed about the change.

We design an algorithm that updates only the distances and the shortest paths that actually change after an edge modification. In particular, if *maxdeg* is the maximum degree of the nodes in the network, then we propose a fully dynamic algorithm for the distributed all-pairs shortest paths problem requiring in the worst case: $O(maxdeg \cdot \Delta_\sigma)$ messages and $O(\Delta_\sigma)$ time for *insert* and *weight decrease* operations; $O(\max\{|\delta_\sigma|, maxdeg\} \cdot \Delta_\sigma)$ messages and time for *delete* and *weight increase* operations. The space complexity is $O(n)$ per node. If $\Delta_\sigma = o(n^2)$, then the given bounds compare favourably with respect to [15].

Some of the ideas the proposed algorithm is based on are borrowed from [6, 13], while others are new. In particular, we borrowed from [6, 13]: *a*) the idea of evaluating the cost of shortest paths algorithms in the output complexity model; *b*) the idea of separating the algorithm for weight increase and delete operations into two activities: *i*) finding the nodes affected by the operation: *ii*) determining new distances and parents for these nodes. Except for the above similarities, we use different algorithmic techniques and data structures with respect to [6, 13].

The paper is organized as follows. In Section 2 we introduce the notation and describe the data structures used throughout the paper. In Sections 3 and 4 we describe the algorithms for weight decrease and weight increase operations, respectively, and prove their correctness and complexity. Finally, in Section 5 we provide some concluding remarks.

2 Preliminaries

We consider *point-to-point communication networks.* In these networks, a processor can generate a single message at a time and send it to all its neighbors in one time step. Messages are delivered to their respective destinations within a finite delay, but they might be delivered out of order; that is, the edges are non-FIFO. The distributed algorithms presented in this paper allow communications only between neighbors. We assume an asynchronous message passing system; that is, a sender of a message does not wait for the receiver to be ready to receive the message.

We represent a computer network, where computers are connected by communication links, by an *undirected weighted graph* $G = (V, E, w)$, where V is a finite set of n *nodes*, one for each computer; E is a finite set of m *edges*, one for each link; and w is a *weight function* from E to positive real numbers. An edge (u, v) is an unordered pair of nodes u and v; u and v are *neighbors*, and the *weight* of (u, v) is denoted as $w(u, v)$. For each node u, $N(u) = \{u_1, u_2, \ldots, u_{deg(u)}\}$ contains the neighbors of u, and $deg(u) = |N(u)|$. A *path* between two nodes u and v is a finite sequence $p = \langle u \equiv v_0, v_1, \ldots, v_k \equiv v\rangle$ of distinct nodes such that, for each $0 \leq i < k$, $(v_i, v_{i+1}) \in E$, and the *weight of the path* is $weight(p) = \sum_{0 \leq i < k} w(v_i, v_{i+1})$. The *shortest path weight*, also called *distance*, between any pair of nodes u and v, denoted as $d(u, v)$, is the minimum weight of all possible paths connecting u to v in G. A *shortest path* from u to v is defined as any path p such that $weight(p) = d(u, v)$. If $s \in V$ is an arbitrary source node, we denote as T_s a shortest paths tree of G rooted at s; for any $u \in V$, $T_s(u)$ denotes the subtree of T_s rooted at u. Every $u \in V$ has one *parent* (except for s), and a set of *children* in T_s. We assume that each node u knows: *i)* the identities of all nodes, $1, 2, \ldots, n$; *ii)* the identity of each node in $N(u)$; *iii)* for each $u_i \in N(u)$, the edge connecting u to u_i and the weight $w(u, u_i)$.

We maintain the following data structures. A *routing table* $RT[\cdot,\cdot]$, needed to store the information on the all-pairs shortest paths. Each node u in G, maintains only the set of records $RT[u,\cdot]$, one record $RT[u, v]$ for each possible destination $v \in V \setminus \{u\}$. Each record has two fields: $RT[u, v].weight$, and $RT[u, v].via$, which denote respectively, the distance between u and v, and the neighbor of u in the path used to determine the *weight*. In the paper, each subcomponent of the routing table $RT[u, v].field$ is also denoted as $field(u, v)$. The space required to store the routing table is clearly $O(n)$ per node. Notice that, the procedures implicitly maintain a shortest paths tree T_s for each source s; T_s is the tree induced by the set of edges $(u, via(u, s))$, for each node u reachable from s.

We assume that, when a modification occurs concerning an edge (u, v) in a *dynamic* network, only nodes u and v are able to detect the change. Furthermore, we do not allow changes to the network that occur during the execution of the proposed algorithm. Finally, for each $v \in V$, $d'(s, v)$ denotes the distance from s to v in the graph G' obtained from G after an edge modification. In general, in the remainder of the paper, we denote by γ' any parameter γ after an edge modification.

We describe the procedures for handling *weight decrease* and *weight increase* operations on the edges of a graph; the extension to *insert* and *delete* operations, respectively, is straightforward. After an edge modification, for each source s, the proposed procedures correctly

Node v receives the message *start-decrease*$(u, s, weight(u, s))$.

```
1.  if via(v, y) = u then
2.    begin
3.      if weight(v, s) > w(v, u) + weight(u, s) then
4.        begin
5.          weight(v, s) := w(v, u) + weight(u, s)
6.          via(v, s) := u
7.          for each v_i ∈ N(v) \ {u} do send start-decrease(v, s, weight(v, s))
8.        end
9.    end
```

Figure 1: The decreasing algorithm of node v.

update $weight(v, s)$ as $d'(v, s)$, and $via(v, s)$ as the neighbor of v in the path used to determine $weight(v, s)$ in G'. Both for weight decrease and for weight increase operations, we describe the behavior of the algorithm with respect to a fixed source s. To obtain the algorithm for updating all-pairs shortest paths, it is sufficient to apply the algorithm with respect to all the possible sources.

It is worth noting that, the procedures proposed to update the shortest paths with respect to a fixed source s, do not represent a space efficient fully dynamic distributed solution for the single-source shortest paths problem. In fact, in order to be run, they need to know the information on the all-pairs shortest paths before the edge modification.

3 Decreasing the weight of an edge

Suppose that a weight decrease operation σ is performed on edge (x, y), that is, $w'(x, y) = w(x, y) - \epsilon$, $\epsilon > 0$. In this case, if $d(s, x) = d(s, y)$, then $\delta_{\sigma,s} = \emptyset$, and no recomputation is needed. Otherwise, without loss of generality, we assume that $d(s, x) < d(s, y)$. In this case, if $d'(s, y) < d(s, y)$ then all the nodes that belong to $T_s(y)$ are in $\delta_{\sigma,s}$. On the other hand, there might exist nodes not contained in $T_s(y)$ that belong to $\delta_{\sigma,s}$. In any case, every node in $\delta_{\sigma,s}$ decreases its distance from s as a consequence of σ.

The proposed solution is based on the following lemma.

Lemma 3.1 *If $v \in \delta_{\sigma,s}$, then there exists a shortest path connecting v to s in G' that contains the path from v to y in T_y as subpath.*

Proof. Let us suppose that $y \in \delta_{\sigma,s}$ and that $d(x, s) < d(y, s)$. Since $v \in \delta_{\sigma,s}$, any shortest path from s to v in G' passes through edge (x, y). It follows that there exists in G' a shortest path P from s to v having the form $P = (s, \ldots, x, y, \ldots v)$. This implies that the subpath of P connecting v to y is a shortest path, and the lemma follows. $\square$

In detail, the proposed solution performs a visit of T_y starting from y. This visit finds the nodes in $\delta_{\sigma,s}$ and updates their routing tables. Each of the visited nodes v performs the algorithm of Figure 1. When v figures out that it belongs to $\delta_{\sigma,s}$ (line 3), it sends the message *start-decrease*$(v, s, weight(v, s))$ to all its neighbors (that says to the receiver that it has to start the decrease algorithm because the distance from s to v has been improved).

This is needed because v does not know its children in T_y (since y is arbitrary, maintaining this information would require $O(n^2)$ space per node). Only when a node, that has received the message *start-decrease*$(u, s, \textit{weight}(u, s))$, performs line 1, it figures out whether it is child of a node in T_y.

Notice that the algorithm of Figure 1 is performed by every node v distinct from y. The algorithm for y is slightly different, as follows:

1. y starts the algorithm when it receives the message *start-decrease*$(u, s, \textit{weight}(u, s))$ from node $u \equiv x$. This message is sent to y as soon as x detects the weight decrease on edge (x, y);

2. y does not perform the test of line 1;

3. the weight $w(v, u)$ at lines 3 and 5 coincides with $w'(x, y)$.

Theorem 3.2 *For each node $v \in T_s$ the algorithm of Figure 1 correcly computes weight(v, s) and via(v, s) after a weight decrease operation on edge (x, y).*

Proof. By Lemma 3.1, a new shortest path from s to a node v in G' can be found as the chaining of a shortest path from s to x, edge (x, y), and a shortest path from y to v.

The algorithm in Figure 1, for any source node s, enforces the following properties:

- the update of a node v is done if and only if the message *start-decrease*$(u, s, \textit{weight}(u, s))$ comes from the parent of v in T_y (see line 1);

- a node v may update its distance from s only if its parent in T_y has been updated (see line 7).

Therefore, for each source s, the computation of the new distances proceeds by visiting the shortest paths tree T_y and, for each visited node v, the search is pruned if and only if the test at line 3 returns a negative answer. On the contrary, if $\textit{weight}(v, s)$ is updated, then all the ancestors of v in T_y have been updated and, by definition, they provide a shortest path from y to v. □

Theorem 3.3 *Updating all-pairs shortest paths over a distributed network with n nodes and positive real edge weights, after a weight decrease operation, requires $O(\textit{maxdeg} \cdot \Delta_\sigma)$ messages, $O(\Delta_\sigma)$ time, and $O(n)$ space per node.*

Proof. After a weight decrease operation σ of edge (x, y), the set of nodes v such that $RT'[v, s] \neq RT[v, s]$ coincides with $\delta_{\sigma,s}$. Each of such nodes v sends at most $\textit{deg}(v)$ messages (see line 7). Hence, the total number of messages sent over the network is $O(\textit{maxdeg} \cdot |\delta_{\sigma,s}|)$. Summing up the values $O(\textit{maxdeg} \cdot |\delta_{\sigma,s}|)$ over all possible nodes in δ_σ, we obtain the following bound for the number of messages required to update the all-pairs shortest paths:

$$\sum_{s \in \delta_\sigma} O(\textit{maxdeg} \cdot |\delta_{\sigma,s}|) = O(\textit{maxdeg} \cdot \Delta_\sigma)$$

The time complexity of the algorithm can be derived from the above analysis by eliminating the factor *maxdeg* from the message complexity. In fact, a node v can send $RT'[v, s]$ to all its neighbors in one time step. The space complexity is clearly $O(n)$ per node. □

4 Increasing the weight of an edge

As in the case of weight decrease operations, we describe the behavior of the algorithm with respect to a fixed source s. Suppose that a *weight increase* σ is performed on edge (x, y), that is, $w'(x, y) = w(x, y) + \epsilon$, $\epsilon > 0$. If $(x, y) \notin T(s)$, then nothing has to be done; in fact, the shortest paths from s to the other nodes are not affected by the change. Otherwise, in order to distinguish the set of required updates determined by σ, we associate a color, denoted as $color(q, s)$, to each node q with respect to s (as in [6]), as follows:

- $color(q, s) = white$ if q changes neither the distance from s nor the parent in T_s (i.e., q is white if $weight'(q, s) = weight(q, s)$ and $via'(q, s) = via(q, s)$);
- $color(q, s) = pink$ if q preserves its distance from s, but it must replace the old parent in T_s (i.e., q is pink if $weight'(q, s) = weight(q, s)$ and $via'(q, s) \neq via(q, s)$);
- $color(q, s) = red$ if q increases the distance from s (i.e., $weight'(q, s) > weight(q, s)$).

According to this coloring, the nodes in $\delta_{\sigma,s}$ are exactly the red and pink nodes.

Remark 4.1 *If we assume that $d(s, x) < d(s, y)$ then the following facts hold:*

F1: If $v \notin T_s(y)$, then $v \notin \delta_{\sigma,s}$; indeed, the shortest path from s to v in T_s is not affected by the change. In other words, all the red and pink nodes belong to $T_s(y)$.

F2: If a node v is pink or red, then either v is a child of a red node in $T_s(y)$, or $v \equiv y$.

F3: If v is pink or white then all the other nodes in $T_s(v)$ are white.

By Fact F1 of the above remark, only the information on the nodes in $T_s(y)$ must be processed. Conversely, if $d(s, x) > d(s, y)$, then only the information on the nodes in $T_s(x)$ must be considered. Since we assume $(x, y) \in T_s$, the case $d(s, x) = d(s, y)$ cannot occur. Without loss of generality, in the remainder of the section we assume $d(s, x) < d(s, y)$. By Fact F3 of Remark 4.1, if $T_s(y)$ contains a pink node v, then all the nodes in $T_s(v)$ remain white and do not require any update. This implies that, if we want to bound the number of messages delivered over the network as a function of the number of output updates, then we cannot search the whole $T_s(y)$.

For each red or pink node v, we introduce the following notation:

- $\mathrm{AP}_s(v)$ denotes the set of *alternative parents* of v with respect to s, that is, a neighbor q of v belongs to $\mathrm{AP}_s(v)$ when $d(s, q) + w(q, v) = d(s, v)$.
- $\mathrm{BNR}_s(v)$ denotes the *best non-red neighbor* of v with respect to s, that is, a non-red neighbor q of v such that the quantity $d(s, q) + w(q, v)$ is minimum.

Notice that, if $\mathrm{AP}_s(v)$ is empty and $\mathrm{BNR}_s(v)$ exists, then $\mathrm{BNR}_s(v)$ represents the best way for v to reach s in G' by means of a path that does not contain red nodes.

Definition 4.1 *Let v be a red node such that $\mathrm{BNR}_s(v)$ exists. Let p be a shortest path from s to v in G, and p' be a shortest path from s to v in G' via $\mathrm{BNR}_s(v)$. Node v is called* boundary *for s if $weight(p') < weight(p) + \epsilon$.*

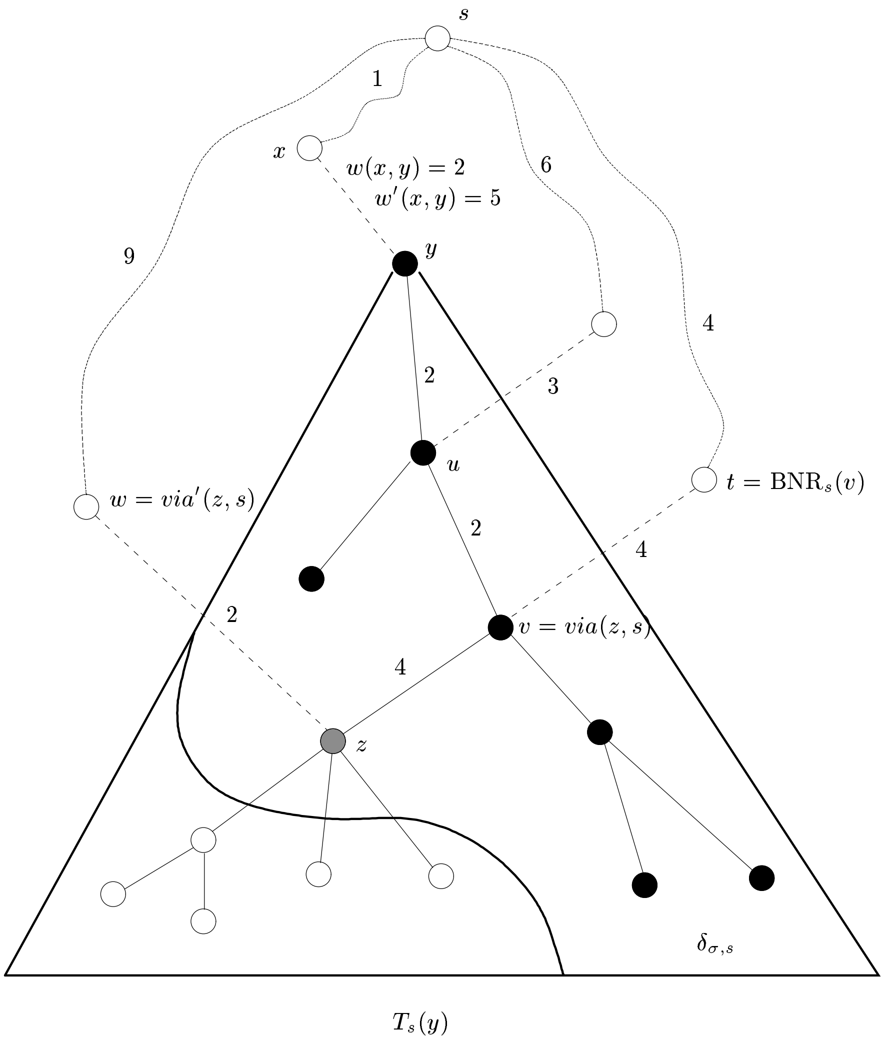

Figure 2: *Coloring the nodes in $T_s(y)$ after increasing the weight of (x, y). Dashed lines and curves represent edges and paths not in $T_s(y)$, respectively.*

The importance of distinguishing the boundary nodes among the red nodes is motivated by the following lemma.

Lemma 4.2 *Let v be a red node, and p be the shortest path from s to v in T_s. Let p' be a shortest path from s to v in G'. Then, either p' coincides with p, or p' contains a boundary node.*

Proof. Let $p' = \langle v \equiv v_1, \ldots, v_t \equiv s \rangle$ a shortest path from v to s in G'. Let v_i, $i > 1$, the first non-red node in p' (v_i exists because at least s is white). By definition of color it follows that $v_i, \ldots, v_t$ are non-red. If $v_{i-1} = y$, then p' coincides with p. Otherwise, v_{i-1} is boundary. □

In Figure 2 we show the concepts introduced so far. In particular, as a consequence of the increase of the weight of edge (x, y) from 2 to 5, node z is colored pink. In fact, there exists a path from s to z through $w \notin T_s(y)$ whose weight is equal to $d(z, s)$, and hence $w = via'(z, s)$.

PHASE	MESSAGE NAME	PARAMETER	COMMENT
coloring	*start-coloring*	(z, s)	node z asks the receiver to start the coloring wrt source s
	end-coloring	(z, s)	node z notifies the receiver the end of its own coloring wrt source s
boundarization	*start-boundarization*	(z, s, ϵ)	node z sends ϵ to notify the receiver to start the boundarization wrt source s
	end-boundarization	$(z, s, B_s(z))$	node z sends $B_s(z)$ to notify the receiver the end of its own boundarization wrt source s
recomputing	*start-recomputing*	$(z, s, B_s(y))$	node z sends $B_s(y)$ to notify the receiver to start the recomputing wrt source s

Figure 3: The messages delivered in the algorithms for weight increase operations

According to Fact $F3$ of Remark 4.1, all the nodes in $T_s(z)$ are white and do not belong to $\delta_{\sigma,s}$. The remaining nodes in $T_s(y)$ are colored red and among them only v is boundary. In fact, the path from s to v through $t = \text{BNR}_s(v)$ is shorter than the path from s to v through (x, y) after the increase.

For each red node u, we denote as $B_s(u)$ the set $\{\langle v; \ell_v \rangle \mid v \in T_s(u) \text{ is boundary for } s\}$, where ℓ_v is the weight of the path from v to s via $\text{BNR}_s(v)$, i.e., $\ell_v = w(v, \text{BNR}_s(v)) + \mathit{weight}(\text{BNR}_s(v), s)$.

The algorithm that we propose for handling a weight increase operation on edge (x, y) consists of the following three phases:

1. *Coloring*: the red and pink nodes with respect to s are found; this phase terminates when y is aware that all the nodes in $T_s(y)$ have been colored.

2. *Boundarization*: this phase starts at the end of the coloring, and performs two tasks: (i) each pink node v computes $\mathit{via}(v, s)$ in G'; (ii) each red node v checks whether it is boundary or not, and computes $B_s(v)$. This phase terminates when y knows $B_s(y)$.

3. *Recomputing*: this phase starts at the end of the boundarization, when y communicates $B_s(y)$ to each red node. By using $B_s(y)$, each red node v computes $\mathit{weight}'(v, s)$ and $\mathit{via}'(v, s)$ by a simple local computation.

We remark that the coloring phase does not perform any update to $RT[\cdot, s]$ (and, as a consequence, to $T_s(y)$). In particular, a pink node v updates $\mathit{via}(v, s)$ during the boundarization phase, whereas a red node v updates both $\mathit{weight}(v, s)$ and $\mathit{via}(v, s)$ during the recomputing phase.

We now provide three algorithms, each corresponding to a phase, and for each algorithm we also give a detailed description. Each algorithm is locally executed when a node receives a specific message. The table in Figure 3 summarizes the messages used by the algorithms.

Two very simple messages have been omitted from the table. The former is the message by which a node z asks a neighbor v for $\mathit{weight}(v, s)$, $\mathit{via}(v, s)$, or $\mathit{color}(v, s)$, while the latter is the corresponding answer of v. Before describing the algorithms in detail, we remark that they work under the following assumptions:

The *red* node v receives the message *start-coloring*(z, s).

```
send to z the message end-coloring(v, s); HALT
```

The *non-red* node v receives the message *start-coloring*(z, s).

```
if color(v, s) = white then
  begin
    if z ≠ via(v, s) then send to z the message end-coloring(v, s); HALT
    AP_s(v) := ∅
    for each v_i ∈ N(v) \ {z} do
      begin
        ask v_i for weight(v_i, s) and color(v_i, s)
        if color(v_i, s) ≠ red and weight(v, s) = w(v, v_i) + weight(v_i, s)
          then AP_s(v) := AP_s(v) ∪ {v_i}
      end
  end
if z ∈ AP_s(v) then AP_s(v) := AP_s(v) \ {z}
if AP_s(v) ≠ ∅
  then color(v, s) := pink
  else begin
         color(v, s) := red
         for each v_i ∈ N(v) \ {z} send to v_i the message start-coloring(v, s)
         for each v_i ∈ N(v) \ {z} wait from v_i the message end-coloring(v_i, s)
       end
send to z the message end-coloring(v, s);
```

Figure 4: The coloring phase of node v

A1. If a node v receives a request for $weight(v, s)$, $via(v, s)$, or $color(v, s)$ then it answers immediately.

A2. If a red node v receives the message *start-coloring*(z, s), then it immediately sends the message *end-coloring*(v, s) to z.

4.1 Coloring phase

This is the first phase of the algorithm. At the beginning all nodes in $T_s(y)$ are assumed white with respect to s, while at the end of the phase each of such nodes has decided its own color. The pink and red nodes are found starting from y and performing a pruned search of $T_s(y)$. The coloring phase of a generic node $v \neq y$ is given in Figure 4; the behaviour of y is sligthly different and is discussed later. Here we just point out that the coloring starts when x and y detect the weight change on edge (x, y): x sends *start-coloring*(x, s) to y, and y sends *start-coloring*(y, s) to x.

Now we describe in detail the coloring phase of $v \neq y$. When v receives the message *start-coloring*(z, s) it understands that has to decide its color. The behavior of v depends on its current color. Three cases may arise:

1. *v is white:* In this case, v tests whether z is its parent in $T_s(y)$ or not. If $z \neq via(v, s)$

(line 3), then by Fact 2 of Remark 4.1 the color of v remains white and v communicates to z the end of its coloring. If $z = via(v, s)$ then v finds all the alternative parents with respect to s, and records them into $\text{AP}_s(v)$ (lines 4–10). After that, v checks how many alternative parents are available. If there is at least one alternative parent (line 13), then v sets its color to pink (line 14) and communicates to z the end of its coloring phase (line 20). Otherwise, if there is no alternative parent (line 15), v performs the following four actions (at lines 16, 17, 18, and 20, respectively) :

- sets its color to red;
- propagates the message *start-coloring*(v, s) to each neighbor but z;
- waits for the message *end-coloring*(v_i, s) from each neighbor $v_i \neq z$;
- communicates the end of its coloring phase (and, implicitly, the end of coloring of each node in $T_s(v)$) by sending *end-coloring*(v, s) to z.

2. *v is pink:* In this case, the test at line 12 is the first action performed by v. If z is an alternative parent of v, then z is correctly removed from $\text{AP}_s(v)$ (since z is now red). After this removing, v could have no further alternative parent. As a consequence, v performs the test at line 13: if there are still elements in $\text{AP}_s(v)$, then v remains pink and sends to z the message concerning the end of its coloring phase (lines 14 and 20); otherwise, v becomes red and propagates the coloring phase to its neighbors (lines 15–19), as already described in case 1 above.

3. *v is red:* In this case, v performs a different procedure: it simply communicates to z the end of its coloring phase (see line 1 for red nodes). This is done to guarantee that Assumption A2 holds.

Notice that, according to this strategy, at the end of the coloring phase node y is aware that each node in $T_s(y)$ has been correctly colored according to the weight increase operation performed on edge (x, y). As remarked above, the algorithm of Figure 4 is performed by every node distinct from y, while the algorithm for y is slightly different. In particular, at line 20, y does not send *end-coloring*(y, s) to $z \equiv x$; instead, y starts the boundarization phase by broadcasting the value ϵ through $T_s(y)$.

4.2 Boundarization phase

This phase performs two main tasks: (i) each pink node v computes $via(v, s)$ in G'; (ii) each red node v checks whether it is boundary or not, and computes $B_s(v)$. This phase terminates when y knows $B_s(y)$. The boundarization phase of $v \neq y$ is shown in Figure 5 and is described below. The behaviour of y is slightly different and is described later.

When a pink node v receives the message *start-boundarization*$(via(v, s), s, \epsilon)$, it understands that the coloring phase is terminated; at this point v needs only to choose arbitrarily $via'(v, s)$ among the nodes in $\text{AP}_s(v)$, and to set its color to white (lines 2–5). At this point task (i) above has been accomplished.

When a red node v receives the message *start-boundarization*$(via(v, s), s, \epsilon)$ it needs to understand whether it is boundary or not. According to Definition 4.1, v has to know which is the shortest between the old path from v to s (whose weight is now increased by ϵ (line 8)), and the path from v to s via $\text{BNR}_s(v)$ (if any). To this aim, v has to find $\text{BNR}_s(v)$, and

Node v receives the message *start-boundarization*$(via(v,s), s, \epsilon)$

if $color(v,s) = pink$ **then**
  begin
    $via(v,s) := q$, where q is an arbitrary node in $\text{AP}_s(v)$
    $color(v,s) := white$; `HALT`
  end
if $color(v,s) = red$ **then**
  begin
    $\ell_v := weight(v,s) + \epsilon$
    $\text{BNR}_s(v) :=$ **nil**
    $\text{PINK-CHILDREN}_s(v) := \emptyset$
    $\text{RED-CHILDREN}_s(v) := \emptyset$
    for each $v_i \in N(v) \setminus \{z\}$ **do**
      begin
        v asks v_i for $weight(v_i,s)$, $via(v_i,s)$, and $color(v_i,s)$
        if $color(v_i,s) \neq red$ **and** $\ell_v > w(v,v_i) + weight(v_i,s)$ **then**
          begin
            $\ell_v := w(v,v_i) + weight(v_i,s)$
            $\text{BNR}_s(v) := v_i$
          end
        if $color(v_i,s) = pink$ **and** $via(v_i,y) = v$
          then $\text{PINK-CHILDREN}_s(v) := \text{PINK-CHILDREN}_s(v) \cup \{v_i\}$
        if $color(v_i,s) = red$ **and** $via(v_i,y) = v$
          then $\text{RED-CHILDREN}_s(v) := \text{RED-CHILDREN}_s(v) \cup \{v_i\}$
      end
    if $\text{BNR}_s(v) =$ **nil**
      then $B_s(v) := \emptyset$ $\{v$ is not boundary for $s\}$
      else $B_s(v) := \{\langle v; \ell_v \rangle\}$ $\{v$ is boundary for $s\}$
    for each $v_i \in \text{PINK-CHILDREN}_s(v) \cup \text{RED-CHILDREN}_s(v)$
          do send *start-boundarization*(v, s, ϵ) to v_i
    for each $v_i \in \text{RED-CHILDREN}_s(v)$ **do**
      begin
        wait the message *end-boundarization*$(v_i, s, B_s(v_i))$
        $B_s(v) := B_s(v) \cup B_s(v_i)$
      end
    send *end-boundarization*$(v, s, B_s(v))$ to $via(v,s)$
  end

Figure 5: The boundarization phase of node v

therefore it asks every neighbor v_i for $weight(v_i, s)$, $via(v_i, s)$ and $color(v_i, s)$ (see lines 13–24). At the same time, v takes advantage from the knowledge on $color(v_i, s)$ and $via(v_i, s)$ to find its pink and red children in $T_s(y)$, and to record them into PINK-CHILDREN$_s(v)$ and RED-CHILDREN$_s(v)$ (see lines 21 and 23). This information will be used to propagate the boundarization phase.

Now, if BNR$_s(v)$ exists and the weight of the path from v to s via BNR$_s(v)$ is smaller than $weight(v, s) + \epsilon$, then v is boundary for s. As a consequence, v initializes $B_s(v)$ as $\{\langle v; \ell_v \rangle\}$ (line 27), where ℓ_v is the weight of the path from v to s via BNR$_s(v)$.

At the end of the boundarization phase of node v, the set $B_s(v)$ contains all the pairs $\langle z; \ell_z \rangle$ such that $z \in T_s(v)$ is a boundary node. In fact, at line 28 v sends to each node $v_i \in$ PINK-CHILDREN$_s(v) \cup$ RED-CHILDREN$_s(v)$ *start-boundarization*(v, s, ϵ) (to propagate the boundarization), and then waits to receive $B_s(v_i)$ from $v_i \in$ RED-CHILDREN$_s(v)$ (lines 30–33). Notice that, v does not wait for any message from a node $v_i \in$ PINK-CHILDREN$_s(v)$, because, by definition, the boundary nodes are red. Whenever v receives $B_s(v_i)$ from a child v_i, it updates $B_s(v)$ as $B_s(v) \cup B_s(v_i)$ (line 32). Finally, at line 34, v sends $B_s(v)$ to y via $via(v, s)$. At this point task (ii) has been accomplished for node v. As a consequence, at the end of the boundarization phase, the set $B_s(y)$, containing all the boundary nodes for s, has been computed and stored in y.

Notice that, the algorithm of Figure 5 is performed by every node distinct from y. The algorithm for y is slightly different. In particular, at line 34, y does not send the message *end-boundarization*$(y, s, B_s(y))$ to $via(y, s) \equiv x$. Instead, y uses this information to start the recomputing phase by broadcasting through $T_s(y)$ the set $B_s(y)$ to each red node.

4.3 Recomputing phase

In this phase, each red node v computes $weight'(v, s)$ and $via'(v, s)$. The recomputing phase of a red node v is shown in Figure 6, and described in what follows. Let us suppose that v has received the message *start-recomputing*$(via(v, y), s, B_s(y))$.

First of all, by using the information contained in RED-CHILDREN$_s(v)$, v propagates $B_s(y)$ to the red nodes in $T_s(v)$.

Then, if v is not boundary, by Lemma 4.2, two cases may arise concerning the shortest path from s to v in G': (a) it coincides with the shortest path from v to s in G (whose weight is increased by ϵ); (b) it contains a boundary node. Otherwise, (c) the shortest path from s to v in G' contains a boundary node.

In any case, v needs to know which is the boundary node giving the shortest connection to s. To this aim, v performs the following local computation: for each $b \in B_s(y)$, it computes w_{min} as $\min_b\{weight(v, b) + \ell_b\}$ (line 2), and b_{min} as the boundary node such that $w_{min} = weight(v, b_{min}) + \ell_{b_{min}}$ (line 3). At this point, the algorithm behaves properly according to cases (a), (b), and (c) (see Figure 6).

4.4 Correctness and complexity

In this section we prove the correctness of the algorithm for *weight increase* operations and its message complexity. The proof of correctness is provided in terms of the correctness of each phase of the algorithm, namely *Coloring* (Lemmas 4.3 and 4.4), *Boundarization* (Lemma 4.6), *Recomputing* (Lemma 4.7).

Lemma 4.3 *The coloring phase is deadlock free.*

The red node v receives *start-recomputing*($via(v, y), s, B_s(y)$).

```
for each v_i ∈ RED-CHILDREN_s(v) do send start-recomputing(v, s, B_s(y)) to v_i
w_min := min{weight(v, b) + ℓ_b | ⟨b; ℓ_b⟩ ∈ B_s(y)}
let b_min be a node such that w_min = weight(v, b_min) + ℓ_{b_min}
weight(v, s) := weight(v, s) + ϵ
if BNR_s(v) = nil
  then                {v is not boundary}
    begin
      if weight(v, s) > w_min then
        begin
          weight(v, s) := w_min
          via(v, s) := via(v, b_min)
        end
    end
  else                {v is boundary}
    begin
      weight(v, s) := w_min
      via(v, s) := via(v, b_min)
    end
color(v, s) := white
```

Figure 6: The recomputing phase of node v

Proof. When a node v performs the coloring phase (see Figure 4), it waits for the end of the coloring phase of all its neighbors at line 18. According to Assumptions A1 and A2, during the execution of line 18, node v can answer immediately to requests of its neighbors asking for *weight*(v, s) and *color*(v, s) (coming from line 7) and to the color notification of some neighbor (coming from line 17).

Let us assume there is a deadlock, that is, there exists a cycle $\langle v_1, v_2, \ldots, v_k\rangle$, $v_k \equiv v_1$, in the network such that node v_i, $2 \le i \le k$, is waiting for the end of the coloring phase of node v_{i-1}. In this case, each node v_i in the cycle is red. In fact, if v_i is performing line 18, then it has previously executed line 16, where the color of v_i is changed to red.

Let us consider an arbitrary node v_i in the cycle. Since v_i is red, then either v_{i+1} is the parent of v_i in $T_s(y)$, or v_{i+1} is the last node in $\mathrm{AP}_s(v_i)$ that changed its color to red. In fact, if this is not the case, the color of v_i must be either white (see lines 2–11), or pink (see line 14). This implies that $d(v_i, y) = w(v_i, v_{i+1}) + d(v_{i+1}, y)$, and then $d(v_i, y) > d(v_{i+1}, y)$. Since the above inequality holds for every i, it follows that $d(v_i, y) > d(v_i, y)$, that is a contradiction. □

Note that, a node v computes its color only if and when it receives *start-coloring*($via(v, s), s$) (see line 3 in Figure 4). Therefore, by inspecting the algorithm of Figure 4, it is possible to check that the following claims hold:

$C1$: a node v is colored either red or pink if and only if $via(v, s)$ has been colored red (in particular each node $v \notin T_s(y)$ remains white);

$C2$: a node v that is pink at the end of the coloring phase has a nonempty set $\mathrm{AP}_s(v)$ of alternative parents that are nonred;

$C3$: if v is red at the end of the coloring phase then $\text{AP}_s(v) = \emptyset$.

It is important to observe that a pink node v sends the message *end-coloring*(v, s) to its parent if $\text{AP}_s(v) \neq \emptyset$, even if v may switch its color to red later. The change from pink to red occurs only if all the alternative parents of v have been colored red.

Lemma 4.4 *At the end of the coloring phase, the nodes in V have been correctly colored.*

Proof. The coloring starts when x and y detect the weight change on edge (x, y): x sends *start-coloring*(x, s) to y, and y sends *start-coloring*(y, s) to x. Since we are assuming that $d(x, s) < d(y, s)$, then node y correctly starts the coloring phase if and only if $via(y, s) = x$ (see line 3).

In the following we show that if the coloring phase starts, all nodes are correctly colored. The proof proceeds by induction as follows. We first prove that y is correctly colored; then we prove that an arbitrary node v is correctly colored, by assuming that all the nodes whose distance from s is smaller than v's distance are correctly colored.
Base case: y is correctly colored.
If $x = via(y, s)$, then y correctly computes $\text{AP}_s(y)$. If $\text{AP}_s(y) \neq \emptyset$ then *color*(y, s) is correctly computed as pink at line 14; otherwise, *color*(y, s) is correctly computed as red at line 16.
Inductive case: Let v be an arbitrary node colored during the coloring phase. Let us assume by inductive hypothesis that all the nodes z such that $weight(z, s) < weight(v, s)$ are correctly colored. One of the following cases arises:

- v is *white*: by claim $C1$ above $via(v, s) \neq red$; since by inductive hypothesis the color of $via(v, s)$ is correct, then the white color of v is correct.

- v is *pink*: by claim $C2$ above and by inductive hypothesis, there exists an alternative parent z of v that has been correctly colored nonred; hence, the pink color of v is correct;

- v is *red*: by claim $C3$ above $\text{AP}_s(v)$ is empty; this implies that all the alternative parents of v have been colored red. By inductive hypothesis the color of these nodes is correct, and hence the red color of v is correct.

□

Lemma 4.5 *The boundarization and recomputing phases are deadlock free.*

Proof. For each red node v, let $RE(v) = \{(v, z) \mid z \in \text{RED-CHILDREN}(v, s)\}$ and let $PE(v) = \{(v, z) \mid z \in \text{PINK-CHILDREN}(v, s)\}$. As consequence of claim $C1$ and by construction of RED-CHILDREN and PINK-CHILDREN (see Figure 5 at lines 10, 11, 21, and 23), the set $RPE = \cup_{v \,:\, v \text{ is } red \text{ or } pink}(RE(v) \cup PE(v))$ induces a tree rooted at y. With the only exception in line 14 of Figure 5 (but in this case the request is immediately answered), all the communications among nodes are done along the edges of this tree (see Figure 5 at lines 28, 31, and 34, and Figure 6 at line 1), hence no circular waiting is possible and then no deadlock can arise. □

Lemma 4.6 *The boundarization phase is correct.*

Proof. First of all, we have to prove that for each pink node v, $via(v,s)$ is correctly computed. Each pink node receives the message *start-boundarization* propagated from y to all the nodes belonging to the tree induced by set RPE defined in Lemma 4.5 (see Figure 5, line 28). Then each pink node v perform lines 2–5 and correctly set $color(v,s)$ and $via(v,s)$ by choosing a node in $\textsc{ap}_s(v)$, which is not empty (see Claim $C2$).

To complete the proof we have to show that the set $B_s(y)$ is correctly computed, that is, it contains all the boundary nodes in $T_s(y)$. By contradiction, let us suppose the information of a boundary node b is not in $B_s(y)$. As the pink nodes, all the red ones receive the message *start-boundarization*. As a consequence, b surely starts the boundarization phase. Since b is boundary, $\textsc{bnr}_s(b)$ exists and the condition at line 25 is false, then line 27 is performed. Hence $B_s(b)$ is correctly initialized with the information related to b. Then b enlarges $B_s(b)$ by composing the information received from its red children (see lines 30–33). At the end, in line 34, b sends the updated set $B_s(b)$ to its parent $via(b,s)$ in $T_s(y)$. Since all the other red nodes in the path from b to y perform the same statement, the information about b reaches y and is added to $B_s(y)$ (line 32), which is a contradiction. □

Lemma 4.7 *For each red node v, the recomputing phase correcly computes $weight'(v,s)$ and $via'(v,s)$.*

Proof. According to line 1, each red node in $T_s(y)$ receives the information contained in the set $B_s(y)$. We now show that, once node v has received $B_s(y)$, it can correcly computes $weight'(v,s)$ and $via'(v,s)$ by local computation.

To this aim, we first show that if v_1 and v_2 are two red nodes, then $weight'(v_1,v_2) = weight(v_1,v_2)$. Let p be the shortest path in G between v_1 and v_2. If $weight'(v_1,v_2) \neq weight(v_1,v_2)$ then p must necessarily contain edge (x,y). In this case, either v_1 or v_2 does not belong to $T_s(y)$, that contradicts fact F1 of Remark 4.1. This property implies that, by using the current shortest path information on G, node v can determine which is the boundary node b_{min} closest to s, and the distance from v to b_{min} in G'.

According to Lemma 4.2, the shortest path p' from v to s in G' either coincides with the old one (but $weight(p')$ is increased by ϵ), or contains b_{min}. If the second case occurs, $weight(p')$ is given by $weight(v,b_{min}) + \ell_{b_{min}}$ ($\ell_{b_{min}}$ is contained in $B_s(y)$). Accordingly, v can compute $via'(v,s)$. □

Theorem 4.8 *Updating all-pairs shortest paths on a distributed network with n nodes and positive real edge weights, after a weight increase operation, requires $O(\max\{|\delta_\sigma|, maxdeg\}\cdot\Delta_\sigma)$ messages, $O(\max\{|\delta_\sigma|, maxdeg\}\cdot\Delta_\sigma)$ time, and $O(n)$ space per node.*

Proof. For each node v, storing $color(v,s)$, $\textsc{pink-children}_s(v)$, and $\textsc{red-children}_s(v)$, for the various sources, require $O(n)$ space. This is possible because $\textsc{pink-children}_s(v)$ and $\textsc{red-children}_s(v)$ are not permanent data structures. As soon as the computation related to a source s has been terminated, and the one for another source s' starts, for each node v, the children of v with respect to s' are stored in place of the children of v with respoect to s. We evaluate the message complexity of the algorithm phase by phase.

Coloring: We first compute the number of messages sent and received by a single node v during the execution of the algorithm of Figure 4, and then we sum up over all possible red and pink nodes.

The worst case, in terms of message complexity, occurs when a white node v that starts the coloring phase receiving the message *start-coloring*(z,s), fulfills the following conditions: *i)* each node in $N(v) \setminus \{z\}$ belongs to $\mathrm{AP}_s(v)$; *ii)* each node q in $\mathrm{AP}_s(v)$ eventually sends to v the message *start-coloring*(q,s). In this case, v sends (and receives) $deg(v)-1$ messages (lines 6–10) to compute $\mathrm{AP}_s(v)$; v sends $deg(v)-1$ messages (line 20) to notify to each node in $\mathrm{AP}_s(v)$ the end of its coloring phase; v sends $deg(v)-1$ messages (line 17) to propagate the coloring phase; v receives $deg(v)-1$ messages (line 19) notifying the end of the coloring phase of the neighbors of v. Since the above discussion applies to every node in $\delta_{\sigma,s}$, then the total number of messages sent over the network during the coloring phase is $O(maxdeg \cdot |\delta_{\sigma,s}|)$.

Boundarization: During the execution of the algorithm of Figure 5, in order to compute $\mathrm{BNR}_s(v)$, node v sends (and receives) $deg(v)-1$ messages in line 14. After that this computation has been performed, v sends at most $deg(v)-1$ messages to propagate the boundarization phase (line 28). In addition, v receives and sends at most $|\delta_{\sigma,s}|$ messages to collect the information on the boundary nodes into $B_s(v)$ (line 31), and to send $B_s(v)$ to $via(v,s)$ (line 34), respectively. Since the number of red nodes with respect to s is at most $|\delta_{\sigma,s}|$, then the total number of messages of constant size sent over the network during the boundarization phase is $O(maxdeg \cdot |\delta_{\sigma,s}| + |\delta_{\sigma,s}|^2)$.

Recomputing: During this phase, node y broadcasts the set $B_s(y)$ to the red nodes through the edges of the portion of $T_s(y)$ spanning the red nodes. Since $B_s(y)$ has size $|\delta_{\sigma,s}|$, and it is sent over at most $|\delta_{\sigma,s}|$ edges, then then the total number of messages of constant size sent during the recomputing phase is $O(|\delta_{\sigma,s}|^2)$.

The total message complexity of the algorithm is $O(maxdeg \cdot |\delta_{\sigma,s}| + |\delta_{\sigma,s}|^2)$, that is, the sum of the message complexities of the three phases. In order to evaluate the message complexity of the proposed algorithm in the case of all-pairs shortest paths, it is sufficient to sum up the message complexity of the three phases over all possible sources in δ_σ. This gives the following bound:

$$\sum_{s\in\delta_\sigma} O(maxdeg \cdot |\delta_{\sigma,s}| + |\delta_{\sigma,s}|^2) = O(maxdeg \cdot \Delta_\sigma + |\delta_\sigma| \cdot \Delta_\sigma)$$

The time complexity of the algorithm is bounded by the above message complexity. □

5 Concluding remarks

We have proposed a fully-dynamic distributed algorithm for the *all-pairs* shortest paths problem on general networks with positive real edge weights. If Δ_σ is the number of pairs of nodes changing the distance after a single edge modification σ (*insert*, *delete*, *weight decrease*, or *weight increase*) then the message complexity of the proposed algorithm is $O(n\Delta_\sigma)$ in the worst case. If $\Delta_\sigma = o(n^2)$, this is better than recomputing everything from scratch after each edge modification.

A problem harder than the one considered in this paper, having strong impact in practical applications, is the problem of updating shortest paths when multiple edge changes occur simultaneously in the network. Several solutions of this problem rely on the classical Ford-Bellman method, originally introduced in the Arpanet [10]. For example, in [8] Humblet proposes a different solution, based on Dijkstra's algorithm for shortest paths, that overcomes some drawbacks of previous protocols. An interesting further research is to apply the new ideas proposed in this paper, both from an algorithmic and a computational complexity point of view, to this more difficult problem.

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. 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] B. Awerbuch, I. Cidon and S. Kutten. Communication-optimal maintenance of replicated information. *Proc. IEEE Symposium on Foundations of Computer Science*, 492–502, 1990.

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

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

[7] S. Haldar. An all pair shortest paths distributed algorithm using $2n^2$ messages. *Journal of Algorithms*, **24** (1997), 20–36.

[8] P. Humblet. Another adaptive distributed shortest path algorithm. *IEEE transactions on communications*, **39**, n. 6 (1991), 995–1003.

[9] G. F. Italiano. Distributed algorithms for updating shortest paths. *Proceedings Int. Workshop on Distributed Algorithms. Lecture Notes in Computer Science* 579, 200–211, 1991.

[10] J. McQuillan. Adaptive routing algorithms for distributed computer networks. BBN Rep. 2831, Bolt, Beranek and Newman, Inc., Cambridge, MA 1974.

[11] J. T. Moy. *OSPF - Anatomy of an Internet Routing Protocol.* Addison-Wesley, 1998.

[12] A. Orda and R. Rom. Distributed shortest-path and minimum-delay protocols in networks with time-dependent edge-length. *Distributed Computing*, **10** (1996), 49-62.

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

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

[15] K. V. S. Ramarao and S. Venkatesan. On finding and updating shortest paths distributively. *Journal of Algorithms*, **13** (1992), 235–257.

[16] A.S. Tanenbaum. Computer networks. Prentice Hall, Englewood Cliffs, NJ (1996).