Engineering a new algorithm for distributed shortest paths on dynamic networks

Serafino Cicerone, Gianlorenzo d'Angelo, Gabriele Di Stefano, Daniele Frigioni, Vinicio Maurizio

Noname manuscript No.
(will be inserted by the editor)

Engineering a new algorithm for distributed shortest paths on dynamic networks

Serafino Cicerone · Gianlorenzo D'Angelo · Gabriele Di Stefano · Daniele Frigioni · Vinicio Maurizio

the date of receipt and acceptance should be inserted later

Abstract We study the problem of dynamically updating *all-pairs shortest paths* in a distributed network while edge update operations occur to the network. We consider the practical case of a dynamic network in which an edge update can occur while one or more other edge updates are under processing. A node of the network might be affected by a subset of these changes, thus being involved in the concurrent executions related to such changes.

In this paper, we provide a new algorithm for this problem, and experimentally compare its performance with respect to those of the most popular solutions in the literature: the classical distributed Bellman-Ford method, which is still used in real network and implemented in the RIP protocol, and DUAL, the Diffuse Update ALgorithm, which is part of CISCO's widely used EIGRP protocol. As input to the algorithms, we used both real-world and artificial instances of the problem. The experiments performed show that the space occupancy per node required by the new algorithm is smaller than that required by both Bellman-Ford and DUAL. In terms of messages, the new algorithm outperforms both Bellman-Ford and DUAL on the real-world topologies, while on artificial instances, the new algorithm sends a number of messages that is more than that of DUAL and much smaller than that of Bellman-Ford.

A preliminary version of this paper appeared in the proceedings of the 9th International Symposium on Experimental Algorithms (SEA2010) [9]

S. Cicerone · G. Di Stefano · D. Frigioni · V. Maurizio
Department of Electrical and Information Engineering
University of L'Aquila
Via G. Gronchi 18, I-67100 L'Aquila, Italy.
E-mail: serafino.cicerone@univaq.it, gabriele.distefano@univaq.it, daniele.frigioni@univaq.it, vinicio.maurizio@cc.univaq.it

G. D'Angelo
MASCOTTE Project I3S(CNRS/UNSA)/INRIA
2004 Route Des Lucioles, BP 93, 06902 Sophia–Antipolis Cedex, France. E-mail: gianlorenzo.d_angelo@inria.fr

Keywords Shortest Paths, Distributed Networks, Dynamic Algorithms, Concurrent Update, Experimental Analysis.

1 Introduction

The problem of efficiently updating *all-pairs shortest paths* in a distributed network whose topology dynamically changes over the time, in the sense that link weights can be modified during the lifetime of the network, is considered crucial in today's practical applications. Hence, it is very important to find efficient dynamic distributed algorithms for shortest paths. This problem has been widely studied in the literature, and the solutions found can be classified as *distance-vector* and *link-state* algorithms.

Distance-vector algorithms require that a node knows the distance from each of its neighbors to every destination and stores this information in a data structure usually called routing table. A node uses its own routing table to compute the distance and the next node in the shortest path to each destination. Most of the distance-vector solutions for the distributed shortest paths problem proposed in the literature (e.g., see [8, 11, 13, 14, 16, 20, 21]) rely on the classical Distributed Bellman-Ford method (DBF from now on), originally introduced in the Arpanet [17], which is still used in real networks and implemented in the RIP protocol. DBF has been shown to converge to the correct distances if the link weights stabilize and all cycles have positive lengths [6]. However, the convergence can be very slow (possibly infinite) due to the well-known *count-to-infinity* phenomenon also know as routing table looping. A loop is a path induced by the routing table entries, such that the path visits the same node more than once before reaching the intended destination. A node "counts to infinity" when it increments its distance to a destination until it reaches a predefined maximum distance value. Furthermore, if the nodes of the network are not synchronized, even though no change occurs in the network, the overall number of messages sent by DBF is exponential in the size of the network (e.g., see [5]).

Link-state algorithms, as for example the *Open Shortest Path First (OSPF)* protocol widely used in the Internet (e.g., see [18]), require that a node knows the entire network topology to compute its distance to any network destination (usually running the centralized Dijkstra's algorithm for shortest paths). Link-state algorithms are free of count-to-infinity, however each node needs to receive and store up-to-date information on the entire network topology after a change, thus requiring quadratic space per node. This is achieved by broadcasting each change of the network topology to all nodes [18, 23, 24] and by using a centralized dynamic algorithm for shortest paths as for example those in [12, 19].

If the topology of a dynamic network is represented as a weighted undirected graph, then the typical update operations on that network can be modeled as insertions and deletions of edges and edge weight changes (weight decrease and weight increase). When arbitrary sequences of the above operations are allowed, we refer to the *fully dynamic problem*; if only insertions and weight decrease (deletion and weight increase) operations are allowed, then we refer to the *incremental* (*decremental*) problem. We are interested in the practical case of a dynamic network in which an edge change can occur while one or more other edge changes are under processing. A node (processor) v of the network might be affected by a subset of these changes. As a consequence, v could be involved in the *concurrent* executions related to such changes.

Many algorithms proposed in the literature for the dynamic distributed shortest paths problem are not able to concurrently update shortest paths as those in [11,16,20,21]. In particular, they 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. For example, the algorithm proposed in [11] to handle weight increase operations works in three phases, and each phase starts when the previous one is completed. In a concurrent scenario, a new edge update can stop one of these phase thus making the algorithm incorrect. Among the algorithms which are able to concurrently update shortest paths (see, e.g., [8,10,13,14,22]) the most interesting is DUAL (Diffuse Update ALgorithm) [13], which is free of the count-to-infinity phenomenon, thus resulting an effective practical solution (it is in fact part of CISCO's widely used EIGRP protocol).

An incremental and a decremental solution have been proposed in [10]. Such solutions cannot deal with fully dynamic sequences of edge weight changes and the decremental solution suffers of count-to-infinity. In this paper, we extend the decremental solution of [10] by providing a new concurrent and fully dynamic algorithm for the distributed shortest paths problem denoted as DUST (Distributed Update of Shortest paThs). DUST suffers of the count-to-infinity problem, but it has been designed to heuristically reduce the cases where this phenomenon occurs. If we denote as n the number of nodes in the network, as $deg(v)$ the degree of a node v, and as $maxdeg$ the maximum node degree, DUST requires the same worst-case space occupancy per node of both DBF and DUAL that is $O(n \cdot maxdeg)$. However, DBF and DUAL store, for each node v, a data structure whose size is proportional to $n \cdot deg(v)$. In fact, they store specific information on each neighbor u of v, as for example, the distance from u to any other node s. Instead, DUST simply stores at node v the distance from v to s and all possible alternative vias to s, and does not store any information regarding its neighbors. This means that the space required by DUST depends on the node degree only in the worst case, which indeed occurs very rarely. In most of the cases the space occupancy of DUST does not depend on the node degree. For these reasons, it is worth measuring the practical performance of DBF, DUAL, and DUST in various dynamic realistic and artificial scenarios, in terms of both the overall number of messages they send, and their space occupancy per node.

With this goal in mind and following an algorithm engineering approach, we implemented DBF, DUAL, and DUST in the OMNeT++ simulation environment [1], an object-oriented modular discrete event network simulator which is widely used in the literature. As input to the algorithms, we used both real-world and artificial instances of the problem. In detail, we used snapshots of the Internet graph provided by the *CAIDA IPv4 topology dataset* [15]. CAIDA (Cooperative Association for Internet Data Analysis) is an association which provides data and tools for the analysis of the Internet infrastructure. Then, we used random Internet-like topologies with a power-law node degree distribution, generated by the *Barabási-Albert* algorithm [2]. Power-law networks have been proven to model many real-world networks such as the Internet, the World Wide Web, citation graphs, and some social networks [3]. Finally, since both CAIDA graphs and Barabási-Albert graphs turn out to be very sparse, we also analyzed random dense graphs generated by the *Erdős-Rényi* algorithm [7].

The experiments performed show that the space occupancy per node required by DUST is much smaller than that required by both DBF and DUAL. In terms of number of messages sent, DUST outperforms both Bellman-Ford and DUAL on the Internet topologies provided by CAIDA. In Barabási-Albert artificial instances, DUST sends more messages than DUAL but fewer than Bellman-Ford. In the case of Erdős-Rényi artificial instances, DUAL is still better than DUST in terms of number of messages, while we noticed a big increase in the space occupancy per node required by DUAL. In all our experiments, we observed that both DUAL (as expected) and DUST never counts to infinity, while in many cases DBF does. The last observation confirms the effectiveness of the heuristics implemented in DUST to reduce the cases where the count-to-infinity phenomenon occurs.

Finally, we compared DUST with the incremental and decremental solutions of [10], which are denoted from now on as INC and DEC, respectively. The comparison has been done separately, once for sequences of only *weight increase* operations and once for sequences of only *weight decrease* operations. While the space occupancy per node is always comparable, in term of number of messages sent, in the incremental case, DUST is slightly worse than INC and, in the decremental case, it is slightly better than DEC.

The paper is organized as follows. In Section 2 we describe the model and the notation used in the paper and the DBF and DUAL algorithms. In Section 3 we describe DUST. In Section 4 we give the correctness analysis of DUST. In Section 5 we report the results of our experimental study. Finally, in Section 6 we give some concluding remarks and outline possible future research directions. In the appendix, we give an example of execution of DBF, DUAL and DUST, useful to appreciate the differences among the algorithms.

2 Preliminaries

We consider a network made of processors linked through weighted communication channels. Each processor can send messages only to its neighbors. We assume that 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. Finally, there is no shared memory among the nodes of the network.

Asynchronous model. We consider an asynchronous system based on that proposed in [4] and summarized below. The *state* of a processor v is the content of the data structure at processor v. The *network state* is the set of states of all the processors in the network plus the network topology and the channel weights. 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 integer 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 time must be non decreasing and must increase without bound if the execution is infinite. Events are ordered according to the time at which they occur. Several events can happen at the same time as long as they do not occur on the same processor. This implies that the times related to a single processor are strictly increasing.

Concurrent executions. We consider a dynamic network in which a change to the network topology or to the channel weights can occur while one or more other changes are under processing. A processor v could be affected by a subset of these changes. As a consequence, v could be involved in the executions related to such changes. Hence, according to the asynchronous model described above we need to define the notion of *concurrent* executions as follows. Let us consider an algorithm A that maintains some data structure on the processors of the network after a change to the network topology or to channel weights. Let c_i and c_j be two of such changes, we denote as: t_i and t_j the times at which c_i and c_j occur, respectively; $\mathcal{A}_i$ and $\mathcal{A}_j$ the executions of A related to c_i and c_j, respectively; and $t_{\mathcal{A}_i}$ 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*.

Graph notation. 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 $w : E \to \mathbb{R}^+ \cup \{\infty\}$. An edge in E that links nodes $u, v \in V$ is denoted as $u \to v$ or (u, v). Given $v \in V$, $N(v)$ denotes the set of neighbors of v and $deg(v)$ the degree of v, which is the size of $N(v)$. The maximum degree of the nodes in G is denoted by *maxdeg*. A path P in G between nodes u and v is denoted as $P = u \leadsto v$. We define the *length* of P as the number of edges of P and denote it by $\ell(P)$, and define the *weight* of P as the sum of the weights of the edges in P and denote it by $weight(P)$. A *shortest path* between nodes u and v is a path from u to v with the minimum weight. The *distance* from u to v is the weight of a shortest path from u to v, and is denoted as $d(u, v)$. Given two nodes $u, v \in V$, the *via* from u to v is the set of neighbors of u that belong to a shortest path from u to v. Formally: $via(u, v) \equiv \{z \in N(u) \mid d(u, v) = w(u, z) + d(z, v)\}$.

Given a graph $G = (V, E, w)$, we suppose that a sequence $\mathcal{C} = (c_1, c_2, ..., c_k)$ of k operations is performed on edges $(x_i, y_i) \in E$, $i \in \{1, 2, ..., k\}$. The operation c_i either inserts a new edge in G, or deletes an edge of G, or modifies (either increases or decreases) the weight of an existing edge in G. We consider the case in which $\mathcal{C}$ is a sequence of *weight increase* and *weight decrease* operations, that is operation c_i either increases or decreases the weight of edge (x_i, y_i) by a quantity $\epsilon_i > 0$, $i \in \{1, 2, ..., k\}$. The extension to *delete* and *insert* operations, respectively, is straightforward: deleting an edge (x, y) is equivalent to increase $w(x, y)$ to $+\infty$, and inserting an edge (x, y) with weight α is equivalent to decrease $w(x, y)$ from $+\infty$ to α. Without loss of generality, we assume that operations in $\mathcal{C} = (c_1, c_2, ..., c_k)$ occur at times $t_1 \leq t_2 \leq ... \leq t_k$, respectively. Assuming $G^0 \equiv G$, we denote as G^i the graph obtained by applying the operation c_i to G^{i-1}. We denote as $d^i()$ and $via^i()$ the distance and the via over G^i, $0 \leq i \leq k$, respectively. Given a path P in G, we denote as $weight^i(P)$ the weight of P in G^i, $0 \leq i \leq k$.

Distance vector algorithms. Here we describe the two most popular distance vector algorithms known in the literature, the classical Bellman-Ford method (DBF) and the Diffuse Update Algorithm (DUAL).

DBF requires each node v in the network to store the last estimated distance towards any other node $s \in V$ received from each neighbor $u \in N(v)$, denoted as $\mathtt{D}_v[u, s]$. In DBF, a node v updates its estimated distance to a node s by simply executing the iteration $\mathtt{D}_v[v, s] := \min_{u \in N(v)}\{w(v, u) + \mathtt{D}_v[u, s]\}$. Like many distance vector algorithms, DBF suffers of the well-known count-to-infinity problem, which arises when a certain kind of link failure or weight increase operation occurs in the network. Figure 1 shows a classical topology where DBF counts to infinity. In particular, the left and right sides of such a figure show a graph G before and after a weight modification on edge (s, v). Figure 2 shows the steps required by DBF to update both the distance and the via to s for each node in G. In detail, when the weight of the edge (s, v) increases to 100, v updates its distance and via to s by setting $\mathtt{D}_v[v, s]$ to 3 and via to $\{a, b\}$. In fact, v knows that the distance from a (and b) to s is 2, while the weight of edge (v, s) is 100. Note that, v cannot know that the path

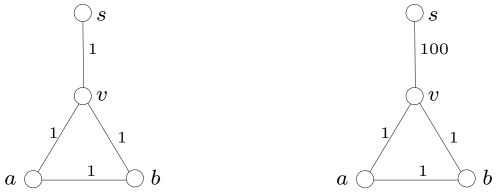


Fig. 1. A graph G before and after a weight modification on the edge (s, v).

from a to s with weight 2 is that passing through edge (v, s) itself. Now, we concentrate on the operations performed by nodes a and b. When node a (b, respectively) performs the updating step, it finds out that its new via to s is b (a, respectively) and its new distance is 3. In fact, according to a's information $\mathtt{D}[v, s] = 3$ and $\mathtt{D}[b, s] = 2$, therefore $w(a, b) + \mathtt{D}[b, s] < w(a, v) + \mathtt{D}[v, s]$. Subsequent updating steps (but the last one) do not change the via to s of 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 steps that depends on the new weight of the edge (s, v) and on the weight of edge (a, b). Note that, if edge (s, v) is deleted (i.e. his weight is set to ∞), the algorithm does not terminate.

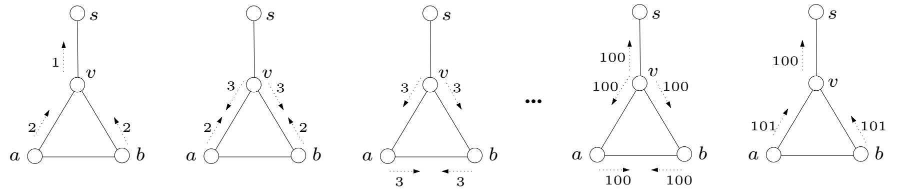


Fig. 2. The sequence of recomputations of $\mathtt{D}[u, s]$ and $\mathtt{VIA}[u, s]$, $u \in G$, performed by DBF. The via to s is represented by a dotted arrow, while the number that lies close to the arrow represents the estimated distance.

DUAL requires, for each node v to maintain, for each destination s a set of neighbors called the feasible successor set $\mathtt{F}[v, s]$, and for each $u \in N(v)$, the distance $\mathtt{D}[u, s]$ from u to s. $\mathtt{F}[v, s]$ is computed using a feasibility condition involving feasible distances from each node u in $N(v)$ to s. In detail, node u is in $\mathtt{F}[v, s]$ if the estimated distance $\mathtt{D}[u, s]$ from u to s is smaller than the estimated distance $\mathtt{D}[v, s]$ from v to s. If the neighbor u, through which the distance to s is minimum, is in $\mathtt{F}[v, s]$, then u is chosen as *successor* to s. If $\mathtt{F}[v, s]$ does not include u, then v initiates a synchronous update procedure, known as *diffuse computation*. Node v sends queries to all its neighbors with its distance through the current successor by using message *query*. From this point onwards v does not change its successor to s until the diffusing computation terminates. When a neighbor $u \in N(v)$ receives a queries, it updates $\mathtt{F}[u, s]$. If u has a successor to s after such update, it replies to the query by sending message *reply* containing its own distance to s. Otherwise, u continues the diffuse computation: it sends out queries and waits for the replies from its neighbors before replying to v's original query. When a node receives messages *reply* by all its neighbors, it updates its distance with the minimal value obtained by its neighbors and finishes the diffuse computation. At the end of a diffuse computation, a node sends message *update* containing the new computed distance to notify it to its neighbors. If there are concurrent updates, the node uses a finite state machine to process these multiple updates sequentially.

3 The new algorithm

In this Section we introduce DUST, our new solution for the concurrent update of distributed all-pairs shortest paths in dynamic networks. In detail, in Section 3.1, we define the data structure per node needed by DUST and give the pseudo-code of the algorithm; in Section 3.2, we outline the main ideas of the behavior of DUST by giving two examples of executions; in Section 3.3, we compare DUST with respect to DBF, DUAL, INC and DEC in terms of number of messages sent and space occupancy per node. The correctness of DUST is given in Section 4.

3.1 Data structures and pseudo-code

Given $G = (V, E, w)$, we assume that each node of G knows the identity of every other node of G, the identity of all its neighbors and the weights of the edges incident to it. The information on the shortest paths in G are stored in a data structure called *routing table* $\texttt{RT}$ distributed over all nodes. Each node v maintains its own routing table $\texttt{RT}_v[\cdot]$, that has one entry $\texttt{RT}_v[s]$, for each $s \in V$. The entry $\texttt{RT}_v[s]$ consists of two fields:

- $\texttt{RT}_v[s].\texttt{D}$, the estimated distance between nodes v and s in G;
- $\texttt{RT}_v[s].\texttt{VIA} \equiv \{u \in N(v) \mid \texttt{RT}_v[s].\texttt{D} = w(v,u) + \texttt{RT}_u[s].\texttt{D}\}$, 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. As the data structures change over time, in what follows 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 the time is clear by the context. Given a destination s the set $\texttt{VIA}[v,s]$ contains at most $deg(v)$ elements. Hence, each node v requires $O(n \cdot deg(v))$ space and the space complexity of DUST is hence $O(maxdeg \cdot n)$ per node in the worst case.

Algorithm DUST is reported in Figure 3, 4 and 5, and is described in what follows with respect to a source $s \in V$. Before the algorithm starts, we assume that, for each $v, s \in V$ and for each $t < t_1$, $\texttt{D}_t[v,s]$ and $\texttt{VIA}_t[v,s]$ are correct, that is $\texttt{D}_t[v,s] = d^0(v,s)$ and $\texttt{VIA}_t[v,s] = via^0(v,s)$. The algorithm starts at each t_i, $i \in \{1, 2, ..., k\}$. The event related to operation c_i on edge (x_i, y_i) is detected only by nodes x_i and y_i. As a consequence, if c_i is a *weight increase* (*weight decrease*) operation, x_i sends the message $increase(x_i, s)$ ($decrease(x_i, s, \texttt{D}_{t_i}[x_i, s])$) to y_i and y_i sends the $increase(y_i, s)$ ($decrease(y_i, s, \texttt{D}_{t_i}[y_i, s])$) message to x_i, for each $s \in V$. Note that message *decrease* includes information about the distance from y_i to s, while message *increase* does not. In the case of a *weight increase*, in fact, such distance will be possibly obtained by x_i in the subsequent REBUILD-TABLE phase (see the following description).

If a node v receives the message $decrease(u, s, \texttt{D}[u,s])$, then it performs Procedure DECREASE in Figure 3. Basically, DECREASE performs a relaxation

```
Event: node $v$ receives the message $decrease(u, s, \mathtt{D}[u, s])$ from $u$
Procedure DECREASE
 1.  if $w(v, u) + \mathtt{D}[u, s] < \mathtt{D}[v, s]$ then
 2.    begin    Lines 2-7: phase IMPROVE-TABLE
 3.      $\mathtt{D}[v, s] := w(v, u) + \mathtt{D}[u, s]$
 4.      $\mathtt{VIA}[v, s] := \{u\}$
 5.      for each $v_i \in N(v)$ do
 6.        send $decrease(v, s, \mathtt{D}[v, s])$ to $v_i$
 7.    end
 8.  else
 9.    if $\mathtt{D}[v, s] = w(v, u) + \mathtt{D}[u, s]$ then
10.      $\mathtt{VIA}[v, s] := \mathtt{VIA}[v, s] \cup \{u\}$    Line 10: phase EXTEND-VIA
```

Fig. 3.

```
Event: node $v$ receives the message $increase(u, s)$ from $u$
Procedure INCREASE
 1.  if $u \in \mathtt{VIA}[v, s]$ then
 2.    begin
 3.      $\mathtt{VIA}[v, s] := \mathtt{VIA}[v, s] \setminus \{u\}$    Line 3: phase REDUCE-VIA
 4.      if $\mathtt{VIA}[v, s] \equiv \emptyset$ then
 5.        begin    Lines 5-17: phase REBUILD-TABLE
 6.          $\mathtt{old\_distance} := \mathtt{D}[v, s]$
 7.          for each $v_i \in N(v)$ do
 8.            receive $\mathtt{D}[v_i, s]$ by sending $get\text{-}dist(v, s)$ to $v_i$
 9.          $\mathtt{D}[v, s] := \min_{v_i \in N(v)} \{w(v, v_i) + \mathtt{D}[v_i, s]\}$
10.          $\mathtt{VIA}[v, s] := \{v_i \in N(v) | w(v, v_i) + \mathtt{D}[v_i, s] = \mathtt{D}[v, s]\}$
11.          for each $v_i \in N(v)$ do
12.            begin
13.              if $\mathtt{D}[v, s] > \mathtt{old\_distance}$ then
14.                send $increase(v, s)$ to $v_i$
15.              send $decrease(v, s, \mathtt{D}[v, s])$ to $v_i$    CRH$_2$
16.            end
17.        end
18.    end
```

Fig. 4.

of edge (u, v). In particular, if $w(v, u) + \mathtt{D}[u, s] < \mathtt{D}[v, s]$ (Line 1), then v needs to update its estimated distance to s. To this aim, v performs phase IMPROVE-TABLE, that updates $\mathtt{D}[v, s]$ and $\mathtt{VIA}[v, s]$ (Lines 3–4), and propagates the updated values to the nodes in $N(v)$ (Line 6). Otherwise, if $w(v, u) + \mathtt{D}[u, s] = \mathtt{D}[v, s]$ (Line 9), then u is a new estimated via for v wrt destination s, and hence v performs phase EXTEND-VIA, that simply adds u to $\mathtt{VIA}[v, s]$ (Line 10).

If a node v receives the message $increase(u, s)$, then it performs Procedure INCREASE in Figure 4. While performing INCREASE, v simply checks whether the message comes from a node in $\mathtt{VIA}[v, s]$ (Line 1). In the affirmative case, v needs to remove u from $\mathtt{VIA}[v, s]$. To this aim, v performs phase REDUCE-VIA (Line 3). As a consequence of this deletion, $\mathtt{VIA}[v, s]$ may become empty. In

```
Event: node $v_i$ receives the message $get\text{-}dist(v,s)$ from $v$
Procedure SEND-DIST
1.  if ($\mathtt{VIA}[v_i,s] \equiv \{v\}$) $\boxed{\text{CRT}_1}$
        or ($v_i$ is performing REBUILD-TABLE or IMPROVE-TABLE wrt destination $s$) $\boxed{\text{CRT}_2}$
2.  then send $\infty$ to $v$ $\boxed{\text{CRH}_1}$
3.  else send $\mathtt{D}[v_i,s]$ to $v$
```

Fig. 5.

this case, v performs phase REBUILD-TABLE, whose purpose is to compute the new estimated distance and via of v to s. To do this, v asks to each node $v_i \in N(v)$ for its current estimated distance, by sending message *get-dist*(v, s) to v_i (Lines 7–8). When v_i receives message *get-dist*(v, s) by v, it performs Procedure SEND-DIST in Figure 5. While performing SEND-DIST, v_i basically sends $\mathtt{D}[v_i, s]$ to v, unless one of the following two conditions holds:

1. $\mathtt{VIA}[v_i, s] \equiv \{v\}$;
2. v_i is performing REBUILD-TABLE or IMPROVE-TABLE wrt destination s.

The test of these two conditions is part of our strategy to reduce the cases in which the count-to-infinity phenomenon appears. The test is performed at Line 1 of SEND-DIST, where the conditions are labeled as CRT_1 and CRT_2, respectively (the acronym stands for Count Reducing Test). If CRT_1 or CRT_2 are true, then v_i sends ∞ to v. This action is performed at Line 2, and is labeled as CRH_1 (the acronym stands for Count Reducing Heuristic). More details on the strategy for the reduction of the count-to-infinity phenomenon are given in Section 3.3.

Once node v has received the answers to the *get-dist* messages by all its neighbors, it computes the new estimated distance and via to s (Lines 9–10). Now, if the estimated distance has been increased, v sends an *increase* message to its neighbors (Line 14). In any case, v sends to its neighbors the message *decrease* (Line 15), to communicate them $\mathtt{D}[v, s]$. This action, that we call CRH_2, is also part of our strategy to reduce the count-to-infinity phenomenon. In fact, at some point, as a consequence of CRH_1, v could have sent ∞ to a neighbor v_j. Then, v_j receives the message sent by v at Line 15, and it performs Procedure DECREASE to check whether $\mathtt{D}[v, s]$ can determine an improvement to the value of $\mathtt{D}[v_j, s]$.

3.2 Behavior of DUST

In what follows, we describe the behavior of DUST by giving two examples which focus on the steps performed by DUST when a sequence of only weight decrease or only weight increase operations occur. The aim of the first example is to show how procedure DECREASE behaves, while the aim of the second example is to show how procedure INCREASE behaves and how the count prevention heuristics work. The steps performed by DUST when sequences of mixed

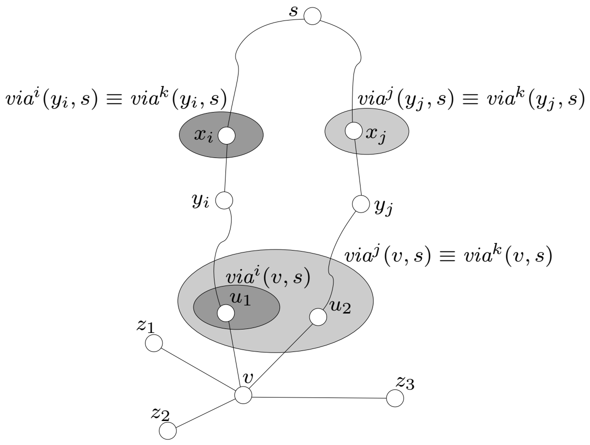

s
$via^i(y_i, s) \equiv via^k(y_i, s)$
$via^j(y_j, s) \equiv via^k(y_j, s)$
x_i
x_j
y_i
y_j
$via^j(v, s) \equiv via^k(v, s)$
$via^i(v, s)$
u_1
u_2
z_1
v
z_3
z_2

Fig. 6. Scenario for Case 1: only shortest paths from v to s in G^k are represented.

increase/decrease operations occur are a combination of those performed in the previous two cases.

In general, given a pair of nodes $s, v \in V$, at the end of the update sequence $\mathcal{C} = (c_1, c_2, ..., c_k)$, that is at time t_k, three cases may arise:

Case 1. The distance from v to s decreases, that is $d^0(v, s) > d^k(v, s)$;
Case 2. The distance from v to s increases, that is $d^0(v, s) < d^k(v, s)$;
Case 3. The distance from v to s does not change, that is $d^0(v, s) = d^k(v, s)$.

Notice that, in each of the above cases, the distance between v and s can change (increase or decrease) many times between t_0 and t_k. We analyze the three cases separately.

Case 1. $d^0(v, s) > d^k(v, s)$. A possible scenario for the shortest paths from v to s in G^k is shown in Figure 6, where the edges are represented as straight lines while the paths are represented as curves. There are two *weight decrease* operations c_i and c_j, occurring at times t_i and t_j, $t_i < t_j$, that decrease the weights of edges (x_i, y_i) and (x_j, y_j), respectively. As a consequence of these operations, the distance from v to s decreases and the edges (x_i, y_i) and (x_j, y_j) belong to the shortest paths from v to s in G^j. Here, we assume that c_i and c_j are the only operations in $\mathcal{C}$ that affect the shortest paths from v to s. This implies that $via^j(v, s) \equiv via^k(v, s)$ and $d^i(v, s) = d^j(v, s) = d^k(v, s)$. In detail, the shortest path from v to s in G^i is the path $P_i = v \rightarrow u_1 \rightsquigarrow y_i \rightarrow x_i \rightsquigarrow s$ and $via^i(v, s) \equiv \{u_1\}$, while the shortest paths from v to s in G^j are the paths P_i and $P_j = v \rightarrow u_2 \rightsquigarrow y_j \rightarrow x_j \rightsquigarrow s$, and $via^j(v, s) \equiv via^k(v, s) \equiv \{u_1, u_2\}$. We show that the algorithm starts by updating the routing table of nodes y_i and y_j, and then it updates the routing tables of the nodes in the sub-paths $v \rightarrow u_1 \rightsquigarrow y_i$ and $v \rightarrow u_2 \rightsquigarrow y_j$ of the shortest paths P_i and P_j, respectively.

When, at time t_i, the *weight decrease* operation c_i occurs, x_i and y_i send their estimated distances to each other. When x_i receives *decrease*$(y_i, s, \mathtt{D}_{t_i}[y_i, s])$ by y_i, it executes procedure Decrease in Figure 3, which first checks whether it is necessary to start the algorithm (see Line 1). In the affirmative case, x_i updates $\mathtt{D}[x_i, s]$ and $\mathtt{VIA}[x_i, s]$ (see Lines 3–4) at a certain time t and sends the message *decrease*$(x_i, s, \mathtt{D}_t[x_i, s])$ to its neighbors (Line 6). The behavior of y_i (when y_i receives the message *decrease*$(x_i, s, \mathtt{D}_{t_i}[x_i, s])$) is symmetric. At most one between x_i and y_i will propagate the *decrease* messages. In fact, if we assume, without loss of generality, that $\mathtt{D}_{t_i}[x_i, s] \leq \mathtt{D}_{t_i}[y_i, s]$, then the test performed by x_i at Line 1 of Procedure Decrease is false. Thus, x_i does not need to 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 its routing table at a certain time t and sends the message *decrease*$(y_i, s, \mathtt{D}_t[y_i, s])$ to its neighbors. If we assume that $\mathtt{D}_{t_j}[x_j, s] \leq \mathtt{D}_{t_j}[y_j, s]$, the behavior of nodes x_j and y_j at time t_j is analogous to that of nodes x_i and y_i at time t_i.

The *decrease* messages sent by y_i (y_j respectively) will update the values of $\mathtt{D}[u, s]$ and $\mathtt{VIA}[u, s]$, for each node u in the subpath from y_i (y_j respectively) to v of P_i (P_j respectively). Let us denote as m_i and m_j the *decrease* messages received by v, and propagated along paths P_i and P_j respectively. Further, let us denote as t_{m_i} and t_{m_j} the time when v receives m_i and m_j respectively. Note that, in an asynchronous system, even if $t_i < t_j$, nothing is known about the ordering of t_{m_i} and t_{m_j}. Let us assume that $t_{m_i} < t_{m_j}$, the case where $t_{m_i} > t_{m_j}$ is symmetric. When, at time t_{m_i}, v receives m_i, it performs improve-table phase of procedure Decrease and then, it updates $\mathtt{D}[v, s]$ and $\mathtt{VIA}[v, s]$ by setting $\mathtt{D}[v, s] = w(u, v) + d^i(u, s) = d^k(v, s)$ and $\mathtt{VIA}[v, s] \equiv \{u_1\} \equiv via^i(v, s)$. At time t_{m_j}, v receives m_j and performs extend-via phase of procedure Decrease and then, it adds u_2 to $\mathtt{VIA}[v, s]$ setting $\mathtt{VIA}[v, s] \equiv \{u_1, u_2\} \equiv via^j(v, s) \equiv via^k(v, s)$. Finally, v sends the message *decrease*$(v, s, \mathtt{D}^k[v, s])$ to nodes in $N(v)$. At this point, node v has correctly computed its current distance and via to s, and it has propagated this information to its neighbors.

Case 2. $d^0(v, s) < d^k(v, s)$. A possible scenario for the shortest paths from v to s in G^0 is represented in Figure 7. In this figure, we consider only *weight increase* operations that occur on edges (x_{i_1}, y_{i_1}), (x_{i_2}, y_{i_2}), (x_{j_1}, y_{j_1}) and (x_{j_2}, y_{j_2}). Note that, $via^0(y_{i_1}, s) \equiv \{x_{i_1}, x_{i_2}\}$ and $via^0(y_{j_1}, s) \equiv \{u_4, x_{j_1}, x_{j_2}\}$.

Since $d^0(v, s) < d^k(v, s)$, each shortest path from v to s in G^0 contains an edge that has been increased by a *weight increase* operation. Moreover, there exist nodes y_i such that, for each $x_i \in via^0(y_{j_1}, s)$, edge (x_i, y_i) has been increased. We call the set of such nodes Y_s. For example, in the scenario of Figure 7, $Y_s \equiv \{y_{i_1}\} \equiv \{y_{i_2}\}$.

In order to update its estimated distance and via to s, node v has to perform Procedure Increase. In particular, v has to perform reduce-via phase to remove from $\mathtt{VIA}[v, s]$ nodes that no longer belong to *via*(v, s). Moreover, if $\mathtt{VIA}[v, s]$ becomes empty, v has to perform rebuild-table phase to update its routing table and propagate the *increase* messages. In what follows we describe

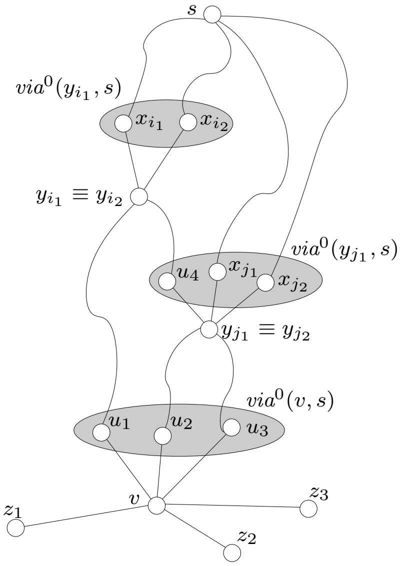


Fig. 7. Scenario for Case 2: only shortest paths from v to s in G^0 are represented.

how nodes that increase their distance to s at time t_k perform Procedure Increase. First, (I) we show that nodes in Y_s starts the updating phase by performing rebuild-table phase; then, (II) we show that node v is reached by the algorithm and it also performs rebuild-table phase; finally, (III) we show how v updates its routing table.

(I) For each $i \in \{1, 2, \ldots, k\}$, such that $y_i \in Y_s$, $x_i \in via^0(y_i, s)$ and c_i is a *weight increase* operation, at time t_i, nodes x_i and y_i send an *increase* message to each other[1]. Each time that y_i receives an *increase* message m, it removes from $\mathtt{VIA}[y_i, s]$ the node of $via^0(y_i, s)$ that sent m (see phase reduce-via). Hence, if $t_\emptyset(y_i, s)$ is the time when y_i has received all the *increase* messages coming from all the nodes in $via^0(y_i, s)$, then $\mathtt{VIA}_{t_\emptyset(y_i,s)}[y_i, s] \equiv \emptyset$. At time $t_\emptyset(y_i, s)$, y_i performs test at Line 4 of Procedure Increase, and the test returns true. Hence, y_i performs rebuild-table phase which updates the routing table of y_i and propagates the *increase* messages. More details on this procedure are discussed later when we analyze the behavior of a generic node v. For example, in Figure 7, node y_{i_1} sends an *increase* message to each node in $N(y_{i_1})$ and hence, to neighbors of y_{i_1} in the shortest paths from v to s in G^0 that contain y_{i_1}.

(II) It can been shown by induction that each node in the shortest paths from y_{i_1} to v and from y_{j_1} to v has the same behavior of y_{i_1}, that is it performs rebuild-table phase and sends *increase* messages to its neighbors. It

[1] If the operation on edge (x_i, y_i) is a *delete* operation, then y_i (x_i, respectively) cannot receive any message by x_i (y_i). In this case, y_i (x_i) simulates the reception of the *increase* message by x_i (y_i) by sending such message to itself.

follows that node v is eventually reached by *increase* messages sent by nodes in $via^0(v,s)$.

(III) Now we can analyze in detail the execution of REBUILD-TABLE phase by a generic node v at a certain time t. In order to update $\mathtt{D}[v,s]$ and $\mathtt{VIA}[v,s]$, v needs to know the estimated distances to s of each node in $N(v)$, that is, $\mathtt{D}_t[v_i,s]$, for each $v_i \in N(v)$. In Figure 7, node v needs the estimated distance of each z_i and u_j, $i,j \in \{1,2,3\}$. To this aim, v sends the message *get-dist*(v,s) to v_i, for each $v_i \in N(v)$ (Line 8 of Procedure INCREASE). When, at a time t_{v_i}, v_i receives *get-dist*(v,s), it performs Procedure SEND-DIST. Note that, in our model, multiple *increase* and *decrease* messages on a single node are processed one by one, while *get-dist* messages are processed immediately. To analyze the behavior of nodes in $N(v)$ when they receive *get-dist* messages, let us assume that, in the scenario of Figure 7, the following conditions hold:

- Node z_1 satisfies CRT$_1$, that is at time t_{z_1}, $\mathtt{VIA}_{t_{z_1}}[z_1,s] \equiv \{v\}$;
- Node z_2 satisfies CRT$_2$, that is at time t_{z_2}, z_2 is performing REBUILD-TABLE phase or IMPROVE-TABLE phase wrt destination s.

Under these hypothesis, the answer to *get-dist*(v,s) messages of nodes z_1 and z_2 is ∞, due to the execution of CRH$_1$, while we can assume that nodes u_1, u_2, u_3 and z_3 answer with their current estimated distances to s. By using the collected information, v updates $\mathtt{D}[v,s]$ and $\mathtt{VIA}[v,s]$ at Lines 9 and 10, respectively. Since z_1 and z_2 sent ∞ as their estimated distance to s, v does not consider such nodes as possible elements of $\mathtt{VIA}[v,s]$. Node z_2 will eventually send its current estimated distance to v as a *decrease* message by performing CRH$_2$. Moreover, at a certain time $\bar{t}$, node z_1 will receive an *increase* message sent by v at the end of REBUILD-TABLE phase. If z_1 does not send further messages to nodes in $N(z_1)$, then $\mathtt{VIA}_{\bar{t}}[z_1,s] \equiv \mathtt{VIA}_{t_{z_1}}[z_1,s] \equiv \{v\}$, and hence z_1 will perform REBUILD-TABLE phase and then it will send a *decrease* message to v containing the current estimated distance from z_1 to s, by performing CRH$_2$. At this point, node v has received all the information needed to correctly compute its current distance and via to s, and to propagate them to its neighbors.

Case 3. $d^0(v,s) = d^k(v,s)$. Two cases may occur:

1. if $d^{i-1}(v,s) = d^i(v,s)$, for each $i \in \{1,2,\ldots,k\}$, then the shortest paths from v to s in G^k are the same of G^0, that is v never updates $\mathtt{RT}_v[s]$.
2. If there exists $i \in \{1,2,\ldots,k\}$ such that $d^{i-1}(v,s) \neq d^i(v,s)$, then there exists $j \in \{1,2,\ldots,k\}$ such that if $d^{i-1}(v,s) < d^i(v,s)$ ($d^{i-1}(v,s) > d^i(v,s)$, respectively), then $d^{j-1}(v,s) > d^j(v,s)$ ($d^{j-1}(v,s) < d^j(v,s)$, respectively). In these cases, the algorithm propagates messages *decrease* and *increase* as in **Cases 1** and **2**.

In any case, v has correctly stored its current distance and via to s, and, possibly, it has propagated this information to its neighbors.

3.3 Comparison to existing algorithms

In what follows, we compare DUST with DBF, DUAL, INC and DEC in terms of number of messages sent and space occupancy per node.

Like many other distance vector algorithms, both DBF and DUST suffer of the count-to-infinity problem and hence the number of messages sent cannot be asymptotically bounded by a function of the size of the graph. However, DUST has been designed to heuristically reduce the number of cases where it counts to infinity. In fact, let us consider again the example of Figure 1 where DBF counts to infinity. In Figure 8 we show the few steps required by algorithm DUST to update both the distance and the via to s for each node in the graph G of Figure 1. When s and v detect the weight change on edge (s, v), they perform Procedure Increase with respect to source s. In particular, s does not perform REBUILD-TABLE phase as $\texttt{VIA}[s, s] = \emptyset$, while v does. When v gets the estimated distances to s from its neighbors (Line 8 of Procedure Increase), it receives ∞ from both a and b. This is due to the fact that since test CRT_1 returns true, a and b apply heuristic CRH_1. At the end of these execution, v correctly updates its routing table and sends messages *increase* and *decrease* to each neighbor. It is easy to verify that, when s receives the *decrease* and *increase* messages from v, it does not perform any local data modification. In contrast, both a and b perform Procedure Increase when they receive the *increase* message from v. In this case, a and b send ∞ (as their estimated distance to s) to each other in response to the *get-dist* message. This is due to the fact that the algorithm applies again CRH_1 but, in this case, CRH_1 is due to test CRT_2, which returns true. Hence, both a and b correctly update their routing tables by using messages sent by v. Subsequent messages sent by a and b do not produce further local data modification.

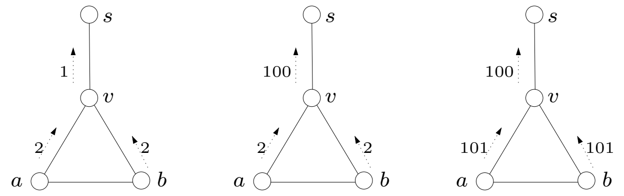


Fig. 8. The sequence of recomputations of $\texttt{D}[u, s]$ and $\texttt{VIA}[u, s]$, $u \in G$, performed by DUST.

In the previous example, there are nodes that apply action CRH_2. However, this action is never effective since the conditions of Lines 1 and 9 of Decrease are always false. In what follows, we describe how to modify the graph G in Figure 1 in order to fully appreciate our strategy to reduce the count-to-infinity phenomenon. Consider a graph G' obtained from G by adding the edge (s, a) with $w(s, a) = 98$. In this case, when a and b receive the *increase* message from v, they update their routing tables as follows: $\texttt{D}[a, s] = 98$ and $\texttt{D}[b, s] = 101$. At this point node a performs CRH_2 by sending a *decrease* message to b and to

v. When b and v perform Procedure Decrease, they set $\mathtt{D}[b,s] = \mathtt{D}[v,s] = 99$ and $\mathtt{VIA}[b,s] = \mathtt{VIA}[v,s] = \{a\}$. Notice that, DBF still counts to infinity on G', but, in this case, the counting stops after a number of updating that depends on the weight of the edge (s,a).

To conclude our comparison of DUST and DBF, we give an example where they both DBF and DUST count to infinity. Let us consider the graph of Figure 1 where the weight of edge (a,v) is set to 2. At time t_0, $\mathtt{D}[a,s] = 3$ and $\mathtt{VIA}[a,s] = \{v,b\}$. Now, the weight of edge (s,v) increases to 100. When v executes rebuild-table phase, it receives ∞ by b but it receives 3 from a and sets $\mathtt{D}[v,s] = 5$ and $\mathtt{VIA}[v,s] = \{a\}$. Then v sends *increase* messages to a and b. Let us assume that the message sent to a is received before that sent to b. Then, when a receives such message, it performs reduce-via and sets $\mathtt{VIA}[a,s] = \{v,b\} \setminus \{v\} = \{b\}$. When b receives the *increase* message by v it performs rebuild-table and sets $\mathtt{VIA}[b,s] = \{v\}$. In the resulting configuration $\mathtt{VIA}[\cdot,s]$ forms a loop, the count-to-infinity phenomenon occurs and the number of messages sent depends on the new weight on the edge (s,v).

Regarding the comparison with DUAL, as already observed, DUAL and DUST asymptotically requires the same space per node in the worst case, that is $O(n \cdot \mathit{maxdeg})$. However, in what follows we observe that in practice the memory requirement of DUST is smaller than that of DUAL. DUAL requires a node v to store, for each destination s, the estimated distance $\mathtt{D}[u,s]$ from each of its neighbors u, hence the space requirement of dual is exactly $O(n \cdot \mathit{deg}(v))$. DUST only needs the estimated distance of v to s and the set $\mathtt{VIA}[v,s]$. In the worst case, $\mathtt{VIA}[v,s]$ can have $\mathit{deg}(v)$ elements, but in practice this happens rarely (indeed, as we will experimentally show, it is not common to have more than one node in such set). It hence follows that the practical space requirements of DUST do not depend on the node degree.

In contrast to the space occupancy, the node degree affects the performances of DUST in terms of the number of messages sent. In fact, as DUST does not store the estimated distances of its neighbors, it needs to ask for them by sending *get-dist* message when they are needed. Hence, the number of *get-dist* messages sent by a node is proportional to the node degree. For this reason, in general DUAL sends a number of messages which is smaller than that of DUST. However, in real-world graphs the average node degree is small and then the number of *get-dist* messages sent is also small. In fact, we will experimentally show that in real-world graphs DUST sends less messages than DUAL.

Regarding the comparison with DEC and INC, we stress that such solutions are only incremental and decremental, respectively, and hence cannot cope with fully dynamic sequences of updates as the two algorithms use different data structure. In particular, INC has an optimal space occupancy per node as it stores only the routing table and it has a message complexity that is a factor *maxdeg* far from the optimal one. Such good theoretical bound of INC are due to the hypothesis that only *weight decrease* operations are assumed to occur in the network. In fact, INC is able to discard many messages thanks to a filtering condition which cannot be used in case of *weight increase* operations. DEC uses the same data structure of DUST and counts to infinity. However, DUST, in addition to being fully dynamic, is able to reduce the cases where the count-to-infinity occurs in sequences of *weight increase* operations, thanks to heuristic $\textsc{crh}_2$ which is not used in DEC.

4 Correctness

In this section, we prove the correctness of DUST. Let G be a graph and $\mathcal{C} = (c_1, c_2, ..., c_k)$ a sequence of *weight increase* and *weight decrease* operations occurring on G at times $t_1 \leq t_2 \leq ... \leq t_k$ and generating graphs $G^1, G^2, \ldots G^k$, respectively. We show that, for any pair of nodes v and s, there exists a time t_F such that for each time $t \geq t_F$ the routing table of v with respect to s is that of G^k, formally:

$$\texttt{D}_t[v, s] = d^k(v, s) \text{ and } \texttt{VIA}_t[v, s] \equiv \mathit{via}^k(v, s).$$

Such statement is proven in Theorem 1, the following lemmata are needed for the proof.

Lemma 1 *Given v, s in G such that $d^0(v, s) < d^k(v, s)$, there exists a node y such that, for each $x \in \mathit{via}^0(y, s)$, $w(x, y)$ has been increased by an operation in $\mathcal{C}$, and $d^0(y, s) < d^k(y, s)$.*

Proof. By contradiction, let us suppose that, for each $y \in \{y_1, y_2, \ldots, y_k\}$, either there exists $z \in \mathit{via}^0(y, s)$ such that $w(z, y)$ has not been increased by any *weight increase* operation, or $d^0(y, s) \geq d^k(y, s)$. This implies that nodes of G can only decrease the distance to s as a consequence of the weight changes $c_1, c_2, ..., c_k$. Hence, $d^0(v, s) \geq d^k(v, s)$, which is a contradiction. □

Let Y_s be the set of nodes y satisfying Lemma 1 with respect to destination s.

Lemma 2 *Given v, s in G such that $d^0(v, s) < d^k(v, s)$, there exists $y \in Y_s$ and a time $t_\emptyset(y, s)$ such that $\texttt{VIA}_{t_\emptyset(y,s)}[y, s] \equiv \emptyset$.*

Proof. By contradiction, let us suppose that, for each node y in Y_s and for each time t, $\texttt{VIA}_t[y, s] \not\equiv \emptyset$. Let y be a node in Y_s. Each node x in $\mathit{via}^0(y, s)$ sends an *increase* message to y as a consequence of the *weight increase* operation on edge (x, y). When y receives this message, it performs the REDUCE-VIA phase of Procedure INCREASE and deletes x from $\texttt{VIA}[y, s]$. Hence, there exists a time when $\texttt{VIA}[y, s]$ contains only nodes that have been added as a consequence of other *increase* or *decrease* messages received by y.

Furthermore, since $\texttt{VIA}[y, s]$ is never empty, the condition in line 4 of procedure INCREASE is always false and y never performs REBUILD-TABLE phase. This implies that for each pair of times t', t'' such that $t' < t''$, $\texttt{D}_{t'}[y, s] \geq \texttt{D}_{t''}[y, s]$ (that is, $\texttt{D}[y, s]$ decreases over time) and that each node u in $\texttt{VIA}[y, s]$ have been added to such set as a consequence of a *decrease* message $m = \mathit{decrease}(u, s, \texttt{D}_{t_u}[u, s])$ sent by u to y. If $\bar{t}_{i_0} > t_u$ is the time when y receives

m, we have $\mathtt{D}_{\bar{t}_{i_0}}[y,s] \geq w(y,u) + \mathtt{D}_{t_u}[u,s]$. Let P be the path from y to s containing u whose estimated weight is $w(y,u) + \mathtt{D}_{t_u}[u,s]$. Since y received m, P must contain an edge whose weight has been changed by an operation $c \in \mathcal{C}$. If c is a *weight decrease* operation, then P must contain also an edge whose weight has been increased by another operation in $\mathcal{C}$. In fact, since $\mathtt{D}[y,s]$ decreases over time, we have that $d^0(y,s) \geq \mathtt{D}_{\bar{t}_{i_0}}[y,s] \geq w(y,u) + \mathtt{D}_{t_u}[u,s]$, that is the estimated weight of P is smaller or equal to $d^0(y,s)$. But, since by hypothesis $d^0(y,s) < d^k(y,s)$, the actual weight of P in G^k is such that $\mathit{weight}^k(P) > d^0(y,s)$. Hence, in any case, P contains an edge whose weight has been increased by another operation in $\mathcal{C}$, that is P has the following structure

$$P = y \rightarrow u \rightsquigarrow \bar{y} \rightarrow \bar{x} \rightsquigarrow s$$

where edge $(\bar{x},\bar{y})$ is the edge whose weight has been increased. Since $d^0(y,s) < d^k(y,s)$, then all paths having the following structure

$$P' = y \rightsquigarrow \bar{y} \rightsquigarrow s$$

where $y \rightsquigarrow \bar{y}$ is the subpath of P from y to $\bar{y}$, satisfy $\mathit{weight}^k(P') > d^0(y,s)$. It follows that, $d^0(\bar{y},s) < d^k(\bar{y},s)$. The set $\mathit{via}^0(\bar{y},s)$ consists of two disjoint subsets, $\bar{X}(\bar{y},s) \equiv \{x \mid w(x,\bar{y})$ has been increased by an operation in $\mathcal{C}\}$ and $\bar{U}(\bar{y},s) \equiv \mathit{via}^0(\bar{y},s) \setminus \bar{X}(\bar{y},s)$, where $\bar{X}(\bar{y},s) \not\equiv \emptyset$, since $\bar{x} \in \bar{X}$. If $\bar{U}(\bar{y},s) \equiv \emptyset$, then $\bar{y} \in Y_s$. If $\bar{U}(\bar{y},s) \not\equiv \emptyset$, then, for each node $\bar{u} \in \bar{U}(\bar{y},s)$, there exists a path from $\bar{u}$ to s that contains a node in Y_s. In any case, there exists a path from y to s having the following structure

$$P_0 = y \rightarrow u \rightsquigarrow y_{i_1} \rightarrow x_{i_1} \rightsquigarrow s$$

where edge (x_{i_1}, y_{i_1}) is an edge whose weight has been increased. Furthermore $y_{i_1} \in Y_s$, hence the same arguments can be used to show that there exists a path:

$$P_1 = y_{i_1} \rightarrow u_{i_1} \rightsquigarrow y_{i_2} \rightarrow x_{i_2} \rightsquigarrow s$$

where the edge (x_{i_2}, y_{i_2}) is involved in a *weight increase* operation, $y_{i_2} \in Y_s$ and $\mathtt{D}_{\bar{t}_{i_1}}[y_{i_1},s] \geq w(y_{i_1},u_{i_1}) + \mathtt{D}_{t_{u_{i_1}}}[u_{i_1},s]$, $t_{u_{i_1}} < t_{y_{i_1}}$. In general, for each $y_j \in Y_s$ there exists a path:

$$P_j = y_j \rightarrow u_j \rightsquigarrow y_{j+1} \rightarrow x_{j+1} \rightsquigarrow s$$

where the edge (x_{j+1}, y_{j+1}) is involved in a *weight increase* operation, $y_{j+1} \in Y_s$ and $\mathtt{D}_{t_j}[y_j,s] \geq w(y_j,u_j) + \mathtt{D}_{t_{u_j}}[u_j,s]$, $t_{u_j} < t_j$. That is, since $|Y_s| < \infty$, there exists a cycle:

$$y \rightarrow u \rightsquigarrow y_{i_1} \rightarrow u_{i_1} \rightsquigarrow y_{i_2} \rightarrow u_{i_2} \rightsquigarrow ... \rightsquigarrow y_{i_h} \rightarrow u_{i_h} \rightsquigarrow y$$

such that:

$$\mathtt{D}_{\bar{t}_{i_0}}[y,s] \geq w(y,u) + \mathtt{D}_{t_u}[u,s] > \mathtt{D}_{t_u}[u,s] \geq$$

$$\mathtt{D}_{\bar{t}_{i_1}}[y_{i_1},s] \geq w(y_{i_1},u_{i_1}) + \mathtt{D}_{t_{u_{i_1}}}[u_{i_1},s] > \mathtt{D}_{t_{u_{i_1}}}[u_{i_1},s] \geq$$

$$\dots$$

$$\mathtt{D}_{\bar{t}_{i_h}}[y_{i_h}, s] \geq w(y_{i_h}, u_{i_h}) + \mathtt{D}_{t_{u_{i_h}}}[u_{i_h}, s] > \mathtt{D}_{t_{u_{i_h}}}[u_{i_h}, s] \geq$$
$$\mathtt{D}_{\bar{t}'_{i_0}}[y, s],$$

where $\bar{t}_{i_0} > t_u > \bar{t}_{i_1} > t_{u_{i_1}} \dots > \bar{t}_{i_h} > t_{u_{i_h}} > \bar{t}'_{i_0}$, that is $\mathtt{D}_{\bar{t}_{i_0}}[y, s] > \mathtt{D}_{\bar{t}'_{i_0}}[y, s]$ with $\bar{t}_{i_0} > \bar{t}'_{i_0}$ that is a contradiction. Hence there exists $y \in Y_s$ and a time $t_\emptyset(y, s)$ such that $\mathtt{VIA}_{t_\emptyset(y,s)}[y, s] \equiv \emptyset$. □

Let us denote as $Y'_s \subseteq Y_s$ the set of nodes satisfying Lemma 2 with respect to destination s. The next Lemma states that condition in Line 13 of Procedure Increase is true for each y in Y'_s, and hence that y sends *increase* messages to nodes in $N(y)$.

Lemma 3 *Given v, s in G such that $d^0(v, s) < d^k(v, s)$, for each $y \in Y'_s$, y sends an* increase *message to each node in $N(y)$.*

Proof. Each node y in Y'_s performs REBUILD-TABLE phase at least once at time $t_\emptyset(y, s)$. By contradiction, let us suppose that each execution of REBUILD-TABLE phase does not send any *increase* message, that is the condition in line 13 of procedure Increase is always false, for each $y \in Y'_s$. Hence, for each $y \in Y'_s$ and for each $t_\emptyset(y, s)$, if we denote as t'_y the time when y performs line 9 of procedure Increase, there exists a node u in $N(y)$ and a time $t_u < t'_y$ such that:

$$\mathtt{D}_{t'_y}[y, s] = w(y, u) + \mathtt{D}_{t_u}[u, s] \leq \mathtt{D}_{t_\emptyset(y,s)}[y, s].$$

Let P_u be the path from y to s containing u whose estimated weight is $w(y, u) + \mathtt{D}_{t_u}[u, s]$. The same arguments used in the proof of Lemma 2 can be used to prove that P_u contains an edge whose weight has been increased by an operation in $\mathcal{C}$ and to derive a contradiction. □

The next Lemma states that each node v such that $d^0(v, s) < d^k(v, s)$ has the same behavior of nodes in Y'_s that is, v performs REBUILD-TABLE phase and sends *increase* message to each node in $N(v)$.

Lemma 4 *Given v, s in G such that $d^0(v, s) < d^k(v, s)$, v performs* REBUILD-TABLE *phase wrt source s and it sends an* increase *message to each node in $N(v)$.*

Proof. We denote as:

$$\mathcal{P}_s(v) = \{P = v \rightsquigarrow y \mid y \in Y'_s \wedge \exists\, P' = v \rightsquigarrow s : P \subseteq P'\}$$
$$L_s(v) = \max\{\ell(P) \mid P \in \mathcal{P}_s(v)\}$$

the proof is by induction on $L_s(v)$.
Inductive basis ($L_s(v) = 0$): a node v is such that $L_s(v) = 0$ and $d^0(v, s) < d^k(v, s)$ if and only if $v \in Y'_s$. By Lemma 2, v performs REBUILD-TABLE phase and by Lemma 3, it sends an *increase* message to each node in $N(v)$.
Inductive step: the inductive hypothesis is: each node v such that $L_s(v) \leq l - 1$ and $d^0(v, s) < d^k(v, s)$ performs REBUILD-TABLE phase and sends and *increase* message to each node in $N(v)$.

Let v be a node such that $L_s(v) = l$ and $d^0(v, s) < d^k(v, s)$ and u be a node in $N(v)$ such that:

$$\begin{array}{l} \mathtt{D}_{t_v}[v, s] = w(v, u) + \mathtt{D}_{t_u}[u, s] \\ u \in \mathtt{VIA}_{t_v}[v, s] \end{array}$$

at certain times t_v and t_u such that $t_u < t_v$.

Two cases may occur:

- $d^0(u, s) < d^k(u, s)$: by inductive hypothesis, u sends an *increase* message to v.
- $d^0(u, s) \geq d^k(u, s)$: let $\mathcal{P}_s(v, u, t_v)$ be the set of estimated shortest paths from v to s via u at time t_v. Since $d^0(v, s) < d^k(v, s)$, each path P in $\mathcal{P}_s(v, u, t_v)$ contains a node $y_P \in Y_s$. For each $P \in \mathcal{P}_s(v, u, t_v)$, y_P is such that $L_s(y_P) \leq l - 1$, then y_P sends an *increase* message to each node in $N(y_P)$. Thus the message is propagated in each path in $\mathcal{P}_s(v, u, t_v)$. Hence u sends an *increase* message to v.

In any case, each node in $\mathtt{VIA}_{t_v}[v, s]$ sends an *increase* message to v and then there exists a time t when $\mathtt{VIA}_t[v, s] \equiv \emptyset$ and v performs REBUILD-TABLE phase.

The same arguments can be used to show that v sends an increase message to each node in $N(v)$. □

In the remainder we will use the following further notation for each pair of nodes v and s:

- $Exe_l(v, s)$ denotes the last local execution by v of phases REBUILD-TABLE or IMPROVE-TABLE with respect to s;
- $t_l(v, s)$ is equal to t_1 if v never performs neither REBUILD-TABLE phase nor IMPROVE-TABLE phase, otherwise, $t_l(v, s)$ is the time when $Exe_l(v, s)$ performs Lines 9 and 10 of Procedure INCREASE or Lines 3 and 4 of procedure DECREASE.
- $t'_l(v, s)$ is the time when v modify $\mathtt{D}[v, s]$ for the last time, $t_l(v, s) \geq t'_l(v, s)$.

Lemma 5 *For each pair of nodes v and s and for any time $t \geq t'_l(v, s)$, $\mathtt{D}_t[v, s] \geq d^k(v, s)$.*

Proof. Two cases may occur:

1. v never updates $\mathtt{D}[v, s]$. For any time t, we have:

$$\mathtt{D}_t[v, s] = d^0(v, s).$$

As v never updates $\mathtt{D}[v, s]$, it never performs REBUILD-TABLE and it never sends *increase* messages, hence, by Lemma 4, $d^0(v, s) \geq d^k(v, s)$. Then, for each time t:

$$\mathtt{D}_t[v, s] \geq d^k(v, s).$$

2. v updates $\mathtt{D}[v,s]$ at least once. By contradiction, let us suppose that v is the first node failing to update its routing table and let $t_v \geq t'_l(v,s)$ be the smallest time such that
$$\mathtt{D}_{t_v}[v,s] < d^k(v,s). \tag{1}$$
For each $z \in \mathtt{VIA}_{t_v}[v,s]$, $\mathtt{D}_{t_v}[v,s] = w(v,z) + \mathtt{D}_{t_z}[z,s]$, $t_z < t_v$.
If there exists a node $z \in \mathtt{VIA}_{t_v}[v,s]$ such that $t_z \geq t'_l(z,s)$, since v is the first node to fail, then $\mathtt{D}_{t_z}[z,s] \geq d^k(z,s)$. Thus $\mathtt{D}_{t_v}[v,s] = w(v,z) + \mathtt{D}_{t_z}[z,s] \geq w(v,z) + d^k(z,s) \geq d^k(v,s)$, a contradiction with respect to Equation 1. Hence, in what follows, we assume that $t_z < t'_l(z,s)$ for each $z \in \mathtt{VIA}_{t_v}[v,s]$.
If there exists a node $z \in \mathtt{VIA}_{t_v}[v,s]$ that never updates $\mathtt{D}[z,s]$, then, by Case 1, we have that $\mathtt{D}_{t_z}[z,s] \geq d^k(z,s)$ and then $\mathtt{D}_{t_v}[v,s] \geq d^k(v,s)$, a contradiction with respect to Equation 1. Hence, in what follows, we assume that $z \in \mathtt{VIA}_{t_v}[v,s]$ updates $\mathtt{D}[z,s]$ at least once.
When a node $z \in \mathtt{VIA}_{t_v}[v,s]$ updates $\mathtt{D}[z,s]$ at time $t'_l(z,s) > t_z$, it sends a *decrease* message m. When v receives m at time $t_m(z,v)$, it performs Procedure Decrease. If, at time $t'_l(z,s)$, z decreases the value of $\mathtt{D}[z,s]$, then, v performs improve-table as a consequence of m and modifies $\mathtt{D}[v,s]$ after the time $t'_l(v,s)$, a contradiction with respect to the definition of $t'_l(v,s)$. In fact, since $t_v \geq t'_l(v,s)$, after t_v, $\mathtt{D}_t[v,s]$ does not change and then $\mathtt{D}_t[v,s] = w(v,z) + \mathtt{D}_{t_z}[z,s]$ for each $t \geq t_v$. Hence, since $\mathtt{D}[z,s]$ as decreased, at time $t_m(z,v)$, the condition at Line 1 of Procedure Decrease is true and v performs improve-table as a consequence of m and modifies $\mathtt{D}[v,s]$ after the time $t'_l(v,s)$. Hence, let us assume that each node $z \in \mathtt{VIA}_{t_v}[v,s]$, at time $t'_l(z,s)$, increases the value of $\mathtt{D}[z,s]$.
Under this hypothesis, each node $z \in \mathtt{VIA}_{t_v}[v,s]$ also sends an *increase* message m' that is received by v at time $t_{m'}(z,v)$. In fact, the value of $\mathtt{D}[z,s]$ can increase only during rebuild-table and, if $\mathtt{D}[z,s]$ increases, the condition at Line 13 of Procedure Increase is true.
Since $t_v \geq t'_l(v,s)$, after t_v, v can only perform phases rebuild-table, reduce-via and extend-via. Let $\mathrm{Ext}(v,s)$ be the set of nodes added to $\mathtt{VIA}[v,s]$ after t_v as a consequence of an rebuild-table or a extend-via phase performed by v. We can assume that each node z in $\mathrm{Ext}(v,s)$ fulfills the same properties of nodes in $\mathtt{VIA}_{t_v}[v,s]$, that is:
   (a) $t_z < t'_l(z,s)$,
   (b) z updates $\mathtt{D}[z,s]$ at least once,
   (c) at time $t'_l(z,s)$, z increases the value of $\mathtt{D}[z,s]$ and then it sends an *increase* message m' that is received by v at time $t_{m'}(z,v)$.

   Let $t_{max} = \max\{t_{m'}(z,v) \mid z \in \mathtt{VIA}_{t_v}[v,s] \cup \mathrm{Ext}(v,s)\}$. Informally, t_{max} is the time when v receives the last *increase* message from nodes in $\mathtt{VIA}_{t_v}[v,s] \cup \mathrm{Ext}(v,s)$. It follows that, at time t_{max}, v performs Procedure Increase and tests at Lines 1 and 4 return true. Then, v performs phase rebuild-table at time $t_{max} > t_v \geq t_l(v,s)$. Furthermore, since each $z \in \mathtt{VIA}_{t_v}[v,s] \cup \mathrm{Ext}(v,s)$ increases the value of $\mathtt{D}[z,s]$, v increases $\mathtt{D}[v,s]$ after the time $t'_l(v,s)$, a contradiction with respect to the definition of $t'_l(v,s)$.

□

Corollary 1 *For each pair of nodes v and s and for any time $t \geq t_l(v,s)$, $\mathtt{D}_t[v,s] \geq d^k(v,s)$.*

Proof. Since $t_l(v,s) \geq t'_l(v,s)$, the statement follows directly by Lemma 5 □

Remark 1 Let v be a node in V and z be a node in $N(v)$. If there exists a pair of times t', t'' such that:

- $t' \leq t''$;
- for each t_z, $t' \leq t_z \leq t''$, $\mathtt{D}_{t_z}[z,s] = \bar{d}$ where $\bar{d}$ is a certain real value;
- at time t_v, $t' \leq t_v \leq t''$, $\mathtt{D}_{t_v}[v,s] \leq w(v,z) + \bar{d}$.

Then, for each time t, $t_v \leq t \leq t''$, $\mathtt{D}_t[v,s] \leq w(v,z) + \bar{d}$.

Theorem 1 *There exists t_F such that, for each pair of nodes $v, s \in V$, and for each time $t \geq t_F$,*

$$\mathtt{D}_t[v,s] = d^k(v,s) \text{ and } \mathtt{VIA}_t[v,s] \equiv via^k(v,s).$$

Proof. The correctness of the algorithm is shown with respect to a fixed source s. The correctness for all pairs of nodes is a straightforward consequence. In fact, since procedures Decrease, Increase and Send-Dist always refer to the record of the routing table related to a single source, then two executions of the algorithm related to two different sources cannot access the same record of the routing table.

If the statement is true for nodes v and s, then we denote as $t_F(v,s)$ the time when the statement occurs for v and s. If there exists $t_F(v,s)$ for each $v, s \in V$, then $t_F = \max_{v \in V}(t_F(v,s))$. Now we show that $t_F(v,s)$ exists for a generic pair (v,s). We need the following definitions:

$$\mathcal{P}_s(v) = \{P = v \rightsquigarrow s \mid P \text{ is a shortest path in } G^k\}$$
$$L_s(v) = \max\{\ell(P) \mid P \in \mathcal{P}_s(v)\}$$

the proof is by induction on $L_s(v)$.
Inductive basis ($L_s(v) = 0$): the unique node such that $L_s(v) = 0$ is s. For any time $t \leq t_1$, we have:

$$\mathtt{D}_t[s,s] = d^0(s,s) = 0$$
$$\mathtt{VIA}_t[s,s] \equiv via^0(s,s) \equiv \emptyset.$$

If s never changes $\mathtt{RT}_s[s]$, then $t_F(s,s) = t_1$. Hence we have to show that s does not perform any of the following phases: improve-table, extend-via, reduce-via and rebuild-table. Node s can perform:

- improve-table or extend-via as a consequence of a *decrease* message;
- reduce-via and rebuild-table as a consequence of an *increase* message.

Let t_m be the time when s receives the first *decrease* or *increase* message m.

- if $m = \mathit{decrease}(z, s, \mathtt{D}_{t_z}[z, s])$, where $z \in N(s)$ and $t_z < t_m$, then, since $\mathtt{D}_{t_m}[s, s] = 0$ and $w(s, z) > 0$, the condition in lines 1 and 9 of procedure Decrease are false. Thus v does not perform neither improve-table nor extend-via phase.
- if $m = \mathit{increase}(z, s)$, where $z \in N(s)$, then, since $\mathtt{VIA}_{t_m}[s, s] \equiv \emptyset$, the condition in line 1 of procedure Increase is false. Thus v does not perform neither reduce-via nor rebuild-table phase.

In any case, s does not change $\mathtt{RT}_s[s]$ then, if s receives further messages, the same arguments can be used to prove the statement. Hence $t_F(s, s) = t_1$.
Inductive step: the inductive hypothesis is: each node v such that $L_s(v) \leq l-1$ correctly assigns $\mathtt{D}_{t_F(v,s)}[v, s]$ and $\mathtt{VIA}_{t_F(v,s)}[v, s]$. Let v be a node such that $L_s(v) = l$. Each node u in $\mathit{via}^k(v, s)$ satisfy the inductive hypothesis.

Now we show that there exists a time $t_{\mathtt{D}}(v, s)$ such that, for each $t \geq t_{\mathtt{D}}(v, s)$, $\mathtt{D}_t[v, s] = d^k(v, s)$. We analyze different cases according to the behavior of nodes in $\mathit{via}^k(v, s)$.

- If each node u in $\mathit{via}^k(v, s)$ does not send any message to v, then, it does not perform rebuild-table and improve-table phases and then it never changes $\mathtt{D}[u, s]$. Hence, by inductive hypothesis, for each time t, $\mathtt{D}_t[u, s] = d^0(u, s) = d^k(u, s)$. Furthermore, since u does not change its distance to s, we have $d^0(v, s) \leq d^k(v, s)$. Hence, at time t_1:

$$\mathtt{D}_{t_1}[v, s] = d^0(v, s) \leq d^k(v, s) = w(v, u) + d^k(u, s)$$

  and, since, for each time $t \geq t_1$, $\mathtt{D}_t[u, s]$ does not change, by Remark 1, we have:

$$\mathtt{D}_t[v, s] \leq w(v, u) + d^k(u, s) = d^k(v, s).$$

  Furthermore, by Corollary 1, for each $t \geq t_l(v, s)$:

$$\mathtt{D}_t[v, s] \geq d^k(v, s).$$

  Thus $t_{\mathtt{D}}(v, s) = \max\{t_1, t_l(v, s)\}$.
- If at least a node in $\mathit{via}^k(v, s)$ sends a message to v, then, let $u_1, u_2, ..., u_h$ be the nodes in $\mathit{via}^k(v, s)$ that send a message to v. By inductive hypothesis, each u_i sends to v at least one of the messages: $\mathit{increase}(u_i, s)$ or $\mathit{decrease}(u_i, s, d^k(u_i, s))$. Let m_{u_i} be the last of such messages sent by u_i and let $m = \mathit{increase}(u, s)$ or $m = \mathit{decrease}(u, s, d^k(u, s))$ be the first message among m_{u_i}, $1 \leq i \leq h$, received by v.
  When v receives m, it performs the procedure Increase or Decrease and, as a consequence, at time t_m stores:

$$\mathtt{D}_{t_m}[v, s] \leq w(v, u) + d^k(u, s).$$

  Since m is the last message sent by u, for each time $t \geq t_m$, $\mathtt{D}_t[u, s]$ does not change and then, by Remark 1, we have:

$$\mathtt{D}_t[v, s] \leq w(v, u) + d^k(u, s) = d^k(v, s).$$

Furthermore, by Corollary 1, for each $t \geq t_l(v,s)$:

$$\mathtt{D}_t[v,s] \geq d^k(v,s).$$

Thus: $t_{\mathtt{D}}(v,s) = \max\{t_m, t_l(v,s)\}$.

Now we show that there exists a time $t_{\mathtt{VIA}}(v,s)$ such that, for each $t \geq t_{\mathtt{VIA}}(v,s)$, $\mathtt{VIA}_t[v,s] \equiv \mathit{via}^k(v,s)$.

Let u be a node in $\mathit{via}^k(v,s)$ and t_{m_u} be the time when v receives the last message m_u giving a correct estimated distance from u to s or a time before t_1 if u never updates $\mathtt{D}[u,s]$. Now we analyze different cases according to t_{m_u}. For each of such cases we show that u is added to $\mathtt{VIA}[v,s]$ and is never removed anymore.

- $t_{m_u} > t_l(v,s)$. We now show that $m_u = \mathit{decrease}(u,s,d^k(u,s))$. By contradiction, let us suppose that $m_u = \mathit{increase}(u,s)$. Then, since u sends an *increase* message only if test at Line 13 of procedure Decrease is true, there exists a time $\tilde{t}$ such that, for each t, $t_1 \leq \tilde{t} \leq t < t_l(v,s)$, $\mathtt{D}_t[u,s] < d^k(u,s)$ and $\tilde{t}$ is the smallest time that fulfill this condition. For each of the following cases we obtain a contradiction:
  - if $\tilde{t} < t_l(v,s)$, then by Remark 1:

    $$\mathtt{D}_{t_l(v,s)}[v,s] \leq w(v,u) + \mathtt{D}_{\tilde{t}}[u,s] < w(v,u) + d^k(u,s) = d^k(v,s)$$

    that is a contradiction with respect to Corollary 1.
  - If $\tilde{t} \geq t_l(v,s)$, then at time $\tilde{t}$, u sends the message $\mathit{decrease}(u,s,\mathtt{D}_{\tilde{t}}[u,s])$ that is received by v at time $\bar{t} > \tilde{t}$. At time $\bar{t}$, we have:

    $$\mathtt{D}_{\bar{t}}[v,s] = d^k(v,s) = w(v,u) + d^k(u,s) < w(v,u) + \mathtt{D}_{\tilde{t}}[u,s].$$

    Hence the condition in line 1 of procedure Decrease is true and v performs improve-table phase at a time greater then $t_l(v,s)$ that is a contradiction with respect to the definition of $t_l(v,s)$.

  Hence, $m_u = \mathit{decrease}(u,s,d^k(u,s))$.
  Since $t_{m_u} > t_l(v,s)$, then $\mathtt{D}_{t_{m_u}}[v,s] = d^k(v,s) = w(v,u) + d^k(u,s)$. Hence, when v receives m_u, the condition in line 9 of procedure Decrease is true and then v adds u to $\mathtt{VIA}[v,s]$ at time t_u.
  Since m_u is the last message sent by u to v and, at time $t > t_u$, v has performed $Exe_l(v,s)$, u will not be removed from $\mathtt{VIA}[v,s]$.
- $t_{m_u} < t_l(v,s)$. We now show that $Exe_l(v,s)$ performs phase rebuild-table. By Remark 1, we have that, for each $t \geq t_{m_u}$,

  $$\mathtt{D}_t[v,s] \leq w(v,u) + d^k(u,s) = d^k(v,s).$$

  By contradiction, let us suppose that $Exe_l(v,s)$ performs improve-table as a consequence of a *decrease* message received at time $\tilde{t} \geq t_{m_u}$. Then the condition at Line 1 of procedure Decrease is true and, at time $t_l(v,s)$, v assigns a value of $\mathtt{D}[v,s]$ such that:

  $$\mathtt{D}_{t_l(v,s)}[v,s] < \mathtt{D}_{\tilde{t}}[v,s] \leq d^k(v,s)$$

that is a contradiction with respect to Corollary 1. Hence $Exe_l(v,s)$ performs phase REBUILD-TABLE.

Since $t_{m_u} < t_l(v,s)$, $\mathtt{D}_{\bar{t}}[u,s] = d^k(u,s)$, where $\bar{t}$ is the time when, during $Exe_l(v,s)$, v receives $\mathtt{D}[u,s]$ as an answer to the message *get-dist*(v,s) sent to u. Hence v adds u to $\mathtt{VIA}[v,s]$ at time $t_u = t_l(v,s)$.

Since m_u is the last message sent by u to v and, at time $t > t_u$, v has performed $Exe_l(v,s)$, u will not be removed from $\mathtt{VIA}[v,s]$.

- $t_{m_u} = t_l(v,s)$. Since two events cannot occur to the same processor at the same time, we have $t_{m_u} = t_l(v,s) < t_1$ that is, neither u nor v changes its distance estimate to s and u does not sends any message to v. Thus $u \in via^0(v,s)$ and v never removes u from $\mathtt{VIA}[v,s]$. In this case we denote $t_u = t_1$.

Thus for each time $t \geq \max\{t_u \mid u \in via^k(v,s)\}$, each node in $via^k(u,s)$ belongs to $\mathtt{VIA}_t[u,s]$.

There may exist a node z in $N(v) \setminus via^k(v,s)$ that belongs to $\mathtt{VIA}_t[u,s]$ for a certain time t. Since $z \notin via^k(v,s)$, then $d^k(v,s) < w(v,s) + d^k(z,s)$. Hence v added z to $\mathtt{VIA}_t[v,s]$ as a consequence of an underestimated value of $\mathtt{D}[z,s]$. By Corollary 1, at time $t_l(z,s)$, z will increase its estimated distance and send to v an increase message $m_z = \mathit{increase}(z,s)$. As a consequence, v will perform phase REDUCE-VIA at time t_z. Thus:

$$t_{\mathtt{VIA}}(v,s) = \max\{\{t_u \mid u \in via^k(v,s)\} \cup \{t_z \mid z \in N(v) \setminus via^k(v,s)\}\}.$$

Hence for a generic node v we have: $t_F(v,s) = \max\{t_{\mathtt{D}}(v,s), t_{\mathtt{VIA}}(v,s)\}$. □

5 Experimental analysis

In this section, we report the results of our experimental study on the performance of DUST against DBF, DUAL, INC and DEC. In detail, we first describe the implementation platform and the executed tests, and then we analyze the results of our study.

Experimental environment. The experiments have been carried out on a workstation equipped with a 2.66 GHz processor (Intel Core2 Duo E6700 Box) and 8Gb RAM. The experiments consist of simulations within the OMNeT++ environment, version 4.0p1 [1]. OMNeT++ is an object-oriented modular discrete event network simulator, useful to model protocols, communication networks, multiprocessors and other distributed systems. An OMNeT++ model consists of hierarchically nested modules, that communicate through message passing. 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). 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 weight of the edge in the graph, and the latter simulates a finite but not null transmission time.

Executed tests. For our experiments we used both real-world and artificial instances of the problem. In detail, we used the *CAIDA IPv4 topology dataset* [15] and two classes of random graphs generated, respectively, by the *Barabási-Albert* algorithm [2] and the *Erdős-Rényi* algorithm [7].

CAIDA (Cooperative Association for Internet Data Analysis) is an association which provides data and tools for the analysis of the Internet infrastructure. The CAIDA dataset is collected by a globally distributed set of monitors. The monitors collect data by sending probe messages continuously to destination IP addresses. Destinations are selected randomly from each routed IPv4/24 prefix on the Internet such that a random address in each prefix is probed approximately every 48 hours. The current prefix list includes approximately 7.4 million prefixes. For each destination selected, the path from the source monitor to the destination is collected, in particular, data collected for each path probed includes the set of IP addresses of the hops which form the path and the Round Trip Times (RTT) of both intermediate hops and the destination.

We parsed the files provided by CAIDA to obtain a weighted undirected graph G_{IP} where a node represents an IP address contained in the dataset (both source/destination hosts and intermediate hops), edges represent links among hops and weights are given by RTTs. As the graph G_{IP} consists of $n \approx 35000$ nodes, we cannot use it for the experiments, as the amount of memory required to store the routing tables of all the nodes is $O(n^2 \cdot maxdeg)$ for any implemented algorithm. Hence, we performed our tests on connected subgraphs of G_{IP} induced by the settled nodes of a breadth first search starting from a node taken at random. We generated a set of different tests, each test consists of a dynamic graph characterized by: a subgraph of G_{IP} of 5000 nodes, a set of k concurrent edge updates, where k assumes values in $\{5, 10, \ldots, 100\}$. An edge update consists of multiplying the weight of a random selected edge by a value randomly chosen in $[0.5, 1.5]$. For each test configuration, we performed 5 different experiments and we report average values.

In order to test the implemented algorithms on random Internet-like topologies, we generated random networks with a power-law node degree distribution by using the Barabási–Albert algorithm. This kind of networks have been proven to model many real-world networks such as the Internet, the World Wide Web, citation graphs, and some social networks [3]. A Barabási–Albert topology is generated by iteratively adding one node at a time, starting from a given connected graph with at least two nodes. A newly added node is connected to any other existing nodes with a probability that is proportional to the degree of the existing nodes. Hence, the more connected a node is, the more likely it is to receive new connections to the new node. This mechanism is known as *preferential attachment* and it has been observed in many real-world networks. In detail, we randomly generated a set of different tests, where a test consists of a dynamic graph characterized by a Barabási–Albert random graphs G_{BA} of 5000 nodes and a set of k concurrent edge updates, where k assumes values in $\{5, 10, \ldots, 100\}$. Edge weights are non-negative real numbers randomly chosen in $[1, 10000]$. Edge updates are randomly chosen as in the CAIDA tests. For each test configuration, we performed 5 different experiments and we report average values.

Graphs G_{IP} and G_{BA} turn out to be very sparse (i.e. $m/n \approx 1.3$), so it is worth analyzing also dense graphs. To this aim we generated Erdős-Rényi random graphs. In detail, we randomly generated a set of different tests, where a test consists of a dynamic graph characterized by: an Erdős-Rényi random graphs G_{ER} of 1000 nodes; the density *dens* of the graph, computed as the ratio between m and the number of the edges of the n-complete graph; and the number k of edge update operations. We chose different values of *dens* ranging from 0.01 to 0.41. The number k assumes values in $\{30, 100, 1000\}$. Edge weights are non-negative real numbers randomly chosen in $[1, 10000]$. Edge updates are randomly chosen as in the CAIDA tests. For each test configuration, we performed 5 different experiments and we report average values.

Finally, in order to compare DUST with DEC and INC, we executed experiments on graphs G_{IP}, G_{BA} and G_{ER} with sequences of only *weight increase* operations or only *weight decrease* operations, respectively. In particular, we used the same graphs and the same number of weight change operations as in the previous experiments, but the edge weights are only increased or decreased by a value randomly chosen in $[1.01, 1.5]$ and $[0.01, 0.5]$, respectively.

Analysis. In our experiments on fully dynamic sequences of changes, we verified that DBF is always outperformed by both DUST and DUAL. In fact, it sends a number of messages that is a factor between 32 and 295 (24 and 166, respectively) higher than the number of messages sent by DUST (DUAL, respectively). The space occupancy per node required by DBF is slightly smaller than that required by DUAL and it is always at least 100 times higher than that required by DUST. Moreover, in the tests where $k \geq 25$, DBF always counts to infinity, while DUST and DUAL always converge to the correct routing tables (see Sections 2 and 3.3). For these reasons, we focus only on the comparison between DUST and DUAL, and the results of these experiments are shown in Figures 9–16.

In Figure 9 we report the number of messages sent by DUST and DUAL on subgraphs of G_{IP} having 5000 nodes and an average value of 6109 edges in the cases where the number k of modifications is in $\{5, 10, \ldots, 100\}$. The figure shows that DUST always sends less messages than DUAL. Figure 10 shows the results of Figure 9 from a different point of view, that is, it shows the ratio between the number of messages sent by DUAL and DUST. It is worth noting that the ratio is within 1.04 and 2.52 which means that DUAL sends a number of messages which is between 4% and 152% higher than the number of messages sent by DUST.

To conclude our analysis on G_{IP}, we experimentally analyze the space occupancy per node. As already observed, in real-world graphs it is not common to have more than one via to a destination. Thus, the memory requirement of DUST in G_{IP} is much smaller than that of DUAL. In particular, DUST requires in average 40000 bytes per node and 40266 bytes per node in the worst case. DUAL requires in average 186090 bytes per node and $5.2M$ bytes per

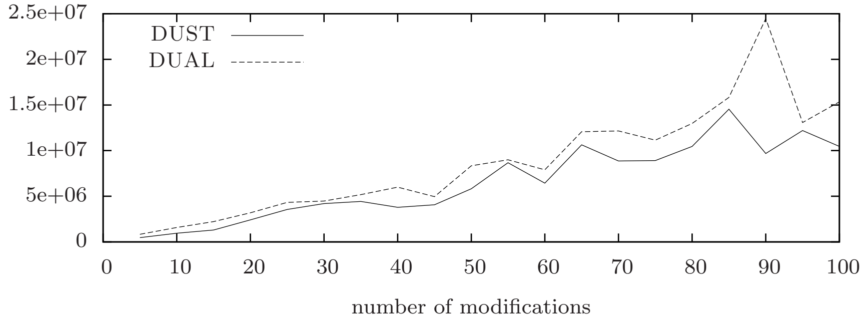


Fig. 9. Number of messages sent by DUST and DUAL on subgraphs of G_{IP}.

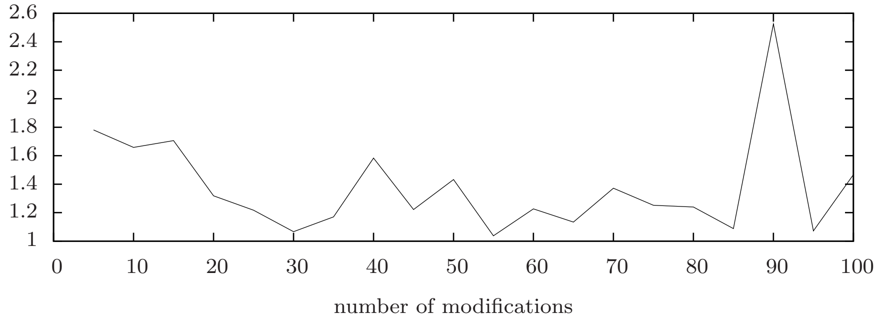


Fig. 10. Ratio between the number of messages sent by DUAL and DUST on subgraphs of G_{IP}.

node in the worst case. This implies that DUAL requires in average 4.65 times the space required by DUST and 129 times the space required by DUST in the worst case. The good performances of DUST in terms of memory consumption are also confirmed by the fact that, in our simulated environment, we have been able to run DUST on subgraphs of CAIDA networks with about 10000 nodes, while we have not been able to run such experiments for DUAL.

Regarding Barabási-Albert graphs, we performed similar tests as for G_{IP}. Figure 11 and 12 report the number of messages sent by DUST and DUAL and the ratio between the number of messages sent by DUAL and DUST on such graphs. In most of the cases, DUAL is better than DUST, in fact DUST sends about twice the number of messages sent by DUAL. This is due to the *get-dist* messages sent by DUST whose number grows proportionally to the average node degree. In contrast, DUAL does not send such messages but it needs to store for each node the estimated distances of all its neighbors. As G_{BA} graphs are in average denser than G_{IP} graphs, this determines an improvement in the number of messages sent and an increase in the space occupancy per node wrt DUST. In fact in G_{BA} graphs, DUST requires in

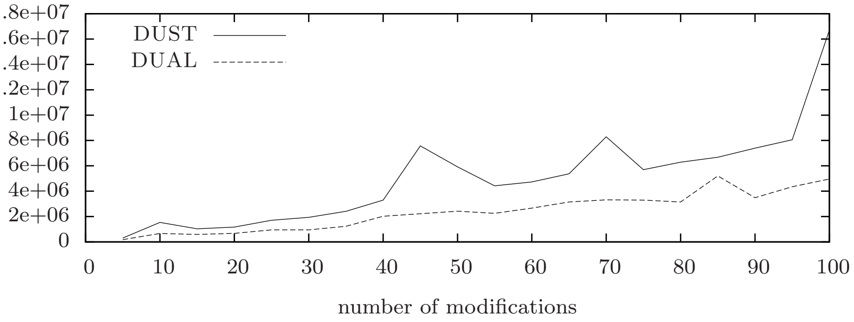


Fig. 11. Number of messages sent by DUST and DUAL on G_{BA} graphs.

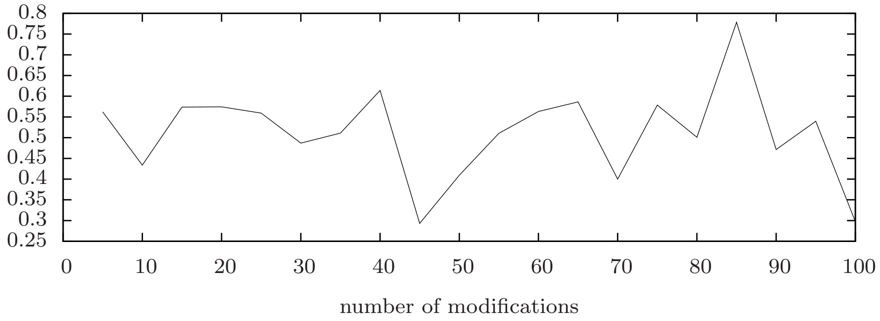


Fig. 12. Ratio between the number of messages sent by DUAL and DUST on G_{BA} graphs.

average 40000 bytes per node and 40014 bytes per node in the worst case. DUAL requires in average 187420 bytes per node and 5.6M bytes per node in the worst case. This implies that DUAL requires in average 4.69 times the space required by DUST and 141 times the space required by DUST in the worst case.

The good performance of DUST is mainly due to the topological structure of graphs G_{IP} and G_{BA}. In fact, as already observed, the node degree affects the performances of DUST and DUAL in terms of both the number of messages sent and space occupancy per node. Therefore, it is worth investigating how the two algorithms perform on dense graphs. Figure 13 shows the number of messages sent by DUST and DUAL on dynamic Erdős-Rényi random graphs with 1000 nodes, 1000 edge cost changes and *dens* ranging from 0.01 to 0.41 which leads to a number m of edges which ranges from about 5000 to about 200000. Figure 14 shows the ratio between the number of messages sent by DUAL and DUST in the same setting as in Figure 13.

As in the case of Barabási-Albert graphs, DUAL is better than DUST in terms of number of messages sent. It is more evident here that this is due to the *get-dist* messages of DUST. It is also more evident the increase in

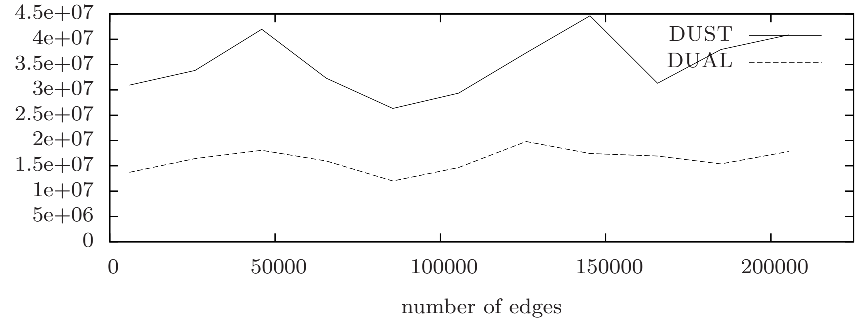


Fig. 13. Number of messages sent by DUST and DUAL on graphs G_{ER}.

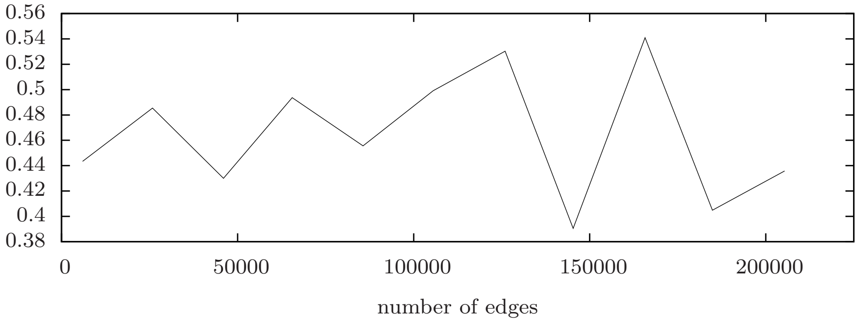


Fig. 14. Ratio between the number of messages sent by DUAL and DUST on graphs G_{ER}.

space occupancy required by DUAL, as highlighted by Figures 15 and 16. In detail, Figure 15 shows the ratio between the average space occupancy per node required by DUAL and DUST in G_{ER}, while Figure 16 shows the ratio between the worst case space occupancy per node required by DUAL and DUST. The average space occupancy ratio grows linearly with the number of edges as the space occupancy of DUST remains almost constant while the space occupancy of DUAL is proportional to the average node degree. The worst case space occupancy of DUAL grows very fast as in the executed tests where $dens > 0.10$ there exists at least a node v such that $deg(v) = n - 1$.

Figures 13–16 refer to the case where $k = 1000$, as it is the case where DUST performs worse. In cases where $k \in \{30, 100\}$ DUST performs better than the case where $k = 1000$, and hence they are not reported.

A summary of the performances of the three implemented algorithms in the three classes of graphs considered in the case of fully dynamic sequences is reported in Table 1, where worst case ratio for each parameter are reported.

To conclude our experimental analysis, we have compared the performances of DUST with those of DEC and INC on sequences of only *weight increase* operations and only *weight decrease* operations, respectively. The results of these

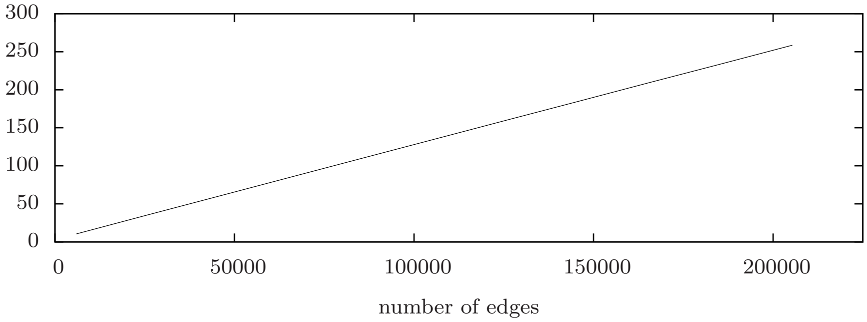


Fig. 15. Ratio between the space required by DUAL and DUST on graphs G_{ER} in the average case.

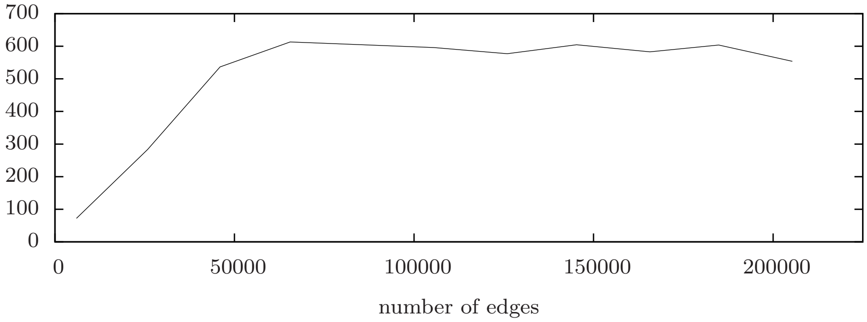


Fig. 16. Ratio between the space required by DUAL and DUST on graphs G_{ER} in the worst case.

	G_{IP}		G_{BA}		G_{ER}	
	msg	space	msg	space	msg	space
DUST	1	1	1	1	1	1
DBF	295	102	/	111	/	500
DUAL	2.52	129	0.48	141	0.43	613

Table 1. Ratio of messages sent (msg) and space occupancy per node (space) with respect to algorithm DUST, in graphs G_{IP}, G_{BA}, and G_{ER}. Message sent by DBF are not reported in the cases where it counts to infinity.

experiments on Barabási-Albert instances are shown in Figures 17 and 18. The results on CAIDA and Erdős-Rényi instances are similar and hence they are not reported. Figure 17 shows that DUST is slightly better than DEC in terms of number of messages sent. We recall that this is due to the application of heuristic CRH$_2$ of DUST which is not used in DEC. The differences in space occupancy per node is negligible as the two algorithms use the same data structures. Regarding INC, it has an almost optimal message complexity and, as expected, it sends less messages than DUST as shown in Figure 18.

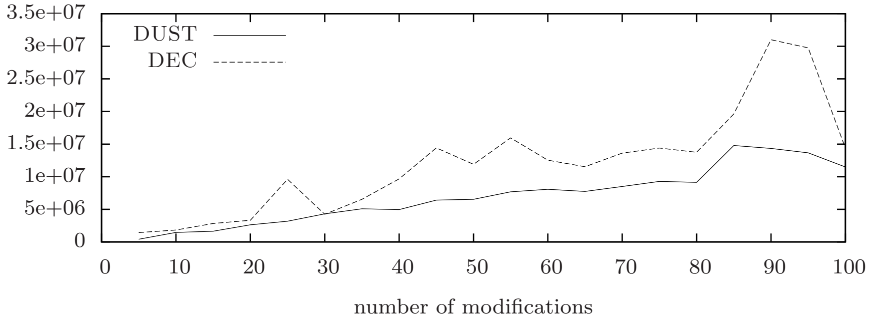


Fig. 17. Number of messages sent by DUST and DEC on graphs G_{BA} on sequences of only *weight increase* operations.

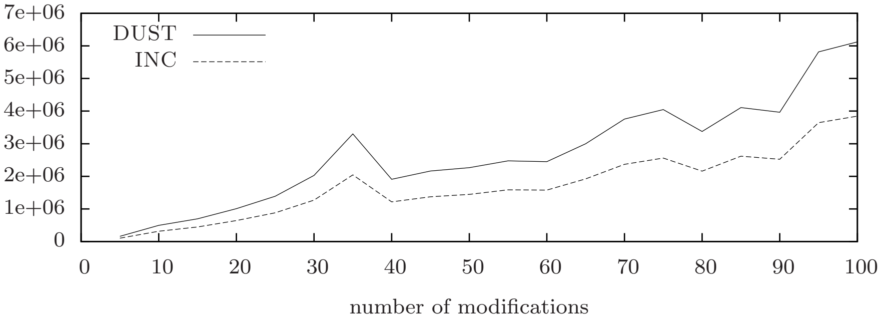


Fig. 18. Number of messages sent by DUST and INC on graphs G_{BA} on sequences of only *weight decrease* operations.

However the number of messages sent by DUST is at most 55% more that that sent by INC. Despite the theoretical bounds, we experimentally verified that DUST and INC have similar space occupancy, again this is due to the fact that it is not common to have more than one via to a destination.

6 Conclusion and future work

We have studied the problem of dynamically update *all-pairs shortest paths* in a distributed network while edge update operations concurrently occur to the network. We have provided a new algorithm, denoted as DUST, for this problem, and experimentally compared its performance on fully dynamic sequences of updates with respect to the most popular solutions in the literature: DBF and DUAL. Furthermore, we have compared DUST with the INC and DEC algorithms of [10] on sequences of only *weight decrease* operations and sequences of only *weight increase* operations, respectively. As input to the algorithms, we used both real-world (Internet-like) and artificial instances of the problem. The results of our experiments show that the space occupancy per node required by DUST is much smaller than that required by both DBF and DUAL. In terms of messages, DUST outperforms both DBF and DUAL on the real-world topologies, while in artificial instances, it sends a number of messages that is more than that of DUAL and much smaller than that of DBF. Regarding the comparison, with INC and DEC, the space occupancy of such algorithms is always comparable to that of DUST. In terms of number of messages sent, in the incremental case, DUST is slightly worse than INC and in the decremental case, it is slightly better than DEC.

A research direction which deserves further investigation is surely that of studying new efficient algorithms or heuristic improvements of the algorithms studied in this paper for the problem of distributed shortest paths. One way could be that of modifying DUST in order to avoid routing table loops. From this point of view, a possibility is that of embedding DUST into the loop-prevention framework given in [22]. Another challenging problem is that of concentrating on real-world networks, to investigate whether structural properties of these networks can lead to the development of efficient algorithms explicitly tailored to them.

Acknowledgments

Support for the IPv4 Routed/24 Topology Dataset is provided by National Science Foundation, US Department of Homeland Security, WIDE Project, Cisco Systems, and CAIDA.

References

1. Omnet++: the discrete event simulation environment. `http://www.omnetpp.org/`.
2. R. Albert and A.-L. Barabási. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.
3. R. Albert and A.-L. Barabási. Statistical mechanics of complex network. *Reviews of Modern Physics*, 74:47–97, 2002.
4. H. Attiya and J. Welch. *Distributed Computing.* John Wiley and Sons, 2004.
5. B. Awerbuch, A. Bar-Noy, and M. Gopal. Approximate distributed bellman-ford algorithms. *IEEE Transactions on Communications*, 42(8):2515–2517, 1994.
6. D. Bertsekas and R. Gallager. *Data Networks.* Prentice Hall International, 1992.
7. B. Bollobás. *Random Graphs.* Cambridge University Press, 2001.
8. S. Cicerone, G. D'Angelo, G. Di Stefano, and D. Frigioni. Partially dynamic efficient algorithms for distributed shortest paths. *Theoretical Computer Science*, 411:1013–1037, 2010.
9. S. Cicerone, G. D'Angelo, G. Di Stefano, D. Frigioni, and V. Maurizio. A new fully dynamic algorithm for distributed shortest paths and its experimental evaluation. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA 2010)*, volume 6049 of *Lecture Notes in Computer Science*, pages 59–70, 2010.
10. S. Cicerone, G. D'Angelo, G. D. Stefano, D. Frigioni, and A. Petricola. Partially dynamic algorithms for distributed shortest paths and their experimental evaluation. *Journal of Computers*, 2(9):16–26, 2007.
11. S. Cicerone, G. D. Stefano, D. Frigioni, and U. Nanni. A fully dynamic algorithm for distributed shortest paths. *Theoretical Computer Science*, 297(1-3):83–102, 2003.

12. D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000.
13. J. J. Garcia-Lunes-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Transactions on Networking*, 1(1):130–141, 1993.
14. P. A. Humblet. Another adaptive distributed shortest path algorithm. *IEEE Transactions on Communications*, 39(6):995–1002, Apr. 1991.
15. Y. Hyun, B. Huffaker, D. Andersen, E. Aben, C. Shannon, M. Luckie, and K. Claffy. The CAIDA IPv4 routed/24 topology dataset. `http://www.caida.org/data/active/ipv4_routed_24_topology_dataset.xml`.
16. G. F. Italiano. Distributed algorithms for updating shortest paths. In *Proceedings of the International Workshop on Distributed Algorithms*, volume 579 of *Lecture Notes in Computer Science*, pages 200–211, 1991.
17. J. McQuillan. Adaptive routing algorithms for distributed computer networks. Technical Report BBN Report 2831, Cambridge, MA, 1974.
18. J. T. Moy. *OSPF - Anatomy of an Internet routing protocol.* Addison-Wesley, 1998.
19. P. Narváez, K.-Y. Siu, and H.-Y. Tzeng. New dynamic algorithms for shortest path tree computation. *IEEE/ACM Transactions on Networking*, 8(6):734–746, 2000.
20. A. Orda and R. Rom. Distributed shortest-path and minimum-delay protocols in networks with time-dependent edge-length. *Distributed Computing*, 10:49–62, 1996.
21. K. V. S. Ramarao and S. Venkatesan. On finding and updating shortest paths distributively. *Journal of Algorithms*, 13:235–257, 1992.
22. S. Ray, R. Guérin, K.-W. Kwong, and R. Sofia. Always acyclic distributed path computation. *IEEE/ACM Transactions on Networking*, 18(1):307–319, 2010.
23. E. C. Rosen. The updating protocol of arpanet's new routing algorithm. *Computer Networks*, 4:11–19, 1980.
24. J. Wu, F. Dai, X. Lin, J. Cao, and W. Jia. An extended fault-tolerant link-state routing protocol in the internet. *IEEE Transactions on Computers*, 52(10):1298–1311, 2003.

APPENDIX

Example of execution

In what follows, we give an example of the execution of DBF, DUAL and DUST on a simple network (which is part of an example in [13]) where a weight increase operation occurs.

Figure 19 shows an example of DBF. The example focuses on the graph of Figure 19(a) and on destination s. In the figure, the value close to a node indicates its distance to node s and an arrowhead from x to y in edge (x, y) indicates that node y is the successor of x towards node s. An arrowhead from x to y close to edge (x, y) denotes that node x is sending a message to y containing the current distance from s to t, the value of such distance is reported close to the arrow.

At a certain point in time, edge (b, s) changes its weight from 2 to 10 (see Figure 19(a)). When node b detects the weight increase, it updates the value of $\mathtt{D}_b[b, s]$ to the minimal possible value, that is $\mathtt{D}_b[b, s] = \min_{u \in N(b)}\{w(b, u) + \mathtt{D}_b[u, s]\} = w(b, c) + \mathtt{D}_b[c, s] = 4$. Then, node b sends $\mathtt{D}_b[b, s]$ to all its neighbors (Figure 19(b)). As a consequence of such messages, nodes a and c update $\mathtt{D}_a[b, s]$ and $\mathtt{D}_c[b, s]$, respectively, compute their optimal distances to s that are 4 and 5, respectively, and send them to their own neighbors (Figure 19(c)). Nodes s and d only update $\mathtt{D}_s[b, s]$ and $\mathtt{D}_d[b, s]$, respectively. In Figure 19(d), node b updates $\mathtt{D}_b[c, s]$ to 5 as a consequence of the message sent

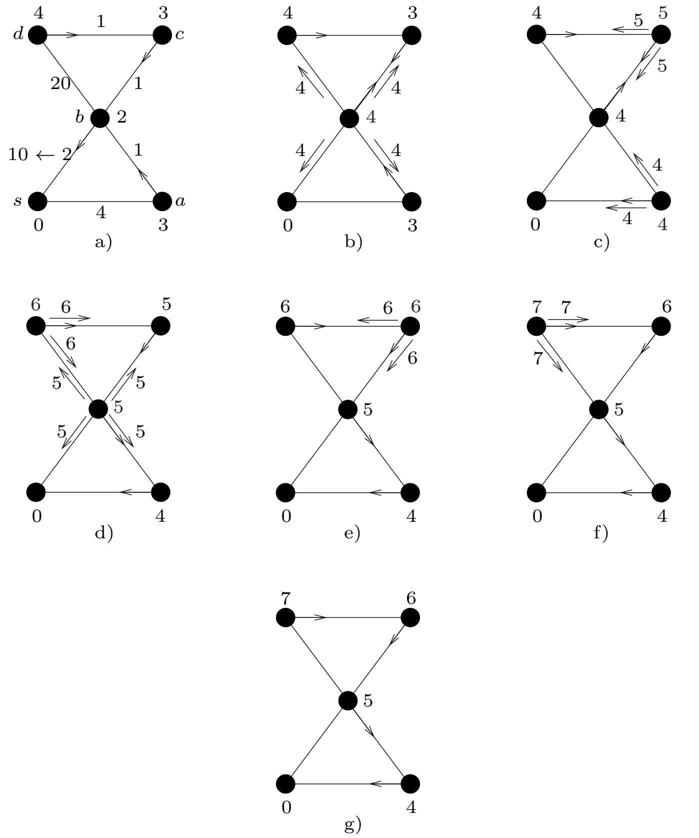


Fig. 19. Example of DBF.

by c. As c was the successor node of b towards s, b needs to update $\mathtt{D}_b[b,s]$ to $\min_{u \in N(b)}\{w(b,u)+\mathtt{D}_b[u,s]\} = c(b,a)+\mathtt{D}_b[a,s] = 5$. After this update, b sends $\mathtt{D}_b[b,s]$ to its neighbors. Node d behaves similarly by updating its distance to s to 6. In Figures 19(e)–19(g), the message sent by b is propagated to nodes c and d in order to update the distances from this nodes to s.

Figure 20 shows an example of DUAL. The example focuses on the same graph of Figure 19(a), which is reported in Figure 20(a), and on destination s. In the figure, the value close to a node indicates its distance to node s, and an arrowhead from x to y in edge (x,y) indicates that node y is the successor of x toward node s. Messages *query*, *reply*, and *update* are denoted by Q, R, and U, respectively. The number in parentheses following R denotes the reported distance contained in the reply message. Nodes involved in a diffuse computation are highlighted in white.

At a certain point in time, edge (b,s) changes its weight from 2 to 10 (Figure 20(a)). When node b detects the weight increase, it determines that it has no feasible successor as none of its neighbors has a distance smaller than

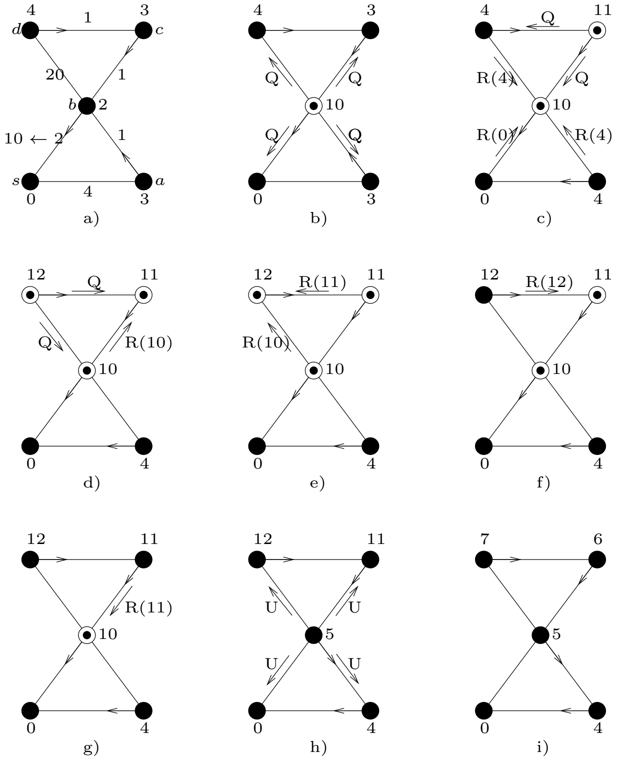
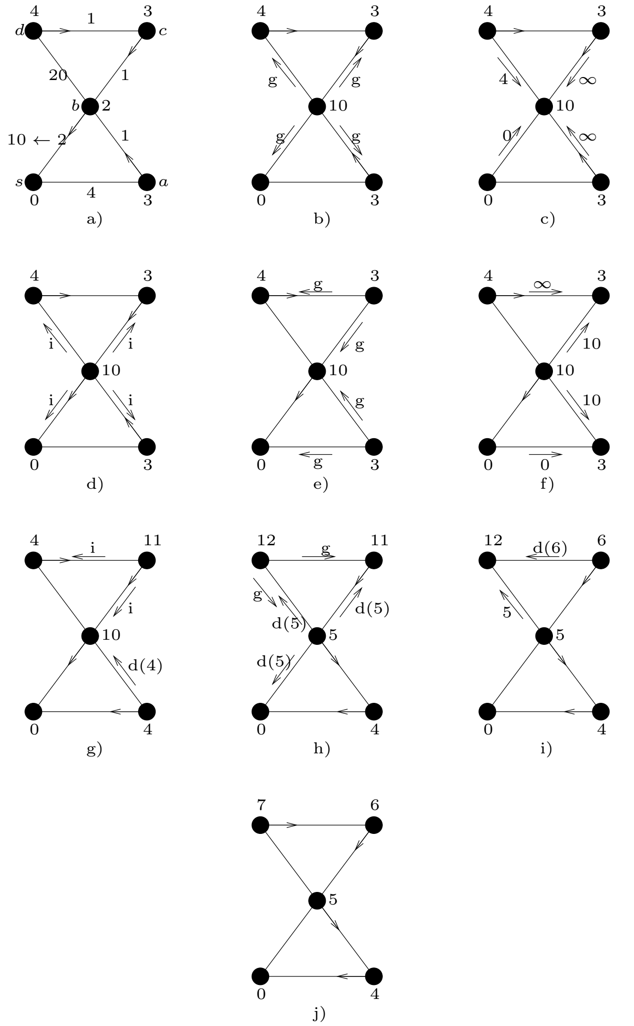


Fig. 20. Example of DUAL.

its current distance, that is 2. Accordingly, it starts a diffuse computation by sending a query to its neighbors (Figure 20(b)). In Figure 20(c), node c forwards the query and continues the diffuse computation, because it has no feasible successor, while node a finds a feasible successor which is node s itself as $0 < 3$ and sends a reply to b. When node d receives node b's query, it simply sends a reply because it has a feasible successor. However, it becomes involved in the diffuse computation when it receives the query from node c (Figure 20(d)). When node d receives all the replies to its query (Figure 20(e)), it computes its new distance and successor (12 and c, respectively), and sends a reply to c's query (Figure 20(f)). Nodes c and b operate in a similar manner when they receives all the replies to their respective queries (Figure 20(f)–20(g)). At this point, the diffuse computation is terminated and node b sends messages *update* containing the new computed distance to notify it to its neighbors (Figure 20(h)). Such messages are propagated to the entire network in order to update the distances according to paths to s induced by successors nodes (Figure 20(i)).

Figure 21 shows an example of DUST. The example focuses on the same graph of Figures 19(a) and 20(a), which is reported in Figure 21(a), and on destination s. As in Figures 19 and 20 , the value close to a node indicates its distance to node s, and an arrowhead from x to y in edge (x, y) indicates that node y is in the set $\mathtt{VIA}[x, s]$, note that in this example these sets have only one element. Messages *get-dist*, *increase*, and *decrease* are denoted by g, i, and d, respectively. The number in parentheses following d denotes the reported distance contained in the *decrease* message. Moreover, messages containing replies to *get-dist* are represented by the reported distance close to the message arrow.



a) b) c) d) e) f) g) h) i) j)

Fig. 21. Example of DUST.

At a certain point in time, edge (b, s) changes its weight from 2 to 10 (Figure 21(a)), as a consequence, node b receives an *increase* message which notifies the weight increase. When node b detects the weight increase, it removes node s from $\texttt{VIA}[b, s]$. As a consequence, $\texttt{VIA}[b, s]$ becomes empty and then b starts procedure Increase by sending *get-dist* messages to its neighbors (Figure 21(b)). In Figure 21(c), $\{b\} \equiv \texttt{VIA}[a, s]$ and $\{b\} \equiv \texttt{VIA}[c, s]$, hence, according to condition $\textsc{crt}_1$, nodes a and c reply to b by sending ∞ while nodes d and s send their own current distances (4 and 0, respectively) to b. By using the information gathered, node b computes its new routing table by setting $\texttt{D}[b, s] = 10$ and $\texttt{VIA}[b, s] = \{s\}$ and then it sends *increase* messages to its neighbors (Figure 21(d)). When nodes a and c receive these messages, they behave like b and hence remove b from their via sets and send *get-dist* messages to their respective neighbors (Figure 21(e)). As a consequence, node b just sends its own current distance (10) to both a and c, node d replies ∞ to c (in fact $\texttt{VIA}[d, s] = \{c\}$) while node s sends its own current distance to a (Figure 21(f)). Hence, node a is able to find an alternative via to s made of node s itself while node c has to set its via again to b and its distance to 11. Then node a sends a *decrease* message to b which will be propagated to the entire network (Figure 21(g)–21(j)) and c sends *increase* messages to its neighbors (Figure 21(g)). Finally, node d behaves like node c in Figures 21(h)–21(j).