Article

Distance-Vector Algorithms for Distributed Shortest Paths on Dynamic Power-Law Networks

Mattia D'Emidio [1,*] and Daniele Frigioni [2,*]

[1] Gran Sasso Science Institute (GSSI), Viale Francesco Crispi, I–67100 L'Aquila, Italy
[2] Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Via Vetoio, I–67100 L'Aquila, Italy
[*] Correspondence: mattia.demidio@gssi.it (M.D.); daniele.frigioni@univaq.it (D.F.)

Abstract: Efficiently solving the problem of computing, in a distributed fashion, the shortest paths of a graph whose topology dynamically changes over time is a core functionality of many today's digital infrastructures, probably the most prominent example being communication networks. Many solutions have been proposed over the years for this problem that can be broadly classified into two categories, namely *Distance-Vector* and *Link-State* algorithms. Distance-Vector algorithms are widely adopted solutions when scalability and reliability are key issues or when nodes have either limited hardware resources, as they result in being very competitive approaches in terms of both the memory and the computational point of view. In this paper, we first survey some of the most established solutions of the Distance-Vector category. Then, we discuss some recent algorithmic developments in this area. Finally, we propose a new experimental study, conducted on a prominent category of network instances, namely *generalized linear preference* (GLP) power-law networks, to rank the performance of such solutions.

Keywords: dynamic algorithms; distributed shortest paths; communication networks; routing protocols; power-law networks

1. Introduction

Efficiently solving the problem of computing, in a distributed fashion, the shortest paths of a graph whose topology dynamically changes over time is a core functionality of essentially all modern digital infrastructures, probably the most prominent example being communication networks. For this reason, the problem has been widely investigated in past decades and many solutions have been proposed over the years, which can be broadly classified into two categories, namely *Distance-Vector* and *Link-State* algorithms, depending on the way they achieve the computation of shortest paths [1–5].

On the one hand, Distance-Vector algorithms, such as, e.g., the *Distributed Bellman-Ford* (DBF) method [3], are characterized by a *local* nature, in the sense that a generic node executing this kind of algorithms usually interacts only with its neighbors and stores minimal little information about the global status of the network. In more details, typically, each node performing a Distance-Vector algorithm maintains only a single data structure (most of the times referred to as *routing table*) containing, for each other node of the network, the *distance*, i.e. the weight of a shortest path, and the *next hop*, i.e. the next node on the same shortest path. Note that, this is, in general, the essential information regarding shortest paths to be stored in most of the applications of interest, like e.g., in data routing where the next hop is used to forward data toward destination nodes of interest.

The computation (and the maintainance under dynamic settings) of the routing table is, in the case of Distance-Vector algorithms, performed by solving very simple equations (see [6] and references therein), thus making them very competitive solutions from the *computational complexity* point of view. However, Distance-Vector based protocols, in dynamic scenarios, can suffer from some well-known and undesired phenomena, namely *looping* and *count-to-infinity*, that can heavily affect their performance in terms of usage of communication resources (a.k.a. *message* or *communication complexity*), tough quite efficient countermeasures for such issues are known.

On the other hand, Link-State algorithms, as for example the widely adopted *Open Shortest Path First* (OSPF) protocol [2], are characterized by a *global* nature, as they require each node to store the entire network topology. Shortest paths, in this case, are usually computed by running a centralized shortest path algorithm, as for example the classic Dijkstra's algorithm [7]. This results in a usage of memory (a.k.a. *space complexity*) which is asymptotically quadratic in the number of nodes of the network, in contrast to the linear requirements of Distance-Vector algorithms. On the positive side, Link-State approaches do not incur in looping and count-to-infinity phenomena, thus being, in static environments, more competitive w.r.t. Distance-Vector algorithms in terms of communication complexity. However, this is counterbalanced in dynamic scenarios, where they perform quite poorly in this sense, since each node needs to receive and store up-to-date information on the entire network topology after any change. This is achieved by broadcasting each modification affecting the network topology to all nodes [2,8,9], and by using a centralized algorithm for dynamic shortest paths, as for example those proposed in [10,11].

In the last few years, there has been a renewed interest in devising new efficient and light-weight distributed shortest-path algorithms for large-scale networks (see, e.g., [6,12–19] and references therein), where Distance-Vector algorithms has been considered as an attractive alternative to Link-State solutions either when scalability and reliability are key issues or when the memory/computational resources of the nodes of the network are limited.

The great majority of Distance-Vector solutions known in the literature (see, e.g., [1,20–24]) are based on the above mentioned DBF, introduced for the first time in Arpanet in the late 60's [25], and still used in some real-world networks, as a part of the RIP protocol [8]. DBF is known to converge to the correct distances (and thus to be able to compute next-hops correctly) if the link weights stabilize and all cycles have positive lengths [26]. However, the convergence time can be very high (and possibly infinite) due to the *looping* and *count-to-infinity* phenomena. Furthermore, if the nodes of the network are not synchronized, even when no change occurs in the network, the overall number of messages sent by DBF is, in the worst case, exponential with respect to the size of the network [27].

Among Distance-Vector algorithms, probably the most prominent one is DUAL (*Diffuse Update ALgorithm*), proposed for the first time in [5] and part of CISCO's widely used *Enhanced Interior Gateway Routing Protocol* (EIGRP) [28]. DUAL is more complex with respect to the baseline DBF since it uses, besides the routing table, several auxiliary data structures to guarantee freedom from looping and count-to-infinity phenomena. Another loop-free Distance-Vector algorithm that is worth to be mentioned is *Loop Free Routing* (LFR), which has been more recently proposed in [29]. Asymptotically, LFR has the same theoretical message complexity of DUAL but it uses an amount of data structures per node which is always smaller than that of DUAL. From the experimental point of view, LFR has been shown in [29] to be very effective in terms of both the number of messages sent and the memory requirements per node in some real-world networks of particular interest.

Recently, in [30], a general technique, named *Distributed Computation Pruning* (DCP) has been proposed in order to overcome some of their main limitations, e.g. high number of messages sent and poor convergence. The methodology is a general one, in the sense that the authors propose a sort of general framework that can be applied to any Distance-Vector algorithm to boost its performance. In particular, DCP has been designed to be effective when the algorithm is run on networks following a power-law node degree distribution, which are often simply referred as *power-law networks*. Such class of networks is of particular practical relevance, since it includes some of the most important modern network applications. Among the others, it is worth to mention the Internet and the majority of wireless sensor networks and social networks. The main idea underlying DCP is based on some well-known properties of the structure of power-law networks. In particular, an n node power-law network typically has average node degree which is much smaller than n (often a small constant) and a very high number of nodes with small degree (less than 3). This can be exploited, to improve the performance of the algorithm, by observing that nodes with small degree often do not provide any useful information for the distributed computation of shortest paths, in the sense that there are many topological situations in which these nodes should neither perform nor be involved in any kind of distributed computation, since their shortest paths depend on those of higher degree nodes. In [30] the effectiveness of DCP has been shown via an extensive experimental evaluation conducted on: i) IPv4 topology datasets collected by the *Cooperative Association for*

Internet Data Analysis (CAIDA) [31], an association that provides data and tools for the analysis of the Internet infrastructure; ii) synthetic topologies generated by the *Barabási-Albert* algorithm [32].

In this paper, we first survey the main features of some of the most established solutions of the Distance-Vector category, namely DBF, DUAL, and LFR. Then, we summarize the main characteristics of DCP. Finally, we assess, via an extensive experimental evaluation, the performance of DUAL, LFR and their combinations with DCP, and provide evidences of its effectiveness, on a practically relevant class of power-law networks, namely power-law artificial instances obtained by the Generalized Linear Preference (GLP) model [33]. The GLP framework has been shown, by many studies focusing on distributed algorithms (see, e.g., [34,35]) to model very well the Internet, and parts of it. In particular, in [36], it has been shown that GLP predicts the structure of real-world communication networks (e.g. the Internet) better than the Barabási–Albert linear preferential model. This behaviour is better captured by the GLP model which adds more flexibility than Barabási–Albert in specifying how nodes connect to other nodes. The results of our experimental evaluation can be summarized as follows: given a generic Distance-Vector algorithm A, we provide strong evidences that combining it with DCP, in GLP network topologies, allows: i) a huge reduction (a couple of orders of magnitude in most of the cases) in the utilization of communication resources (measured in terms of messages sent) with respect to A without DCP; ii) a significant improvement in terms of memory requirements with respect to A without DCP. As a side result, the experiments also show that LFR outperforms DUAL in terms of number of messages sent and is very effective from the memory requirements point of view in GLP networks.

The paper is organized as follows. In Section 2 we give all the necessary background and notation. In Sections 3, 4 and 5 we review DBF, DUAL, and LFR, respectively. In Section 6 we describe DCP and its combinations with both DUAL and LFR, and overview the experimental study of [30]. In Section 7 we show the results of our new experimental study on the GLP power-law networks. Finally, Section 8 concludes the paper.

2. Background

In this section, we provide all the necessary background and notation that will be used through the paper. We consider the classic distributed scenario where we have a network made of processors that are connected through (bidirectional) communication channels and exchange data using a message passing model, in which:

- each processor can send messages only along its own communication channels, i.e. to processors it is connected with;
- messages are delivered to their destination within a finite delay;
- there is no shared memory among the processors;
- the system is *asynchronous*, that is a sender of a message does not wait for the receiver to be ready to receive the message. The message is delivered within a finite but unbounded time.

Moreover, we assume the system to be asynchronous, as well as that described in [37], which is briefly summarized below. The *state* of a processor v is the content of the data structure stored by 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 located at q. When q reads m from its buffer and processes it, the event "reception of m" occurs. Messages are trasmitted through the channels in *First-In-First-Out (FIFO)* order, that is, messages arriving at processor q are always received in the same order as they are sent by p. An *execution* is a (possibly infinite) sequence of network states and events. A non-negative integer number is associated to each event, the *time* at which that event occurs. Time is a *global* parameter and is not accessible to the processors of the network. Moreover, time must be non-decreasing and must increase without any bound, if the execution is infinite. Finally, 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.

2.1. Graph Notation

We represent a network by an undirected weighted connected 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 $w : E \to \mathbb{R}^+$ that assigns to each edge a real value representing the optimization parameter associated to the corresponding channel, such as, e.g. the time needed to traverse the corresponding link if a packet is sent on it. Given a graph $G = (V,E,w)$, we will denote by: (v,u) an edge of E that connects nodes $v,u \in V$, and by $w(v,u)$ its weight, respectively; $N(v) = \{u \in V : (v,u) \in E\}$ the set of neighbors of a node $v \in V$; $deg(v) = |N(v)|$ the degree of v, for each $v \in V$; $maxdeg = \max_{v \in V} deg(v)$ the maximum degree among the nodes in G. Furthermore, we will use $\{u,\dots,v\}$ to represent a generic path in G between nodes u and v and, given a path P, we will use $w(P)$ to denote its *weight*, i.e. the sum of the weights associated to its edges. A path $P = \{u,...,v\}$ is called a *shortest path* between u and v if and only if P is a path having minimum weight among all possible paths between u and v in G. Given two nodes $u,v \in V$, we will denote by $d(u,v)$ the topological *distance* between u and v, i.e. the weight of a *shortest path* between u and v. Finally, we will call $via(u,v)$ the *via* from u to v, i.e. the set of neighbors of u (there might be more than one) that belong to a shortest path from u to v. More formally, $via(u,v) \equiv \{z \in N(u) \mid d(u,v) = w(u,z) + d(z,v)\}$.

2.2. Performance Model

As shown in [38], the performance of a distributed algorithm in the asynchronous model depend on the time needed by processors to execute the local procedures of the algorithm and on the delays incurred in the communication among nodes. These parameters heavily influence the scheduling of the distributed computation and hence the number of messages sent. For these reasons, to properly analyze the behaviour of the solutions described in this paper with respect to convergence time, in what follows we consider the so-called FIFO *network scenario* as it is universally considered the most suited for analyzing the performance of distributed algorithms in the considered setting.

The FIFO network scenario can be briefly summarized as follows: the weight of an edge models the time needed to traverse the corresponding link (the delay occurring on that edge if a packet is sent on it) and all the processors require the same time to process every procedure (the delay occurring on a processor if a procedure is performed on it), which is assumed to be instantaneous. In this way, the distance between two nodes models the minimum time that such nodes need to communicate. Then, the time complexity is measured as the number of steps performed by the processors, that is the number of times that a processor performs a procedure.

In this paper, we concentrate on the realistic case of *dynamic networks*, i.e. networks that vary over time due to change operations occurring on the processors or on the communication channels, respectively. We denote a sequence of update operations on the edges of graph G representing the network by $\mathscr{C} = (c_1, c_2, ..., c_k)$. Assuming $G_0 \equiv G$, we denote by G_i, $0 \le i \le k$, the graph obtained by applying c_i to G_{i-1}. Without loss of generality, we restrict our focus on the case where operation c_i either increases or decreases the weight of an existing edge in G_i, as insertions and deletions of nodes and edges can be easily modelled as weight changes (see, e.g., [6] for more details). Moreover, we consider the case of networks in which a change in the weight of an edge (either increase or decrease) can occur while one or more other edge weight changes are under processing. A 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. We will use $w^t()$, $d^t()$, and $via^t()$ to denote a given edge weight, distance, or via in graph G_t, respectively.

2.3. Complexity Measures

In the remainder of the paper, the performance of some of the considered algorithms will be measured in terms of two parameters, namely δ and Δ, which have been considered in several works on the matter (see, e.g. [5,20,21,29] and reference therein) since they capture pretty well the amount of distributed computation that has to be carried out to update the shortest paths in dynamic networks, as a consequence of one or more update operations.

In more details, given a sequence $\mathscr{C} = (c_1, c_2, ..., c_k)$ of update operations, we define parameter $\sigma_{c_i,s}$ to represent, for each operation c_i and for each node s, the set of nodes that change either the distance or the via toward s as a consequence of c_i. More formally, such parameter is defined as

$$\sigma_{c_i,s} = \{v \in V \mid d^{t_i}(v,s) \neq d^{t_{i-1}}(v,s) \text{ or } via^{t_i}(v,s) \neq via^{t_{i-1}}(v,s)\}.$$

If a node $v \in \cup_{i=1}^{k}\{\cup_{s\in V}\sigma_{c_i,s}\}$, then v is said to be *affected*. We denote by Δ the overall number of affected nodes, $\Delta = \sum_{i=1}^{k}\sum_{s\in V}|\sigma_{c_i,s}|$. Furthermore, given a generic destination s in V, $\sigma_s = \cup_{i=1}^{k}\sigma_{c_i,s}$ and $\delta = \max_s|\sigma_s|$. Note that, it follows that a node can be affected for at most δ different destinations.

2.4. Distance-Vector Algorithms

Most of Distance-Vector algorithms are able to handle concurrent updates, and share a set of common features which can be briefly summarized as follows. Given a weighted graph $G = (V,E,w)$, a generic node v of G executing a Distance-Vector algorithm:

- knows the identity of any other node of G, as well as the identity of its neighbors and the weights of its adjacent edges;
- maintains a routing table that has n entries, one for each $s \in V$, which consists of at least two fields:
  - the *estimated distance* $\mathrm{D}_v[v,s]$ towards s, i.e. an estimation of $d(v,s)$;
  - the *estimated via* $\mathrm{VIA}_v[s]$ towards s, i.e. an estimation of $via(v,s)$;
- handles edge weight increases and decreases either all together, by a single procedure, or by two separate routines; in the former case (see, e.g., [5]), we will denote such unified routine by HANDLECHANGEW, while in the latter case (see, e.g., [6]), we will denote the two procedures by HANDLEINCREASEW and HANDLEDECREASEW, respectively;
- requests data to neighbors, regarding estimated distances, and receives the corresponding replies from them, through a dedicated exchange of messages (for instance, by sending a *query* message, like in [5], or by sending a $get.feasible.dist$ message, like in [29]);
- propagates a variation, occurring on an estimation on the distance or on the via, to the rest of the network as follows:
  - if v is performing HANDLECHANGEW, then it sends out to its neighbors a dedicated notification message (from now on denoted by *update*); a node that receives this kind of message executes a corresponding routine, from now on denoted by HANDLEUPDATE;
  - if v is performing HANDLEINCREASEW (HANDLEDECREASEW, respectively) then it sends to its neighbors a dedicated notification message (denoted from now on by *increase* or *decrease*, respectively); a node that receives an *increase* (*decrease*, respectively) message executes a corresponding routine, from now on denoted by HANDLEINCREASE (HANDLEDECREASE, respectively).

Moreover, it is known that a Distance-Vector algorithm can be designed to be free of looping or count-to-infinity phenomena by incorporating suitable sufficient conditions in the routing table update procedures. Three of such conditions are given in [5]. The less restrictive, and easier to implement, of the conditions in [5], is the so-called SOURCE NODE CONDITION (SNC), which can be implemented to work in combination with a Distance-Vector algorithm if and only if such an algorithm maintains, besides the already mentioned routing table, a so-called *topology table*. The topology table of a node v must contain enough information to allow the node to compute, for each $u \in N(v)$ and for each $s \in V$, the value $\mathrm{D}_v[u,s]$, i.e. an estimation on the distance $d(u,s)$ from u to s as it is known to v [5]. These values are then exploited by the SNC to establish whether a path is free of loops as follows. If, at time t, v needs to change $\mathrm{VIA}_v[s]$ for some $s \in V$, then it can select as new via any neighbor $k \in N(v)$ satisfying both the conditions of the following *loop-free test*:

1. $\mathrm{D}_v[k,s](t) + w^t(v,k) = \min_{v_i \in N(v)}\{\mathrm{D}_v[v_i,s](t) + w^t(v_i,v)\}$, and
2. $\mathrm{D}_v[k,s](t) < \mathrm{D}_v[v,s](t)$,

where $\mathrm{D}_v[i,s](t)$ denotes, in this case, the estimated distance of neighbor $i \in N(v)$ as it is known to v at time t. If no such neighbor exists, then $\mathrm{VIA}_v[s]$ does not change.

If $\mathrm{VIA}_G[s](t)$ denotes the directed subgraph of G induced by the set $\{\mathrm{VIA}_v[s](t),$ for each $v \in V\}$, of the estimated vias, at time t, then the following result holds.

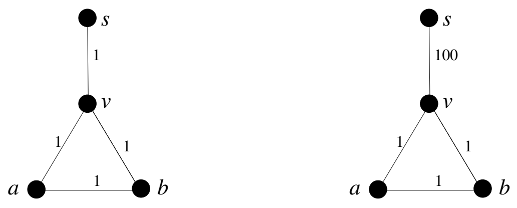

Figure 1. A graph G before and after a weight increase on the edge (s,v). Edges here are labelled with their weights.

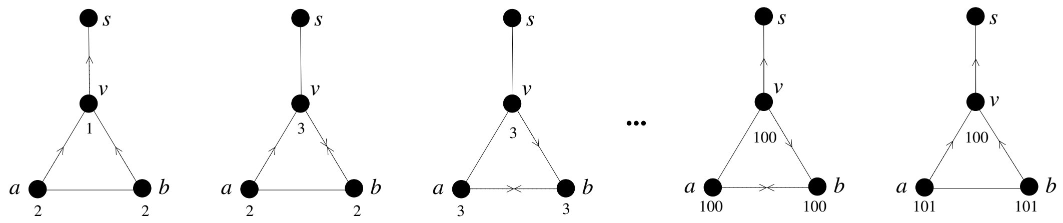

Figure 2. The sequence of computations of $\mathrm{D}_u[u,s]$ and $\mathrm{VIA}_u[s]$ by a node u executing DBF. The value close to a node denotes its distance towards s while an arrowhead from x to y in edge (x,y) indicates that node y is the estimated via of x towards s.

Theorem 2 ([5]). *Given a graph G, let us suppose that $\mathrm{VIA}_G[s](t_0)$ is loop-free at time t_0. If G undergoes a sequence of updates starting at a time $t' \geq t_0$ and* SNC *is used when nodes have to change their via, then $\mathrm{VIA}_G[s](t)$ remains loop-free, for any $t \geq t' \geq t_0$.*

3. Distributed Bellmann-Ford Algorithm

This section summarizes the main characteristics of the Distributed Bellmann-Ford (DBF) algorithm.

DBF requires each node v in the network to store the last known estimated distance $\mathrm{D}_v[u,s]$ towards any other node $s \in V$, received from each neighbor $u \in N(v)$. In DBF, a node v updates its estimated distance $\mathrm{D}_v[v,s]$ toward a node s by simply executing the iteration $\mathrm{D}_v[v,s] := \min_{u\in N(v)}\{w(v,u) + \mathrm{D}_v[u,s]\}$, when needed.

As already mentioned in the Introduction, DBF can incur in the well-known looping and count-to-infinity problems, which arise when a certain kind of link failure or weight increase operation occurs in the network. In Figure 1, we show 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 occurring on edge (s,v). In Figure 2, we show the corresponding steps required by DBF to update both the distance and the via towards a distinguished node s, for each node of G, as a consequence of the change. In detail, when the weight of edge (s,v) increases to 100, node v updates its distance and via towards s by setting $\mathrm{D}_v[v,s]$ to 3 and $\mathrm{VIA}_v[s]$ to node 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 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 estimated via towards s is b (a, respectively) and its new distance is 3. In fact, according to a's information $\mathrm{D}_a[v,s] = 3$ and $\mathrm{D}_a[b,s] = 2$, therefore $w(a,b) + \mathrm{D}_a[b,s] < w(a,v) + \mathrm{D}_a[v,s]$. Subsequent updating steps (but the last one) do not change the estimated 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., by the weight of edge (a,b)). The counting stops after a number of updating steps that depends on the new weight of 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.

In Figure 3 we give an example of the execution of DBF on another (simple) network (which is part of an example in [5]) where a weight increase operation occurs and the algorithm does not count to infinity. 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.

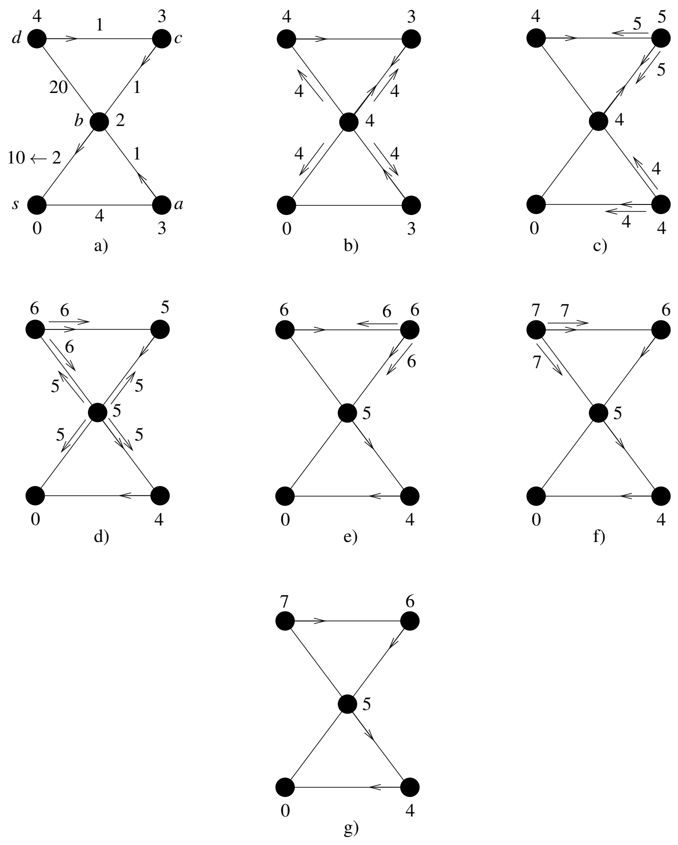

Figure 3. Example of execution of DBF.

At a certain point in time, edge (b,s) changes its weight from 2 to 10 (see Figure 3(a)). When node b detects the weight increase, it updates the value of $\mathrm{D}_b[b,s]$ to the minimum possible value, that is $\mathrm{D}_b[b,s] = \min_{u\in N(b)}\{w(b,u)+\mathrm{D}_b[u,s]\} = w(b,c)+\mathrm{D}_b[c,s] = 4$. Then, node b sends $\mathrm{D}_b[b,s]$ to all its neighbors (Figure 3(b)). As a consequence of such messages, nodes a and c update $\mathrm{D}_a[b,s]$ and $\mathrm{D}_c[b,s]$, respectively, compute their optimal distances to s that are 4 and 5, respectively, and send them to their own neighbors (Figure 3(c)). Nodes s and d only update $\mathrm{D}_s[b,s]$ and $\mathrm{D}_d[b,s]$, respectively. In Figure 3(d), node b updates $\mathrm{D}_b[c,s]$ to 5 as a consequence of the message sent by c. As c was the successor node of b towards s, b needs to update $\mathrm{D}_b[b,s]$ to $\min_{u\in N(b)}\{w(b,u)+\mathrm{D}_b[u,s]\} = w(b,a)+\mathrm{D}_b[a,s] = 5$. After this update, b sends $\mathrm{D}_b[b,s]$ to its neighbors. Node d behaves similarly by updating its distance to s to 6. In Figures 3(e)–3(g), the message sent by b is propagated to nodes c and d in order to update the distances from this nodes to s.

As a concluding remark of this section, we recall the reader that, if the nodes of the network are not synchronized, even in the *static* case, i.e. when no change occurs in the network, it can be shown that the overall number of messages sent by DBF is, in the worst case, exponential with respect to the number of nodes in the network.

4. Diffuse Update Algorithm

This section describes the main characteristics of the Diffuse Update ALgorithm (DUAL) of [5]. The algorithm is described with respect to a source $s \in V$, and it starts every time a weight change $c_i \in \mathscr{C} = (c_1, c_2, ..., c_k)$ occurs on an edge (x_i, y_i).

4.1. Data Structures

DUAL is more complex than DBF, and uses different data structures in order to guarantee freedom from looping and counting to infinity phenomena. In detail, it stores, for each node v and for each destination s, a slightly modified routing table where the two fields are the *estimated distance* $\mathrm{D}_v[v,s]$ and the so-called *feasible successor* $\mathrm{FS}_v[s]$, respectively. This latter value takes the place of the standard value of $\mathrm{VIA}_v[s]$, and represents an estimation on $via(v,s)$ that is always guaranteed to induce a loop-free $\mathrm{VIA}_G[s](t)$ at any time t [5]. In order to compute $\mathrm{FS}_v[s]$, DUAL requires that each node v be able to determine, for each destination s, a set of neighbors called the Feasible Successor Set, denoted as $\mathrm{FSS}_v[s]$. To this aim, each node v explicitly stores the topology table, which contains, for each $u \in N(v)$, the distance $\mathrm{D}_v[u,s]$ from u to s. Then, it computes $\mathrm{FSS}_v[s]$ by using the SNC sufficient condition. In more details, node $u \in N(v)$ is inserted in $\mathrm{FSS}_v[s]$ if the estimated distance $\mathrm{D}_v[u,s]$ from u to s is smaller than the so-called *feasible distance* $\mathrm{FD}_v[v,s]$ from v to s. If a neighbor $u \in N(v)$, through which the distance from v to s is minimum, is in $\mathrm{FSS}_v[s]$, then u is chosen as feasible successor. Moreover, in order to guarantee mutual exclusion in case multiple weight change operations occur, each node v performing DUAL uses some auxiliary data structures: (i) an auxiliary distance $\mathrm{RD}_v[v,s]$, for each $s \in V$; and (ii) a finite state machine to process these multiple updates sequentially. The state of the machine consists, for each $s \in V$, of three variables: the query origin flag $\mathrm{O}_v[s]$ and the state $\mathrm{ACTIVE}_v[s]$, which contain an integer and a boolean entry, respectively, and the replies status flag $\mathrm{R}_v[u,s]$, which contains a boolean entry for each neighbor $u \in N(v)$. It follows that DUAL requires $\Theta(n \cdot maxdeg)$ space per node, as all the data structures stored by a node v are arrays of size n, with the exception of the topology table $\mathrm{D}_v[u,s]$ and the replies status flag, which are permanently allocated and require $\Theta(n \cdot maxdeg)$ space.

4.2. Algorithm

The main core of DUAL is a sub-routine, named DIFFUSE-COMPUTATION, which is performed by a generic node v, every time $\mathrm{FSS}_v[s]$ does not include the node $u \in N(v)$ through which the distance from v to s is minimum. The DIFFUSE-COMPUTATION works as follows: node v sends queries to all its neighbors with its distance through $\mathrm{FS}_v[s]$ by using message *query*. Accordingly, v sets $\mathrm{R}_v[u,s]$ to true, for each $u \in N(v)$, in order to keep trace of which neighbor has answered to the *query* message (the value is set to false when a corresponding *reply* message is received). From this point onwards v does not change its feasible successor to s until the DIFFUSE-COMPUTATION terminates.

When a neighbor $u \in N(v)$ receives a *query*, which triggers the execution of procedure QUERY whose purpose is to try to determine if a feasible successor to s, after such update, exists. If so, it replies to the *query* by sending message *reply* containing its own distance to s. Otherwise, u propagates the DIFFUSE-COMPUTATION toward the rest of the network. In details, it sends out queries and waits for the replies from its neighbors before replying to v's original *query*. To guarantee that each node is involved in one DIFFUSE-COMPUTATION phase at the time, for a certain $s \in V$, an appropriate finite state machine behaviour is implemented by variables $\mathrm{O}_v[s]$ and $\mathrm{ACTIVE}_v[s]$. Changes to distances, feasible distances and successors are allowed only under specific circumstances. Moreover, an auxiliary variable $\mathrm{RD}_v[v,s]$, representing an upper bound to $\mathrm{D}_v[v,s]$ is used by each node v, for each $s \in V$, to answer to certain types of queries, under the same circumstances, in order to avoid loops. We refer the reader to [5] for an exhaustive discussion on the subject. In the same paper, the authors show that the DIFFUSE-COMPUTATION always terminates, i.e. that there exists, under the FIFO assumption, a time when a node receives messages *reply* by all its neighbors. At that point, it updates its distance and feasible successor, with the minimum value obtained by its neighbors and the neighbor that provides such distance. This is done during the execution of procedure REPLY, which is invoked upon the reception of each REPLY message. At the end of a DIFFUSE-COMPUTATION execution, a node sends message *update* containing the new computed distance to its neighbors. As mentioned above, DUAL starts every time a node x_i detects a weight change operation c_i occurring on one of its adjacent edges, say (x_i, y_i). In what follows, the cases in which c_i is a weight decrease and a weight increase operation are considered separately.

Weight decrease. If c_i is a weight decrease operation on (x_i, y_i), node x_i first tries to determine whether node y_i can be chosen as new $\mathrm{FS}_{x_i}[s]$ or not, for each $s \in V$, without performing DIFFUSE-COMPUTATION. In fact, since

c_i can induce only decreases in the distances, SNC is trivially always satisfied by at least one neighbor, which is either the current $\mathrm{FS}_{x_i}[s]$ or y_i itself. This is done by invoking procedure DISTANCEDECREASE (y_i, s) for all $s \in V$, and the same routine is performed, symmetrically, by node y_i. In any of the two cases, propagates the change by sending *update* messages to its neighbors, with the aim of notifying either a change in the distance or in the distance and the feasible successor. Each node in the graph, which receives such *update* message, in turn, determines whether $\mathrm{FS}_v[s]$ has to be updated or not in the same way, and possibly propagates the change. Note that, as the FIFO case is under consideration, each node of the graph updates its data structures related to s at most once as a consequence of c_i. Hence, since there are $|\sigma_{c_i,s}|$ nodes that change their distance or feasible successor to s as a consequence of c_i and since each node v in $\sigma_{c_i,s}$ sends at most *maxdeg update* messages, the number of messages, related to a source s, sent as a consequence of a weight decrease operation c_i is $O(maxdeg \cdot |\sigma_{c_i,s}|)$, while the number of steps required to converge is $O(|\sigma_{c_i,s}|)$.

Weight increase. If c_i is a weight increase operation, the only nodes that sends messages, as a consequence of operation c_i and w.r.t. a source s, are those in $\sigma_{c_i,s}$ and their neighbors. In particular, after c_i occurs on (x_i, y_i), node x_i tries, for each $s \in V$, to determine whether a feasible successor still exists or not, by checking if nodes in $\mathrm{FSS}_v[s]$ still satisfy SNC. This is done by inkoving procedure DISTANCEINCREASE (y_i, s) for all $s \in V$. The same routine is performed, symmetrically, by node y_i. In the affirmative case, node x_i immediately terminates its computation and sends an *update* message to each $u \in N(x_i)$ with the updated value of distance. Since we are considering the FIFO case, by SNC we know that, in the above case, the path in $\mathrm{VIA}_G[s]$ from u to s does not contain x_i. Then, it follows that also node u does not execute a DIFFUSE-COMPUTATION nor send *update* to x_i, as a consequence of c_i. In the negative case, i.e. node x_i performs a DIFFUSE-COMPUTATION, and sens *query* messages to all its neighbors, and possibly (depending on the presence of alternative paths) induces other nodes in $\mathrm{VIA}_G[s]$ to perform DIFFUSE-COMPUTATION. When x_i receives all the *reply* messages, it chooses as new feasible successor a neighbor $u \in N(x_i)$ which, in turn, does not perform DIFFUSE-COMPUTATION nor send *update* messages to x_i, with respect to s, as a consequence of c_i.

In any of the above cases, node x_i sends $O(|N(x_i)|)$ *update* messages while only in the second case, it sends $O(|N(x_i)|)$ *query* messages and each of the nodes in $N(x_i)$ sends $O(1)$ *reply* messages. Note that each node $v \in \sigma_{c_i,s}$ behaves as x_i when it receives either *update* or *query* messages from $\mathrm{FS}_v[s]$. As a consequence, it follows that the total number of messages sent by each node $v \in \sigma_{c_i,s}$ is $O(|N(v)|) = O(maxdeg)$ and that the overall number of messages related to the source s sent as a consequence of a weight increase operation c_i is $O(maxdeg \cdot |\sigma_{c_i,s}|)$ while the number of steps required to converge is $O(|\sigma_{c_i,s}|)$.

Since $\sum_{i=1}^{k} \sum_{s \in V} |\sigma_{c_i,s}| = \Delta$, it follows that the overall number of messages sent by DUAL during a sequence of weight modifications $\mathscr{C} = (c_1, c_2, ..., c_k)$, in the FIFO case, and for each possible source s, is given by $\sum_{i=1}^{k} \sum_{s \in V} O\left(maxdeg \cdot |\sigma_{c_i,s}|\right) = O(maxdeg \cdot \Delta)$, while the overall number of steps required by the algorithm to converge is $\sum_{i=1}^{k} \sum_{s \in V} O(|\sigma_{c_i,s}|) = O(\Delta)$. By the above discussion the next theorem follows.

Theorem 4 ([5]). DUAL *requires* $O(maxdeg \cdot \Delta)$ *messages,* $O(\Delta)$ *steps, and* $\Theta(n \cdot maxdeg)$ *space occupancy per node.*

4.3. Example of Execution

In Figure 4 we propose an example of execution of DUAL, which is inspired by an example given in [5]. The example focuses on the graph of Figure 4(a), and on destination s. In the figure, the value close to a node denotes its distance to node s, and an arrowhead from x to y in edge (x, y) represents node y being the successor of x toward 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) increases its weight from 2 to 10 (Figure 4(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 its current distance, that is 2. Accordingly, it starts a DIFFUSE-COMPUTATION by sending a query to its neighbors (Figure 4(b)). In Figure 4(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

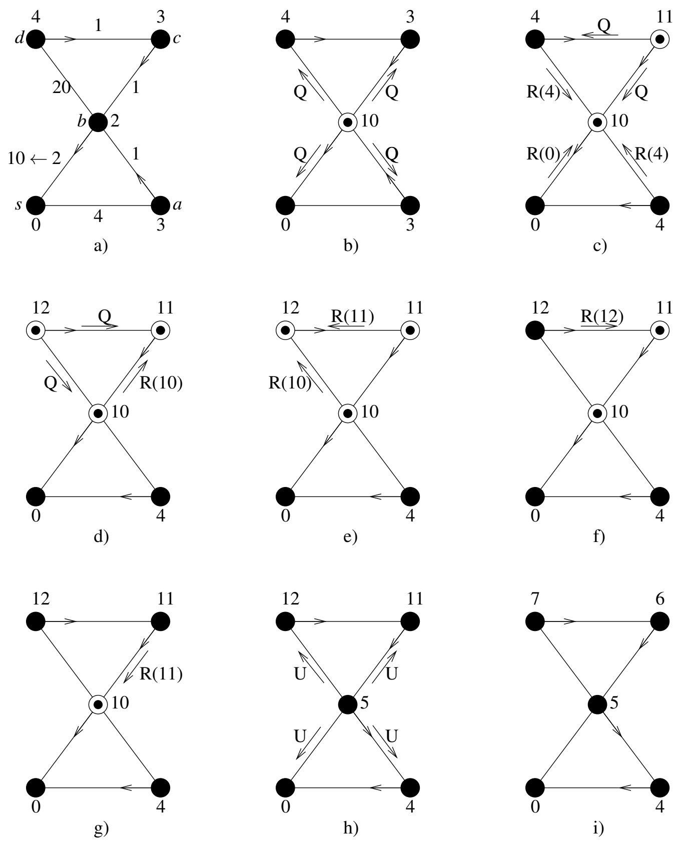
Figure 4. Example of execution of DUAL.

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 4(d)). When node *d* receives all the replies to its *query* (Figure 4(e)), it computes its new distance and successor (12 and *c*, respectively), and sends a *reply* to *c*'s *query* (Figure 4(f)). Nodes *c* and *b* operate in a similar manner when they receives all the replies to their respective queries (Figure 4(f)–4(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 4(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 4(i)).

As a final observation, notice that, the undesirable count-to-infinity phenomenon shown in Figure 2 of Section 3, induced by the use of DBF, does not occur if DUAL is used, with SNC. In fact, for instance, at step 2, the SNC prevents node *v* to choose *b* as its successor, since the loop-free test fails. This triggers an execution of DIFFUSE-COMPUTATION, which is guaranteed to always produce an acyclic sub-graph induced by the feasible successors [5].

5. Loop Free Routing Algorithm

This section describes the Loop Free Routing (LFR) algorithm, introduced for the first time in [39], and then extended and established in [29]. The algorithm consists of four procedures named UPDATE, DECREASE, INCREASE and SENDFEASIBLEDIST, respectively. The algorithm is described with respect to a source $s \in V$, and it starts every time a weight change $c_i \in \mathscr{C} = (c_1, c_2, ..., c_k)$ occurs on an edge (x_i, y_i).

5.1. Data Structures

As well as DUAL, LFR is more complex than DBF, and uses a different set of data structures to realize a loop-free behaviour. In more details, it maintains, for each node *v*, a customized version of the standard routing table, that consists of two arrays $\mathrm{D}_v[v,s]$ and $\mathrm{FS}_v[s]$, that store the *estimated distance* and the so-called *feasible via*, respectively. This latter value represents a different kind of estimation on $via(v,s)$ that is always guaranteed to induce loop-free $\mathrm{VIA}_G[s](t)$ at any time t [29]. In addition, for each $s \in V$, a node v executing LFR stores the following data structures: $\mathrm{STATE}_v[s]$: the state of node v with respect to source s (v is in *active* state and $\mathrm{STATE}_v[s] = true$ if and only if it is performing procedure INCREASE or procedure SENDFEASIBLEDISTwith respect to s); $\mathrm{UD}_v[s]$: the estimated distance from v to s through the current $\mathrm{FS}_v[s]$ (in particular, if v is active $\mathrm{UD}_v[s]$ is always greater than or equal to $\mathrm{D}_v[v,s]$, otherwise they coincide). In addition, to implement the topology table and thus SNC, a node v stores a *temporary* data structure TEMPD_v. Such array TEMPD_v is allocated at v for a certain s only when needed, that is when v becomes active with respect to a certain s, and it is deallocated right after v turns back in passive state with respect to the same s. The entry $\mathrm{TEMPD}_v[u][s]$ contains $\mathrm{UD}_u[s]$, for each $u \in N(v)$, and hence TEMPD_v takes $O(maxdeg)$ space per node.

5.2. Algorithm

At any time $t < t_1$, before LFR starts, it is assumed that, for each pair of nodes $v,s \in V$, the values stored in $\mathrm{D}_v[v,s](t)$ and $\mathrm{FS}_v[s](t)$ are correct, that is $\mathrm{D}_v[v,s](t) = d^t(v,s)$ and $\mathrm{FS}_v[s](t) \in via^t(v,s)$. The description focuses on a distinguished node $s \in V$ and each node $v \in V$, at time t, is assumed to be passive with respect to s.

The algorithm starts when the weight of an edge (x_i,y_i) changes. As a consequence, x_i (y_i, respectively) sends to y_i (x_i, respectively) message $update(x_i,s,\mathrm{D}_{x_i}[x_i,s])$ ($update(y_i,s,\mathrm{D}_{y_i}[y_i,s])$, respectively). Messages received at a node are stored in a queue and processed in FIFO order to guarantee mutual exclusion. If an arbitrary node v receives $update(u,s,\mathrm{D}_u[u,s])$ from $u \in N(v)$, then it performs procedure UPDATE, which simply compares $\mathrm{D}_v[v,s]$ with $\mathrm{D}_u[u,s] + w(u,v)$ to determine whether v needs to update its estimated distance and or its estimated feasible via to s.

If node v is active, then the processing of the message is postponed by enqueueing it into the FIFO queue associated to s. Otherwise, we distinguish three cases, and discuss them separately, depending on the type of change in the estimated distance (or feasible via) that is induced by the message. In particular, if $\mathrm{D}_v[v,s] > \mathrm{D}_u[u,s] + w(u,v)$, then v performs procedure DECREASE, while if $\mathrm{D}_v[v,s] < \mathrm{D}_u[u,s] + w(u,v)$, then v performs procedure INCREASE. Finally, if node v is passive and $\mathrm{D}_v[v,s] = \mathrm{D}_u[u,s] + w(u,v)$ then it follows that there is more than one shortest path from v to s. In this case the message is discarded and the procedure ends.

Weight decrease. When a node v performs procedure DECREASE, it simply updates D, UD and FS data structures by using the updated information provided by u. Then, the update is forwarded to all neighbors of v with the exception of $\mathrm{FS}_v[s]$ which is node u.

Weight increase. When a node v performs procedure INCREASE, it first checks whether the update has been received from $\mathrm{FS}_v[s]$ or not. In the negative case, the message is simply discarded while, in the affirmative case (only) v needs to change its estimation on distance and feasible via to s. To this aim, node v becomes active, allocates the temporary data structure TEMPD_v, and sets $\mathrm{UD}_v[s]$ to the current distance through $\mathrm{FS}_v[s]$. At this point, v first performs the so called LOCAL-COMPUTATION, which involves all the neighbors of v. If the LOCAL-COMPUTATION does not succeed, then node v initiates the so called GLOBAL-COMPUTATION, which involves in the worst case all the other nodes of the network.

During the LOCAL-COMPUTATION, node v sends *get.dist* messages, carrying $\mathrm{UD}_v[s]$, to all its neighbors, with the exception of u. A neighbor $k \in N(v)$ that receives a *get.dist* message, immediately replies with the value $\mathrm{UD}_k[s]$, and if k is active, it updates $\mathrm{TEMPD}_k[v][s]$ to $\mathrm{UD}_v[s]$. When node v receives these values from its neighbors, it stores them in the array TEMPD_v, and it uses them to compute the minimum estimated distance D_{min} to s and the neighbor VIA_{min} which gives such a distance.

At the end of the LOCAL-COMPUTATION v checks whether a feasible via exists, by executing the loop-free test, according to the SNC. If the test fails, then v initiates the GLOBAL-COMPUTATION, in which it entrusts the neighbors the task of finding a loop-free path. In this phase, v sends $get.feasible.dist(v,s,\mathrm{UD}_v[s])$ message to each of its neighbors. This message carries the value of the temporary estimated distance through its current feasible via. This distance is not guaranteed to be minimum but it is guaranteed to be loop-free. When v receives the answers to $get.feasible.dist$ messages from its neighbors, again it stores them in TEMPD_v and it uses them to compute the minimum estimated distance D_{min} to s and the neighbor VIA_{min} which gives such a distance. At this point, v has surely found a feasible via to s and hence it deallocates TEMPD_v, updates $\mathrm{D}_v[v,s]$, $\mathrm{UD}_v[s]$ and $\mathrm{FS}_v[s]$ and propagates the change by sending *update* messages to all its neighbors. Finally, v turns back in passive state and starts processing another message in the queue, if any.

A node $k \in N(v)$ that receives a *get.feasible.dist* message performs procedure SENDFEASIBLEDIST. If $\mathrm{FS}_k[s] = v$ and k is passive, then procedure SENDFEASIBLEDIST behaves similarly to procedure INCREASE. The only difference is that SENDFEASIBLEDIST needs to 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, it becomes active, allocates the temporary data structure TEMPD_k, and sets $\mathrm{UD}_k[s]$ to the current distance through $\mathrm{FS}_v[s]$. Then, as in procedure INCREASE, k first performs the LOCAL-COMPUTATION, which involves all the neighbors of k. If the LOCAL-COMPUTATION fails, that is SNC is violated, then node k initiates the GLOBAL-COMPUTATION, 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 as answer to the *get.feasible.dist* message. Now, as in procedure INCREASE, node k can deallocate TEMPD_v, update its local data structures $\mathrm{D}_v[v,s]$, $\mathrm{UD}_v[s]$ and $\mathrm{FS}_v[s]$, and propagate the change by sending *update* messages to all its neighbors. Finally, v turns back in passive state and starts processing another message in the queue, if any.

Concerning the space complexity, LFR takes $O(n + maxdeg \cdot \delta)$ space per node, as all the data structures, stored by a node v, are arrays of size n, with the exception of $\mathrm{TEMPD}_v[\cdot][s]$ which is allocated only when node v becomes active for a certain destination s, that is only if $v \in \delta_{c_i,s}$, and deallocated when v turns back in passive state for s, that is at most δ times. As each entry $\mathrm{TEMPD}_v[\cdot][s]$ requires $O(maxdeg)$ space, the total space per node is $O(n + maxdeg \cdot \delta)$ in the worst case.

Concerning the message and time complexity, given a source s and a weight change operation $c_i \in \mathscr{C}$ on edge (x_i, y_i), the cases in which c_i is a weight decrease or a weight increase operation are considered separately. If c_i is a weight decrease operation, only nodes in $\sigma_{c_i,s}$ update their data structures and send messages to their neighbors. In detail, a node v can update its data structures related to s at most once as a consequence of c_i and, in this case, it sends $|N(v)|$ messages. Hence, v sends at most *maxdeg* messages. Since there are $|\sigma_{c_i,s}|$ nodes that change their distance or via to s as a consequence of c_i, the number of messages related to the source s sent as a consequence of a weight decrease operation c_i is $O(maxdeg \cdot |\sigma_{c_i,s}|)$, while the number of steps required to converge is $O(|\sigma_{c_i,s}|)$.

If c_i is a weight increase operation, the only nodes which send messages with respect to operation c_i and source s are those in $\sigma_{c_i,s}$ and the neighbors of such nodes. In detail, each time that a node $v \in \sigma_{c_i,s}$ executes procedures INCREASE and SENDFEASIBLEDIST, it sends $O(|N(v)|)$ messages and each of the nodes in $N(v)$ sends $O(1)$ messages, for a total number of $O(|N(v)|) = O(maxdeg)$ messages sent. In the FIFO case, node v performs either procedure INCREASE or procedure SENDFEASIBLEDIST at most once with respect to c_i and s. It follows that each $v \in \sigma_{c_i,s}$ sends at most $O(maxdeg)$ messages. Therefore, the overall number of messages related to the source s sent as a consequence of operation c_i is $O(maxdeg \cdot |\sigma_{c_i,s}|)$ while the number of steps required to converge is $O(|\sigma_{c_i,s}|)$. Now, since $\sum_{i=1}^{k} \sum_{s \in V} |\sigma_{c_i,s}| = \Delta$, it follows that the overall number of messages sent during the whole sequence $\mathscr{C}$ and for each possible source s, is given by $\sum_{i=1}^{k} \sum_{s \in V} O\left(maxdeg \cdot |\sigma_{c_i,s}|\right) = O(maxdeg \cdot \Delta)$, while the overall number of steps required by the algorithm to converge is $\sum_{i=1}^{k} \sum_{s \in V} O(|\sigma_{c_i,s}|) = O(\Delta)$.

The next theorem follows from the above discussion.

Theorem 5 ([29]). LFR *requires* $O\left(maxdeg \cdot \Delta\right)$ *messages,* $O(\Delta)$ *steps, and* $O(n + maxdeg \cdot \delta)$ *space occupancy per node.*

5.3. Example of Execution

Figure 5 shows an example of execution of LFR on the same graph of Figure 4(a), where the focus is on shortest paths towards node s. Given a node v, $\mathrm{FS}_v[s]$ is represented by an arrow from v to $\mathrm{FS}_v[s]$, and $\mathrm{D}_v[v,s]$

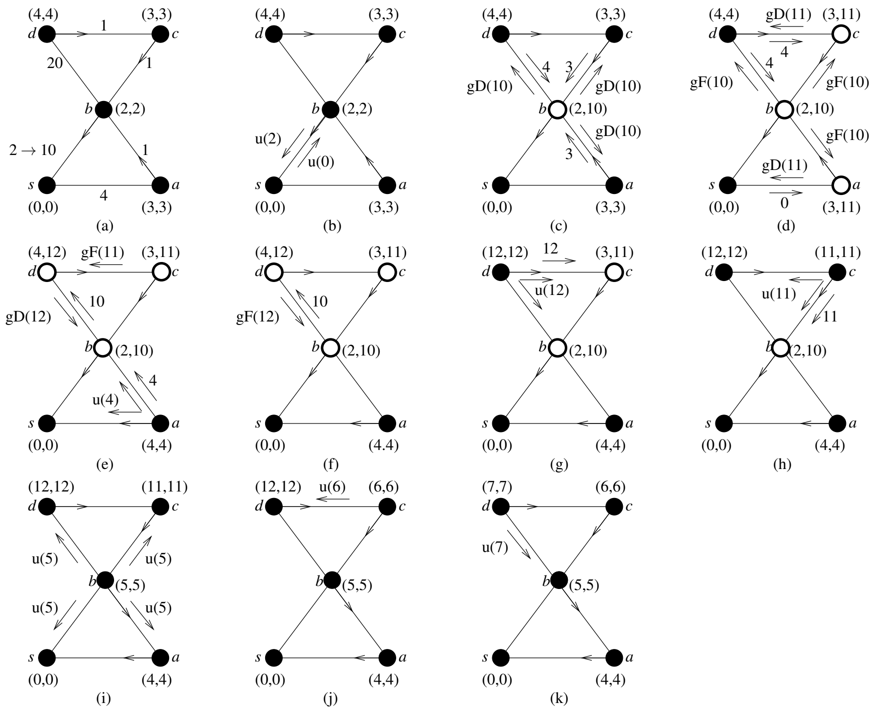

Figure 5. Example of execution of LFR.

and $UD_v[s]$ by a pair of values associated to v. Before LFR starts it is assumed that all nodes are in *passive* state, that is none of them is involved in a computation with respect to s. Passive nodes are represented as black circles, while active nodes by white circles. In the example, the algorithm starts when the weight of (b,s) increases from 2 to 10 (Figure 5(a)). As a consequence b sends to s message $update(b,s,D_b[b,s])$, and s sends to b message $update(s,s,D_s[s,s])$, denoted as u(2) and u(0), respectively (Figure 5(b)).

When a node v receives an *update* message with an increased distance to s, it checks whether it comes from $FS_v[s]$ and, in the affirmative case, it performs Procedure INCREASE, otherwise it discards the message. Hence, when s receives u(2), it immediately discards it and terminates, as $FS_s[s] \neq b$. On the other hand, when node b, receives u(0), since $FS_b[s] = s$ and $D_b[b,s] < D_s[s,s] + w(b,s)$, it needs to update its own routing table and hence it performs Procedure INCREASE. In detail, b first performs the LOCAL-COMPUTATION in which it tries to understand whether its routing table can be updated by using only the distances to s of its neighbors. To this aim, b switches to the *active* state, sets $\text{TEMPD}_b[s][s] = 0$ and $UD_b[s] = \text{TEMPD}_b[s][s] + w(b,s) = 0 + 10 = 10$, and it sends to all its neighbors, except $FS_b[s] = s$, a *get.dist* message carrying $UD_b[s]$ denoted as gD(10) (Figure 5(c)). When a node $k \in N(b)$ receives gD(10), it immediately replies to b by sending $UD_k[s]$ (Figure 5(c)). By using the replies of its neighbors, b computes $D_{min} = \min\{\text{TEMPD}_b[k][s] + w(b,k) \mid k \in N(b)\}$ and $\text{VIA}_{min} = \text{argmin}\{\text{TEMPD}_b[k][s] + w(b,k) \mid k \in N(b)\}$ and performs the loop-free test of the SNC, to check whether the provided distance corresponds to a loop-free path or not. Then b compares $\text{TEMPD}_b[\text{VIA}_{min}][s]$ with $D_b[b,s]$, which represents the last value of a loop-free distance computed by b. If the test succeeds, it follows that b has a feasible via to s. Then it turns back in passive state, updates its routing table and propagates the change to its neighbors. Otherwise, b performs the GLOBAL-COMPUTATION, where it computes a loop-free path by involving the other nodes of the graph. In this phase b sends to its neighbors a *get.feasible.dist* message (denoted as gF in the figure), bringing the most up to date estimated distance to s through the current successor $FS_b[s]$ of b. In this case, $D_{min} = D_a[a,s] + w(b,a) = 4$ and $\text{TEMPD}_b[a][s] = 3 > D_b[b,s] = 2$, hence b performs GLOBAL-COMPUTATION (Fig. 5(d)).

When a neighbor k of b receives *get.feasible.dist*, it performs Procedure SENDFEASIBLEDIST. In detail, k first checks whether $FS_k[s] \neq b$ or $\text{STATE}_k[s] = true$. In this case, k immediately replies to b with $UD_k[s]$ (node d replies to b with 4 in Figure 5(d)). If $FS_k[s] = b$ and $\text{STATE}_k[s] = false$ then node k, sets

$\text{UD}_k[s] = \text{TEMPD}_b[b][s] + w(b,k)$ (for example node a sets $\text{UD}_a[s] = 10+1 = 11$ in Figure 5(d)), performs first LOCAL-COMPUTATION and then GLOBAL-COMPUTATION (nodes a and c in Figures 5(d)–5 (f)), in a way similar to Procedure INCREASE. To this aim, node a performs LOCAL-COMPUTATION (Figure 5(d)), which succeeds since $\text{TEMPD}_a[s][s] = 0$, and replies to b with 4 (Figure 5(e)). Differently from a, the LOCAL-COMPUTATION of node c fails, and hence c performs GLOBAL-COMPUTATION as well, by sending gF(11) to d (Fig. 5(e)). When node d receives such message, it performs LOCAL-COMPUTATION by sending gD(12) to b which replies with $\text{UD}_b[s] = 10 \neq \text{D}_b[b,s] = 2$. Since the SNC is not satisfied at d, because

- $min\{\text{TEMPD}_d[z][s] + w(d,z) \mid z \in N(d)\} = 12$
- $argmin\{\text{TEMPD}_d[z][s] + w(d,z) \mid z \in N(d)\} = c$ and
- $\text{TEMPD}_d[c][s] = 11 > \text{D}_d[d,s] = 4$

then d performs GLOBAL-COMPUTATION by sending gF(12) to b, which immediately replies with 10, and updates $\text{TEMPD}_b[d][s] = 12$ (Fig. 5(f)).

At the end of this process, node d finds a new feasible via, node c, and replies to c with the corresponding minimum estimated distance $\text{D}_c[c,s] + w(d,c) = 12$. In addition, node d updates its routing table and propagates the change to its neighbors. When c receives the answer to the gF message from d (Figure 5(g)) it behaves as d, by sending $\text{D}_b[b,s] + w(b,c) = 11$ to b, updating its routing table and propagating the change to its neighbors.

When b receives all the replies to its gF messages (Fig. 5(h)), it is able to compute the new loop-free shortest path to s, to update $\text{UD}_b[s] = \text{D}_b[b,s] = 5$, to turn back in passive state and to propagate the change by means of *update* messages (Fig. 5(i)). The *update* messages induce the neighbors of b to update their routing tables as described above for b. As the value $\text{UD}_b[s]$, sent by b during the GLOBAL-COMPUTATION is an upper bound to $\text{D}_b[b,s]$, these *update* messages induce the neighbors to perform procedure DECREASE (Figure 5(j–k)).

As a final observation of the section, notice that, the undesirable count-to-infinity phenomenon shown in Figure 2 of Section 3, induced by the use of DBF, does not occur if LFR is used, with SNC, as well as for DUAL. In fact, for instance, at step 2, the SNC prevents node v to choose b as its successor, since the loop-free test fails after the LOCAL-COMPUTATION phase. This triggers an execution of GLOBAL-COMPUTATION, which is guaranteed to always produce an acyclic sub-graph induced by the feasible successors [29].

6. Distributed Computation Pruning

This section describes the Distributed Computation Pruning (DCP) technique, introduced for the first time in [40] and then extended in [30]. The approach is not an algorithm by itself. Instead, it is a general technique that can be applied on top of a Distance-Vector algorithm for distributed shortest paths with the aim of improving its performance. Given a generic Distance-Vector algorithm **A**, the combination of DCP with **A** induces a new algorithm, denoted by **A**-DCP. The DCP technique is designed to be efficient mainly in power-law networks, by forcing the distributed computation to be carried out only by a subset of few nodes of the network.

6.1. Power-law Networks

A *power-law network*, in the most general meaning, is a network where the distribution of the nodes' degree follows a power-law trend, thus having many nodes with low degree and few (core) nodes with very high degree (see e.g. [32]). Such class of networks is very important from the practical point of view, since it includes many of the currently implemented communication infrastructures, like the Internet, the World Wide Web, some social networks, and so on. For this reason, a number of methods to generate artificial topologies exhibiting a power-law behaviour have been proposed in the literature. Among them, the most prominent are the *Barabási-Albert* model [32] and the Generalized Linear Preference (GLP) model [33].

In the following, we summarize some definitions that are useful to capture the typical structure of a power-law network, and some properties of the shortest paths in graphs of this kind, that are exploited by DCP. Given a graph $G = (V,E,w)$, nodes, edges and paths in G are classified as follows. A node $v \in V$ is *central* if $deg(v) \geq 3$ and non-central otherwise. A non-central node v can be either *semi-peripheral*, if $deg(v) = 2$, or *peripheral*, if $deg(v) = 1$. A node $v \in V$ is: *peripheral*, if $deg(v) = 1$; *semi-peripheral*, if $deg(v) = 2$; and *central* if $deg(v) \geq 3$. A peripheral or semi-peripheral node is *non-central*. A path $P = \{v_0, v_1, \ldots, v_j\}$ of G is *central* if v_i is central, for each $0 \leq i \leq j$. Any edge belonging to a central path is called *central edge*. A path $P = \{v_0, v_1, \ldots, v_j\}$ of G is *peripheral* if v_0 is central, v_j is peripheral, and all v_i, for each $1 \leq i \leq j-1$, are semi-peripheral. In this case, v_0 is called the *owner* of P and of any node belonging to P. Any edge belonging to a peripheral path, accordingly, is called *peripheral edge*. Finally, a path $P = \{v_0, v_1, \ldots, v_j\}$ of G is *semi-peripheral* if v_0 and v_j are two distinct central nodes, and all v_i are semi-peripheral nodes, for each $1 \leq i \leq j-1$. Nodes v_0 and v_j are called the *semi-owners* of P and of any node belonging to P. Any edge belonging to a semi-peripheral path is called semi-peripheral edge. A further distinction for semi-peripheral paths occurs if $v_0 \equiv v_j$. In this case P is called a *semi-peripheral cycle*, and node $v_0 \equiv v_j$ is called the *cycle-owner* of P and of any node belonging to P. Each edge belonging to such a path is called *cyclic edge* and each node $u \neq v_0$ in P is called *cyclic node*.

DCP has been designed to exploit the aforementioned topological properties of power-law networks in order to reduce the communication overhead induced by distributed computations executed by Distance-Vector algorithms. In particular, it forces the distributed computation to be carried out by the central nodes only (which are few in power-law networks). Non-central nodes, which are instead the great majority, play a passive role and receive updates about routing information from the respective owners, without taking part to any kind of distributed computation and by performing few trivial operation to update their routing data. Hence, it is clear that the larger is the set of non-central nodes of the network, the bigger is the improvement in the pruning of the distributed computation and, consequently, in the global number of messages sent by **A**-DCP.

6.2. Data Structures

In order to be implemented, DCP requires a generic node of G to store some additional information with respect to those required by **A**. In particular, each node v needs to store and update information about adjacent non-central paths of G. To this aim, v maintains a data structure called CHAINPATH, denoted as CHP_v, which is an array containing one entry $\text{CHP}_v[s]$, for each central node s. $\text{CHP}_v[s]$ stores the list of all edges, along with their weights, belonging to those non-central paths that contain s. To build the CHAINPATH data structure, it is assumed that v knows the degree of all nodes of the network belonging to non-central paths. The following properties clearly hold: i) a central node obviously does not appear in any list of CHP_v; ii) a peripheral node appears in exactly one list $\text{CHP}_v[s]$, where $s \in V$ is its owner; iii) a semi-peripheral node appears in exactly two lists $\text{CHP}_v[v_0]$ and $\text{CHP}_v[v_{j-1}]$, if it belongs to a semi-peripheral path (v_0 and v_{j-1} are its semi-owners), while it appears in a single list $\text{CHP}_v[v_0]$, if it belongs to a semi-peripheral cycle (v_0 is its cycle-owner). Hence, a node v, by using CHP_v is able to determine locally its type, and, in the case it is not central, it is also able to compute its owner, semi-owners, or cycle-owner. As shown in [30], the worst case space occupancy overhead per node due to the use of CHAINPATH is $O(n)$.

6.3. Description

The behaviour of a generic algorithm **A**, when combined with DCP, can be summarized as follows. The main difference resides in the fact that in a classic routing algorithm every node performs the same code thus having the same behaviour, while in **A**-DCP, given the nodes' and edges' classification, central and non-central nodes are forced to have different behaviours. In particular, central nodes detect (and handle) changes concerning all kinds of edges, while peripheral, semi-peripheral, and cyclic nodes detect (and handle) changes concerning only peripheral, semi-peripheral, and cyclic edges, respectively. In what follows, we summarize how each of the possible changes is handled by **A**-DCP.

Case (i). If the weight of a central edge (x,y) changes, then node x (y, respectively) performs the procedure provided by **A** for handling changes of this kind only with respect to central nodes. During this computation, if x (y, respectively) needs to know the estimated distances of its neighbors toward a central node s, it asks for it only to its central neighbors. If x (y, respectively) is the semi-owner of one or more semi-peripheral paths, it also asks for information to the other semi-owner of each semi-peripheral path, by means of a strategy called TRAVERSE-PATH. In details, node x (y, respectively) sends, for each semi-peripheral path he belongs to, a $sp.query(s)$ message to the corresponding semi-peripheral neighbor. The aim of this message is to traverse the semi-peripheral path in order to get the estimated distance, toward the considered node s, of the other semi-owner of the path. The $sp.query$ message contains only one field, i.e. the object s of the computation that has originated the message. When a semi-peripheral node receives a $sp.query(s)$ message from one of its two neighbors j, it simply performs a store-and-forward step and sends a $sp.query(s)$ message to the other neighbor $k \neq j$. The store-and-forward step is performed in a way that the ordering of the messages is preserved. A central node r that receives a $sp.query(s)$ message from one of its semi-peripheral neighbors u, simply replies to u with a $sp.reply(s, \mathrm{D}_r[r,s])$ message, which carries the estimated distance of r towards s, which was requested by x (y, respectively). When a semi-peripheral node receives a $sp.reply(s, \mathrm{D}_r[r,s])$ message from one of its two neighbors j, it simply performs a store-and-forward step and sends a $sp.reply(s, \mathrm{D}_r[r,s])$ message to the other neighbor $k \neq j$. The strategy terminates whenever the central node x (y, respectively) receives $sp.reply(s, \mathrm{D}_r[r,s])$: upon that event, x (y, respectively) stores $\mathrm{D}_r[r,s]$ and uses it, if needed, while executing the routine provided by **A** for the distributed computation of shortest paths.

Once x (y, respectively) has updated its own routing data toward a certain central node s, it propagates the variation to all its neighbors through a $gen.update(s, \mathrm{D}_x[x,s])$ ($gen.update(s, \mathrm{D}_y[y,s])$, respectively), which carries an updated value of $\mathrm{D}_x[x,s]$ ($\mathrm{D}_y[y,s]$, respectively). When a generic node v receives a $gen.update$ message from a neighbor u, it executes a procedure, called GENERALIZEDUPDATE, which, as first step, stores the current value of $\mathrm{D}_v[v,s]$ in a temporary variable $\mathrm{D}^{old}_v[s]$. Then, according to its status, the node performs different steps, which can be summarized as follows. If v is central, then it handles the change and updates its routing information toward the central node s by using the proper procedure of **A**, i.e. HANDLEUPDATE, HANDLEINCREASE, or HANDLEDECREASE, depending on the original structure of **A**, and forwards the change through the network accordingly.

If, instead, v is a peripheral node whose owner is node r, then $\mathrm{D}_v[v,s]$ is trivially updated by setting $\mathrm{D}_v[v,s] = \mathrm{D}_v[v,r] + \mathrm{D}_v[r,s]$. Moreover, any specific data structure of **A** is accordingly updated, and $\mathrm{D}_v[r,s]$ is propagated to the other neighbor of v. Moreover, if v is a cyclic node whose cycle-owner is node r, then v sets $\mathrm{D}_v[v,s] = \mathrm{D}_v[v,r] + \mathrm{D}_v[r,s]$. Moreover, since $\mathrm{D}_v[r,s]$ is not changed, any specific data structure of **A** is accordingly updated, and $\mathrm{D}_v[r,s]$ is propagated to the other neighbor of v.

Finally, if v is a semi-peripheral node whose semi-owners are nodes r_1 and r_2, then the message carries the estimated distance from either r_1 or r_2 to s which can be used to update distances. In details, let us assume that the message carries $\mathrm{D}_{r_1}[r_1,s]$, the other case is symmetric. Let u and z be the neighbors of v which are closer (in terms of number of edges) to r_1 and r_2, respectively. If the distance from v to s is not affected by the change of $\mathrm{D}_{r_1}[r_1,s]$, that is $\mathrm{D}_{r_1}[r_1,s]$ increases but $\mathrm{VIA}_v[s] \neq u$, then v simply discards the message. Otherwise, two cases may arise:

- (i) if $\mathrm{D}_{r_1}[r_1,s]$ is increased and $\mathrm{VIA}_v[s] = u$, then node v update $\mathrm{D}_v[v,s]$ as the weight of the shortest between two paths: that formed by the shortest path from r_1 to s plus the path from r_1 to v, and that formed by the shortest path from z to s plus edge (z,v);
- (ii) if $\mathrm{D}_{r_1}[r_1,s]$ is decreased enough to induce a decrease also to $\mathrm{D}_v[v,s]$, then v updates $\mathrm{D}_v[v,s]$ as the weight of the path formed by the shortest path from r_1 to s plus the unique path from r_1 to v.

In both the above mentioned cases, any specific data structure of **A** is updated accordingly, and $\mathrm{D}_{r_1}[r_1,s]$ is propagated to z. This behaviour mimics the distributed Bellman-Ford algorithm equipped with the split horizon heuristic [41, Section 6.6.3]: the information about the route for a particular node is never sent back in the direction from which it was received.

After updating the routing information toward the central node s, v calls a procedure called PERIPHERYUPDATE, using s and $\mathrm{D}^{old}_v[s]$ as parameters. This routine first verifies whether the routing table entry of s is changed or not and, in the affirmative case, it updates the routing information about the non-central nodes whose owner, semi-owner, or cycle-owner is s, if they exist, as follows:

- for each peripheral node z whose owner is s, node v sets $\mathrm{D}_v[v,z]$ equal to the weight of the unique path from s to z plus the weight of the shortest path from v to s.
- for each cyclic node z whose cycle-owner is s, node v sets $\mathrm{D}_v[v,z]$ equal to weight of the shortest between the two possible paths from s to z plus the weight of the shortest path from v to s.

- for each semi-peripheral node z such that one of the semi-owner nodes is s, node v performs procedure INNERSEMIPERIPHERYUPDATE, if z and v lie on the same semi-peripheral path, and procedure OUTERSEMIPERIPHERYUPDATE, otherwise. These procedures update the routing information toward z by exploiting the data stored in the CHAINPATH. In detail, procedure INNERSEMIPERIPHERYUPDATE updates $\mathrm{D}_v[v,y]$ by comparing the weight of the only two paths that connect v and z. Note that, such two paths include the shortest paths between the semi-owners of v and v itself.

  In particular, if S is the semi-peripheral path that includes v and z whose semi-owners are r_1 and r_2, node v computes the weight D_1 of the unique sub-path of S from v to z, and determines the neighbor VIA_1 of v that belongs to such sub-path. Then, if v belongs to the sub-path of S that connects z to r_1 (this detail can easily be deduced from the CHAINPATH), it computes the weight D_2 of the path formed by the sub-path of S from z to r_2 plus the shortest path from v to r_2. Note that such shortest path might contain node r_1. Otherwise, node v computes the weight D_2 of the path formed by the sub-path of S from z to r_1 plus the shortest path from v to r_1. Finally, node v sets $\mathrm{D}_v[v,z]$ to the minimum weight of the above two possible paths and, accordingly, it updates $\mathrm{VIA}_v[z]$.

  Similarly, procedure OUTERSEMIPERIPHERYUPDATE updates $\mathrm{D}_v[v,z]$ by comparing the weight of the only two paths that connect v and z. Note that such paths include the shortest paths between the semi-owners of z and v. In detail, if S is the semi-peripheral path that contains z and r_1 and r_2 are the semi-owners of S, node v first computes two values D_{r1} and D_{r2}, equal to the weight of the path formed by the path between z and r_1 (z and r_2, respectively) plus the shortest path between v and r_1 (v and r_2, respectively), then sets $\mathrm{D}_v[v,z]$ equal to the minimum of the weights of these two possible paths and, accordingly, it updates $\mathrm{VIA}_v[z]$.

Case (ii). If a weight change occurs on a peripheral edge (x,y_1), belonging to a peripheral path $P = \{r,\ldots,x,y_1,\ldots,y_n\}$ whose owner is r, then node x (y_1, respectively), handles the change by sending a $peri.change(x,y_1,w(x,y_1))$ message to each of its neighbors. In this case, the distance from each node of the network to x does not change, except for those nodes y_p with $p=1,\ldots,n$ (which are topologically further than x from r). Each of these nodes, after receiving the $peri.change(x,y_1,w(x,y_1))$ message, first updates the CHP with the new value of $w(x,y_1)$ and then computes the distance to x and to all the other nodes s of the network by simply adding to $\mathrm{D}_{y_p}[y_p,s]$ the weight change on edge (x,y_1).

When a generic node v, different from nodes y_p, receives message $peri.change(x,y_1,w(x,y_1))$, it first verifies whether the update has been already processed or not, by comparing the new value of $w(x,y_1)$ with the one stored in its CHP. In the first case the message is discarded. Otherwise, it updates its CHP with the updated value $w(x,y_1)$ and its routing information only toward nodes y_p, as the shortest path toward x does not change. In particular, node v sets $\mathrm{D}_v[v,y_p] = \mathrm{D}_v[v,r] + \mathrm{D}_v[r,y_p]$, where $\mathrm{D}_v[r,y_p]$ is the weight of the peripheral path from r to y_p (note that, for each $v \in P$, $v \neq y_p$, $\mathrm{D}_v[v,y_p]$ is computed by using only the information stored inside the CHP because it is equal to the weight of the peripheral path from v to y_p). Then, it propagates $peri.change(x,y_1,w(x,y_1))$ over the network by a flooding algorithm.

Case (iii). If the weight of a semi-peripheral edge (x,y), whose semi-owner nodes are r_1 and r_2, changes, then node x (y, respectively) sends two kinds of messages: a $semi.change(x,y,w(x,y))$, to each of its neighbors, and a $gen.update(s,\mathrm{D}.[\cdot,s]$ to x (y, respectively), for each central node s such that $\mathrm{VIA}_x[s] \neq y$ ($\mathrm{VIA}_y[s] \neq x$, respectively), where $\mathrm{D}.[\cdot,s]$ is the distance toward s of the semi-owner node of x (y, respectively) that belongs to the sub-path of the semi-peripheral path that does not include the edge (x,y). When a generic node v receives message $semi.change$, it first verifies whether the update has been already processed or not, by comparing the new value of $w(x,y)$ with the one stored in its CHP. In the first case the message is discarded. Otherwise, node v updates its CHP with the new value of $w(x,y)$ and it propagates $semi.change(x,y,w(x,y))$ over the network by a flooding algorithm. Moreover, if v is the semi-owner node of the semi-peripheral path P that includes edge (x,y), it also performs the procedure provided by **A** for the distributed computation of shortest paths with respect to central nodes. This step basically considers P as a single edge that connects the two semi-owner nodes of P itself, and induces such semi-owner nodes to behave like the weight of one of their adjacent edges has changed.

When a generic node v receives a $gen.update(s, \mathrm{D}.[\cdot, s]$ message from a neighbor u, it executes procedure GENERALIZEDUPDATE$(s, \mathrm{D}.[\cdot, s]$. After updating the routing information toward a central node s, node v calls the procedure PERIPHERYUPDATE using s and $\mathrm{D}_v^{old}[s]$ as parameters. The procedure works as in the case of a central edge weight change (see Case (i)).

Case (iv). If the weight of a cyclic edge (x, y) changes, both nodes x and y send a $cycl.change(x, y, w(x, y))$ message to each of their neighbors. Let r be the cycle-owner node of both x and y. When a generic node v receives message $cycl.change$, it first verifies whether the update has been already processed or not, by comparing the new value of $w(x, y)$ with the one stored in its CHP. In the first case the message is discarded. Otherwise, node v first updates its CHP with the updated value of $w(x, y)$ and propagates $cycl.change(x, y, w(x, y))$ over the network by a flooding algorithm. Then, two cases can occur: either node v belongs to the same semi-peripheral cycle of x and y or not.

- In the first case, node v first computes $d^\alpha = \mathrm{D}_v[v, s] - \mathrm{D}_v[v, r]$ and, hence, updates the routing information toward all the nodes of the semi-peripheral cycle, including r, by using the CHP data structure. Then, if $\mathrm{D}_v[v, r]$ changes, it updates the routing information toward all the other central nodes s of the network by setting $\mathrm{D}_v[v, s] = \mathrm{D}_v[v, r] + d^\alpha$. After updating the routing information toward a central node s, node v calls the procedure PERIPHERYUPDATE using s and $\mathrm{D}_v^{old}[s]$ as parameters. The procedure works as in the case of a central edge weight change (see Case (i)).
- In the second case, v computes, for each node z of the semi-peripheral cycle, the shortest path distance between z and its cycle-owner node r by using the CHP data structure. Finally, it assigns $\mathrm{D}_v[v, z] = \mathrm{D}_v[v, r] + \mathrm{D}_v[r, z]$.

6.4. *Combining* DCP *with* DUAL

This section describes the combination of DCP to DUAL, denoted as DUAL-DCP. The main changes deriving by the application of DCP to DUAL can be summarized as follows.

On the one hand, if the weight of a central edge (u, v) changes, then node v verifies, only with respect to each central node s, whether $\mathrm{D}_v[v, s] > \mathrm{D}_v[u, s] + w(u, v)$ or not (note that the behaviour of node u is symmetric with respect to the weight change operation). In the first case, node v sets $\mathrm{D}_v[v, s] = \mathrm{D}_v[u, s] + w(u, v)$ and $\mathrm{FS}_v[s] = u$ and propagates the change to all its neighbors. In the second case, node v first checks whether $\mathrm{FS}_v[s] = u$ or not. If $\mathrm{FS}_v[s] \neq u$, the node terminates the update procedure. Otherwise, node v tries to compute a new $\mathrm{FS}_v[s]$. In this phase, if no neighbor of v satisfies SNC and node v needs to perform the DIFFUSE-COMPUTATION, it sends out *query* messages only to its central neighbors. Moreover, with the aim of knowing the estimated distance of each of the semi-owner of the semi-peripheral paths which node v belongs to, node v performs the TRAVERSE-PATH phase and sends $sp.query$ messages to each of its semi-peripheral neighbors. When node v receives all the replies to these messages, it updates its routing information towards s and propagates the change to all its neighbors. In all the cases, if the distance towards s changes, node v is able to update its routing information towards all the nodes in the non-central paths of s, if they exists.

On the other hand, if a weight change occurs on either a peripheral or a cyclic edge, then the nodes adjacent to the edge behave as described in Section 6. The only difference with respect to the generic case is that involved nodes also update their topology table. Finally, if a weight change occurs on a semi-peripheral edge, differently from the general case, semi-peripheral nodes do not need to ask information to their neighbors, as DUAL permanently stores the topology table.

6.5. *Combining* DCP *with* LFR

This section describes the combination of DCP to LFR, denoted as LFR-DCP. The main changes deriving by the application of DCP to LFR can be summarized as follows.

If the weight of a central edge (u, v) changes, then node v verifies, only with respect to central nodes $s \in V_c$, whether $\mathrm{D}_v[v, s] > \mathrm{D}_v[u, s] + w(u, v)$ or not (note that the behaviour of node u is symmetric with respect to the weight change operation). In the first case, node v sets $\mathrm{D}_v[v, s] = \mathrm{D}_v[u, s] + w(u, v)$ and $\mathrm{FS}_v[s] = u$ and propagates the change to all its neighbors. In the second case, node v first checks whether $\mathrm{FS}_v[s] = u$ or not. If

$\text{FS}_v[s] \neq u$, the node terminates the update procedure. Otherwise, node v performs LOCAL-COMPUTATION, by sending *get.dist* message to all its neighbors. If the LOCAL-COMPUTATION succeeds, node v updates its routing information and propagates the change. Otherwise, node v needs to perform the GLOBAL-COMPUTATION and it sends out $get.feasible.dist$ messages only to its central neighbors. Moreover, with the aim of knowing the estimated distance of each of the semi-owner of the semi-peripheral paths which node v belongs to, node v performs the TRAVERSE-PATH phase and sends *sp.query* messages to each of its semi-peripheral neighbors. When node v receives all the replies to these messages, it updates its routing information towards s and propagates the change to all its neighbors. In all the cases, if the distance to s changes, node v is able to update its routing information towards all nodes in the non-central paths of s, if they exists. If a weight change occurs on a peripheral, semi-peripheral or a cyclic edge, then the nodes adjacent to the edge behave as described in Section 6.

6.6. Practical Effectiveness of DCP

This section describes the results of the experimental study proposed in [30], which considers algorithms DUAL, LFR, DUAL-DCP and LFR-DCP, and shows the practical effectiveness of the use of DCP. Experiments have been performed both on real-world and artificial instances of the problem, subject to randomly generated sequences of updates. In detail, both the power-law networks of the *CAIDA IPv4 topology dataset* [31], and the random power-law networks generated by the *Barabási-Albert* algorithm [32] were used. In particular, simulations have been ran on a CAIDA instance with 8000 nodes and 11141 edges, named $G_{IP-8000}$, and on a Barabási–Albert instance with 8000 nodes and 12335 edges, named $G_{BA-8000}$. $G_{IP-8000}$ has average node degree equal to 2.8, a percentage of degree 1 nodes approximately equal to 38.5%, and a percentage of degree 2 nodes approximately equal to 33%; $G_{BA-8000}$ has average node degree equal to 3.1, a percentage of degree 1 nodes approximately equal to 45%, and a percentage of degree 2 nodes approximately equal to 26%.

The experimental results provided in the considered paper have shown that the combinations of both DUAL and LFR with DCP provide a huge improvement in the global number of messages sent on $G_{IP-8000}$. In particular, the ratio between the number of messages sent by DUAL-DCP and those sent by DUAL is within 0.03 and 0.16 which means that DUAL-DCP sends a number of messages which is between 3% and 16% that of DUAL. The ratio between the number of messages sent by LFR-DCP and those sent by LFR is within 0.10 and 0.26.

Similar results are obtained on $G_{BA-8000}$. In more details, the ratio between the number of messages sent by DUAL-DCP and those sent by DUAL is within 0.22 and 0.47, while the ratio between the number of messages sent by LFR-DCP and those sent by LFR is within 0.26 and 0.34. It is worth noting that the improvement provided by DCP in these artificial instances is smaller than in the real-world ones. This is due to the fact that the part of the distributed computation pruned by DCP in the case of Barabási-Albert networks is smaller with respect to the case of CAIDA networks, as they have: (i) a slightly higher average degree (ii) a wider range of the degree of the central nodes, i.e. the standard deviation of the node degree is slightly larger. In fact, for instance, $G_{IP-8000}$ has an average degree equal to 2.8 and *maxdeg* equal to 203 while $G_{BA-8000}$ has an average degree equal to 3.1 and *maxdeg* equal to 515.

Notice also that this behaviour is more emphasized for LFR-DCP as LFR includes two sub-routines (called LOCAL-COMPUTATION and GLOBAL-COMPUTATION, respectively) where the worst case message complexity depends on the maximum degree, while DUAL uses a single sub-routine (namely the DIFFUSE-COMPUTATION) where the worst case message complexity depends on the same parameter.

Another experimental evidence provided by the mentioned work is that the use of DCP has positive effects on the space requirements per node of the algorithms. In particular, it is shown that, in $G_{IP-8000}$, DUAL-DCP (LFR-DCP, respectively) requires a maximum space occupancy per node that is 0.30 (0.72, respectively) times that of DUAL (LFR, respectively). The behavior is similar in $G_{IP-8000}$ where DUAL-DCP (LFR-DCP, respectively) requires a maximum space occupancy per node that is 0.29 (0.83, respectively) times that of DUAL (LFR, respectively). Notice that: i) the improvement is more evident in the case of DUAL, as its maximum space occupancy per node is by far higher than that of LFR; ii) concerning DUAL, this behaviour is confirmed also in the average case, where DUAL-DCP requires 0.81 and 0.92 times the average space occupancy per node of DUAL, in $G_{IP-8000}$ and $G_{BA-8000}$, respectively. On the contrary, this is not true for LFR-DCP, whose space occupancy per node is slightly higher than that of LFR. The use of DCP here induces an overhead in the average space occupancy per node which is equal to 53% and 77%, in $G_{IP-8000}$ and $G_{BA-8000}$, respectively. Further evidences of the the above observations, and a corresponding analysis, will be given in Section 7.

7. Experimental Evaluation

In this section we present the results of our new experimental study of algorithms DUAL, LFR, DUAL-DCP, and LFR-DCP, which have been tested on networks generated by the Generalized Linear Preference (GLP) model [42]. 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++ 4.0p1 environment [43]. All software has been written in C++ and compiled with GNU g++ compiler v.4.8 under Linux (Kernel 2.6.32).

7.1. Generalized Linear Preference model

The Generalized Linear Preference (GLP) model has been introduced in [42], and it is a variant of the Barabási–Albert model. 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. In detail, let $P(i)$ denote the probability that a new node will be connected to node i, the *linear preference* connectivity rule of the Barabási–Albert model is defined as follows:

$$P(i) = \frac{deg(i)}{\sum_j deg(j)} \tag{1}$$

As a result of the above connectivity mechanism, the power-law graphs generated by the Barabási–Albert algorithm have average node degree approximately equal to 3 and a number of nodes with degree smaller than 3 approximately equal to $7/10n$, where n is the number of nodes of the graph. In [36] it has been shown that in the real Internet, new Autonomous Systems (AS) have a much stronger preference to connect to high degree ASs than predicted by the linear preference model. This behaviour is better captured by the GLP model which adds more flexibility than Barabási–Albert in specifying how nodes connect to other nodes [33]. In detail, the GLP model reflects the fact that the evolution of the AS graph is mostly due to two operations, the addition of new nodes and the addition of new links between existing nodes. It starts with n_0 nodes connected through $n_0 - 1$ links. At each time-step, one of the following two operations is performed:

1. with probability $p \in [0,1]$, $k < n_0$ new links are added between existing nodes;
2. with probability $1 - p$, a new node is added and connected to k existing nodes.

If $P(i)$ denotes again the probability that a new node will be connected to node i, the *generalized linear preference* connectivity rule of the GLP model is defined as follows:

$$P(i) = \frac{deg(i) - \beta}{\sum_j (deg(j) - \beta)} \tag{2}$$

where parameter $\beta \in (-\infty, 1)$ is a tunable parameter that can be adjusted such that nodes have a stronger preference of being connected to high degree nodes than predicted by Equation 1. In particular, it indicates the preference for a new node (edge) connecting to high degree nodes. The smaller the value of β is, the less preference is given to high degree nodes. Power-law graphs generated by the Generalized Linear Preference model have average node degree of 4.8 and a number of nodes with degree smaller than 3 that is on average around $8/10n$.

7.2. Executed tests

We implemented the GLP generation algorithm and first generated a large connected graph, having 16 031 nodes, by fixing β to 0.6447, since this choice has been shown to allow the generation of networks that are similar to real-world ones in terms of the two properties of the small world [33]. Since DUAL has rather high memory

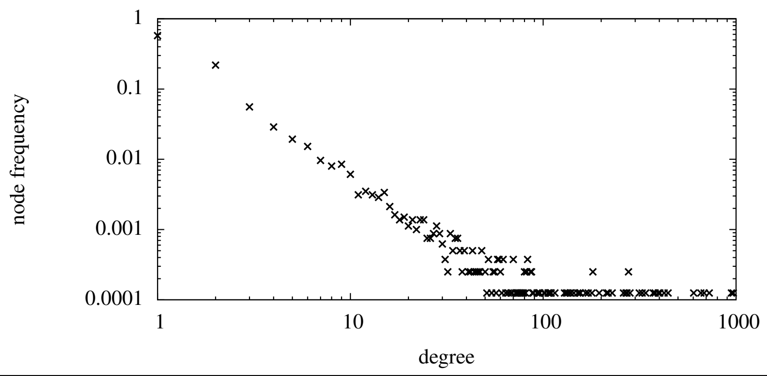

Figure 6. Power-law node degree distribution of a GLP graph with 8000 nodes and 19158 edges.

requirements, we were unable to perform tests on such large graph on our experimental platform. Therefore, we randomly extracted different subgraphs of smaller sizes, by performing pruned BFSs rooted at different randomly chosen nodes.

Finally, we generated a set of different experimental settings, where each setting consists of a n-node artificial subgraph of the above graph, denoted as G_{GLP-n}, and a set of k edge updates, with k assuming values in $\{5,10,\ldots,30\}$. Edge weights are non-negative real numbers randomly chosen in $[1,1\,000\,000]$ and an edge update consists of multiplying the weight of a random selected edge by a percentage value randomly chosen in $[50\%,150\%]$. For each test configuration (a graph and a fixed value of k) we performed 5 different experiments (for a total amount of 30 runs) measured performance metrics of interest, and computed average values for each metric. We ran simulations on GLP instances with $n \in \{1200,5000,8000\}$. The results of our experiments on the different instances are similar, hence we report only those on the bigger instance $G_{GLP-8000}$, having 8000 nodes and 19158 edges. Notice that, $G_{GLP-8000}$ has average node degree equal to 4.8, around 58% of nodes having degree one, and around 22% of nodes having degree two. In Figure 6 we report the distribution of the nodes' degree of $G_{GLP-8000}$, which is clearly a power-law one (note that the y–axis is log scaled).

7.3. Analysis

A summary of our results is shown in Figures 7 and 8, where we report the average number of messages sent by DUAL and DUAL-DCP, and by LFR and LFR-DCP, respectively, on $G_{GLP-8000}$, per test.

Our data suggest that the use of DCP provides a huge improvement in the global number of messages sent. In more details, in the tests of Figure 7 the ratio between the number of messages sent by DUAL-DCP and those sent by DUAL is within $73\cdot10^{-5}$ and $457\cdot10^{-5}$ while in the tests of Figure 8 the ratio between the number of messages sent by LFR-DCP and those sent by LFR is within $138\cdot10^{-5}$ and $2555\cdot10^{-5}$. Note that, the improvement provided by DCP here is by far more significant than that provided in case of Barabási–Albert graphs where, as shown in [30] (see the paper for details), the ratio between the number of messages sent by DUAL-DCP (LFR-DCP, respectively) and those sent by DUAL (LFR, respectively) is always larger than 10^{-2}. Since the GLP model is known to better predict real-world network infrastructures w.r.t. the Barabási–Albert one, this provides even more evidences of the effectiveness of DCP in real-world network instances.

Notice that the very good performance of DCP in this case is probably due to the structure of GLP graphs, which have more nodes with degree smaller than 3 with respect to the Barabási–Albert ones. This results in a much more effective pruning of the distributed part of the computation of both DUAL and LFR w.r.t. what happens in Barabási–Albert instances, as the set of central nodes, which handle the distributed computations, is smaller. Notice also that this behavior is more evident for LFR-DCP as LFR includes two sub-routines (the LOCAL-COMPUTATION and the GLOBAL-COMPUTATION) which worst case message complexity depends on the size of the set of central nodes, while DUAL uses a single sub-routine (the DIFFUSE-COMPUTATION) which worst case message complexity depends on the same parameter.

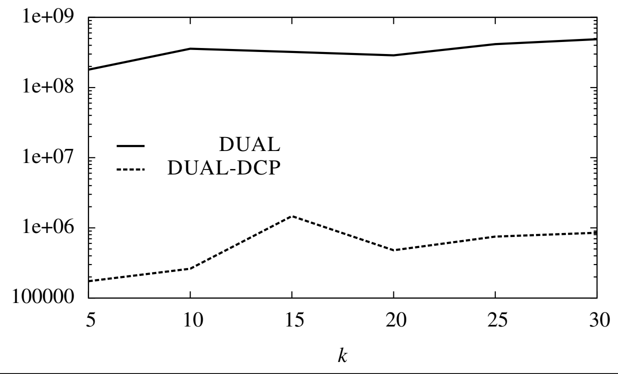

Figure 7. Number of messages sent by DUAL and DUAL-DCP on $G_{GLP-8000}$.

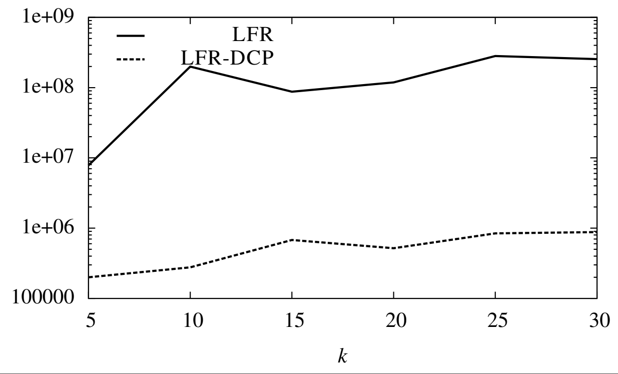

Figure 8. Number of messages sent by LFR and LFR-DCP on $G_{GLP-8000}$.

By the two diagrams of Figures 7 and 8, it is clear that LFR outperforms DUAL on $G_{GLP-8000}$. This is due to the fact that, in the literature, GLP graphs has been shown to model very accurately real world topologies and that, in such topologies, LFR is very effective in terms of number of messages sent. This consideration about the good performance of LFR also explains why the reduction in the number of messages sent induced by DCP is less emphasized for LFR. As a side remark, it can be observed that the number of messages sent by all the algorithms is by far bigger on GLP graphs with respect to Barabási–Albert ones. This is clearly due to the fact that $G_{GLP-8000}$ has much more edges than $G_{BA-8000}$.

To conclude our analysis, we consider the space occupancy per node of each algorithm. In particular, note that, when DCP is combined with DUAL, a subset of nodes of the network can avoid to maintain some data structures of DUAL, as either the information stored in them can be inferred by using the ChainPath, or it is not needed due to the pruning mechanism of DCP. For instance, each node of the network executing DUAL-DCP does not need to store the data structure of DUAL that implements the finite state machine with respect to non-central nodes, as no distributed computation can be initiated for this kind of nodes. The same considerations hold for the topology table.

Similar observations can be done with respect to LFR, since also here some of the nodes of the network can avoid to maintain some of the data structures used by the algorithm. In more details, in this case, the information stored in them can be either inferred by using the ChainPath or it is not needed at all due to the pruning mechanism of DCP. For instance, each node of the network executing LFR-DCP does not need to allocate the temporary data structure tempD with respect to non-central nodes, as no node of the network can become active with respect to a non-central node. Moreover, this data structure, when allocated for some central node s, has a reduced size, equal to the number of central neighbors of s plus the number of semi-peripheral path which s belongs to.

To support the above considerations, and assess their benefits, we provide measurements on the space occupancy of the nodes which are summarized in Table 1 where we report the maximum and the average space occupancy per node (in Bytes) of each algorithm on $G_{GLP-8000}$. We also report the ratio between the space occupancy per node of the algorithms integrating DCP and that of the original algorithms, for each test instance. Note that, since the space occupancy per node of LFR and LFR-DCP depends on the number of weight change operations, we report median values for each of these algorithms.

Table 1. Space occupancy per node of the implemented algorithms on $G_{GLP-8000}$.

Algorithm	MAX		AVG	
	Bytes	Ratio	Bytes	Ratio
DUAL	38 888 000	1	359 580	1
DUAL-DCP	8 163 826	0.21	327 366	0.91
LFR	241 115	1	192 069	1
LFR-DCP	354 212	1.47	348 636	1.82

Our experiments show that the use of DCP induces, in most of the cases, a clear improvement also in the space requirements per node. In particular, DUAL-DCP requires a maximum space occupancy per node which is 0.29 times that of DUAL in $G_{GLP-8000}$. Notice that, the improvement is more evident in the case of DUAL, as its maximum space occupancy per node is by far higher than that of LFR. Concerning DUAL, this behavior is confirmed also in the average case, where DUAL-DCP requires 0.91 times the average space occupancy per node of DUAL, in $G_{GLP-8000}$. On the contrary, our data show that the average space occupancy per node of LFR-DCP is slightly larger than that of LFR and that the use of DCP induces an overhead in the average space occupancy per node which is equal to 82% on $G_{GLP-8000}$, respectively. This is due to the fact that the space occupancy of LFR is quite low by itself and that, in this case, the space occupancy overhead due to the data structures required by DCP (see [30]) is larger than the space occupancy reduction induced by the use of DCP on the data structures of the original algorithm. As a final observation, it is worth noting that LFR is the best solution in terms of average space occupancy per node in all the power-law networks considered in this paper and in [30].

8. Conclusions

In the last few years, there has been a renewed interest in devising new efficient light-weight distributed shortest paths solutions for large-scale networks, where distance-vector algorithms are an attractive alternative to link-state solutions, when scalability and reliability are key issues or when the memory resources of the nodes of the network are limited.

In this paper, we have surveyed established Distance-Vector approaches, and outlined the most recent and efficient solutions of this category. In particular, we have considered the classic *Distributed Bellman-Ford* (DBF) algorithm [25], the well-known *Diffuse Update ALgorithm* (DUAL) [5], and the recent *Loop Free Routing* algorithm (LFR) [29]. Furthermore, we have discussed the general framework *Distributed Computation Pruning* (DCP) [30]. We have analyzed differences and similarities, pros and cons, of the various algorithm, and we have also provided a summary of the experimental results given in [30], which show how the above mentioned algorithms behave in practice.

We have also proposed a new experimental evaluation on power-law networks generated by the GLP model [33] of the mentioned algorithms. Our experiments show that in the case of GLP networks, DCP is very effective in terms of improving the usage of communication resources, since the overall number of messages sent is reduced up to a couple of orders of magnitudes. As a side result, we have given further experimental evidences that LFR outperforms DUAL in terms of number of messages sent and is very effective from the memory requirements point of view in power-law artificial networks.

The study of new techniques, which take advantage of the structural properties of real-world networks, deserves further investigation. In particular, it would be interesting to: (i) evaluate how DCP scales to bigger a/o more dynamic networks; (ii) develop techniques for shortest paths distributed algorithms to be efficient in other practically interesting scenarios; (iii) investigate the possibility of devising techniques to enhance the performances of distributed algorithms for other important network problems.

Author Contributions: Daniele Frigioni contributed mainly in writing the paper and in the analysis of the experimental results. Mattia D'Emidio conceived, designed, and performed the experiments, analysed the data, and contributed in the writing process.

Conflicts of Interest: The authors declare no conflicts of interest.

1. Orda, A.; Rom, R. Distributed shortest-path and minimum-delay protocols in networks with time-dependent edge-length. *Distributed Computing* **1996**, *10*, 49–62.
2. Moy, J.T. *OSPF: Anatomy of an Internet routing protocol*; Addison-Wesley, 1998.
3. Bellman, R. On a routing problem. *Quarterly of Applied Mathematics* **1958**, *16*, 87–90.
4. Garcia-Luna-Aceves, J.J.; Murthy, S. A Path-finding Algorithm for Loop-free Routing. *IEEE/ACM Trans. Netw.* **1997**, *5*, 148–160.
5. Garcia-Lunes-Aceves, J.J. Loop-Free Routing Using Diffusing Computations. *IEEE/ACM Transactions on Networking* **1993**, *1*, 130–141.
6. Cicerone, S.; D'Angelo, G.; Di Stefano, G.; Frigioni, D.; Maurizio, V. Engineering a new algorithm for distributed shortest paths on dynamic networks. *Algorithmica* **2013**, *66*, 51–86.
7. Dijkstra, E.W. A note on two problems in connexion with graphs. *Numerische Mathematik* **1959**, *1*, 269–271.
8. Rosen, E.C. The updating protocol of ARPAnet's new routing algorithm. *Computer Networks* **1980**, *4*, 11–19.
9. Wu, J.; Dai, F.; Lin, X.; Cao, J.; Jia, W. An extended fault-tolerant link-state routing protocol in the Internet. *IEEE Transactions on Computers* **2003**, *52*, 1298–1311.
10. Frigioni, D.; Marchetti-Spaccamela, A.; Nanni, U. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms* **2000**, *34*, 251–281.
11. Narváez, P.; Siu, K.Y.; Tzeng, H.Y. New dynamic algorithms for shortest path tree computation. *IEEE/ACM Transactions on Networking* **2000**, *8*, 734–746.
12. D'Angelo, G.; D'Emidio, M.; Frigioni, D.; Maurizio, V. A speed-up technique for distributed shortest paths computation. Proceedings 11th International Conference on Computational Science and its Applications (ICCSA2011), 2011, Vol. 6783, *Lecture Notes in Computer Science*, pp. 578–593.
13. Elkin, M. Distributed Exact Shortest Paths in Sublinear Time. arXiv:1703.01939, 2017.
14. Elmeleegy, K.; Cox, A.L.; Ng, T.S.E. On count-to-infinity induced forwarding loops in Ethernet networks. Proceedings 25th IEEE Conference on Computer Communications (INFOCOM2006), 2006, pp. 1–13.
15. Henzinger, M.; Krinninger, S.; Nanongkai, D. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. Proceedings 48th Annual ACM SIGACT Symposium on Theory of Computing, (STOC2016). ACM, 2016, pp. 489–498.
16. Myers, A.; Ng, E.; Zhang, H. Rethinking the service model: Scaling Ethernet to a million nodes. Proceedings 3rd Workshop on Hot Topics in Networks (ACM HotNets). ACM Press, 2004.
17. S.Ray.; Guérin, R.; Kwong, K.; Sofia, R. Always acyclic distributed path computation. *IEEE/ACM Transactions on Networking* **2010**, *18*, 307–319.
18. Yao, N.; Gao, E.; Qin, Y.; Zhang, H. RD: Reducing message overhead in DUAL. Proceedings 1st International Conference on Network Infrastructure and Digital Content (IC-NIDC2009). IEEE Press, 2009, pp. 270–274.
19. Zhao, C.; Liu, Y.; Liu, K. A More Efficient Diffusing Update Algorithm For Loop-Free Routing. Proceedings 5th International Conference on Wireless Communications, Networking and Mobile Computing (WiCom2009). IEEE Press, 2009, pp. 1–4.
20. Cicerone, S.; D'Angelo, G.; Di Stefano, G.; Frigioni, D. Partially Dynamic Efficient Algorithms for Distributed Shortest Paths. *Theoretical Computer Science* **2010**, *411*, 1013–1037.
21. Cicerone, S.; Di Stefano, G.; Frigioni, D.; Nanni, U. A Fully Dynamic Algorithm for Distributed Shortest Paths. *Theoretical Computer Science* **2003**, *297*, 83–102.
22. Humblet, P.A. Another Adaptive Distributed Shortest Path Algorithm. *IEEE Transactions on Communications* **1991**, *39*, 995–1002.
23. Italiano, G.F. Distributed algorithms for updating shortest paths. International Workshop on Distributed Algorithms (WADS1991), 1991, Vol. 579, *Lecture Notes in Computer Science*, pp. 200–211.

24. Ramarao, K.V.S.; Venkatesan, S. On finding and updating shortest paths distributively. *Journal of Algorithms* **1992**, *13*, 235–257.
25. McQuillan, J. Adaptive routing algorithms for distributed computer networks. Technical Report BBN Report 2831, Cambridge, MA, 1974.
26. Bertsekas, D.; Gallager, R. *Data Networks*; Prentice Hall International, 1992.
27. Awerbuch, B.; Bar-Noy, A.; Gopal, M. Approximate Distributed Bellman-Ford Algorithms. *IEEE Transactions on Communications* **1994**, *42*, 2515–2517.
28. EIGRP. Enhanced Interior Gateway Routing Protocol. `http://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/16406-eigrp-toc.html`.
29. D'Angelo, G.; D'Emidio, M.; Frigioni, D. A loop-free shortest-path routing algorithm for dynamic networks. *Theoretical Computer Science* **2014**, *516*, 1–19.
30. D'Angelo, G.; D'Emidio, M.; Frigioni, D.; Romano, D. Enhancing the Computation of Distributed Shortest Paths on Power-law Networks in Dynamic Scenarios. *Theory of Computing System* **2015**, *57*, 444–477.
31. Hyun, Y.; Huffaker, B.; Andersen, D.; Aben, E.; Shannon, C.; Luckie, M.; Claffy, K. The CAIDA IPv4 Routed/24 Topology Dataset. `http://www.caida.org/data/active/ipv4_routed_24_topology_dataset.xml`.
32. Albert, R.; Barabási, A.L. Emergence of scaling in random networks. *Science* **1999**, *286*, 509–512.
33. Bu, T.; Towsley, D. On distinguishing between Internet power law topology generators. Proceedings 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM). IEEE, 2002, pp. 37–48.
34. Coudert, D.; Hogie, L.; Lancin, A.; Papadimitriou, D.; Pérennes, S.; Tahiri, I. Feasibility Study on Distributed Simulations of BGP. 26th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS2012). IEEE, 2012, pp. 96–98.
35. Hernandez, J.M.; Kleiberg, T.; Wang, H.; Mieghem, P.V. A Qualitative Comparison of Power Law Generators. Int. Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2007), 2007.
36. Chen, Q.; Chang, H.; Govindan, R.; Jamin, S.; Shenker, S.; Willinger, W. The Origin of Power-Laws in Internet Topologies Revisited. Proceedings IEEE INFOCOM, 2002, pp. 608–617.
37. Attiya, H.; Welch, J. *Distributed Computing*; John Wiley and Sons, 2004.
38. Dijkstra, E.W.; Scholten, C.S. Termination detection for diffusing computations. *Information Processing Letters* **1980**, *11*, 1–4.
39. D'Angelo, G.; D'Emidio, M.; Frigioni, D.; Maurizio, V. Engineering a new loop-free shortest paths routing algorithm. Proceedings 11th International Symposium on Experimental Algorithms (SEA2012), 2012, Vol. 7276, *Lecture Notes in Computer Science*, pp. 123–134.
40. D'Angelo, G.; D'Emidio, M.; Frigioni, D.; Romano, D. Enhancing the computation of distributed shortest paths on real dynamic networks. Proceedings 1st Mediterranean Conference on Algorithms (MEDALG2012). Springer, 2012, Vol. 7659, *Lecture Notes in Computer Science*, pp. 148–158.
41. Pahlavan, K.; Krishnamurthy, P. *Networking Fundamentals: Wide, Local and Personal Area Communications*; Wiley, 2009.
42. Albert, R.; Jeong, H.; Barabási, A.L. Error and attack tolerance of complex networks. *Nature* **2000**, *406*, 378–381.
43. OMNeT++. Discrete event simulation environment. `http://www.omnetpp.org`.