Engineering a new loop-free shortest paths routing algorithm

Gianlorenzo d'Angelo, Mattia d'Emidio, Daniele Frigioni, Vinicio Maurizio

Engineering a new loop-free shortest paths routing algorithm⋆

Gianlorenzo D'Angelo[1], Mattia D'Emidio[2], Daniele Frigioni[2], and Vinicio Maurizio[2]

[1] MASCOTTE Project I3S(CNRS/UNSA)/INRIA `gianlorenzo.d_angelo@inria.fr`
[2] Department of Electrical and Information Engineering, University of L'Aquila, `mattia.demidio@univaq.it, daniele.frigioni@univaq.it, vinicio.maurizio@cc.univaq.it`

Abstract. We present LFR (Loop Free Routing), a new loop-free distance vector routing algorithm, which is able to update the shortest paths of a distributed network with n nodes in fully dynamic scenarios. If Φ is the total number of nodes *affected* by a set of updates to the network, and ϕ is the maximum number of destinations for which a node is affected, then LFR requires $O(\Phi \cdot \Delta)$ messages and $O(n + \phi \cdot \Delta)$ space per node, where Δ is the maximum degree of the nodes of the network. We experimentally compare LFR with DUAL, one of the most popular loop-free distance vector algorithms, which is part of CISCO's EIGRP protocol and requires $O(\Phi \cdot \Delta)$ messages and $\Theta(n \cdot \Delta)$ space per node. The experiments are based on both real-world and artificial instances and show that LFR is always the best choice in terms of memory requirements, while in terms of messages LFR outperforms DUAL on real-world instances, whereas DUAL is the best choice on artificial instances.

1 Introduction

Updating shortest paths in a distributed network whose topology dynamically changes over the time is considered crucial in today's communication networks. This problem has been widely studied in the literature, and the solutions found can be classified as *distance-vector* and *link-state*.

Distance-vector algorithms require that a node knows the distance from each of its neighbors to every destination and stores them in a data structure 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 known distance-vector solutions (e.g., see [6, 9, 10, 15, 16]) are based on the classical Distributed Bellman-Ford method (DBF), originally introduced in the Arpanet [12], which is implemented in the RIP protocol. The convergence of DBF can be very slow (possibly infinite) due to the well-known *looping* phenomenon which occurs when a path induced by the routing table entries visits the same node more than

---

once before reaching the intended destination. 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 [1]).

Link-state algorithms, as for example the *Open Shortest Path First (OSPF)* protocol widely used in the Internet (e.g., see [13]), require that a node must know the entire network topology to compute its distance to any destination, usually running the centralized Dijkstra's algorithm. Link-state algorithms are free of looping, but 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 [13, 18] and by using a centralized algorithm for shortest paths.

Related works. In the last years, there has been a renewed interest in devising new efficient light-weight loop-free distributed shortest paths solutions for large-scale Ethernet networks (see, e.g., [3, 7, 8, 17, 19, 20]), where usually the routing devices have limited storage capabilities, and hence distance-vector algorithms seem to be an attractive alternative to link-state solutions. For example, in [17] a new technique has been introduced, named DIV, which is not a routing algorithm by itself, rather it can run on top of any routing algorithm to guarantee loop freedom. A distance vector algorithm has been recently introduced in [5] and successively developed in [4], where it has been named DUST (Distributed Update of Shortest paThs), which suffers of looping, although it has been designed to heuristically reduce the cases where this phenomenon occurs.

Despite the renewed interest of the last years, the most important distance vector algorithm in the literature is surely DUAL (Diffuse Update ALgorithm) [9], which is free of looping and is indeed part of CISCO's widely used EIGRP protocol. DUAL has been experimentally tested in [4] against DUST and DBF in various artificial and real-world scenarios. It has been shown that DUST is always the best choice in terms of space per node. In terms of messages, DUST is the best choice on those real-world topologies in which it does not fall in looping, while DUAL is better than DUST and DBF in all other cases.

Results of the paper. We propose a new loop-free distance vector algorithm, named LFR (Loop Free Routing), which is able to update the shortest paths of a distributed network subject to arbitrary modifications on the edges of the network. Let us denote by n the number of nodes in the network, by Δ the maximum node degree, by Φ the number of nodes *affected* by a sequence of updates on the edges of the network, that is the nodes changing their routing table during that sequence, and by ϕ the maximum number of destinations for which a node is affected. Then LFR requires $O(\Phi \cdot \Delta)$ messages and $O(\Phi)$ steps to converge and requires $O(n + \phi \cdot \Delta)$ space per node. Compared with DUAL, LFR sends the same number of messages but requires less memory per node. In fact, DUAL requires $O(\Phi \cdot \Delta)$ messages and $O(\Phi)$ steps to converge and $\Theta(n \cdot \Delta)$ space per node. Compared with DUST, LFR is better in terms of both number of messages sent and memory requirement per node. In fact, the number

of messages sent by DUST cannot be bounded, as it suffers of looping, and its space requirement per node is $O(n \cdot \Delta)$.

From the experimental point of view, we conducted an extensive study with the aim of comparing the performances of the loop-free algorithms LFR and DUAL also in practical cases. Our simulations were performed in the OMNeT++ simulation environment [14]. As input to the algorithms, we used both real-world and artificial networks. In detail, we considered some of the Internet topologies of the *CAIDA IPv4 topology dataset* [11] (CAIDA - Cooperative Association for Internet Data Analysis provides data and tools for the analysis of the Internet infrastructure) and *Erdös-Rényi* random graphs [2]. The results of our experiments can be summarized as follows: in real-world networks LFR outperforms DUAL in terms of both number of messages and space occupancy per node. In the experiments, we observe that this is in part due to the topological structure of the CAIDA instances which are sparse and contain a high number of nodes of small degree. Therefore, we considered also *Erdös-Rényi* random instances with a variable degree of density. In this case, DUAL sends a number of messages smaller than that of LFR. However, the space requirements of DUAL grow drastically in these random networks while that of LFR are the same of real-word networks. Since CAIDA instances used in the experiments follow a power-law node degree distribution, we embedded the two algorithms in DLP (Distributed Leaf Pruning), a general framework recently proposed in [7] which can run on top of any routing algorithm and is able to reduce the number of messages sent by such algorithms in this kind of graphs. These further experiments show that the good performances of LFR in real-world instances improve when it is used in combination with DLP.

2 Preliminaries

We consider a network made of processors linked through communication channels that exchange data using a message passing model, in which: each processor can send messages only to its neighbors; messages are delivered to their destination within a finite delay but they might be delivered out of order; there is no shared memory among the nodes; the system is asynchronous, that is, a sender of a message does not wait for the receiver to be ready to receive the message.

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 $w : E \to \mathbb{R}^+ \cup \{\infty\}$ on the edges. An edge in E that links nodes u and v is denoted as $\{u, v\}$. Given $v \in V$, $N(v)$ denotes the set of neighbors of v. A *shortest path* between nodes u and v is a path from u to v with the minimum weight. The *distance* $d(u, v)$ from u to v is the weight of a shortest path from u to 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: $via(u, v) \equiv \{z \in N(u) \mid d(u, v) = w(u, z) + d(z, v)\}$. We denote as $w^t()$, $d^t()$ and $via^t()$ an edge weight, a distance and a via in G at time instant t, respectively. We denote a sequence of update

Procedure: $\textsc{Update}(u, s, \mathtt{D}_u[s])$

Input: Node v receives the message $update(u, s, \mathtt{D}_u[s])$ from u

1 if $STATE_v[s] = false$ then
2 $\quad$ if $\mathtt{D}_v[s] > \mathtt{D}_u[s] + w(u,v)$ then $\textsc{Decrease}(u, s, \mathtt{D}_u[s])$;
3 $\quad$ else if $\mathtt{D}_v[s] < \mathtt{D}_u[s] + w(u,v)$ then $\textsc{Increase}(u, s, \mathtt{D}_u[s])$;

Fig. 1. Pseudocode of procedure $\textsc{Update}$.

---

operations on the edges of G by $\mathcal{C} = (c_1, c_2, ..., c_k)$. Assuming $G_0 \equiv G$, we denote as G_i, $0 \le i \le k$, the graph obtained by applying c_i to G_{i-1}. We consider the case in which c_i occurs at time $t_i \ge t_{i-1}$ and either increases or decreases the weight of $\{x_i, y_i\}$ by a quantity $\epsilon_i > 0$. The extension to *delete* and *insert* operations is straightforward. In what follows, given a sequence $\mathcal{C} = (c_1, c_2, ..., c_k)$ of update operations, we denote as $\phi_{c_i,s}$ the set of nodes that change the distance or the via to s as a consequence of c_i, formally: $\phi_{c_i,s} = \{v \in V \mid d^{t_i}(v,s) \ne d^{t_{i-1}}(v,s)$ or $via^{t_i}(v,s) \ne via^{t_{i-1}}(v,s)\}$. If $v \in \cup_{i=1}^{k} \cup_{s \in V} \phi_{c_i,s}$ we say that v is *affected* by c_i. We denote as $\Phi = \sum_{i=1}^{k} \sum_{s \in V} |\phi_{c_i,s}|$. Furthermore, given a generic destination s in V, we denote as $\phi_s = \cup_{i=1}^{k} \phi_{c_i,s}$ and by $\phi = \max_s |\phi_s|$. Note that a node can be affected for at most ϕ different destinations.

Conditions for loop freedom. A distance vector algorithm can be designed to be loop-free by using sufficient conditions as for example those described in [9]. In particular, we focus on SNC (SOURCE NODE CONDITION), which can be implemented and work in combination with a distance vector algorithm that maintains at least the *routing table* and the so-called *topology table*. The routing table of a node v has two entries for each $s \in V$: the estimated distance $\mathtt{D}_v[s]$ between v and s in G; the node $\mathtt{VIA}_v[s] \in via(v,s)$ representing the via from v to s in G. We denote as $\mathtt{D}_v[s](t)$ and $\mathtt{VIA}_v[s](t)$ the estimated distance, and the estimated via at a certain time t. The topology table of v has to contain enough information for v to compute, for each $u \in N(v)$ and for each $s \in V$, the quantity $\mathtt{D}_u[s]$. These values are used in SNC to determine whether a path is free of loops as follows: if, at time t, v needs to change $\mathtt{VIA}_v[s]$ for some $s \in V$, it can select as $\mathtt{VIA}_v[s](t)$ any neighbor $k \in N(v)$ satisfying the following *loop-free test*: $\mathtt{D}_k[s](t) + w^t(v,k) = \min_{v_i \in N(v)}\{\mathtt{D}_{v_i}[s](t) + w^t(v_i, v)\}$ and $\mathtt{D}_k[s](t) < \mathtt{D}_v[s](t)$. If no such neighbor exists, then $\mathtt{VIA}_v[s]$ does not change. Let $\mathtt{VIA}_G[s](t)$ be the directed subgraph of G induced by the set $\mathtt{VIA}_v[s](t)$, for each $v \in V$. In [9] it is proved that if $\mathtt{VIA}_G[s](t_0)$ is loop-free and SNC is used when nodes change their via, then $\mathtt{VIA}_G[s](t)$ remains loop-free, for each time $t \ge t_0$.

3 The new algorithm

In this section, we describe LFR which consists of four procedures named $\textsc{Update}$, $\textsc{Decrease}$, $\textsc{Increase}$ and $\textsc{SendFeasibleDist}$, which are reported in Fig.s 1, 2, 3 and 4, resp. The algorithm is described wrt a source $s \in V$, and it starts every time a weight change $c_i \in \mathcal{C} = (c_1, c_2, ..., c_k)$ occurs on edge $\{x_i, y_i\}$.

Procedure: DECREASE$(u, s, \mathtt{D}_u[s])$

1 $\mathtt{D}_v[s] := \mathtt{D}_u[s] + w(u,v);\ \mathtt{UD}_v[s] := \mathtt{D}_v[s];\ \mathtt{VIA}_v[s] := u;$
2 **foreach** $k \in N(v) \setminus \{\mathit{VIA}_v[s]\}$ **do**
3 $\quad$ send $update(v, s, \mathtt{D}_v[s])$ to k ;

Fig. 2. Pseudo-code of procedure DECREASE.

Data Structures. LFR stores, for each node v, the arrays $\mathtt{D}_v[s]$ and $\mathtt{VIA}_v[s]$ plus, for each $s \in V$, the following data structures: $\mathtt{STATE}_v[s]$, which represents the state of node v wrt source s, v is in *active* state and $\mathtt{STATE}_v[s] = true$, if and only if it is performing procedure INCREASE or procedure SENDFEASIBLEDIST with respect to s; $\mathtt{UD}_v[s]$ which represents the distance from v to s through $\mathtt{VIA}_v[s]$. In particular, if v is active $\mathtt{UD}_v[s]$ is always greater or equal to $\mathtt{D}_v[s]$, otherwise they coincide; in addition, node v stores a temporary data structure $\mathtt{tempD}_v$ needed to implement the topology table. $\mathtt{tempD}_v$ is allocated for a certain s only when needed, that is when v is active wrt s, and it is deallocated when v turns back in passive state wrt s. The entry $\mathtt{tempD}_v[u][s]$ contains $\mathtt{UD}_u[s]$, for each $u \in N(v)$, and hence $\mathtt{tempD}_v$ requires Δ space per node for each source for which v is active. The number of nodes for which v is active is at most ϕ, thus giving an $O(n + \phi \cdot \Delta)$ bound for the space requirement per node, which is better than DUAL. In Section 4 we will show that also in real practical cases the space per node needed by LFR is always smaller than that of DUAL.

Description of LFR. Before LFR starts, at time $t < t_1$, we assume that, for each $v, s \in V$, $\mathtt{D}_v[s](t)$ and $\mathtt{VIA}_v[s](t)$ are correct, that is $\mathtt{D}_v[s](t) = d^t(v,s)$ and $\mathtt{VIA}_v[s](t) \in via^t(v,s)$. We focus the description on a source $s \in V$ and we assume that each node $v \in V$, at time t, is passive wrt s. The algorithm starts when the weight of an edge $\{x_i, y_i\}$ changes. As a consequence, x_i (y_i resp.) sends to y_i (x_i resp.) message $update(x_i, s, \mathtt{D}_{x_i}[s])$ ($update(y_i, s, \mathtt{D}_{y_i}[s])$ resp.). Messages received at a node wrt a source s are stored in a queue and processed in a FIFO order to guarantee mutual exclusion. If an arbitrary node v receives $update(u, s, \mathtt{D}_u[s])$ from $u \in N(v)$, then it performs procedure UPDATE in Fig. 1. Basically, UPDATE compares $\mathtt{D}_v[s]$ with $\mathtt{D}_u[s] + w(u,v)$ to determine whether v needs to update its estimated distance and its estimated via to s. If node v is active, the processing of the message is postponed by enqueueing it into the FIFO queue associated to s. Otherwise, if $\mathtt{D}_v[s] > \mathtt{D}_u[s] + w(u,v)$ (Line 2), then v performs procedure DECREASE, while if $\mathtt{D}_v[s] < \mathtt{D}_u[s] + w(u,v)$ (Line 3) v performs procedure INCREASE. Finally, if node v is passive and $\mathtt{D}_v[s] = \mathtt{D}_u[s] + w(u,v)$ then there is more than one shortest path from v to s. In this case the message is discarded and the procedure ends.

When a node v performs procedure DECREASE, it simply updates $\mathtt{D}$, $\mathtt{UD}$ and $\mathtt{VIA}$ by using the updated information provided by u. Then, the update is forwarded to all neighbors of v with the exception of $\mathtt{VIA}_v[s]$ (Line 3).

When a node v performs procedure INCREASE, it first verifies whether the update has been received from $\mathtt{VIA}_v[s]$ or not (Line 1). In fact, only in the affirmative case v changes its distance to s and needs to find a new via. To this

Procedure: INCREASE$(u, s, \mathtt{D}_u[s])$

1 **if** $\mathit{VIA}_v[s] = u$ **then**

2 $\mathtt{STATE}_v[s] := \mathit{true}$;

3   Allocate $\mathtt{tempD}_v[\cdot][s]$;

4 $\mathtt{tempD}_v[u][s] := \mathtt{D}_u[s]$;

5 $\mathtt{UD}_v[s] := \mathtt{tempD}_v[u][s] + w(u,v)$;

6   **foreach** $v_i \in N(v) \setminus \{\mathit{VIA}_v[s]\}$ **do**

7   receive $\mathtt{UD}_{v_i}[s]$ and store it in $\mathtt{tempD}_v[v_i][s]$ by sending $\mathit{get.dist}(v, s, \mathtt{UD}_v[s])$ to v_i;

8 $\mathtt{Dmin} := \min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u,v)\}$;

9 $\mathtt{VIAmin} := \arg\min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u,v)\}$;

10   **if** $\mathit{tempD}_v[\mathit{VIAmin}][s] \geq D_v[s]$ **then**

11   **foreach** $v_i \in N(v) \setminus \{\mathit{VIA}_v[s]\}$ **do**

12   receive loop-free distance $\mathtt{UD}_{v_i}[s]$ and store it in $\mathtt{tempD}_v[v_i][s]$ by sending $\mathit{get.feasible.dist}(v, s, \mathtt{UD}_v[s])$ to v_i;

13 $\mathtt{Dmin} := \min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u,v)\}$;

14 $\mathtt{VIAmin} := \arg\min_{u \in N(v)}\{\mathtt{tempD}_v[u][s] + w(u,v)\}$;

15   Deallocate $\mathtt{tempD}_v[\cdot][s]$;

16 $\mathtt{D}_v[s] := \mathtt{Dmin}$;

17 $\mathtt{UD}_v[s] := \mathtt{D}_v[s]$;

18 $\mathit{VIA}_v[s] := \mathtt{VIAmin}$;

19   **foreach** $k \in N(v)$ **do** send $\mathit{update}(v, s, \mathtt{D}_v[s])$ to k;

20 $\mathtt{STATE}_v[s] := \mathit{false}$;

Fig. 3. Pseudo-code of procedure INCREASE.

aim, node v becomes active, allocates the temporary data structure $\mathtt{tempD}_v$, and sets $\mathtt{UD}_v[s]$ to the current distance through $\mathtt{VIA}_v[s]$ (Lines 2–5). At this point, v first performs the so called Local-Computation (Lines 6–9), which involves all the neighbors of v. If the Local-Computation does not succeed, then node v initiates the so called Global-Computation (Lines 11–14), which involves in the worst case all nodes of the network.

In the Local-Computation, node v sends $\mathit{get.dist}$ messages, carrying $\mathtt{UD}_v[s]$, to all its neighbors, except u. A neighbor $k \in N(v)$ that receives a $\mathit{get.dist}$ message, immediately replies with the value $\mathtt{UD}_k[s]$, and if k is active, it updates $\mathtt{tempD}_k[v][s]$ to $\mathtt{UD}_v[s]$. When v receives these values from its neighbors, it stores them in $\mathtt{tempD}_v$, and it uses them to compute the minimal estimated distance $\mathtt{Dmin}$ to s and the neighbor $\mathtt{VIAmin}$ which gives such a distance (Lines 8–9).

At the end of the Local-Computation v checks whether a feasible via exists, according to SNC, by executing the loop-free test at line 10. If the test does not succeed, then v initiates the Global-Computation (Line 11), in which it entrusts the neighbors the task of finding a loop-free path. In this phase, v sends $\mathit{get.feasible.dist}(v, s, \mathtt{UD}_v[s])$ message to each of its neighbors. This message carries the value of the estimated distance through its current via. This distance is not guaranteed to be minimum but it is guaranteed to be loop-free. When v receives the answers to $\mathit{get.feasible.dist}$ messages from its neighbors, again it stores them in $\mathtt{tempD}_v$ and it uses them to compute the minimal estimated distance $\mathtt{Dmin}$ to s and the neighbor $\mathtt{VIAmin}$ which gives such a distance (Lines 13–14).

Procedure: SENDFEASIBLEDIST$(u, s, \overline{D})$
Input: Node v receives $get.feasible.dist(u, s, \texttt{UD}_u[s])$ from u

1 **if** $\mathit{VIA}_v[s] = u$ and $\mathit{STATE}_v[s] = false$ **then**
2 $\quad$ $\texttt{STATE}_v[s] := true$;
3 $\quad$ Allocate $\texttt{tempD}_v[\cdot][s]$;
4 $\quad$ $\texttt{tempD}_v[u][s] := \overline{D}$;
5 $\quad$ $\texttt{UD}_v[s] := \texttt{tempD}_v[u][s] + w(u, v)$;
6 $\quad$ **foreach** $v_i \in N(v) \setminus \{\mathit{VIA}_v[s]\}$ **do**
7 $\quad\quad$ receive $\texttt{UD}_{v_i}[s]$ and store it in $\texttt{tempD}_v[v_i][s]$ by sending $get.dist(v, s, \texttt{UD}_v[s])$ to v_i;
8 $\quad$ $\texttt{Dmin} := \min_{u \in N(v)}\{\texttt{tempD}_v[u][s] + w(u, v)\}$;
9 $\quad$ $\texttt{VIAmin} := \arg\min_{u \in N(v)}\{\texttt{tempD}_v[u][s] + w(u, v)\}$;
10 $\quad$ **if** $\mathit{tempD}_v[\mathit{VIAmin}][s] \geq D_v[s]$ **then**
11 $\quad\quad$ **foreach** $v_i \in N(v) \setminus \{\mathit{VIA}_v[s]\}$ **do**
12 $\quad\quad\quad$ receive loop-free distance $\texttt{UD}_{v_i}[s]$ and store it in $\texttt{tempD}_v[v_i][s]$ by sending
$\quad\quad\quad$ $get.feasible.dist(v, s, \texttt{UD}_v[s])$ to v_i;
13 $\quad\quad$ $\texttt{Dmin} := \min_{u \in N(v)}\{\texttt{tempD}_v[u][s] + w(u, v)\}$;
14 $\quad\quad$ $\texttt{VIAmin} := \arg\min_{u \in N(v)}\{\texttt{tempD}_v[u][s] + w(u, v)\}$;
15 $\quad$ send $\texttt{Dmin}$ to $\texttt{VIA}_v[s]$;
16 $\quad$ Deallocate $\texttt{tempD}_v[\cdot][s]$;
17 $\quad$ $\texttt{D}_v[s] := \texttt{Dmin}$;
18 $\quad$ $\texttt{UD}_v[s] := \texttt{D}_v[s]$;
19 $\quad$ $\texttt{VIA}_v[s] := \texttt{VIAmin}$;
20 $\quad$ **foreach** $k \in N(v)$ **do** send $update(v, s, \texttt{D}_v[s])$ to k;
21 $\quad$ $\texttt{STATE}_v[s] := false$;
22 **else**
23 $\quad$ **if** $\mathit{STATE}_v[s] = true$ **then**
24 $\quad\quad$ $\texttt{tempD}_v[u][s] := \overline{D}$;
25 $\quad$ send $\texttt{UD}_v[s]$ to u ;

Fig. 4. Pseudo-code of procedure SENDFEASIBLEDIST.

At this point v has surely found a feasible via to s and it deallocates $\texttt{tempD}_v$, updates $\texttt{D}_v[s]$, $\texttt{UD}_v[s]$ and $\texttt{VIA}_v[s]$ and propagates the change by sending *update* messages to its neighbors (Lines 15–19). Finally, v turns back in passive state.

A node $k \in N(v)$ which receives a *get.feasible.dist* message performs the procedure SENDFEASIBLEDIST. If $\texttt{VIA}_k[s] = v$ and k is passive (Line 1), then procedure SENDFEASIBLEDIST behaves similarly to procedure INCREASE, notice indeed that Lines 2–21 of SENDFEASIBLEDIST are basically identical to Lines 2–20 of INCREASE. The only difference is Line 15 of SENDFEASIBLEDIST which is not present in INCREASE, and represents the answer to the *get.feasible.dist* message. However, within SENDFEASIBLEDIST the Local-Computation and the Global-Computation are performed with the aim of sending a reply with an estimated loop-free distance in addition to that of updating the routing table. In particular, node k needs to provide to v a new loop-free distance. To this aim, node k becomes active, allocates the temporary data structure $\texttt{tempD}_k$, and sets $\texttt{UD}_k[s]$ to the current distance through $\texttt{VIA}_v[s]$ (Lines 2–5). Then, as in procedure INCREASE, k first performs the Local-Computation (Lines 6–9), which involves

all the neighbors of k. If the Local-Computation does not succeed, that is the SNC is violated, then node k initiates the Global-Computation (Lines 11–14), which involves in the worst case all nodes of the network. At this point k has surely found an estimated distance to s which is guaranteed to be loop-free and hence, differently from INCREASE, it sends this value to its current via v (Line 15) as answer to the *get.feasible.dist* message. Now, as in procedure INCREASE, node k can deallocate tempD_v, update its local data structures $\text{D}_v[s]$, $\text{UD}_v[s]$ and $\text{VIA}_v[s]$, and propagate the change by sending *update* messages to all its neighbors (Lines 15–19). Finally, v turns back in passive state.

Differently from the *get.dist* case, k immediately replies with $\text{UD}_k[s]$ only if $\text{VIA}_k[s] \neq v$ or $\text{STATE}_v[s] = false$ (Line 25). If $\text{VIA}_k[s] \neq v$, that is k does not use v to reach s, and k is active with respect to s, then it updates $\text{tempD}_k[v][s]$ with the received most recent value (Line 24) before sending to v the distance to s through its current via. This is done to send to v the most up to date value.

The next theorems, whose proofs will be given in the full paper, show the loop-freedom, correctness, and complexity of LFR. As highlighted in [9], the complexity of a distributed algorithm in the asynchronous model depends on the time needed by processors to execute the local procedures of the algorithm and on the delays incurred in the communication among nodes. Moreover, these parameters influence the scheduling of the distributed computation and hence the number of messages sent. For these reasons, we consider the *realistic* case where the weight of an edge models the time needed to traverse such edge and all the processors require one unit of time to process a procedure. In this way, the distance between two nodes models the minimum time that such nodes need to communicate. We then measure the time complexity by the number of times that a processor performs a procedure. Note that, in the *realistic* case, DUAL has the same worst case complexity of LFR.

Theorem 1 (Loop-Freedom). *Let s be a node in G and let $\mathcal{C} = (c_1, c_2, ..., c_k)$ be a sequence of edge weight changes on G, if $\text{VIA}_G[s](t_0)$ is loop-free, then $\text{VIA}_G[s](t)$ is loop-free for each $t \geq t_0$.*

Theorem 2 (Correctness). *There exists a time $t_F \geq t_k$ such that, for each pair of nodes $v, s \in V$, and for each time $t \geq t_F$, $\text{D}_v[s](t) = d^{t_k}(v, s)$ and $\text{VIA}_t[v, s] \in via^{t_k}(v, s)$.*

Theorem 3 (Complexity). *Given a sequence of weight change operations $\mathcal{C} = (c_1, c_2, ..., c_k)$, LFR requires $O(n + \phi \cdot \Delta)$ space per node and sends $O(\Phi \cdot \Delta)$ messages and needs $O(\Phi)$ steps to converge.*

4 Experimental analysis

In this section, we report the results of our experimental study on LFR and DUAL. Our experiments have been performed on a workstation equipped with a Quad-core 3.60 GHz Intel Xeon X5687 processor, with 12MB of internal cache and 24 GB of main memory, and consist of simulations within the OMNeT++
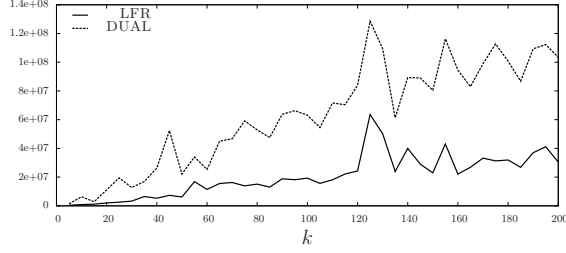
Fig. 5. Number of messages sent by LFR and DUAL on $G_{IP-8000}$

4.0p1 environment [14]. The program has been compiled with GNU g++ compiler 4.4.3 under Linux (Kernel 2.6.32).

Executed tests. For the experiments we used both the real-world instances of the *CAIDA IPv4 topology dataset* [11] and *Erdős-Rényi* random graphs [2]. CAIDA is an association which provides data and tools for the analysis of the Internet infrastructure. We parsed the files in the CAIDA dataset to obtain a weighted undirected graph G_{IP} where nodes represent routers, edges represent links among routers and weights are given by Round Trip Times (RTT). As the graph G_{IP} consists of almost 35000 nodes, we cannot use it for the experiments, due to the memory requirements of DUAL. Hence, we performed our tests on connected subgraphs of G_{IP}, with a variable number of nodes and edges, 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 consisting of a dynamic graph characterized by a subgraph of G_{IP} with $n \in \{1200, 5000, 8000\}$ nodes (we denote an n nodes subgraph of G_{IP} with G_{IP-n}) and a set of k concurrent edge updates, where k assumes values in $\{5, 10, \ldots, 200\}$. 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 (a dynamic graph with a fixed value of k) we performed 5 different experiments and we report average values. We performed the experiments in the *realistic* case, that is we considered the RTT as the time delay for receiving packets.

Graphs G_{IP} turns out to be very sparse (i.e., $m/n \approx 1.3$), so it is worth analyzing LFR and DUAL also on graphs denser than G_{IP}. To this aim we considered Erdős-Rényi random graphs [2]. In detail, we randomly generated a set of different tests, where a test consists of a dynamic graph characterized by: an Erdős-Rényi random graph G_{ER} of 2000 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* in $[0.01, 0.61]$ and $k = 200$. Edge weights are 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. As in G_{IP}, time delays are equal to the edge weights.
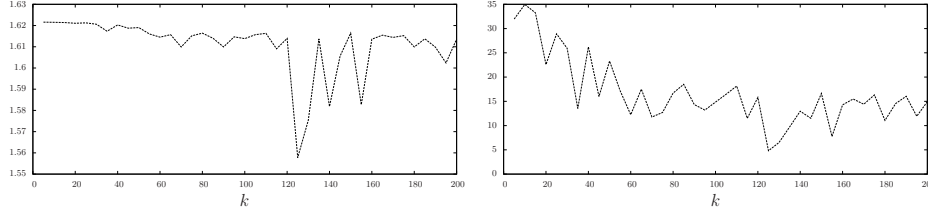
Fig. 6. Ratio between the average space occupancy of DUAL and LFR (left); ratio between the maximum space occupancy of DUAL and LFR (right) on $G_{IP-8000}$.

Analysis. In Fig. 5 we report the number of messages sent by LFR and DUAL on $G_{IP-8000}$. The diagram shows that LFR always outperforms DUAL. The ratio between the number of messages sent by DUAL and LFR is between 2.02 and 7.61. The results of our experiments on $G_{IP-1200}$ and $G_{IP-5000}$ give similar results. These good performances of LFR are due in part to the topological structure of G_{IP} in which the average node degree is almost one. In fact, LFR uses *get.dist* messages in order to know the estimated distances of its neighbors. It follows that the number of *get.dist* messages sent by a node is proportional to the node degree, and hence the contribution to the message complexity of LFR of these messages is basically irrelevant. Note that, DUAL does not need to use *get.dist* messages as it stores, for each node, the estimated distances of its neighbors. We can hence conclude that the better performance of LFR on G_{IP} with respect to DUAL are basically due to the different way in which the two algorithms manage the distributed computation of shortest paths.

To conclude our analysis on G_{IP}, we considered the space occupancy per node of the implemented algorithms (recall that DUAL and LFR require $O(n \cdot \Delta)$ and $O(n + \phi \cdot \Delta)$ space per node, resp.). The experimental results on the space occupancy are reported in Fig. 6. Since in these sparse graphs the average degree is almost one, the average space occupancy of the two algorithms are almost equivalent although LFR is always slightly better than DUAL (Fig. 6 (left)). If we consider the maximum space occupancy, that is the space occupied by the node with the highest space requirements, then LFR is by far better than DUAL (Fig. 6 (right)). In fact DUAL requires a maximum space occupancy per node which is between 4.81 and 34.97 times the maximum space occupancy required by LFR. This is due to the fact that the topology table, implemented by the `tempD` data structure, is allocated by LFR only when needed.

As already observed, graph G_{IP} and its subgraphs have a high number of nodes with small degree. For instance, $G_{IP-8000}$ has 3072 degree-one nodes which corresponds to 38.4% of the nodes of the graph. Indeed, these graphs follow a power-law degree distribution. Hence, we embedded LFR and DUAL in DLP [7], a recently proposed framework which is able to reduce the number of messages sent by any distance vector algorithm in this kind of graphs by avoiding any computation involving degree-one nodes. The algorithms obtained by combining LFR and DUAL with DLP are denoted as LFR-DLP and DUAL-DLP,
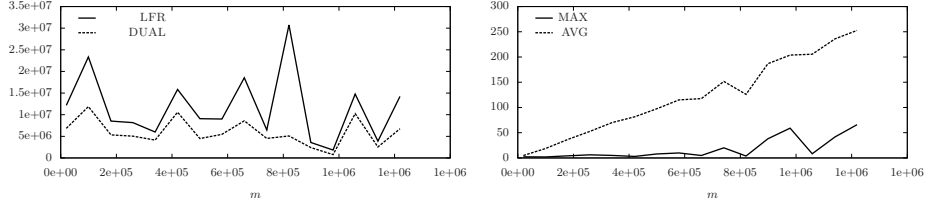
Fig. 7. Experiments on G_{ER} with $k = 200$ and $dens = 0.01, 0.05, ..., 0.61$: number of messages sent (left); ratio between the space occupancies, maximum (MAX) and average (AVG), of DUAL and LFR (right).

resp. Our experiments on these algorithms show that the use of DLP reduced the number of messages sent by a factor in $[1.96, 2.78]$ in LFR and $[2, 17, 8.33]$ in DUAL. This allows us to state that the good performance of LFR obtained in real-world instances are confirmed if it is used in combination with DLP. The ratio between the number of messages sent by DUAL-DLP and LFR-DLP is very similar to the ratio between the number of messages sent by DUAL and LFR since it always lies between 1.47 and 6.25. Finally, our experiments show that the space overhead resulting from the application of DLP is irrelevant. In fact, the ratio between the average space occupancy of DUAL and DUAL-DLP is 1.04 while the ratio between the average space occupancy of LFR and LFR-DLP is in $[1.04, 1.06]$. Since the performances of LFR and DUAL on the CAIDA graphs are in part influenced by the topological structure of such graphs, it is worth investigating how these algorithms perform also on dense graphs. To this aim we considered Erdős-Rényi random graphs G_{ER} with 2000 nodes, 200 weight changes and $dens$ ranging from 0.01 to 0.61, which leads to a number of edges ranging from about 20000 to about 1200000. Fig. 7 (left) shows the number of messages sent by LFR and DUAL on these instances. Contrarily to the case of G_{IP}, DUAL is better than LFR on G_{ER}. In fact, in most of the cases DUAL sends half the number of messages sent by LFR. This is due to the high number of *get.dist* messages sent by LFR which in these dense graphs are of course relevant. However, from the space occupancy point of view, we notice that the space requirements of DUAL increase more than those of LFR with the node degree, as highlighted in Fig. 7 (right). In detail, Fig. 7 (right) shows the ratio between the average space occupancy per node of DUAL and that of LFR in G_{ER} and the ratio between the maximum space occupancy per node of DUAL and that of LFR in G_{ER}. The average space occupancy ratio grows almost linearly with m, as the space occupancy of LFR depends on the degree with a factor ϕ while that of DUAL is proportional to the node degree with a factor n. Similar observations hold for the maximum space occupancy.

In conclusion, our experiments show that LFR is always the best choice in terms of memory requirements, while in terms of messages LFR outperforms DUAL on real-world instances and DUAL is the best choice on artificial ones.

References

1. B. Awerbuch, A. Bar-Noy, and M. Gopal. Approximate distributed bellman-ford algorithms. *IEEE Trans. on Communications*, 42(8):2515–2517, 1994.
2. B. Bollobás. *Random Graphs.* Cambridge University Press, 2001.
3. 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.
4. S. Cicerone, G. D'Angelo, G. Di Stefano, D. Frigioni, and V. Maurizio. Engineering a new algorithm for distributed shortest paths on dynamic networks. *Algorithmica*. To appear. Prel. version in [5].
5. 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 *SEA 2010*, volume 6049 of *LNCS*, pages 59–70, 2010.
6. S. Cicerone, G. D. Stefano, D. Frigioni, and U. Nanni. A fully dynamic algorithm for distributed shortest paths. *Theoretical Comp. Science*, 297(1-3):83–102, 2003.
7. G. D'Angelo, M. D'Emidio, D. Frigioni, and V. Maurizio. A speed-up technique for distributed shortest paths computation. In *ICCSA 2011*, volume 6783 of *LNCS*, pages 578–593, 2011.
8. K. Elmeleegy, A. L. Cox, and T. S. E. Ng. On count-to-infinity induced forwarding loops in ethernet networks. In *Proceedings IEEE INFOCOM*, pages 1–13, 2006.
9. J. J. Garcia-Lunes-Aceves. Loop-free routing using diffusing computations. *IEEE/ACM Trans. on Networking*, 1(1):130–141, 1993.
10. P. A. Humblet. Another adaptive distributed shortest path algorithm. *IEEE Trans. on Communications*, 39(6):995–1002, Apr. 1991.
11. 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`.
12. J. McQuillan. Adaptive routing algorithms for distributed computer networks. Technical Report BBN Report 2831, Cambridge, MA, 1974.
13. J. T. Moy. *OSPF: Anatomy of an Internet routing protocol.* Addison-Wesley, 1998.
14. OMNeT++. Discrete event simulation environment. `http://www.omnetpp.org`.
15. A. Orda and R. Rom. Distributed shortest-path and minimum-delay protocols in networks with time-dependent edge-length. *Distr. Computing*, 10:49–62, 1996.
16. K. V. S. Ramarao and S. Venkatesan. On finding and updating shortest paths distributively. *Journal of Algorithms*, 13:235–257, 1992.
17. S. Ray, R. Guérin, K.-W. Kwong, and R. Sofia. Always acyclic distributed path computation. *IEEE/ACM Trans. on Networking*, 18(1):307–319, 2010.
18. E. C. Rosen. The updating protocol of arpanet's new routing algorithm. *Computer Networks*, 4:11–19, 1980.
19. N. Yao, E. Gao, Y. Qin, and H. Zhang. Rd: Reducing message overhead in DUAL. In *Proceedings 1st International Conference on Network Infrastructure and Digital Content (IC-NIDC09)*, pages 270–274. IEEE Press, 2009.
20. C. Zhao, Y. Liu, and K. Liu. A more efficient diffusing update algorithm for loop-free routing. In *5th International Conference on Wireless Communications, Networking and Mobile Computing (WiCom09)*, pages 1–4. IEEE Press, 2009.