Partially Dynamic Algorithms for Distributed Shortest Paths and their Experimental Evaluation

Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, Daniele Frigioni, Alberto Petricola

Dipartimento di Ingegneria Elettrica e dell'Informazione
Università dell'Aquila, I-67040 Monteluco di Roio, L'Aquila, Italy
E-mail: {cicerone,gdangelo,gabriele,frigioni,petricol}@ing.univaq.it

***Abstract*— In this paper, we study the dynamic version of the *distributed all-pairs shortest paths* problem. Most of the solutions given in the literature for this problem, either (i) work under the assumption that before dealing with an edge operation, the algorithm for the previous operation has to be terminated, that is, they are not able to update shortest paths *concurrently*, or (ii) concurrently update shortest paths, but their convergence can be very slow (possibly infinite). In this paper we propose a partially dynamic algorithm that overcomes most of these limitations. In particular, it is able to concurrently update shortest paths and in many cases its convergence is quite fast. These properties are highlighted by an experimental study whose aim is to show the effectiveness of the proposed algorithms also in the practical case.**

***Index Terms*— Distributed networks, dynamic algorithms, shortest paths, routing, experimental evaluation, network simulation environment**

I. Introduction

We consider the *distributed all-pairs shortest paths* problem in a network whose topology dynamically changes over the time, in the sense that communication links can change status during the lifetime of the network. This problem arises naturally in practical applications. For instance, the *OSPF* protocol, widely used in the Internet (e.g., see [2]), basically updates shortest paths after a network change by distributing the network topology to all processors and using centralized Dijkstra's algorithm for shortest paths on every node.

If the topology of a network is represented as a weighted graph, where nodes represent processors, edges represent links between processors, and edge weights represent costs of communication among processors, then the typical update operations on a dynamic network can be modelled as insertions and deletions of edges and edge weight changes. 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. Incremental and decremental problems are usually called *partially dynamic*.

In many crucial routing applications the worst case complexity of the adopted protocols is never better than recomputing the shortest paths from scratch after each change to the network. Therefore, it is important to find efficient dynamic distributed algorithms for shortest paths, since the recomputation from scratch could result very expensive in practice. The efficiency of a distributed algorithm is evaluated in terms of *message* and *space* complexity (e.g., see [3]). The *message complexity* is the total number of messages sent over the edges. The *space complexity* is the space usage per node.

In this paper we consider a dynamic network in which a change can occur while another change is under processing. A processor v could be affected by both these changes. As a consequence, v could be involved in the *concurrent* executions related to both the changes.

Previous works. Given a weighted graph G with n nodes and m edges, 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. The state of the art is that no efficient fully dynamic solution is known for general graphs that is faster than recomputing single-source shortest paths from scratch after each update. Actually, only *output bounded* fully dynamic solutions are known on general graphs [4], [5]. In the case of all-pairs shortest paths an efficient fully dynamic solution has been proposed in [6] that works in $O(n^2 \log^3 n)$ amortized time per update.

A number of solutions have been proposed in the literature also for the dynamic distributed shortest paths problem (see [7]–[12]). Some of these solutions rely on the classical Bellman-Ford method, whose distributed version has been originally introduced in the Arpanet [13]. This algorithm, and a number of its variations, has been shown to converge to the correct distances if the edge weights stabilize and all cycles have positive lengths (e.g., see [14]). However, the convergence can be very slow in the case of *weight increase* operations (possibly infinite), due to the well-known *looping* and *count-to-infinity* phenomena (see, e.g., [15]). This is a major

This paper is based on "Partially Dynamic Concurrent Update of Distributed Shortest Paths" by Gianlorenzo D'Angelo, Serafino Cicerone, Gabriele Di Stefano, Daniele Frigioni, which appeared in the Proceedings of the IEEE International Conference on Computing: Theory and Applications, March 2007, Kolkata, India. © 2007 IEEE [1].

Work partially supported by the Future and Emerging Technologies Unit of EC (IST priority, 6th FP), under contract no. FP6-021235-2 (project ARRIVAL).

drawback of the Bellman-Ford algorithm and its variations that is avoided in many protocols by broadcasting the whole topology of the network to all nodes [2], [16]. Furthermore, if the network is asynchronous and static, the message complexity of the Bellman-Ford method can be exponential in the size of the network (see, [17]). In [9] Humblet proposes a variation of the Bellman-Ford algorithm, that has the same message and space complexity, and, under certain conditions, avoids the looping phenomenon thus converging in a finite number of steps.

In [10], an efficient incremental solution has been proposed for the distributed all-pairs shortest paths problem, requiring $O(n \log(nW))$ amortized number of messages, and the difficulty of dealing with edge deletions has been addressed. Here, W is the largest positive *integer* edge weight. In [7], a general technique is proposed that allows to update the all-pairs shortest paths in a distributed network in $\Theta(n)$ amortized number of messages, by using $O(n^2)$ space per node. In [12], algorithms are given for both finding and updating shortest paths distributively. In particular, the authors propose a distributed algorithm for finding single source shortest paths (all pairs shortest paths) of a network with positive real edge weights requiring $\Theta(n^2)$ ($O(n^3)$) messages and $O(n)$ space per node. Furthermore, they propose a distributed incremental algorithm requiring $O(n^2)$ messages for updating all-pairs shortest paths. Finally, they give fully dynamic algorithms for single-source (all-pairs) shortest paths that work in $O(n^2)$ ($O(n^3)$) messages, and show that, in the worst case, updating shortest paths is as difficult as computing shortest paths.

In [8] a solution for the fully dynamic distributed all-pairs shortest paths problem is presented whose message complexity is evaluated in terms of *output complexity* (see [4], [5]). 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 change. The algorithm in [8] is able to update only the distances and the shortest paths that actually change after an edge modification σ. It requires in the worst case $O(\mathit{maxdeg} \cdot \Delta_\sigma)$ messages per edge update operation. The space complexity is $O(n)$ per node. Here, *maxdeg* is the maximum degree of the nodes in the network and Δ_σ is the number of pairs of nodes affected by σ. On one hand, if $\Delta_\sigma = o(n^2)$, then these bounds compare favorably with respect to those in [12]. On the other hand, the algorithm is not robust, in fact for *weight increase* operations it works in three phases and requires that a phase is terminated before the execution of the subsequent one, while in the case of *weight decrease* operations it works under the assumption that before dealing with an edge operation, the algorithm for the previous operation has to be terminated.

Summarizing, we can conclude that most of the algorithms of the literature for the dynamic distributed shortest paths problem fall in one of the following categories:

- algorithms which are not able to *concurrently* update shortest paths when multiple edge changes occur in the network, as those in [7], [8], [10], [12]. In particular, algorithms that work under the assumption that before dealing with an edge operation, the algorithm for the previous operation has to be terminated. This is a limitation in real networks, where changes can occur in an unpredictable way;
- algorithms which are able to *concurrently* update shortest paths as those in [9], [13], but *(i)* either they suffer of the looping and count-to-infinity phenomena, or *(ii)* their convergence can be very slow in the case of *weight increase* operations (possibly infinite).

Results of the paper. In this paper we provide *partially dynamic* solutions that do not belong to any of the previous categories. In particular, our algorithms are able to concurrently update shortest paths, they avoid the looping and count-to-infinity phenomena and their convergence is fast in the case of *weight increase* operations. The details of this contribution can be summarized as follows:

1) We propose a new decremental algorithm that is robust since it works in one phase (thus avoiding the main drawback of [8]). Furthermore, it is able to *concurrently* update shortest paths in the case of multiple *weight increase*/*delete* operations. The algorithm requires $O(\mathit{maxdeg} \cdot \Delta^2)$ messages and $O(\mathit{maxdeg} \cdot n)$ space per node. Here, Δ is the number of nodes affected by a set of *weight increase*/*delete* operations.
2) We propose an extension of the incremental algorithm in [8] for *weight decrease*/*insert* operations that works also in the *concurrent* case, within the same bounds of [8], that is $O(\mathit{maxdeg} \cdot \Delta)$ messages per operation and $O(n)$ space per node. Here, Δ is the number of nodes affected by a set of *weight decrease*/*insert* operations. This is only a factor *maxdeg* far from the optimal incremental solution.
3) We propose an experimental study whose aim is to highlight the merits of the proposed algorithms also from a practical point of view. In detail, we experimentally show that our incremental algorithm sends a number of messages that is 10%–15% less than Bellman-Ford.
   In the decremental case, we compare our algorithm with two variants of the Bellman-Ford method: BF.1, that stores in each node the estimated distances of its neighbors, and BF.2 that does not store such information. Our decremental algorithm sends a number of messages that is 10–50 times less than BF.2, while using the same space occupancy per node. BF.1 outperforms most of the times our decremental algorithm in terms of number of messages; however, BF.1 requires a space occupancy per node which is 20–85 times our space occupancy. Moreover, we experimentally show that our decremental solution does not suffers of the looping and count-

to-infinity phenomena in some classical cases where BF.1 and BF.2 do.

Structure of the paper. The paper is organized as follows. In Section II we introduce the notation and the computation model used throughout the paper. In Sections III and IV we describe the algorithms for weight increase and weight decrease operations, respectively, and show their complexity in terms of number of messages. In Section V we describe the experiments performed to check the effectiveness of our algorithms in the practical case. Finally, in Section VI we provide some concluding remarks and future research directions.

II. PRELIMINARIES

We consider a network made of processors linked through communication channels. Each processor can send messages only to its neighbors. Messages are delivered to their destination within a finite delay but they might be delivered out of order. We consider an asynchronous system, that is, a sender of a message does not wait for the receiver to be ready to receive the message. There is no shared memory, that is, each processor has its own storage system and the other processors cannot access it.

We represent the network by an undirected weighted graph $G = (V, E, w)$, where V is a finite set of n nodes, one for each processor; E is a finite set of m edges, one for each communication channel; and w is a weight function $w : E \to \mathbb{R}^+$. An edge $e \in E$ that links the pair of nodes $u, v \in V$ is represented with $u \to v$. If $v \in V$, $N(v)$ denotes the set of neighbors of v and $\deg(v)$ the degree of v.

We define the *weight* of a path P as the sum of the weights of the edges in P. The *distance* between nodes u and v is the weight of a shortest path from u to v, and is denoted as $d(u, v)$. Given $u, v \in V$, the *via* from u to v is the set of neighbors of u that belong to a shortest path from u to v. Formally,

$$\mathit{via}(u, v) \equiv \{z \in N(u) \mid d(u, v) = w(u, z) + d(z, v)\}$$

Complexity measures. Given a weighted undirected graph G, a set of k weight changes $\sigma_1, \sigma_2, ..., \sigma_k$ and a source s, we denote as $\delta_{\sigma_i,s}$ the set of nodes that change the distance to s as a consequence of σ_i. If $v \in \bigcup_{s \in V} \delta_{\sigma_i,s}$ we say that v is *affected* by σ_i. The total number of times that nodes of G are affected by the k weight changes is at most $\Delta = \sum_{i=1}^{k} \sum_{s \in V} |\delta_{\sigma_i,s}|$. We give the complexity bounds of our algorithms as a function of Δ.

Asynchronous model. Given an asynchronous system, the model summarized below is based on that proposed in [3]. The *state* of a processor v is the content of the data structure at node v. The *network state* is the set of states of all the processors in the network plus the network topology. An *event* is the reception of a message by a processor or a change to the network state. When a processor p sends a message m to a processor q, m is stored in a buffer in q. When q reads m from its buffer and processes it, the event "reception of m" occurs.

An *execution* is an alternate sequence (possibly infinite) of network states and events. A non negative real number is associated to each event, the *time* at which that event occurs. The time is a *global* parameter and is not accessible to the processors of the network. The times must be non decreasing and must increase without bound if the execution is infinite. Events are ordered according to the times at which they occur. Several events can happen at the same time as long as they do not occur at the same processor. This implies that the times related to a single processor are strictly increasing.

Concurrent executions. In this paper we consider a dynamic network in which a change can occur while another change is under processing. A processor v could be affected by both these changes. As a consequence, v could be involved in the executions related to both the changes. Hence, according to the asynchronous model we need to define the notion of *concurrent* executions as follows.

Let us consider an algorithm A that maintains shortest paths on G after a weight change operation. Given two operations σ_i and σ_j we denote as:

- t_i and t_j the times at which σ_i and σ_j occur respectively.
- $\mathcal{A}_i$ ($\mathcal{A}_j$) the execution of A related to σ_i (σ_j).
- $t_{\mathcal{A}_i}$ the time when $\mathcal{A}_i$ terminates.

If $t_i \leq t_j$ and $t_{\mathcal{A}_i} \geq t_j$, then $\mathcal{A}_i$ and $\mathcal{A}_j$ are *concurrent*, otherwise they are *sequential*.

III. DECREMENTAL ALGORITHM

In this Section we describe our new decremental solution for the concurrent update of distributed all-pairs shortest paths in the case of multiple operations. We consider the algorithm to handle *weight increase* operations, since the extension to *delete* operations is straightforward (deleting an edge (x, y) is equivalent to increase $w(x, y)$ to $+\infty$).

Given the input graph $G = (V, E, w)$, we suppose that k *weight increase* operations $\sigma_1, \sigma_2, ..., \sigma_k$ are performed on edges $(x_i, y_i) \in E$, $i \in \{1, 2, ..., k\}$, at times $t_1, t_2, ..., t_k$, respectively. The operation σ_i increases the weight $w(x_i, y_i)$ by a quantity $\epsilon_i > 0$, $i \in \{1, 2, ..., k\}$. Without loss of generality, we assume that $t_1 \leq t_2 \leq ... \leq t_k$. We denote as G' the graph after t_k, as $d'()$ and $\mathit{via}'()$ the distance and the via over G', respectively.

Data structures. A node knows the identity of each node of the graph, the identity of all its neighbors and the weight of the edges incident to it. The information on the shortest paths in G are stored in a data structure called *routing table* RT distributed over all nodes. Each node v maintains its own routing table $\mathtt{RT}_v[\cdot]$; this table has one entry $\mathtt{RT}_v[s]$ for each $s \in V$. The entry $\mathtt{RT}_v[s]$ consists of two fields:

- $\mathtt{RT}_v[s].\mathtt{d}$ that stores the estimated distance between nodes v and s in G.

- $\texttt{RT}_v[s].\texttt{via} = \{v_i \in N(v) \mid \texttt{RT}_v[s].\texttt{d} = w(v, v_i) + \texttt{RT}_{v_i}[s].\texttt{d}\}$ that stores the estimated *via* from v to s.

For sake of simplicity, we write $\texttt{d}[v, s]$ and $\texttt{via}[v, s]$ instead of $\texttt{RT}_v[s].\texttt{d}$ and $\texttt{RT}_v[s].\texttt{via}$, respectively.

The values stored in the routing table of each node change over the time during the execution of the update algorithms. Hence, we denote as $\texttt{d}_t[v, s]$ and $\texttt{via}_t[v, s]$ the value of the data structures at time t; we simply write $\texttt{d}[v, s]$ and $\texttt{via}[v, s]$ when time is clear by the context.

Algorithm. Before the decremental algorithm starts, we assume that $\texttt{d}_t[v, s]$ and $\texttt{via}_t[v, s]$ are correct, for each $v, s \in V$ and for each $t < t_1$.

The decremental algorithm starts at each t_i, $i \in \{1, 2, ..., k\}$. For instance, the *weight increase* operation σ_i represents an event that is detected only by nodes x_i and y_i; as a consequence:

- x_i sends the message *increase*$(x_i, s, \texttt{d}_{t_i}[x_i, s])$ to y_i, for each $s \in V$;
- y_i sends the message *increase*$(y_i, s, \texttt{d}_{t_i}[y_i, s])$ to x_i, for each $s \in V$.

When, at a certain time t, node v receives the message *increase*$(u, s, \texttt{d}_{\tilde{t}}[u, s])$, $\tilde{t} < t$, by a generic node u, v executes procedure INCREASE (see Figure 1), which is designed to update $\texttt{RT}_v[s]$, if necessary. To this aim, v may need to know the estimated distances of its neighbors from s, that is, $\texttt{d}_t[v_i, s]$ for each $v_i \in N(v)$. Hence, v sends messages *get-dist*(v, s); when v_i receives such message, it performs procedure DIST (see Figure 2).

Event: node v receives the message *increase*$(u, s, \texttt{d}[u, s])$ by u

Procedure INCREASE

1. **if** $u \in \texttt{via}[v, s]$ **then**
2. **begin**
3. $\texttt{via}[v, s] := \texttt{via}[v, s] \setminus \{u\}$ Line 3: phase REDUCE-VIA
4. **if** $\texttt{via}[v, s] \equiv \emptyset$ **then**
5. **begin** Lines 5-11: phase BUILD-TABLE
6. **for each** $v_i \in N(v)$ **do** send *get-dist*(v, s) to v_i
7. $\texttt{d}[v, s] := \min_{v_i \in N(v)} \{w(v, v_i) + \texttt{d}[v_i, s]\}$
8. $\texttt{via}[v, s] := \{v_i \in N(v) | w(v, v_i) + \texttt{d}[v_i, s] = \texttt{d}[v, s]\}$
9. **for each** $v_i \in N(v)$ **do** Lines 9-10: phase PROPAGATE_1
10. send *increase*$(v, s, \texttt{d}[v, s])$ to v_i
11. **end**
12. **end**
13. **else**
14. **if** $\texttt{d}[v, s] > w(v, u) + \texttt{d}[u, s]$ **then**
15. **begin** Lines 15-20: phase IMPROVE-TABLE
16. $\texttt{d}[v, s] := w(v, u) + \texttt{d}[u, s]$
17. $\texttt{via}[v, s] := \{u\}$
18. **for each** $v_i \in N(v)$ **do** Lines 18-19: phase PROPAGATE_2
19. send *increase*$(v, s, \texttt{d}[v, s])$ to v_i
20. **end**
21. **else**
22. **if** $\texttt{d}[v, s] = w(v, u) + \texttt{d}[u, s]$ **then**
23. $\texttt{via}[v, s] := \texttt{via}[v, s] \cup \{u\}$ Line 23: phase EXTEND-VIA

Figure 1. The INCREASE algorithm

Notice that, in our model, multiple *increase* messages received by a node are stored and processed in an arbitrary order, while each message *get-dist* is processed immediately.

Event: node v receives the message *get-dist*(u, s) by u

Procedure DIST

1. **if** $(\texttt{via}[v, s] \equiv \{u\})$ **or** (v is performing phase BUILD-TABLE or phase IMPROVE-TABLE of procedure INCREASE with respect to source s)
2. **then** send $+\infty$ to u
3. **else** send $\texttt{d}[v, s]$ to u

Figure 2. Procedure DIST performed by a node v when it receives a *get-dist* message

Now we provide an informal description of the algorithm. The purpose of this description is to give an intuition of both the behavior and correctness of the algorithm (the formal correctness proof is given in [18]). The description is focused on the execution of the algorithm by a generic node v with respect to a source s, and uses the scenario for node v depicted in Figure 3 as a representative case.

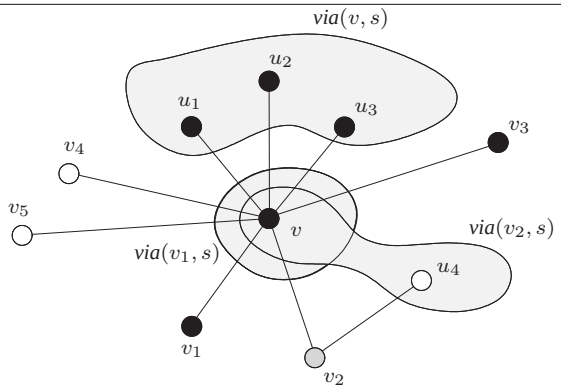

Figure 3. A representative scenario

In such a figure nodes are colored *white*, *gray* and *black*. These colors are assigned according to the following definitions.

- a node v is *white* with respect to s if v does not change both its distance and its via to s. Formally: $d(v, s) = d'(v, s)$ and *via*$(v, s) \equiv$ *via*$'(v, s)$.
- a node v is *gray* with respect to s if v does not change its distance from s, but it changes its via to s. Formally: $d(v, s) = d'(v, s)$, and *via*$(v, s) \not\equiv$ *via*$'(v, s)$. Notice that, in this case, *via*$(v, s) \supsetneq$ *via*$'(v, s)$.
- a node v is *black* with respect to s if v changes its distance from s. Formally: $d(v, s) \neq d'(v, s)$. Notice that, in this case, $d(v, s) < d'(v, s)$.

The following properties trivially hold:

P1: If v is *gray* or *black* with respect to s, then there exists $u \in$ *via*(v, s) which is *black* with respect to s. If v is *black* with respect to s, then all nodes in *via*(v, s) are *black* with respect to s.

P2: If v is *white* or *gray* with respect to s, then, each node z such that $v \in$ *via*(z, s) is *white* with respect to s.

Node v in Figure 3 is *black* with respect to s since, according to property P1, all nodes in *via*$(v, s) \equiv \{u_1, u_2, u_3\}$

are *black*. As a consequence, v surely receives messages *increase*$(u_i, s, \mathtt{d}[u_i, s])$, $1 \leq i \leq 3$, in some order. This implies that v performs three times procedure INCREASE. The first two executions simply perform phase REDUCE-VIA, while the third one performs REDUCE-VIA and BUILD-TABLE.

Let us suppose that the third execution is related to u_3. During the execution of BUILD-TABLE, node v sends the message *get-dist*(v, s) to each node $v_i \in N(v) \setminus \{u_3\}$. We assume that this message is received by v_i at time $\tilde{t}_{1,i}$. In this phase, let us assume that the following conditions hold for nodes v_1 and v_3, respectively:

(a) $\mathtt{via}_{\tilde{t}_{1,1}}[v_1, s] \equiv \{v\}$
(b) at time $\tilde{t}_{1,3}$, node v_3 is performing either BUILD-TABLE or IMPROVE-TABLE phases of procedure INCREASE with respect to source s

According to these conditions and to test at line 1 of procedure DIST, nodes v_3 and v_1 send $+\infty$ to v.

By using the collected information, v performs the instructions $\mathtt{d}[v, s] := \min_{v_i \in N(v)} \{w(v, v_i) + \mathtt{d}[v_i, s]\}$ and $\mathtt{via}[v, s] := \{v_i \in N(v) \mid w(v, v_i) + \mathtt{d}[v_i, s] = \mathtt{d}[v, s]\}$. Let us assume that now $\mathtt{via}_{\tilde{t}_2}[v, s] = \{v_2\}$. Notice that, since v has received partial information, the content of $\mathtt{RT}_v[s]$ at time $\tilde{t}_2$ could be not correct. Now, two relevant observations have to be remarked:

(i) since nodes v_1 and v_3 sent $+\infty$ to v, then v does not consider such nodes as possible new elements of via; this is done to prevent the looping and count-to-infinity phenomena.
(ii) in the subsequent Items 1 and 2 we show that nodes v_1 and v_3 will eventually send $\mathtt{d}[v_1, s]$ and $\mathtt{d}[v_3, s]$ to v.

The BUILD-TABLE phase of v is completed by the PROPAGATE_1 phase. In this phase v broadcast to $N(v)$ the message *increase*$(v, s, \mathtt{d}_{\tilde{t}_2}[v, s])$; it may seem useless to send the message to nodes u_i, $1 \leq i \leq 3$, (the old via of v) and to node v_2 (the new via of v). The former will be explained later (last paragraph of Item 1), while the latter is due to the fact that $v \in$ *via*(v_2, s), and hence v_2 has to perform the REDUCE-VIA phase.

Let us now analyze what happens to the nodes v_1, v_3, v_4 and v_5.

1. node v_1 receives message *increase*$(v, s, \mathtt{d}_{\tilde{t}_2}[v, s])$ at time $\tilde{t}_3 > \tilde{t}_2$, and it executes INCREASE. Since $\mathtt{via}_{\tilde{t}_{1,1}}[v_1, s] \equiv \mathtt{via}_{\tilde{t}_3}[v_1, s] \equiv \{v\}$, v_1 performs the BUILD-TABLE phase. At the end of this phase, at time $\tilde{t}_4 > \tilde{t}_3$, v_1 updates $\mathtt{RT}_{v_1}[s]$. Now, two major cases may occur:
   - v is in $\mathtt{via}_{\tilde{t}_4}[v_1, s]$;
   - v is not in $\mathtt{via}_{\tilde{t}_4}[v_1, s]$. This means that v_1 now uses a new via to s.

   In both cases, at the end of the BUILD-TABLE phase, v_1 broadcast the message *increase*$(v_1, s, \mathtt{d}_{\tilde{t}_4}[v_1, s])$ to $N(v_1)$, and hence to v also (with reference to Item (ii) above).
   In the first case, v performs tests at lines 1, 14 and 22 of INCREASE. All such tests return false, and hence, node v terminates INCREASE without modifying its routing tables and without propagating the decremental algorithm.
   In the second case, one of the tests performed by v at lines 14 and 22 may return true. If test at line 14 returns true, then v has to perform the IMPROVE-TABLE phase to rebuild $\mathtt{RT}_v[s]$. If test at line 22 returns true, then v has to perform the EXTEND-VIA phase to add v_1 to $\mathtt{via}[v, s]$.
   Notice that the behavior of v after receiving message *increase*$(v_1, s, \mathtt{d}_{\tilde{t}_4}[v_1, s])$ is essentially the same of nodes u_i, $1 \leq i \leq 3$, after receiving message *increase*$(v, s, \mathtt{d}_{\tilde{t}_2}[v, s])$.
2. node v_3, once terminated the execution of phase BUILD-TABLE or phase IMPROVE-TABLE of procedure INCREASE with respect to source s (see item (b) above), executes phase PROPAGATE_1 or phase PROPAGATE_2. This implies that node v restarts INCREASE now using the current estimated distance from v_3 to s (with reference to Item (ii) above).
3. since nodes v_4 and v_5 are *white*, once received message *increase*$(v, s, \mathtt{d}_{\tilde{t}_2}[v, s])$ they perform tests at lines 1, 14 and 22 of procedure INCREASE. All such tests return false, and hence according to property P2, nodes v_4 and v_5 terminate INCREASE without modifying their routing tables and without propagating the decremental algorithm.

The correctness of the decremental algorithm is given in [18]. The complexity bounds of the algorithm in the absence of looping are stated in the next theorem.

Theorem 1: The concurrent update of all-pairs shortest paths over a graph G with n nodes and positive real edges weights, after a set of *weight increase* operations, requires $O\left(\textit{maxdeg} \cdot \Delta^2\right)$ messages and $O\left(\textit{maxdeg} \cdot n\right)$ space per node.

Proof: Only *black* nodes send messages with respect to a source s. Given a source s and a *weight increase* operation σ_i, a *black* node v with respect to s can update the value of $\mathtt{d}[v, s]$ at most $|\delta_{\sigma_i, s}|$ times. Each time that v updates $\mathtt{d}[v, s]$, it sends $\deg(v)$ messages, then at most it sends *maxdeg* $\cdot\, |\delta_{\sigma_i, s}|$ messages. Since there are at most $|\delta_{\sigma_i, s}|$ nodes that are *black* with respect to s as a consequence of σ_i, the number of messages related to source s sent as a consequence of operation σ_i is *maxdeg* $\cdot\, |\delta_{\sigma_i, s}|^2$. The sum of this value over all sources $s \in V$ and *weight increase* operations σ_i, $i \in \{1, 2, ..., k\}$ is:

$$\sum_{i=1}^{k} \sum_{s \in V} \left(\textit{maxdeg} \cdot |\delta_{\sigma_i, s}|^2\right) \leq \textit{maxdeg} \cdot \Delta^2$$

Thus, the message complexity is $O\left(\textit{maxdeg} \cdot \Delta^2\right)$.

Each node stores only its routing table. Given a node v and a source s the set $\mathtt{via}[v, s]$ contains at most $\deg(v)$ elements. Hence, each node v requires $O\left(n \cdot \deg(v)\right)$ space and the space complexity is $O\left(\textit{maxdeg} \cdot n\right)$. ■

IV. INCREMENTAL ALGORITHM

In this Section we describe a new incremental algorithm for the concurrent update of distributed all-pairs shortest paths in the case of multiple operations. This algorithm is an extension of the incremental solution proposed in [8] that has been shown to work only in the sequential case. Our solution works correctly also in the concurrent case and differs from that in [8] in how the algorithm starts and in the message delivering policy. In particular, we force the messages between two neighbors to be delivered in a FIFO order. We consider only *weight decrease* operations, since the extension to *insert* operations is straightforward (inserting edge $x \rightarrow y$ with weight w is equivalent to decrease $w(x, y)$ from $+\infty$ to w).

Given the input graph $G = (V, E, w)$, we suppose that k *weight decrease* operations $\sigma_1, \sigma_2, ..., \sigma_k$ are performed on edges $x_i \rightarrow y_i \in E$, $i \in \{1, 2, ..., k\}$, at times $t_1, t_2, ..., t_k$, respectively. The operation σ_i decreases the weight $w(x_i, y_i)$ by a quantity $\epsilon_i > 0$, $i \in \{1, 2, ..., k\}$. Without loss of generality, we assume that $t_1 \leq t_2 \leq ... \leq t_k$. We denote as G' the graph after t_k, as $d'()$ and *via*$'()$ the distance and the via in G', respectively.

Data structures. As in the case of the decremental algorithm:

- a node knows the identity of each node of the graph, the identity of all its neighbors and the weight of the edges incident to it;
- the information on the estimated shortest paths are stored in a routing table RT distributed over all nodes; the entry $\mathtt{RT}_v[s]$ locally at v consists of the fields $\mathtt{RT}_v[s].\mathtt{d}$ and $\mathtt{RT}_v[s].\mathtt{via}$.

Here, differently from the decremental case, the field $\mathtt{RT}_v[s].\mathtt{via}$ represents just one neighbor of v. Formally:

$$\mathtt{RT}_v[s].\mathtt{via} \in \{v_i \in N(v) |\ \mathtt{RT}_v[s].\mathtt{d} = w(v, v_i) + \mathtt{RT}_{v_i}[s].\mathtt{d}\}$$

Again we use $\mathtt{d}_t[v, s]$ and $\mathtt{via}_t[v, s]$ to denote the estimated distance and via from s to v at time t.

Algorithm. Before the incremental algorithm starts, we assume that $\mathtt{d}_t[v, s]$ and $\mathtt{via}_t[v, s]$ are correct, for each $v, s \in V$ and for each $t < t_1$. The algorithm starts at each t_i, $i \in \{1, 2, ..., k\}$. For instance, the *weight decrease* operation σ_i represents an event that is detected only by nodes x_i and y_i, at time t_i; as a consequence:

- y_i sends the message *init*$(y_i, s, \mathtt{d}_{t_i}[y_i, s])$ to x_i, for each $s \in V$;
- x_i sends the message *init*$(x_i, s, \mathtt{d}_{t_i}[x_i, s])$ to y_i, for each $s \in V$.

Event: node v receives the message *init*$(u, s, \mathtt{d}[u, s])$.

Procedure INIT

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

Figure 4. The initialization algorithm

When x_i receives *init*$(y_i, s, \mathtt{d}_{t_i}[y_i, s])$ by y_i, x_i executes procedure INIT (see Figure 4). This procedure is responsible for checking if it is necessary to start the incremental algorithm. In the affirmative case, x_i updates $\mathtt{RT}_{x_i}[s]$ at a certain time t and, in order to propagate the incremental algorithm, sends the message *decrease*$(x_i, s, \mathtt{d}_t[x_i, s], x_i)$ to its neighbors (line 6). The first three arguments of the message have the same meaning as in *init*, while the fourth argument is one of the endpoints of the edge changed by σ_i.

The behavior of y_i (when y_i receives the message *init*$(x_i, s, \mathtt{d}_{t_i}[x_i, s])$) is symmetric. At most one between x_i and y_i will propagate the incremental algorithm. In fact, if we assume, without loss of generality, that $\mathtt{d}_{t_i}[s, x_i] \leq \mathtt{d}_{t_i}[s, y_i]$, then the test performed by x_i at Line 1 of procedure INIT is false. Thus, x_i does not update $\mathtt{RT}_{x_i}[s]$ and does not propagate the *decrease* message to its neighbors.

Conversely, under the same assumptions, y_i may improve its distance from s. In this case y_i updates $\mathtt{RT}_{y_i}[s]$ at a certain time t and, in order to propagate the incremental algorithm, sends the message *decrease*$(y_i, s, \mathtt{d}_t[y_i, s], y_i)$ to its neighbors. When a node v receives the message *decrease*$(u, s, \mathtt{d}_{\tilde{t}}[u, s], y_i)$, $\tilde{t} \geq t$, from a node u, it performs procedure DECREASE (see Figure 5).

Notice that, in our model, multiple messages *init* and *decrease* received by a node are stored and processed in a certain order.

Event: node v receives the message *decrease*$(u, s, \mathtt{d}[u, s], y)$.

Procedure DECREASE

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

Figure 5. The DECREASE algorithm

Procedure DECREASE differs from the classical distributed Bellman-Ford algorithm (e.g., see [14]) in the way in which messages are propagated. In the Bellman-Ford algorithms messages containing the estimated distances are sent to all the nodes in the graph. In the algorithm described in this section these messages are sent only to the nodes that change the shortest path with respect to at least one source as a consequence of the operations σ_i.

The correctness proof of the incremental algorithm is given in [18]. The complexity bounds of the algorithm are stated in the next theorem.

Theorem 2: The concurrent update of all-pairs shortest paths over a graph G with n nodes and positive real edges weights, after a set of *weight decrease* operations, requires $O(\mathit{maxdeg} \cdot \Delta)$ messages and $O(n)$ space per node.

Proof: Given a source s and a *weight decrease* operation σ_i, a node v can update $\mathtt{RT}_v[s]$ at most $|\delta_{\sigma_i,s}|$ times. Each time that v updates $\mathtt{RT}_v[s]$, it sends $\deg(v)$ messages. Hence, v sends at most *maxdeg* $\cdot\, |\delta_{\sigma_i,s}|$ messages. Since there are $|\delta_{\sigma_i,s}|$ nodes that change their distance from s as a consequence of σ_i, the number of messages related to the source s sent as a consequence of operation σ_i is *maxdeg* $\cdot\, |\delta_{\sigma_i,s}|$. The sum of this value over all sources $s \in V$ and *weight decrease* operations σ_i, $i \in \{1, 2, ..., k\}$ is:

$$\sum_{i=1}^{k} \sum_{s \in V} (\textit{maxdeg} \cdot |\delta_{\sigma_i,s}|) = \textit{maxdeg} \cdot \Delta$$

Thus, the message complexity is $O(\textit{maxdeg} \cdot \Delta)$. The space complexity is $O(n)$ per node because a node stores only $\mathtt{RT}_v[\cdot]$. ■

V. EXPERIMENTS

In this section we describe the experiments we performed to check the effectiveness of our algorithms also in the practical case.

Experimental environment. All the experiments have been carried out on a workstation equipped with a 2,66 GHz processor (Intel Core2 Duo E6700 Box) and a 2Gb RAM (PC6400 PRO Series, 800 MHz). The experiments consist of simulations within the OMNeT++ environment [19].

OMNeT++ is an object-oriented modular discrete event network simulator, useful to model protocols, telecommunication networks, multiprocessors and other distributed systems. It also provides facilities to evaluate performance aspects of complex software systems where the discrete event approach is suitable. An OMNeT++ model consists of hierarchically nested modules, that communicate through message passing. Modules and messages can have their own parameters, stored in arbitrarily complex data structures, that can be used to customize specific behaviors or topologies.

In our model, we defined a basic module *node* to represent a node in the network. A node v has a communication *gate* with each node in $N(v)$. Each node can send messages to a destination node through a *channel* which is a module that connects gates of different nodes (both gate and channel are OMNeT++ predefined modules). In our model, a channel connects exactly two gates and represents an edge between two nodes. We associate two parameters per channel: a *weight* and a *delay*. The former represents the cost of the edge in the graph, and the latter simulates a finite but not null transmission time.

Implemented algorithms. We implemented the algorithms described in Sections III and IV, that in the remainder we denote as DECR and INCR. In order to compare their performances with respect to known algorithms in literature, we also implemented three different versions of the Bellman-Ford algorithm. They are denoted as BF.1, BF.2 and BF.3 and briefly described as follows.

BF.1 In this version, described in [14], each node v updates its estimated distance to a node s, by simply executing the iteration

$$\mathtt{d}[v,s] := \min_{u \in N(v)} \{w(v,u) + \mathtt{d}[u,s]\}$$

using the last estimated distances $\mathtt{d}[u,s]$ received from the neighbors $u \in N(v)$ and the latest status of its links. Eventually, node v transmits the new estimated distance to its neighbors. It requires $O(n \cdot \textit{maxdeg})$ space per node to store the last estimated distance vector $\{\mathtt{d}[u,s] \mid s \in V\}$ received from each neighbor $u \in N(v)$.

BF.2 The only difference with the previous version is that the node v does not explicitly store the estimated distances $\mathtt{d}[u,s]$ which are asked to the neighbors when needed. This results in $O(n)$ space per node.

BF.3 This version is described in [20]. It assumes that each node v initially overestimates the distance with the remaining nodes in the network. Then, for each new $\mathtt{d}[u,s]$ received from a neighbor $u \in N(v)$, it first checks whether its estimated distance to s can be improved, and, in the affirmative case, it sends the new estimated distance to each neighbor but u. It requires $O(n)$ space per node.

Executed tests. We compared the experimental performances of DECR against those of BF.1 and BF.2, as follows. We randomly generated a set of different tests, where a test consists of a dynamic graph characterized by the following parameters:

- n, the number of nodes of the graph;
- $dens$, the density of the graph. It is computed as the ratio between m and the number of the edges of the n-complete graph;
- k, the number of edge update operations.

For n we used three values: 100, 300, and 500. For each possible value of n, we chosen different values of $dens$ ranging from $(\log n + 3)/n$ – a value that guarantees a connected graph with a probability of 95% – to 0.3. The number k ranges from $0.02m$ to $0.16m$. Edge weights are non-negative real numbers randomly chosen in $[1, 200]$. Edge delays are expressed in milliseconds, and are randomly chosen in $[100, 1000]$. For INCR, edge weights are decreased by a percentage randomly chosen in the range $[10\%, 90\%]$, while for DECR, edge weights are increased by a percentage randomly chosen in the range $[10\%, 400\%]$. For each test configuration – represented by the triple $(n, dens, k)$ – we performed at least four different experiments. All the obtained data has to be intended as average values together with the standard deviations.

Due to memory and execution time limits, we were not able to run experiments for values of n larger than 500 and values of k larger than 5000 (with a few exceptions).

Furthermore, we built a test configuration which shows that a weight increase let BF.1 and BF.2 to fall into a loop, while DECR is able to prevent this phenomenon.

Decremental algorithm: results. The performances of DECR have been compared to those of BF.1 and BF.2 (BF.3 cannot be used when edge weights increase).

In Figure 6 we report the number of messages used by the three algorithms when $n = 100$ and $dens = 0.0964$. BF.2 is worse than the others: it requires from 10 to 15 times the number of messages needed to both BF.1 and DECR. Considering the experiments over all the test configurations, this ratio ranges from 10 to 50. Hence, in what follows, we report a more detailed comparison of only BF.1 and DECR.

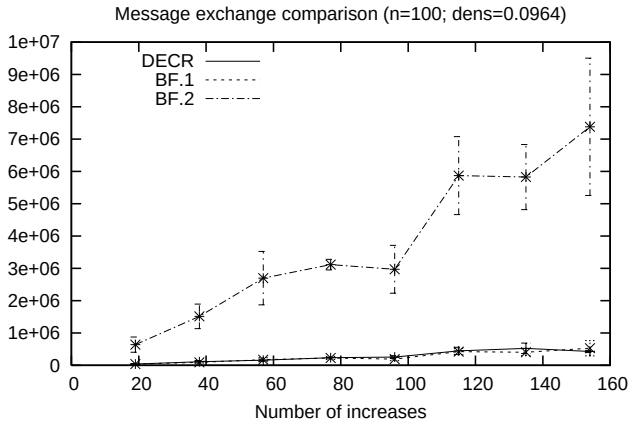

Figure 6. Number of messages needed by DECR, BF.1 and BF.2

Figure 7 shows the same results of Figure 6, but BF.2 is omitted. Here, it is evident that the two algorithms require approximately the same number of messages: in general, BF.1 requires less messages, but the there are instances in which DECR performs better.

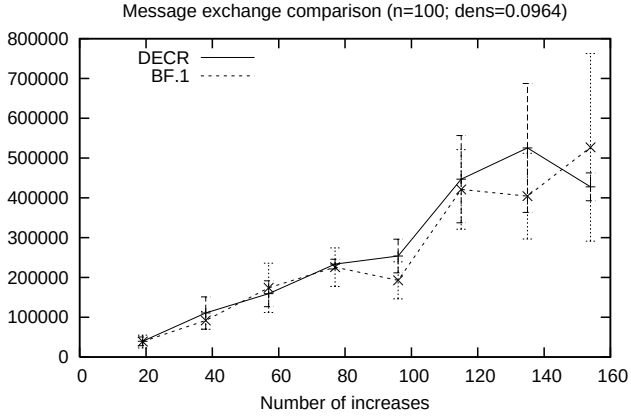

Figure 7. Number of messages needed by DECR and BF.1

A different view of the messages sent by the two algorithms is given in Figure 8. Given a set of experiments, let $m(A)$ be the average number of messages sent by a generic algorithm A. Then, we define the *message exchange percentage gain (gain%)* as:

$$\frac{m(\text{BF.1}) - m(\text{DECR})}{m(\text{DECR})} \cdot 100.$$

Figure 8 shows the gain% for the set of experiments represented in Figure 7.

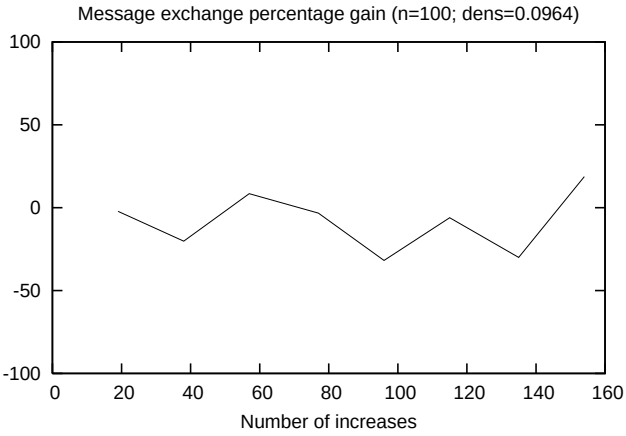

Figure 8. Message exchange percentage gain in the decremental case

Regarding all the remaining experiments, the results are summarized in Table I.

TABLE I.
SUMMARY OF RESULTS

n	$dens$	gain%	std.dev. gain%
100	0.096	-8.28	16.74
100	0.122	-27.92	32.99
100	0.148	-28.81	14.11
100	0.174	-33.36	11.34
100	0.2	-43.54	13.20
300	0.037	-11.35	7.78
300	0.078	-48.72	14.16
300	0.118	-49.01	7.89
300	0.159	-64.90	3.60

Each row represents a set of experiments characterized by a test configuration. The first two values represent the number of nodes and the density of the networks. For each test configuration, at least 20 experiments have been performed by taking different values of k in the range $0.02m$ to $0.16m$. The remaining two columns report the average values gain% and their standard deviations. Notice that the values of gain% are negative, meaning that, in average, BF.1 performs better than DECR. However, the extra number of messages used by DECR can be seen as the price to avoid space consumption, looping and count-to-infinity phenomena.

This is explained in Figure 9 that shows a classical topology where BF.1 and BF.2 count to infinity and create a loop in which nodes a chooses b as via to s and viceversa. In particular, when the weight of the edge (s, v) changes, v updates its distance to s. Now, we concentrate on the operations performed by nodes a and b. When node a (b, resp.) performs the updating step, it finds out that its new via to s is b (a, resp.). Subsequent updating steps (but the last one) do not change the via to s of

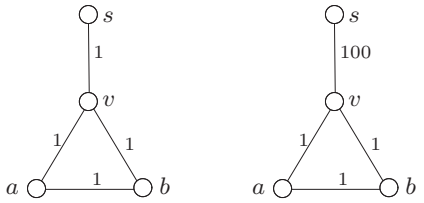

Figure 9. *A graph G before and after a weight increase on edge (s, v).*

both a and b, but only the estimated distances. For each updating step the estimated distances increase by 1 (i.e., the weight of the edge (a, b)). The counting stops after a number of updating that depends on the new weight of the edge (u, v). If the new weight is ∞ - that is, the link (u, v) breaks - then the algorithm counts to infinity.

Conversely, DECR requires few steps to update both the estimated distance and via to s for each node in G. When s and v detect the weight change, they perform Procedure INCREASE with respect source s. In particular, s does not performs BUILD-TABLE phase, while v does. When v gets the estimated distances to s from its neighbors (Line 6 of Procedure INCREASE), it receives $+\infty$ from both a and b. This is due to the fact that, when a (b, resp.) performs Procedures DIST, the test at Line 1 returns true. At the end of these executions, v correctly updates its routing table and sends messages *increase* to each neighbor. Hence, s, a, and b perform Procedure INCREASE, but only a and b perform the BUILD-TABLE phase. In this phase, a and b send ∞ (as their estimated distance to s) to each other in response to the *get-dist* message (see Line 6). This is due to the fact that, during the executions of Procedure DIST, both a and b are performing the BUILD-TABLE phase and then, the test at Line 1 returns true. Hence, both a and b correctly update their routing tables in one step. Subsequent messages sent by a and b do not produce further local data modification.

The tests on the example shown in Figure 9 are reported in Table II.

TABLE II.
COUNT-TO-INFINITY

$w(v, s)$	DECR	BF.1	BF.2
100	39	702	3491
200	39	1402	6991
300	39	2102	10491
400	39	2802	13991
500	39	3502	17491
600	39	4202	20991
700	39	4902	24491
800	39	5602	27991
900	39	6302	31491
1000	39	7002	34991

The results show that DECR requires a constant number of messages, while, as expected, BF.1 and BF.2 require a number of messages that depends on the new weight on the edge (s, v). Notice that, the table shows the *total* number of messages required by the algorithms, while in the previous discussion we have only considered the messages required to update the routing table with respect to s.

To conclude our discussion on the performances of the three implemented algorithms, we show the results about the space occupancy. BF.1 requires to store, for each destination, the estimated distance given by each of its neighbors, BF.2 only its estimated distance, whereas DECR is something in between: the estimated distance and the set `via`. Since it is not common to have more than one via to a destination, the size to store the routing table for DECR is very close to the size required by BF.2.

TABLE III.
SPACE REQUIREMENTS IN THE DECREMENTAL CASE

n	*dens*	DECR	BF.1 (max)	BF.1 (avg)	BF.2
100	0.096	206	1900	476	200
100	0.122	203	2300	604	200
100	0.148	201	2500	726	200
100	0.174	201	3100	862	200
100	0.200	201	3300	992	200
300	0.037	607	7500	1677	600
300	0.078	627	12300	3498	600
300	0.118	608	18000	5352	600
300	0.159	604	21300	8634	600
300	0.200	610	25800	8979	600
500	0.023	1006	13500	3005	1000
500	0.067	1010	29000	8470	1000
500	0.111	1009	41500	13935	1000
500	0.155	1010	56000	19565	1000
500	0.200	1009	68000	24950	1000

Table III summarizes the data relative to the space used by the three algorithms, assuming that the cost to store either a destination or an estimated distance is unitary. The third column gives the space used by DECR. In particular, it reports the space consumption of the node with the maximum size of `via`. BF.1 requires much more space (that is given by n times the degree of each node): the fourth column reports the space consumption of the node with maximum degree while the average space consumption per node is listed in the fifth column. The last columns gives the space used by BF.2 for each node. This value is given by two times n.

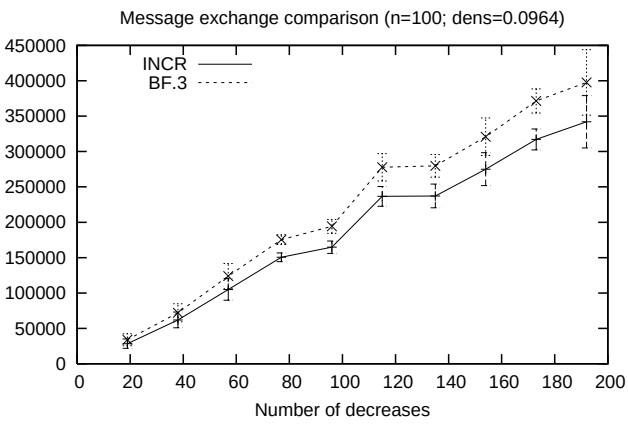

Figure 10. Number of messages needed by INCR and BF.3

Incremental algorithm: results. In the following, we will compare the performance of BF.3 with respect to INCR.

Since, the space required by the two algorithms is the same, then we just focus on the number of messages they send.

In Figure 10 we report the number of messages used by the two algorithms when $n = 100$ and $dens = 0.0964$. The average number of messages required by INCR is always less than that required by BF.3. The gain%, defined in this case as

$$\frac{m(BF.3) - m(INCR)}{m(BF.3)}$$

is about 15% and it is shown in Figure 11.

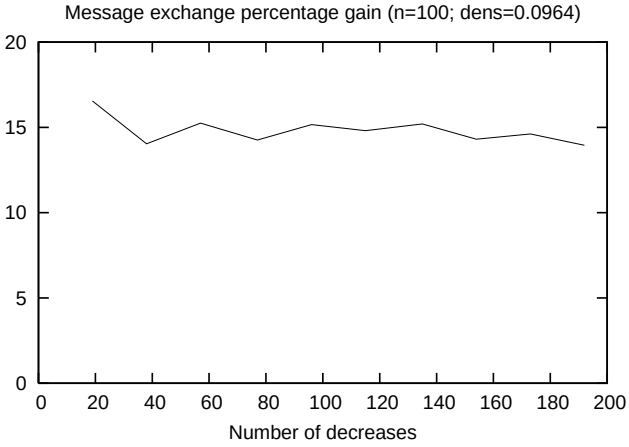

Figure 11. Message exchange percentage gain in the incremental case

We repeated the experiments for many different test configurations, always reaching results qualitatively similar to those shown in Figure 10. These results are summarized in Table IV. Each row in the table refers to at least 40 tests and it is worth to note that the gain% is always in favor of INCR.

TABLE IV.
PERCENTAGE GAIN IN THE INCREMENTAL CASE

n	$dens$	gain%	std.dev. gain%
100	0.096	14.81	0.73
100	0.147	11.30	0.46
100	0.198	9.93	0.28
100	0.249	8.68	0.66
100	0.3	8.30	0.67
300	0.037	13.10	0.43
300	0.103	7.94	0.33
300	0.168	6.78	0.26
300	0.234	6.05	0.27
300	0.3	5.76	0.23
500	0.024	12.77	0.22
500	0.093	6.85	0.19
500	0.162	5.57	0.22
500	0.231	5.20	0.17
500	0.3	4.69	0.11

VI. FUTURE WORK

Future work will explore the possibility to extend the partially dynamic solutions proposed here to the more realistic fully dynamic case, while keeping the merits of the partially dynamic solution that is: being concurrent and, in many cases, free of the looping and count-to-infinity phenomena. Another research direction is that of experimentally compare our solution with other variants of the Bellman-Ford methods known in the literature.

REFERENCES

[1] G. D'Angelo, S. Cicerone, G. D. Stefano, and D. Frigioni, "Partially dynamic concurrent update of distributed shortest paths," in *International Conference on Computing: Theory and Applications (ICCTA'07)*. IEEE Computer Society, 2007, pp. 32–38.

[2] J. T. Moy, *OSPF - Anatomy of an Internet routing protocol*. Addison-Wesley, 1998.

[3] H. Attiya and J. Welch, *Distributed Computing*. John Wiley and Sons, 2004.

[4] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni, "Fully dynamic algorithms for maintaining shortest paths trees." *Journal of Algorithms*, vol. 34, no. 2, pp. 251–281, 2000.

[5] G. Ramalingam and T. Reps, "On the computational complexity of dynamic graph problem." *Theoretical Computer Science*, vol. 158, pp. 233–277, 1996.

[6] C. Demetrescu and G. F. Italiano, "A new approch to dynamic all pairs shortest paths." *Jou. of ACM*, vol. 51, no. 6, pp. 968–992, 2004.

[7] B. Awerbuch, I. Cidon, and S. Kutten, "Communications-optimal maintenance of replicated information." in *Proc. IEEE Symposium on Foundation of Computer Science*, 1990, pp. 492–502.

[8] S. Cicerone, G. D. Stefano, D. Frigioni, and U. Nanni, "A fully dynamic algorithm for distributed shortest paths." *Theoretical Computer Science*, vol. 297, no. 1-3, pp. 83–102, Mar. 2003.

[9] P. A. Humblet, "Another adaptive distributed shortest path algorithm." *IEEE Transactions on Communications*, vol. 39, no. 6, pp. 995–1002, Apr. 1991.

[10] G. F. Italiano, "Distributed algorithms for updating shortest paths." *Proceedings of Int. Workshop on Distributed Algorithms.*, vol. LNCS, 579, pp. 200–211, 1991.

[11] A. Orda and R. Rom, "Distributed shortest-path and minimum-delay protocols in networks with time-dependent edge-length." *Distributed Computing*, vol. 10, pp. 49–62, 1996.

[12] K. V. S. Ramarao and S. Venkatesan, "On finding and updating shortest paths distributively." *Journal of Algorithms*, vol. 13, pp. 235–257, 1992.

[13] J. McQuillan, "Adaptive routing algorithms for distributed coomputer networks," Cambridge, MA, Tech. Rep. BBN Report 2831, 1974.

[14] D. Bertsekas and R. Gallager, *Data Networks*. Prentice Hall International, 1992.

[15] A. S. Tanenbaum, *Computer Networks*. Prentice Hall, 1996.

[16] E. C. Rosen, "The updating protocol of arpanet's new routing algorithm." *Computer Networks*, vol. 4, pp. 11–19, 1980.

[17] B. Awerbuch, A. Bar-Noy, and M. Gopal, "Approximate distributed bellman-ford algorithms." *IEEE Transactions on Communications*, vol. 42, no. 8, pp. 2515–2517, 1994.

[18] S. Cicerone, G. D'Angelo, G. Di Stefano, and D. Frigioni, "Partially dynamic efficient algorithms for distributed shortest paths," ARRIVAL Project, Tech. Rep. 0022, 2006.

[19] "OMNeT++: the discrete event simulation environment." `http://www.omnetpp.org/`.

[20] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

Serafino Cicerone received his Ph.D. in February 1998 from the University of Rome "La Sapienza". He has been visiting scientist at the Institute of Theoretical Computer Science, University of Rostock (Germany). Currently he is associate professor in Computer Science at the University of L'Aquila. His main research areas concern network algorithms, combinatorial optimization, algorithmic graph theory, and he is (co-)author of several papers published in both international journals and conferences. He has contributed to the organization of international and national conferences. He participated and participates in several EU and national funded projects.

Gianlorenzo D'Angelo is currently a Ph.D. student in the Department of Electrical and Information Engineering of the University of L'Aquila. His research interests are in the area of graphs and networks algorithms. In particular, he is involved in the development of sequential and distributed algorithms for the shortest paths problem in a dynamic environment.

Gabriele Di Stefano obtained his Ph.D. at the University "La Sapienza" of Rome in 1992. Currently he is associate professor for computer science at the University of L'Aquila; his current research interests include network algorithms, combinatorial optimization, algorithmic graph theory; he is (co-)author of more than 50 publications in journals and international conferences. He had key-participation in several EU funded projects. Among them: MILORD (AIM 2024), COLUMBUS (IST 2001-38314), AMORE (HPRN-CT-1999-00104), and, currently, ARRIVAL (IST FP6-021235-2).

Daniele Frigioni received his Ph.D. in February 1997 by the University of Rome "La Sapienza". He his currently associate professor for computer science at the University of L'Aquila (Italy). His main research areas concern the design, analysis and engineering of graph and network algorithms. He is (co-)author of several publications in this area in both international journals and conferences. He has contributed to the organization of several international and national conferences. He participated and participates in several EU and national funded projects. (http://www.diel.univaq.it/frigioni)

Alberto Petricola is a Ph.D. student in the Department of Electrical and Information Engineering of the University of L'Aquila. His research interests are in the areas of parallel and distributed processing. He has worked for the Italian National Institute of Nuclear Physics (INFN) contributing to the development of the APE project. His current research area is the development of distributed algorithms for Ad Hoc Networks.