Fully Dynamic 2-Hop Cover Labeling

GIANLORENZO D'ANGELO, Gran Sasso Science Institute (GSSI), L'Aquila, Italy
MATTIA D'EMIDIO and DANIELE FRIGIONI, University of L'Aquila, L'Aquila, Italy

The 2-hop Cover labeling of a graph is currently the best data structure for answering shortest-path distance queries on large-scale networks, since it combines low query times, affordable space occupancy, and reasonable preprocessing effort. Its main limit resides in not being suited for *dynamic* networks since, after a network change, (1) queries on the distance can return incorrect values and (2) recomputing the labeling *from scratch* yields unsustainable time overhead.

In this article, we overcome this limit by introducing the first *decremental* algorithm able to update 2-hop Cover labelings under node/edge *removals* and edge *weight increases*. We prove the new algorithm to be (1) correct, i.e., after each update operation queries on the updated labeling return exact values; (2) efficient with respect to the number of nodes that change their distance as a consequence of a graph update; and (3) able to preserve the *minimality* of the labeling, a desirable property that impacts on both query time and space occupancy.

Furthermore, we provide an extensive experimental study to demonstrate the effectiveness of the new method. We consider it both alone and in combination with the unique known *incremental* approach (Akiba et al. 2014), thus obtaining the first *fully dynamic* algorithm for updating 2-hop Cover labelings under general graph updates. Our experiments show that the new dynamic algorithms are orders of magnitude faster than the from-scratch approach while at the same time being able to preserve the quality of the labeling in terms of query time and space occupancy, thus allowing one to employ the 2-hop Cover labeling approach in dynamic networks with practical performance.

CCS Concepts: • **Mathematics of computing → Graph algorithms**; • **Theory of computation → Shortest paths**; **Dynamic graph algorithms**; • **Information systems →** *Data mining*; *Web mining*; Information retrieval;

Additional Key Words and Phrases: Distance queries, large graph mining, shortest paths, labeling, dynamic networks, graph algorithms

ACM Reference format:
Gianlorenzo D'Angelo, Mattia D'Emidio, and Daniele Frigioni. 2019. Fully Dynamic 2-Hop Cover Labeling. *J. Exp. Algorithmics* 24, 1, Article 1.6 (January 2019), 36 pages.
https://doi.org/10.1145/3299901

Part of this work was done while the author was with Gran Sasso Science Institute (GSSI), L'Aquila, Italy.
Part of this work has been published in a preliminary form in D'Angelo et al. (2016).
This research has been partially supported by the Italian National Group for Scientific Computation GNCS-INdAM.
Authors' addresses: G. D'Angelo, Gran Sasso Science Institute (GSSI), Viale F. Crispi 7, I–67100, L'Aquila, Italy; email: gianlorenzo.dangelo@gssi.it. M. D'Emidio (corresponding author) and D. Frigioni, Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Via Vetoio, I–67100, L'Aquila, Italy; emails: {mattia.demidio, daniele.frigioni}@univaq.it.

1084-6654/2019/01-ART1.6 $15.00
https://doi.org/10.1145/3299901

1 INTRODUCTION

Answering *shortest-path distance queries* between pairs of nodes of a graph is a fundamental operation on graph data, as it is a building block of some of the most important applications in modern networked systems, such as social networks analysis (Vieira et al. 2007), context-aware search (Potamias et al. 2009), route planning in road networks (Abraham et al. 2012; Delling et al. 2014a), journey planning in transport systems (Cionini et al. 2014), and routing and management of resources in computer networks (Boccaletti et al. 2006).

Distance queries can be easily answered by using either a breadth-first search (BFS) on unweighted graphs or Dijkstra's algorithm on positively weighted graphs. Unfortunately, networks deriving from real-world applications tend to be huge, yielding unsustainable times to compute shortest paths on the corresponding graphs. For this reason, many smarter methods for efficiently answering distance queries in different application scenarios have been proposed (Abraham et al. 2012; Akiba et al. 2013; Bruera et al. 2008; Cheng and Yu 2009; Cohen et al. 2002; Delling et al. 2014a; Fu et al. 2013; Jin et al. 2012; Potamias et al. 2009; Qin et al. 2015; Wei 2010), the majority of which rely on a *preprocessing* phase that precomputes data to be exploited for reducing the time required for answering queries. Some of the most efficient of these methods are based on the notion of *2-hop Cover labeling* (Cohen et al. 2002). Among them, the recent *pruned landmark labeling* (PLL) (Akiba et al. 2013) achieves considerably better scalability than other methods in several real-world networks (Delling et al. 2014a).

However, while most of the above methods have been developed for *static* networks, i.e., networks whose topology is assumed to be fixed over time, many real-world networks are inherently *dynamic*, i.e., have topologies that evolve over time. In this latter case, every time the graph representing the network is subject to a change, like an edge insertion or an edge removal, the preprocessed data need to be updated in order to keep queries correct. Otherwise, queries could return either underestimated or overestimated values of distance, depending on the type of change, that can arbitrarily deviate from exact ones.

The trivial way to produce updated preprocessed data is to recompute everything from scratch. Nonetheless, in modern large-scale network applications, where graph sizes tend to be massive, this is in general infeasible since even the fastest methods can require hundreds to thousands of minutes. For this reason, in the recent past, researchers have worked to adapt known static methodologies to function in dynamic scenarios. For instance, some techniques have been developed to solve this issue for classes of dynamic graphs exhibiting a well-defined structure, such as road networks (Chabini and Lan 2002; D'Angelo et al. 2012, 2014a; Delling et al. 2011). However, for general *complex* networks, i.e., for networks that do not exhibit any topological features to be exploited (such as regularity or low highway dimension), very little has been done.

In Akiba et al. (2014), a dynamic version of PLL has been proposed, which, however, focuses on the *incremental* problem only, i.e., on handling *edge additions* and *edge weight decreases*. The authors purposely neglect the *decremental* problem, i.e., when *edge removals* and *edge weight increases* need to be managed, and do not give an analogous solution. They motivate this choice by a series of considerations. First, they claim that solving the incremental problem is already quite technically challenging and that the decremental problem appears to be much harder, in the sense that it seems to be very difficult to devise a solution able to deal with the problem without making big compromises on performance (such as labeling size or query time). Second, the authors claim that *decremental update operations* either never occur or happen with very low frequency in most real-world complex dynamic networks. To support this statement, they mention some typical examples of complex networks such as interaction networks and coauthor networks where this is in fact the case. Nonetheless, is several scenarios, edge removals are not only possible but also occur very frequently. Prominent examples are public transportation systems and road networks,

which have been the subject of a plethora of studies on dynamic algorithms in the recent past (see, e.g., Abraham et al. (2012) and Delling et al. (2011) and references therein). In these cases, decremental operations, representing, for instance, disruptions, traffic jams, and delays, are essentially the most important graph update operation to be supported, while incremental update operations basically never occur. Other examples that are worth mentioning are communication networks, where frequent decremental operations are, for instance, those related to congested links, or network-aware search indices, where the typical decremental update operation is that related to the removal of links (Boccaletti et al. 2006; Cionini et al. 2017; Delling et al. 2014c).

More in general, it can be claimed that efficiently solving the decremental problem is a crucial building block of all those applications that rely on dynamic large-scale graph-like data, such as graph database systems or software modeling tools. This latter statement is somehow supported by a recent work (Hayashi et al. 2016) that has tried to tackle this aspect in a deeper way as compared to the past. In particular, the authors tackle the problem from a completely different perspective. A framework for dealing with distance queries in large-scale graphs that is not based on the 2-hop Cover is introduced, along with a corresponding *fully dynamic algorithm*, i.e., a dynamic algorithm able to handle both decremental and incremental update operations. Its performance is experimentally shown to be pretty competitive with respect to 2-hop Cover-based solutions. In particular, the approach combines a pretty compact data structure, which can be precomputed via a rather fast preprocessing phase, with a very fast dynamic algorithm. Nonetheless, queries are much slower with respect to 2-hop Cover-based solutions, even by three orders of magnitude in some cases. Another drawback of this approach resides in its lack of generality. In fact, the proposed method, differently from methods based on 2-hop Cover, cannot be used on weighted graphs, since it heavily relies on a parallel strategy that exploits some properties of the BFS algorithm that hold in unweighted graphs only.

In this article, we move forward toward a general method for efficiently answering distance queries in fully dynamic complex networks. In particular, we give a new dynamic algorithm for updating 2-hop Cover labelings under decremental update operations. To the best of our knowledge, no algorithm of this kind was previously known. We show that the new method is (1) efficient in terms of the number of nodes that change their distance toward some other node of the network, as a consequence of a decremental update operation, and (2) able to preserve the *minimality* of the labeling, which is a desirable property of 2-hop Cover labelings that has an impact on both size and query time.

Then, we combine our new algorithm with the unique algorithm able to update 2-hop Cover labelings in case of incremental update operations (Akiba et al. 2014), thus obtaining the first *fully dynamic* algorithm for updating 2-hop Cover labelings under any kind of graph update operation. For each of the given algorithms, we give correctness proofs and complexity analyses. Moreover, we show their generality by discussing how to extend each of the considered algorithms to work in either undirected or directed graphs, as well as in either unweighted or weighted ones.

Finally, we propose an extensive experimental study to demonstrate the efficiency and the scalability of our new methods, which are simple enough to be quickly implemented. We considered a large set of both real-world and synthetic network input instances and various practically interesting settings.

We provide experimental evidence that shows that our approaches are practical since (1) they require orders of magnitude less time to update 2-hop Cover labeling with respect to the recomputation from scratch via PLL, even on massive networks, and (2) labelings updated by means of our dynamic algorithms are equivalent, in terms of performance, to those computed via PLL. In fact, their size does not increase as graph updates occur, thus allowing them to answer distance queries in microseconds, even in very large-scale evolving networks.

Our experimental data, on the one hand, confirm that the decremental problem is much harder to solve with respect to the incremental one. In fact, the time required for updating the labeling in the case of decremental update operations is higher than that required for handling incremental ones only. On the other hand, our experimental findings clearly contradict the conjecture given in Akiba et al. (2014) by showing that a practically efficient method for the fully dynamic case can be developed without any compromise on performance.

The remainder of the article is organized as follows. In Section 2, we give the notation needed in the article and describe the 2-hop Cover technique, the two algorithms known for its from-scratch computation, and the incremental algorithm of Akiba et al. (2014). In Section 3, we present our new decremental algorithm for undirected unweighted graphs, show its correctness, and analyze its computational complexity. We also provide some heuristics to improve its practical performance. In Section 4, we discuss how to adapt the new algorithms to work in the case of both directed and weighted instances. In Section 6, we provide our experimental evaluation. Finally, in Section 7, we conclude the article and outline possible future research directions.

2 BACKGROUND

Given an undirected unweighted graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, we denote by $N(v)$ the set of the neighbors of v in G, i.e., $N(v) = \{u \in V \mid \{u, v\} \in E\}$, and by $d(u, v)$ the distance between nodes u and v, i.e., the number of edges in a shortest path between u and v. If u and v are not connected, then $d(u, v) = \infty$. A *modification* or *update* to a graph can be *incremental* if it is an insertion of a new edge or *decremental* if it is a removal of an existing edge. Node insertions and removals can be modeled as modifications of multiple edges incident to the same node.

We assume that time is described by positive integers and that graph updates occur at each integer k. Symbol $G_\tau = (V_\tau, E_\tau)$ denotes the graph after τ modifications, where $G_0 = (V, E)$ is the initial graph. Similarly, N_τ and d_τ denote neighbor and distance functions in G_τ, respectively. We omit parameter τ when it is clear by the context or irrelevant.

2.1 2-Hop Cover Labeling

For each node v of G, the *label* $L(v)$ of v is a set of pairs (u, δ_{uv}), where u is a node in V and $\delta_{uv} = d(u, v)$. The set $\{L(v)\}_{v \in V}$ is referred to as a *labeling* of G. We use $u \in L(v)$ instead of $(u, \delta_{uv}) \in L(v)$ whenever the meaning is clear from the context. Labels can be used to answer to a *query* on the distance between two nodes s and t as follows:

$$\textsc{Query}(s, t, L) = \begin{cases} \min\limits_{v \in V} \{\delta_{sv} + \delta_{vt} \mid v \in L(s) \wedge v \in L(t)\} & \text{if } L(s) \cap L(t) \neq \emptyset \\ \infty & \text{otherwise.} \end{cases}$$

A labeling L is called a *2-hop Cover labeling* of G if, for each pair $s, t \in V$, $L(s) \cap L(t)$ contains at least a node u in a shortest path between s and t (or it is empty if s and t are disconnected). If such a node u exists, it is said to be a *hub* of pair (s, t). A *hub* node is said to *cover* pair (s, t), and, symmetrically, the pair is said to be *covered* by L. Moreover, if each pair u, v of nodes of the graph is covered by L, then the graph is said to be *covered* by the labeling, or similarly the labeling is said to satisfy the *cover property* for the considered graph.

It can be easily proven that in a 2-hop Cover labeling $\textsc{Query}(s, t, L) = d(s, t)$, for each $s, t \in V$ (Cohen et al. 2002). For each node $v \in V$, if $L(v)$ is sorted according to the IDs of its nodes, then computing $\textsc{Query}(s, t, L)$ takes $O(|L(s)| + |L(t)|)$ time.

Given a graph G, two nodes s and t in V, and a labeling L of G, a shortest path P between s and t is *induced* by L if, for any two nodes u and v in P, there exists a hub h of pair (u, v) such that

$h \in P$, or $h = u$, or $h = v$. The set of shortest paths between nodes s and t induced by L is denoted by $\textsc{Path}(s, t, L)$.

Definition 2.1. A labeling L of a graph G is *minimal* if and only if, for each $v \in V$ and for each $(u, \delta_{uv}) \in L(v)$, there exist two nodes s, t such that $\textsc{Query}(s, t, L') \neq d(s, t)$, where L' is obtained from L by removing (u, δ_{uv}) from $L(v)$.

In what follows, we first describe two different algorithms for the from-scratch computation of a 2-hop Cover labeling of a graph (Akiba et al. 2013) and then describe the unique algorithm known in the literature for the incremental maintenance of a 2-hop Cover labeling (Akiba et al. 2014).

2.2 From-Scratch Computation

In the literature, two algorithms for the from-scratch computation of a 2-hop Cover labeling of a graph G are known (Akiba et al. 2013). The first method is called *naive landmark labeling* and is based on a full computation of n Breadth-First Searches (BFSs) starting from all the nodes of the graph. The second method is called *pruned landmark labeling* and consists of a tailored pruning of the BFSs of the previous method. For more details on both algorithms we refer to Akiba et al. (2013). In both methods, we assume that the node set V is arbitrarily sorted and we denote the nodes as $v_1, v_2, \ldots, v_n$ according to such sorting.

The naive landmark labeling starts with labels L_0, where $L_0(v) = \emptyset$, for each $v \in V$, and runs n BFSs starting from each node, according to the sorting of V. Let L_k be the labels after k BFS searches from nodes $v_1, v_2, \ldots, v_k$. During the BFS starting at v_k, $L_k(u)$ is updated for each node u that is reachable from v_k as $L_k(u) = L_{k-1}(u) \cup \{(v_k, d(v_k, u))\}$. For each node u that is not reachable from v_k, $L_k(u) = L_{k-1}(u)$. It is easy to see that L_n is a 2-hop Cover since, for each $s, t \in V$, $(s, 0) \in L(s)$ and $(s, d(s, t)) \in L(t)$ and then $s \in L(s) \cap L(t)$. This method requires $O(n(m + n)) = O(nm + n^2)$ worst-case time and $O(n^2)$ space.

The PLL is based on the former method, but it *prunes* a BFS if a particular condition is satisfied. In detail, it starts with empty labels L_0, where $L_0(v) = \emptyset$, for each $v \in V$, and runs n BFSs starting from each node, according to the sorting of V. Again, let L_k be the labels after k BFS searches from nodes $v_1, v_2, \ldots, v_k$.

Consider a node u that is reached during the BFS started at v_k and assume that δ is the distance between v_k and u computed during the BFS. Then, the algorithm checks whether

$$\textsc{Query}(v_k, u, L_{k-1}) \leq \delta.$$

If the above condition holds, then labeling L_{k-1} contains a hub for (v_k, u) and for all pairs (v_k, x) such that there exists a shortest path between v_k and x passing through node u. Therefore, the BFS starting from v_k is pruned at u. Otherwise, the algorithm sets $L_k(u) = L_{k-1}(u) \cup \{(v_k, d(v_k, u))\}$ and continues the BFS as in the naive method. Again, for each node u that is not reachable from v_k, $L_k(u) = L_{k-1}(u)$. It can be proven that L_n is a 2-hop Cover (Akiba et al. 2013).

Even if the two methods exhibit the same asymptotic worst-case time and space complexity, it has been shown that PLL performs very well in practice (Akiba et al. 2013; Delling et al. 2014a). In particular, the performance of PLL heavily depends on the ordering of V. It has been experimentally observed that the average label size (and the time complexity of the query algorithm) decreases by several orders of magnitude if the nodes are sorted according to a centrality measure, like the degree or the closeness, instead of a random ordering (Akiba et al. 2013). The best way to select an ordering depends on the graph structure; only recently has a sorting algorithm been proposed that leads to small label sizes in many graph classes (Delling et al. 2014a). Note that, given a graph, it has been shown that computing an ordering on the nodes that deliver a 2-hop Cover of minimum size is NP-Hard (Delling et al. 2014b).

The PLL method guarantees that, if $v < u$, then u is not in $L(v)$, while v might be in $L(u)$. This property, called *well-ordering* (Qin et al. 2015), is quite important, as it can be used to prove that the computed labeling is minimal (Akiba et al. 2013), according to Definition 2.1. Note that minimality is a highly desirable property given the hardness of finding 2-hop Cover of minimum size.

2.3 Incremental Algorithm

In this section, we summarize the incremental algorithm introduced in Akiba et al. (2014). Note that the authors propose two algorithms: one based on the naive landmark labeling method, which we will denote by Naive–IncPLL, and another based on the pruned landmark labeling method, which we will denote by IncPLL. In the reminder of the article, we will concentrate on IncPLL only, since it has been shown to be the most efficient of the two. However, since IncPLL is based on Naive–IncPLL, for the sake of clarity we will describe Naive–IncPLL first. We refer the reader to Akiba et al. (2014) for a thorough description of the algorithms.

Given a graph G and a 2-hop Cover labeling L of G, algorithm Naive–IncPLL is able to update L in order to reflect a newly added edge $\{a, b\}$ that was previously absent. In particular, if we denote by G' the new graph, i.e., G plus the new edge $\{a, b\}$, Naive–IncPLL computes a 2-hop Cover labeling L' of G' by updating L. In what follows, we will denote by $d'(u, v)$ the distance between u and v in G'.

First of all, notice that Naive–IncPLL does not remove *outdated label entries* in L, i.e., entries that correspond to distances in G that have decreased in G' as a consequence of the insertion of $\{a, b\}$, and hence they are present also in L'. Formally, an *outdated entry* is a pair $(u, \delta_{uv}) \in L(v)$ for some u and v in G, such that $\delta_{uv} = d(u, v) \neq d'(u, v)$.

The above observation is motivated by the fact that distances can only decrease as a consequence of insertions and hence they cannot be underestimated because of the outdated label entries; i.e., it is possible to correctly answer distance queries even in the presence of such pairs, since the query algorithm searches for the minimum. Moreover, removing outdated entries is avoided also because detecting them could be too costly. Hence, Naive–IncPLL simply adds new label entries or overwrite distances of existing label entries. Under this strategy, the minimality of the resulting labeling L' as a whole is broken after updates, even though the set of newly added entries is minimal to answer correct distances in G'.

Algorithm Naive–IncPLL is based on the following two key insights: (1) if the distance between a node v_k and a node u changes, then all the new shortest paths between them pass through the new edge $\{a, b\}$, and (2) if the shortest path P between v_k and $u \neq a, b$ changes, then the distance between v_k and w changes, where w is the penultimate node in P.

Assuming without loss of generality that $d(v_k, a) \leq d(v_k, b)$, and based on the above insights, for every node v_k, the idea is that it suffices to resume the BFS from b originally rooted at v_k and to stop at unchanged nodes. That is, instead of inserting $(v_k, 0)$ into the initial queue of the BFS, the algorithm inserts pair $(b, d(v_k, a) + 1)$, which corresponds to the position after passing through the new edge $\{a, b\}$. Then, edges adjacent to u are not traversed if $\delta \geq d(v_k, u)$, where δ is the tentative distance for u drawn from the queue.

Algorithm IncPLL simply introduces pruning to Naive–IncPLL using the notion of *prefixal query*, denoted as PrefQuery. In particular, let s and t be two nodes and k be an integer; then:

$$\textsc{PrefQuery}(s, t, L, k) = \begin{cases} \min\limits_{v_i \in V} \{\delta_{sv_i} + \delta_{tv_i} \mid v_i \in L(s) \wedge v_i \in L(t) \wedge i \leq k\} & \text{if } L(s) \cap L(t) \neq \emptyset \\ \infty & \text{otherwise.} \end{cases}$$

In other words, $\textsc{PrefQuery}(s, t, L, k)$ is the answer to the query between nodes s and t computed from labeling L only using distances to nodes $v_1, v_2, \ldots, v_k$. Suppose now a resumed BFS is running

originally rooted at v_k and visiting node u with distance δ; then it is possible to prune the search at u if $\textsc{PrefQuery}(v_k, u, L, k) \leq \delta$.

The nodes for which it is needed to resume BFS searches are exactly those in $L(a)$ and $L(b)$. That is, it suffices to conduct resumed searches originally rooted at v_k if $v_k \in L(a) \cup L(b)$. In fact, if $v_k \notin L(a) \cup L(b)$, then both a and b are pruned or unreached during previous (resumed) searches rooted at v_k, and since the shortest path between v_k and them has not changed, the situation does not change at all. It is worth mentioning the fact that the second value $d(v_k, a)$ of pair $(b, d(v_k, a) + 1)$, to be inserted into the initial queue of the resumed BFS, can be computed from L as it represents the distance in G before the insertion.

3 DECREMENTAL ALGORITHM

In this section, we present our new dynamic algorithm to update a 2-hop Cover labeling L of a given graph G under decremental update operations, called DecPLL.

First of all, notice that, when handling decremental update operations, outdated label entries cannot be ignored, as done by IncPLL in the case of incremental ones. This is due to the fact that decremental update operations induce increases of distances. Thus, in this case, outdated label entries might induce the labeling to encode underestimated, uncorrect distances. For this reason, the detection and the removal of such kinds of entries cannot be avoided. This leads to the following definitions.

Definition 3.1 (Correct Label Entry). Let us assume that an edge $\{x, y\}$ is removed from G_{i-1} and that G_i is the resulting graph for $i \geq 1$. An entry $(u, \delta_{vu}) \in L(v)$ is correct for G_i if $\delta_{vu} = d_i(v, u)$.

Definition 3.2 (Affected Node). Let us assume that an edge $\{x, y\}$ is removed from G_{i-1} and that G_i is the resulting graph for $i \geq 1$. We say that a node v is *affected* by such a removal if there exists a shortest path induced by L between v and any other node u, i.e., a path in $\textsc{Path}(u, v, L)$, that passes through edge $\{x, y\}$ in G_{i-1}. The set of affected nodes is denoted as $\textsc{Aff}_{xy}$.

The set $\textsc{Aff}_{xy}$ of affected nodes can be exploited to detect label entries whose associated distance changes as a consequence of the removal of $\{x, y\}$ as follows. Assume that, for a pair of nodes u and v, one of the paths in $\textsc{Path}(u, v, L)$ passes through edge $\{x, y\}$ in G_{i-1}. Let h be a hub of pair (u, v) corresponding to such path. Then, one pair among $(h, \delta_{uh}) \in L(u)$ and $(h, \delta_{vh}) \in L(v)$ might not be correct and hence it must be removed from either $L(u)$ or $L(v)$, respectively.

Moreover, in this case, as a consequence of the removal of one of the mentioned pairs, a new hub for nodes u and v must be computed in order to restore the cover property. This might also hold for those label entries of each other node z that contains u (or v) in $L(z)$ and might exploit it as a hub from z to v (or u).

For each affected node v, we have that either $d_{i-1}(v, x) < d_{i-1}(v, y)$ or $d_{i-1}(v, x) > d_{i-1}(v, y)$. Therefore, we can partition set $\textsc{Aff}_{xy}$ into two disjoint subsets $\textsc{Aff}_x$ and $\textsc{Aff}_y$ that contain the affected nodes closer to x or to y, respectively. Moreover, we observe that if u and v are two affected nodes such that $u \in \textsc{Aff}_x$ and $v \in \textsc{Aff}_y$ and a shortest path between u and v passes through $\{x, y\}$, then a hub h of pair (u, v) is also affected and either $h \in \textsc{Aff}_x$ or $h \in \textsc{Aff}_y$ (see Figure 1). The next lemma gives us a way to exploit the set of affected nodes to identify the pairs in L that must be updated.

Lemma 3.3. *Let L be a minimal 2-hop Cover labeling of graph G_{i-1}. Let us assume that an edge $\{x, y\}$ is removed from G_{i-1} and that G_i is the resulting graph for $i \geq 1$. Then, for each $u, v \in V$ we have that $\textsc{Query}(u, v, L) \neq d_i(u, v)$ only if $v \in \textsc{Aff}_x$ and $u \in \textsc{Aff}_y$ or $v \in \textsc{Aff}_y$ and $u \in \textsc{Aff}_x$.*

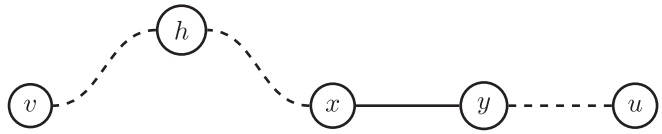

Fig. 1. A shortest path between nodes u and v. The solid line represents edge $\{x, y\}$, while dashed lines represent shortest paths. Assume that $h \in L(u) \cap L(v)$; then h is a hub for pair (u, v). If $\{x, y\}$ is removed, then $v, h \in \text{AFF}_x$, $u \in \text{AFF}_y$, and label entry (h, δ_{uh}) in $L(u)$ is not correct and must be updated.

Proof. We prove the statement by contradiction. Let us assume that $\text{QUERY}(u, v, L) \neq d_i(u, v)$ for a pair of nodes u and v that do not satisfy the statement, that is,

$$(v \notin \text{AFF}_x \vee u \notin \text{AFF}_y) \wedge (v \notin \text{AFF}_y \vee u \notin \text{AFF}_x). \tag{1}$$

Since L is a 2-hop Cover labeling of G_{i-1}, then $\text{QUERY}(u, v, L) = d_{i-1}(u, v)$. In what follows, we prove that $d_{i-1}(u, v) = d_i(u, v)$. To satisfy Equation (1), we obtain the following cases:[2]

- $v \notin \text{AFF}_x$ and $v \notin \text{AFF}_y$. In this case, for each node $z \in V$, all the shortest paths between v and z induced by L in G_{i-1} do not include edge $\{x, y\}$ and therefore $d_i(v, z) = d_{i-1}(v, z)$. It follows that $d_i(v, u) = d_{i-1}(v, u)$.
- $v \notin \text{AFF}_x$, $u \notin \text{AFF}_x$, and $v \in \text{AFF}_y$. In this case, either $u \notin \text{AFF}_y$ or $u \in \text{AFF}_y$. In the former case, $u \notin \text{AFF}_x \cup \text{AFF}_y$ and hence the distance between u and any other node in V in G_i is equal to the same distance in G_{i-1}. Therefore, $d_i(v, u) = d_{i-1}(v, u)$. In the latter case, since $v \in \text{AFF}_y$, we have that $d_{i-1}(v, y) < d_{i-1}(v, x)$ and $d_{i-1}(u, y) < d_{i-1}(u, x)$. Therefore, no shortest path between v and u in G_{i-1} passes through edge $\{x, y\}$. In fact, if we assume that a shortest path P passes through $\{x, y\}$ in G_{i-1}, then either the weight of P is $d_{i-1}(v, x) + 1 + d_{i-1}(y, u)$ or $d_{i-1}(v, y) + 1 + d_{i-1}(x, u)$ or in both cases the weight of P is greater than $d_{i-1}(v, y) + d_{i-1}(y, u)$ and then P is not a shortest path. Hence, $d_i(v, u) = d_{i-1}(v, u)$. □

Now, by Lemma 3.3 we know that, if L is a 2-hop Cover labeling of G_{i-1}, then $\text{QUERY}(u, v, L) \neq d_i(u, v)$ only if $v \in \text{AFF}_x$ and $u \in \text{AFF}_y$ (or symmetrically $v \in \text{AFF}_y$ and $u \in \text{AFF}_x$). Moreover, there exists a hub h of pair (u, v) that is also affected and either $h \in \text{AFF}_x$ or $h \in \text{AFF}_y$. Otherwise, we would have $\text{QUERY}(u, v, L) = d_i(u, v)$. In the former case, we have that $\text{QUERY}(u, h, L) \neq d_i(u, h)$, while in the latter case, we have that $\text{QUERY}(v, h, L) \neq d_i(v, h)$ (Figure 1 depicts the former case). By repeating this argument, we obtain a pair of affected nodes h', h'' in $\text{PATH}(u, v, L)$ such that $(h', \delta_{h'h''}) \in L(h'')$ but $\delta_{h'h''} \neq d_i(h', h'')$. In this case, we know that pair $(h', \delta_{h'h''})$ must be removed from $L(h'')$ if we want to update L in order to generate a labeling that covers G_i. In the remainder of the article, labels (hubs, respectively) of the above kind will be denoted as *affected labels* (*hubs*, respectively). The reasoning clearly holds for all hubs in the considered path between u and v. Thus, we can give the following result.

Lemma 3.4. *Let L be a minimal 2-hop Cover labeling of graph G_{i-1}. Let us assume that an edge $\{x, y\}$ is removed from G_{i-1} and that G_i is the resulting graph for $i \geq 1$. Then, for all nodes v and u such that either $v \notin \text{AFF}_x \vee u \notin \text{AFF}_y$ or $v \notin \text{AFF}_y \vee u \notin \text{AFF}_x$, all entries $(u, \delta_{vu}) \in L(v)$ are correct for G_i.*

Proof. By contradiction, let us assume that there exists a pair of nodes u and v such that $(u, \delta_{vu}) \in L(v)$, $v \notin \text{AFF}_x$ or $u \notin \text{AFF}_y$, and δ_{vu} is not correct (the case in which $v \notin \text{AFF}_y$ or $u \notin \text{AFF}_x$ is symmetric). Since pair $(u, 0)$ belongs to $L(u)$, then $\text{QUERY}(u, v, L) \leq \delta_{uv} = d_{i-1}(u, v)$.

[2]Cases $u \notin \text{AFF}_y \wedge u \notin \text{AFF}_x$ and $u \notin \text{AFF}_y \wedge v \notin \text{AFF}_y \wedge u \in \text{AFF}_x$ are symmetric, and hence they are not discussed.

By Lemma 3.3, $\textsc{Query}(u, v, L) = d_i(u, v)$ and $d_{i-1}(u, v) \leq d_i(u, v)$. It follows that $\textsc{Query}(u, v, L) = d_{i-1}(u, v) = d_i(u, v) = \delta_{uv}$ and (u, δ_{vu}) in $L(v)$ is correct. □

The two above lemmas imply that all pairs (u, δ_{vu}) in $L(v)$, for all nodes v and u such that $v \in \textsc{Aff}_x$ and $u \in \textsc{Aff}_y$ or $v \in \textsc{Aff}_y$ and $u \in \textsc{Aff}_x$, may not be correct in G_i, and hence the corresponding labeling does not cover G_i. Since we are trying to obtain a correct labeling without recomputing everything from scratch (i.e., starting from L), our approach is to remove them from $L(v)$ and recompute a new hub for (u, v). In fact, note that to cover G_i, it is not enough to update (u, δ_{vu}) in $L(v)$ by setting $\delta_{vu} = d_i(u, v)$, but the computation of a new hub might be required.

Therefore, our new algorithm DecPLL works in three phases as follows:

(1) *Detecting affected nodes*: computes sets $\textsc{Aff}_x$ and $\textsc{Aff}_y$
(2) *Removing affected hubs*: removes the pairs satisfying the condition of Lemma 3.3
(3) *Computing new hubs*: recomputes the missing hubs in order to restore the 2-hop Cover labeling according to Lemma 3.4

In what follows, we describe and analyze the three phases in detail. Given a 2-hop Cover labeling L of G, we denote by Λ the maximum size of the labels in L, i.e., $\Lambda = \max_{v \in V} |L(v)|$, and by m_S the number of edges incident to the nodes of a given subset of nodes $S \subseteq V$, i.e., $m_S = \sum_{v \in S} |N(v)|$.

3.1 Detecting Affected Nodes

The pseudo-code of the algorithm to compute set $\textsc{Aff}_x$ is given in Algorithm 1, while a symmetric algorithm for determining $\textsc{Aff}_y$ can be obtained by replacing, in the mentioned pseudo-code, $\mathtt{A}(x)$, x, and y with $\mathtt{A}(y)$, y, and x, respectively (i.e., by rooting the visit at y). The set of affected nodes is stored in variable $\mathtt{A}(x)$ ($\mathtt{A}(y)$, respectively). In order to perform efficiently (in logarithmic time) membership operations, we assume that such sets are kept sorted by the node ordering; this could be done, e.g., by maintaining them as balanced binary search trees.

Algorithm 1 mimics a BFS rooted at x, but it prunes some of the branches when the reached node is not affected. We use a queue `Q` to store the nodes to visit and a Boolean vector `mark` to keep track of the visited nodes. Lines 1 through 6 initialize $\mathtt{A}(x)$, `Q`, and `mark`. Then, the pruned BFS is performed at lines 7 through 15 starting from x. A node v is enqueued in `Q` only if it is affected. Let v be a node extracted from `Q`; then v is inserted into $\mathtt{A}(x)$ and $\mathtt{mark}[v]$ is set to `true` at line 14. Then, each neighbor u of v is analyzed in order to check whether it is affected and hence must be inserted into `Q`. Let H be the set of hubs of pair (u, y) in L of G_{i-1}; then u is inserted into `Q` if there exists at least a hub $h \in H$ that satisfies one of the two following conditions (see line 13):

(i) $h \in \mathtt{A}(x)$
(ii) $d_{i-1}(v, y) = d_{i-1}(v, x) + 1$ and either $h = v$ or $h = y$

Condition (i) checks whether a hub h has been already identified as affected. In this case, there exists a shortest path from h to y in $\textsc{Path}(h, y, L)$ that contains edge $\{x, y\}$, and therefore also one of the shortest paths between u and y in $\textsc{Path}(u, y, L)$ must contain edge $\{x, y\}$. Note that, since we are performing a BFS starting from x, then the hub h of pair (u, y) is analyzed by the algorithm before u, and if it is affected, then it is inserted in $\mathtt{A}(x)$ before u is analyzed from the condition at line 13. Condition (ii) handles the case in which a hub of pair (u, y) is either u or y. In this case, u is affected if any shortest path between u and y passes through edge $\{x, y\}$, and therefore we check whether $d_{i-1}(v, y) = d_{i-1}(v, x) + 1$. In fact, in such a case, there exists at least a shortest path between u and y passing through edge $\{x, y\}$. If the condition at line 13 is not satisfied, node u is not inserted into `Q` and then the search is pruned at u.

Lemma 3.5. *At the end of Algorithm 1,* $\mathtt{A}(x) = \textsc{Aff}_x$.

ALGORITHM 1: Computation of $\mathtt{A}(x)$

```
1  A(x) ← ∅;
2  foreach v ∈ V do
3  |   mark[v] ← false;
4  Q ← ∅;
5  mark[x] ← true;
6  Q.Enqueue(x);
7  while Q ≠ ∅ do
8  |   v ← Q.Dequeue();
9  |   A(x) ← A(x) ∪ {v};
10 |   foreach u ∈ N_i(v) : ¬mark[u] do
11 |   |   Let H be the set of hubs of pair (u, y) in L;
12 |   |   foreach h ∈ H do
13 |   |   |   if h ∈ A(x) ∨ ((h = u ∨ h = y) ∧ d_{i−1}(u, y) = d_{i−1}(u, x) + 1) then
14 |   |   |   |   mark[u] ← true;
15 |   |   |   |   Q.Enqueue(u);
16 |   |   |   |   break;
```

Proof. We prove the lemma by induction on the distance $d_i(v, x)$ between x and any other node v. The inductive claim is: for each v such that $d_i(v, x) \leq \ell$, $\ell \geq 0$, $v \in \textsc{Aff}_x$ if and only if $v \in \mathtt{A}(x)$.

The only node for which $\ell = 0$ is x, and it is inserted into $\mathtt{A}(x)$ at the first iteration of the while loop at lines 7 through 15. Moreover, x is affected since the unique shortest path between x and y is $\{x, y\}$. This proves the inductive basis.

Let us assume that for each v such that $d_i(v, x) \leq \ell$, $v \in \textsc{Aff}_x$ if and only if $v \in \mathtt{A}(x)$ and prove that the inductive claim holds for nodes u such that $d_i(u, x) = \ell + 1$. If $u \in \textsc{Aff}_x$, then there exists at least a neighbor v of u such that $v \in \textsc{Aff}_x$ and $d_i(v, x) = \ell$. By induction, $v \in \mathtt{A}(x)$, and at the iteration of the while loop in which v is inserted into $\mathtt{A}(x)$, all its neighbors are analyzed and hence u is analyzed at lines 10 through 15. Since $u \in \textsc{Aff}_x$, then either one of the hubs of pair (u, y) is in $\textsc{Aff}_x$ or such hubs are u or y themselves. In the former case, let h be a hub of (u, y) such that $h \in \textsc{Aff}_x$; then, by inductive hypothesis, $h \in \mathtt{A}(x)$. In the latter case, u is affected if any shortest path between u and y passes through edge $\{x, y\}$, which implies that $d_{i-1}(u, y) = d_{i-1}(u, x) + 1$. In both cases the condition at line 13 holds and therefore u is inserted into $\mathtt{Q}$ and, when extracted, is inserted into $\mathtt{A}(x)$. If $u \notin \textsc{Aff}_x$, then all its hubs toward y are not in $\mathtt{A}(x)$. Since all its hubs have a distance to y that is smaller than that of u, then by the inductive hypothesis they are not in $\mathtt{A}(x)$ when u is analyzed. Moreover, if a hub of (u, y) is u or y, then $d_{i-1}(u, y) < d_{i-1}(u, x) + 1$; otherwise, a shortest path between u and y passes through $\{x, y\}$. Hence, the condition at line 13 does not hold for u and it is not inserted into $\mathtt{Q}$ and $\mathtt{A}(x)$. □

By elaborating on the proof of Lemma 3.5, it can be easily shown that the symmetric version of Algorithm 1, where the visit is rooted at y, is able to compute $\mathtt{A}(y) = \textsc{Aff}_y$. Moreover, we can prove the following.

Lemma 3.6. *Algorithm 1 requires* $O(m_{\mathtt{A}(x)} \Lambda \log |\mathtt{A}(x)| + m_{\mathtt{A}(y)} \Lambda \log |\mathtt{A}(y)|)$ *worst-case computational time.*

Proof. Algorithm 1 performs a BFS-like visit rooted at x, which settles only nodes in $\mathtt{A}(x)$. Now, we bound the time needed to perform the visit rooted at x. In detail, for each node $v \in \mathtt{A}(x)$, and for each $u \in N(v)$, Algorithm 1, in order to perform the test of Line 13, computes the set of hubs

H toward y (line 11). Such a computation, for a single pair u, y, requires $O(|L(u)| + |L(y)|) = O(\Lambda)$ time, since the two ordered labels $L(u)$ and $L(y)$ have to be scanned. Performing the test of line 13 requires $O(\Lambda \log |\mathsf{A}(x)|)$ time, since checking whether a hub h belongs to the ordered set $\mathsf{A}(x)$ requires logarithmic time. Hence, the overall cost of lines 10 through 13 is $O(m_{\mathsf{A}(x)}\Lambda \log |\mathsf{A}(x)|)$. It follows that the overall time complexity of the visit rooted at x, performed by Algorithm 1, is $O(m_{\mathsf{A}(x)}\Lambda \log |\mathsf{A}(x)|)$, since line 1 requires $O(1)$ time. Therefore, Algorithm 1, on the whole, requires $O(m_{\mathsf{A}(x)}\Lambda \log |\mathsf{A}(x)| + m_{\mathsf{A}(y)}\Lambda \log |\mathsf{A}(y)|)$. □

By elaborating on the proof of Lemma 3.6, we can obtain an equivalent complexity result for the symmetric version of Algorithm 1 that computes $\mathsf{A}(y) = \textsc{Aff}_y$; namely, we can claim that said algorithm requires $O(m_{\mathsf{A}(y)}\Lambda \log |\mathsf{A}(y)| + m_{\mathsf{A}(x)}\Lambda \log |\mathsf{A}(x)|)$ worst-case computational time.

3.2 Removing Affected Hubs

Given the two sets $\mathsf{A}(x)$ and $\mathsf{A}(y)$, Algorithm 2 removes outdated label entries (u, δ_{uv}) from $L(v)$ for all affected nodes u and v that satisfy the condition of Lemma 3.3. The following lemma follows by simply observing that, by Lemma 3.5, $\mathsf{A}(x) = \textsc{Aff}_x$.

ALGORITHM 2: Remove affected hubs from affected labels

```
1 foreach (u, v) : (v ∈ A(x) ∧ u ∈ A(y)) ∨ (v ∈ A(y) ∧ u ∈ A(x)) do
2 |   if u ∈ L(v) then
3 |   |   Remove (u, δ_uv) from L(v);
```

Lemma 3.7. *At the end of Algorithm 2, L does not contain any pair that satisfies the condition of Lemma 3.3.*

Lemma 3.8. *Algorithm 2 requires $O(|\mathsf{A}(x)|\Lambda \log |\mathsf{A}(y)| + |\mathsf{A}(y)|\Lambda \log |\mathsf{A}(x)|)$ worst-case computational time.*

Proof. In Algorithm 2, for each affected node $v \in \mathsf{A}(x)$ ($v \in \mathsf{A}(y)$, respectively), each element of $L(v)$ is scanned in order to test whether the corresponding node belongs to the ordered set $\mathsf{A}(y)$ ($\mathsf{A}(x)$, respectively) or not. Since such test requires logarithmic time and since it is performed for each element of $L(v)$, it follows that the time complexity of Algorithm 2 is $O(|\mathsf{A}(x)|\Lambda \log |\mathsf{A}(y)| + |\mathsf{A}(y)|\Lambda \log |\mathsf{A}(x)|)$. □

3.3 Computing New Hubs

In this section, we present the third phase of DecPLL, whose aim is restoring the cover property of those pairs that might be not covered by L after phase 2 as a consequence of removals of outdated label entries, i.e., for all pairs u, v such that $u \in \mathsf{A}(x)$ and $v \in \mathsf{A}(y)$. First of all, we assume that $d_i(x, y) \neq \infty$, since otherwise L is already correct and DecPLL can be stopped after the execution of Algorithm 2 and after detecting that condition $d_i(x, y) = \infty$ is true.

Now, to achieve a labeling L that covers G_i we basically need to

- —analyze all pairs s, t such that s and t are affected and, in particular, all those pairs s, t such that $s \in \mathsf{A}(x)$ and $t \in \mathsf{A}(y)$ or vice versa, and
- —test, for each of these pairs, whether the pair is covered and, if not, add some suitable label entries to restore the cover property.

To this regard, we first need to check for each $s, t \in V$ whether $L(s) \cap L(t)$ contains at least a node u in a shortest path between s and t. In the affirmative case, the pair is covered, so the algorithm does not need to add any label entry to either $L(s)$ or $L(t)$. In the negative case, instead, we need

to add some suitable hub node to $L(s)$ or to $L(t)$. This requires the computation of a shortest path between s and t in G_i. For this purpose, in the following we propose two possible strategies, namely, Greedy–Restore and Order–Restore, whose common underlying idea is that of discovering (suitable) shortest paths in G_i and compare their weight with that of paths induced by L.

3.3.1 Greedy–Restore. The first approach, shown in Algorithm 3, follows a greedy paradigm and performs the greedy choice of running a set of BFS-like visits rooted only on nodes of the set, among $\mathtt{A}(x)$ and $\mathtt{A}(y)$, having minimum cardinality. In more detail, let us define SA to be such a set, i.e.,

$$\mathtt{SA} = \begin{cases} \mathtt{A}(x) & \text{if } |\mathtt{A}(x)| < |\mathtt{A}(y)| \\ \mathtt{A}(y) & \text{otherwise.} \end{cases}$$

Symmetrically, we call LA the largest of the two sets, i.e.,

$$\mathtt{LA} = \begin{cases} \mathtt{A}(y) & \text{if } |\mathtt{A}(x)| < |\mathtt{A}(y)| \\ \mathtt{A}(x) & \text{otherwise.} \end{cases}$$

Algorithm 3 performs a BFS rooted at each node $a \in \mathtt{SA}$. As in Algorithm 1, we use a queue Q to store the nodes to visit and a Boolean vector `mark` to keep track of the visited nodes. Moreover, we use a vector `dist` to store the distances between a and any other node reached during the visit. At lines 7 through 12, we initialize Q, `mark`, and `dist`. At lines 13 through 22, the BFS-like visit, rooted at each $a \in \mathtt{SA}$, is actually run. Let v be the node extracted from Q at line 14 and let L be the current label set. If $v \in \mathtt{LA}$, then the result of a query between a and v that uses the label set might be wrong due to Lemma 3.3. Therefore, we check whether the value $\mathtt{dist}[v]$ of the discovered distance satisfies $\mathtt{dist}[v] < \textsc{Query}(a, v, L)$ or not. In the affirmative case, it follows that the path between a and v found by the BFS is shorter than all those currently induced by L. Hence, we update L either by inserting pair $(v, \mathtt{dist}[v])$ in $L(a)$ or by inserting pair $(a, \mathtt{dist}[v])$ in $L(v)$, depending on the relative ordering of v and a (see lines 15 through 18). This is done in order to preserve the *well-ordering* property (see Section 2). Note that the test of line 16 allows us to avoid the computation of distances between all pairs (u, v) such that $v \in \mathtt{LA}$ and $u \in \mathtt{SA}$ since, in some case, a hub h between u and v might have been found in previous iterations of the algorithm.

Theorem 3.9. *At the end of Algorithm 3, L is a 2-hop Cover labeling that covers G_i.*

Proof. By contradiction, let (a, v) be a pair of nodes such that, at the end of Algorithm 3, $\textsc{Query}(a, v, L) \neq d_i(a, v)$. Since $a \in \mathtt{SA}$ and $v \in \mathtt{LA}$, by Lemma 3.3, it follows that $a \in \mathtt{A}(x)$ and $v \in \mathtt{A}(y)$ (or vice versa). Moreover, by Lemma 3.7, $\textsc{Query}(a, v, L) > d_i(a, v)$. Consider the BFS rooted at a. When the algorithm extracts v from Q, we have that $\mathtt{dist}[v] = d_i(a, v, L)$ by the property of BFS. Then, since we are assuming that $\textsc{Query}(a, v, L) > d_i(a, v)$ at the end of the algorithm, it follows that this was also true when v is extracted. Thus, the test of line 16 succeeds and the algorithm reaches line 17. Now, either $(v, d_i(a, v))$ is inserted into $L(a)$ or $(a, d_i(a, v))$ is inserted into $L(v)$. In both cases, since $(a, 0) \in L(a)$ and $(v, 0) \in L(v)$, then $\textsc{Query}(a, v, L) = d_i(a, v)$. □

The next theorem shows that the labeling resulting from Algorithm 3 is minimal. On the one hand, this leads to a reduced query time. On the other hand, it implies that the deletion of any pair in the labeling would lead to some incorrect query (i.e., breaks the cover property for at least a pair of nodes). However, in Algorithm 1, we exactly identify the labels that need to be updated in order to keep all the queries correct.

Theorem 3.10. *If L is a minimal 2-hop Cover labeling of G_{i-1} that satisfies the well-ordering property, then at the end of Algorithm 3, L is a minimal 2-hop Cover labeling of G_i that satisfies the well-ordering property.*

ALGORITHM 3: Computation of new hubs: algorithm GREEDY–RESTORE

```
1  SA ← A(x);
2  LA ← A(y);
3  if |A(x)| > |A(y)| then
4  |   SA ← A(y);
5  |   LA ← A(x);
6  foreach a ∈ SA do
7  |   foreach v ∈ V \ {a} do
8  |   |   mark[v] ← false;
9  |   |   dist[v] ← ∞;
10 |   Q ← ∅;
11 |   mark[a] ← true;
12 |   dist[a] ← 0;
13 |   while Q ≠ ∅ do
14 |   |   v ← Q.Dequeue();
15 |   |   if v ∈ LA then
16 |   |   |   if dist[v] < QUERY(a, v, L) then
17 |   |   |   |   if v < a then Insert (v, dist[v]) in L(a);
18 |   |   |   |   else Insert (a, dist[v]) in L(v);
19 |   |   foreach u ∈ N_i(v) : ¬mark[u] do
20 |   |   |   dist[u] ← dist[v] + 1;
21 |   |   |   mark[u] ← true;
22 |   |   |   Q.Enqueue(u);
```

PROOF. We first show that if L is a 2-hop Cover labeling of G_{i-1} that satisfies the well-ordering property, then, at the end of Algorithm 3, L is a 2-hop cover labeling of G_i that satisfies the well-ordering property. Let us suppose by contradiction that such statement is not true. Then, it follows that there exist at least two nodes u and v of G_i such that $u < v$ and $(v, \delta_{uv}) \in L(u)$. Since this is false for L before the edge removal, as it is a well-ordering cover by hypothesis, and since Algorithm 3 only inserts pairs (a, δ_{ab}) in $L(b)$ if and only if $a < b$, the contradiction follows.

We now show that L is minimal. Let us denote the labeling L before and after applying the DECPLL algorithm by L_{i-1} and L_i, respectively. Let us suppose by contradiction that the statement is not true, i.e., that there exists, for some node s, a pair $(u, \delta_{su}) \in L_i(s)$ whose removal does not affect the value returned by the query; i.e., $\text{QUERY}(s, t, L_i) = \text{QUERY}(s, t, L') = d_i(s, t)$ for any other t of the graph, where L' is obtained from L_i by removing (u, δ_{su}) in $L_i(s)$. Since this is claimed to be true for all other nodes of the graph, it follows that $\text{QUERY}(s, u, L') = d_i(s, u)$. Therefore, there exists another hub, say, h, such that $h \in \{L'(s) \cap L'(u)\}$ and $d_i(s, u) = \delta_{sh} + \delta_{hu}$. Since we are assuming that the removed pair is (u, δ_{su}), it clearly follows that h must also be in both $L_i(s)$ and $L_i(u)$,

Now, we can observe that only the two following cases can occur: either (1) $h \in L_{i-1}(u)$ and $u \notin L_{i-1}(s)$ or (2) $h \notin L_{i-1}(u)$ and $u \in L_{i-1}(s)$. In fact: (1) having both $h \in L_{i-1}(u)$ and $u \in L_{i-1}(s)$ would imply the violation of the well-ordering property, because if $u < h$, then h cannot belong to $L_{i-1}(u)$, while if $u > h$, then h cannot be in $L_{i-1}(s)$; (2) having neither $h \in L_{i-1}(u)$ nor $u \in L_{i-1}(s)$ would contradict the fact that $h \in L_i(s) \cap L_i(u)$ and $h \in L'(s) \cap L'(u)$.

In the first case, i.e., if $h \in L_{i-1}(u)$ and $u \notin L_{i-1}(s)$, we immediately reach an absurd conclusion. In fact, this implies that $h < u$ and that the pair (u, δ_{su}) cannot be inserted in $L_i(s)$ (see line 15 of Algorithm 3), which contradicts the hypothesis (u, δ_{su}) in $L_i(s)$.

In the second case, i.e., if $h \notin L_{i-1}(u)$ and $u \in L_{i-1}(s)$, we have that h has been added to $L_i(u)$ by Algorithm 3, since by the hypothesis $h \in L_i(u)$. Therefore, we have that either $h \in \textsc{Aff}_x$ and $u \in \textsc{Aff}_y$ or $h \in \textsc{Aff}_y$ and $u \in \textsc{Aff}_x$, as Algorithm 3 adds new label entries only for such kinds of pairs of nodes. This again leads to an absurd conclusion. In fact, let us denote by $\{x, y\}$ the removed edge. It follows that, in G_i, the following three statements must be true at the same time: (1) a shortest path between u and h includes $\{x, y\}$, (2) a shortest path in G_i between s and h includes $\{x, y\}$, and (3) a shortest path in G_i between s and u passing through h includes $\{x, y\}$. This is a contradiction, as it implies that a shortest path between s and u is not a simple path in G_i. □

Lemma 3.11. *Algorithm 3 requires* $O(|\mathsf{SA}|(m + n \log |\mathsf{LA}| + n\Lambda))$ *worst-case computational time.*

Proof. Algorithm 3 performs a BFS-like visit for each node of SA, where the difference with respect to a standard BFS essentially consists of lines 16, 15, and 18. Performing such lines requires $O(\Lambda + \log |\mathsf{LA}|)$ worst-case time, as performing $\textsc{Query}(s, t, L)$ for a pair $s, t \in V$ requires $O(\Lambda)$ time, while the membership test on LA takes $O(\log |\mathsf{LA}|)$ time per test. Since these lines, in the worst case, are performed n times in the while loop of Line 13, the overall cost of Algorithm 3 is $O(|\mathsf{SA}|(m + n \log |\mathsf{LA}| + n\Lambda))$ as a BFS-like visit has to be performed for each of the nodes in SA. □

By Lemmas 3.6, 3.8, and 3.11, it follows that DecPLL with Greedy–Restore requires $O(m_{\mathsf{A}(x)}\Lambda \log |\mathsf{A}(x)| + m_{\mathsf{A}(y)}\Lambda \log |\mathsf{A}(y)| + |\mathsf{A}(x)|\Lambda \log |\mathsf{A}(y)| + |\mathsf{A}(y)|\Lambda \log |\mathsf{A}(x)| + |\mathsf{SA}|(m + n \log |\mathsf{LA}| + n\Lambda)) = O(m_{\mathsf{LA}}\Lambda \log |\mathsf{LA}|) + |\mathsf{LA}|\Lambda \log |\mathsf{LA}| + |\mathsf{SA}|(m + n \log |\mathsf{LA}| + n\Lambda))$ worst-case time. Since $|\mathsf{LA}| \leq m_{\mathsf{LA}}$, the following theorem follows.

Theorem 3.12. *Algorithm* DecPLL*, with* Greedy–Restore*, requires* $O(m_{\mathsf{LA}}\Lambda \log |\mathsf{LA}| + |\mathsf{SA}|(m + n \log |\mathsf{LA}| + n\Lambda))$ *worst-case computational time.*

Notice that, since in the worst case we have that $\Lambda = O(n)$, $|\mathsf{LA}| = O(n)$, $|\mathsf{SA}| = O(n)$, and $m_{\mathsf{LA}} = O(m)$, if we analyze the complexity of DecPLL as a function of n and m only, it requires $O(nm \log n + n^3)$ worst-case time. This is clearly worse than the cost of PLL, which requires $O(nm + n^2)$ time in the worst case. However, in practice, the sizes of SA and LA are much smaller than n, as well as Λ, thus suggesting that DecPLL will behave well in practice. This observation will be confirmed by our experimental results, which are shown in Section 6.

3.3.2 Order–Restore. Note that the Greedy–Restore approach restores the cover properties for all pairs of affected nodes s, t by running a set of BFS-like visits rooted at nodes of SA only. This choice causes the shortest paths of pairs s, t in G_i to be discovered in a shuffled order that is not coherent with the node ordering used for building the labeling; i.e., when a path between s and t is found by Algorithm 3, then either $s < t$ or vice versa. This is true even if SA is kept sorted by the node ordering, since nodes in LA are arbitrarily met during the searches. Therefore, in order to preserve the well-ordering property, and thus the minimality, of the labeling, Algorithm 3 needs to perform the test of line 17. Moreover, when performing the visit rooted at s, we cannot prune the visit on the basis of the ordering, as done, for instance, by using PrefQuery within IncPLL. Both of the above issues might induce Algorithm DecPLL to spend very large computational time on performing Greedy–Restore, especially when sets SA and LA are balanced in size.

In what follows, we propose an alternative strategy, named Order–Restore, for restoring the cover property that takes into account the above considerations. This second approach is shown in Algorithm 4 and, unlike Greedy–Restore, performs a BFS-like visit rooted at v, for each affected node v, regardless of the sizes of the two sets.

This of course leads to a larger number of BFS-like visits to be performed with respect to Greedy–Restore. However, if such visits are performed according to the node ordering (i.e., if s, t are affected, then the visit rooted at s is performed before the visit rooted at t if and only if

ALGORITHM 4: Computation of new hubs: algorithm Order–Restore

```
1  FA ← A(x) ∪ A(y);
2  Sort FA by the node ordering;
3  foreach a ∈ FA in increasing order do
4      foreach v ∈ V \ {a} do
5          mark[v] ← false;
6          dist[v] ← ∞;
7      Q ← ∅;
8      mark[a] ← true;
9      dist[a] ← 0;
10     while Q ≠ ∅ do
11         v ← Q.Dequeue();
12         if v < a then continue;
13         if (a ∈ A(x) ∧ v ∈ A(y)) ∨ (a ∈ A(y) ∧ v ∈ A(x)) then
14             if dist[v] < Query(a, v, L)) then Insert (a, dist[v]) in L(v);
15         foreach u ∈ N_i(v) : ¬mark[u] do
16             dist[u] ← dist[v] + 1;
17             mark[u] ← true;
18             Q.Enqueue(u);
```

$s < t$), we can exploit the ordering itself to avoid the test of line 17 of Algorithm 3 and to prune the search in a more aggressive way, with respect to Greedy–Restore. In particular, when a shortest path in G_i between a node a and a node v is found by the visit rooted at a (see line 11), we are sure that all shortest paths between s and t such that $s < t$ have been discovered and tested in previous visits. Therefore, if $v < a$, we can prune the search at v (see line 12). Moreover, after checking whether the value $\texttt{dist}[v]$ of the discovered distance satisfies $\texttt{dist}[v] < \textsc{Query}(a, v, L)$ or not, we can directly insert, in the affirmative case, pair $(a, \texttt{dist}[v])$ in $L(v)$ without testing the relative ordering of v and a (see line 14). This behavior can be easily achieved by considering sets $\texttt{A}(x)$ and $\texttt{A}(y)$ as a single set, say, FA; by sorting it in accordance with the node ordering; and by performing BFS-like visits accordingly. The next theorem can be easily derived by elaborating on Theorems 3.9 and 3.10.

Theorem 3.13. *At the end of Algorithm 4, L is a 2-hop Cover labeling that covers G_i. Moreover, if L is a minimal 2-hop Cover labeling of G_{i-1} that satisfies the well-ordering property, then at the end of Algorithm 4, L is a minimal 2-hop Cover labeling of G_i that satisfies the well-ordering property.*

In addition, by using arguments that are similar to those used in the proofs of both Lemma 3.11 and Theorem 3.12, the two following results concerning Algorithm 4 immediately follow.

Lemma 3.14. *Algorithm 4 requires $O(|\texttt{FA}|(m + n \log |\texttt{LA}| + n\Lambda))$ worst-case computational time.*

Theorem 3.15. *Algorithm* DecPLL*, with* Order–Restore*, requires $O(m_{\texttt{LA}}\Lambda \log |\texttt{LA}| + |\texttt{FA}|(m + n \log |\texttt{LA}| + n\Lambda))$ worst-case time.*

Clearly, Order–Restore might require either larger or smaller computational time with respect to Greedy–Restore, depending on the size of the sets $\texttt{A}(x)$ and $\texttt{A}(y)$. In fact, Order–Restore performs more visits that, however, touch a number of nodes that decreases as the number of performed visits increases. Indeed, it is hard to determine which of the strategies works better in any given case, since the computational time spent by both methods highly depends on the structure of the network. Therefore, in our experimental study, we have conducted a set of preliminary

experiments to estimate, empirically, when it is convenient to use Order–Restore rather than Greedy–Restore and vice versa.

3.4 Improved Detection of Affected Nodes

In this section, we provide an alternative definition of the sets of affected nodes that allows us to efficiently compute them. The motivation comes from the observation that Algorithm 1 requires, at line 11, computing the set H of all hubs between pairs (u, y). This is computationally expensive as it requires scanning $L(u)$ and $L(y)$ for each pair u, y, which takes $O(|L(u)| + |L(y)|)$ time in the worst case. Moreover, the condition at line 13 requires, for each element of H, testing whether this belongs to $\mathtt{A}(x)$ or not, which takes $\Theta(|H| \log |\mathtt{A}(x)|)$ overall worst-case time if $\mathtt{A}(x)$ is stored as a balanced binary search tree. Equivalently, this could be done by scanning H and $\mathtt{A}(x)$ to check whether $\{H \cap \mathtt{A}(x)\} \neq \emptyset$ in $O(|H| + |\mathtt{A}(x)|)$ time.

In order to introduce the new definition of the affected node, for each pair of nodes (s, t), we consider a particular path in $\textsc{Path}(s, t, L)$, where L is a 2-hop Cover of G_{i-1}. In detail, let P be a path between s and t in $\textsc{Path}(s, t, L)$ such that, for any two nodes u and v in P, the *smallest* (with respect to the node ordering) hub h of pair (u, v) is such that $h \in P$, $h = u$, or $h = v$. The set of such shortest paths between nodes s and t is denoted by $\textsc{Path}'(s, t, L)$. We define $\overline{\textsc{Aff}_x}$ as the set of nodes v such that $d_{i-1}(v, x) < d_{i-1}(v, y)$ and there exists a path in $\textsc{Path}'(v, u, L)$ that passes through edge $\{x, y\}$ in G_{i-1} for some node u. Set $\overline{\textsc{Aff}_y}$ is defined symmetrically. In the following, we only discuss the cases in which $v \in \overline{\textsc{Aff}_x}$ and the cases in which $v \in \overline{\textsc{Aff}_y}$ are symmetrical. The difference from the previous definition essentially consists of pairs (u, v) that have more than one hub. Let us assume that (u, v) has two hubs h_1 and h_2, with $h_1 < h_2$. The following cases occur:

- A shortest path between u and v that contains h_2 passes through $\{x, y\}$, while the shortest paths that contain h_1 do not. In this case, $v \in \textsc{Aff}_x$ and $v \notin \overline{\textsc{Aff}_x}$. However, $\textsc{Query}(u, v, L)$ is correct, that is, $\textsc{Query}(u, v, L) = d_i(u, v)$, since $d_i(u, h_2) = d_{i-1}(u, h_2)$, $d_i(v, h_2) = d_{i-1}(v, h_2)$, $(h_2, d_{i-1}(u, h_2)) \in L(u)$, and $(h_2, d_{i-1}(v, h_2)) \in L(v)$.
- A shortest path between u and v that contains h_1 passes through $\{x, y\}$, while the shortest paths that contain h_2 do not. In this case, $v \in \textsc{Aff}_x$ and $v \in \overline{\textsc{Aff}_x}$, and therefore, $\textsc{Query}(u, v, L)$ is not correct and L must be updated as in the definition of the affected node given in Section 3.1.
- A shortest path between u and v that contains h_1 passes through $\{x, y\}$, and a shortest path that contains h_2 passes through $\{x, y\}$ as well. Also, in this case $v \in \textsc{Aff}_x$ and $v \in \overline{\textsc{Aff}_x}$ as in the previous case.
- No shortest path between u and v induced by L passes through $\{x, y\}$. In this case, $v \notin \textsc{Aff}_x$ and $v \notin \overline{\textsc{Aff}_x}$, and then $\textsc{Query}(u, v, L) = d_{i-1}(u, v) = d_i(u, v)$.

The generalization of the above arguments to the case of pairs having more than two hubs is straightforward. This implies that we do not need to consider the nodes in $\textsc{Aff}_x \setminus \overline{\textsc{Aff}_x}$ as affected. Therefore, the next result can be given.

Lemma 3.16. *Let L be a minimal 2-hop Cover labeling of graph G_{i-1}. Let us assume that an edge $\{x, y\}$ is removed from G_{i-1} and that G_i is the resulting graph for $i \geq 1$. Then, for each $u, v \in V$, we have that $\textsc{Query}(u, v, L) \neq d_i(u, v)$ only if $v \in \overline{\textsc{Aff}_x}$ and $u \in \overline{\textsc{Aff}_y}$ or $v \in \overline{\textsc{Aff}_y}$ and $u \in \overline{\textsc{Aff}_x}$.*

Similarly, the next corollary trivially follows.

Corollary 3.17. *Let L be a minimal 2-hop Cover labeling of graph G_{i-1}. Let us assume that an edge $\{x, y\}$ is removed from G_{i-1} and that G_i is the resulting graph for $i \geq 1$. Then, for all nodes v and*

u such that either $v \notin \overline{\text{AFF}_x} \vee u \notin$ or $v \notin \overline{\text{AFF}_y} \vee u \notin \overline{\text{AFF}_x}$, all entries $(u, \delta_{vu}) \in L(v)$ are correct for G_i, i.e., $\delta_{vu} = d_i(v, u)$.

It follows that we can use set $\overline{\text{AFF}_x}$ instead of AFF_x in the DecPLL algorithm. The pseudo-code of the new algorithm to detect $\overline{\text{AFF}_x}$ is given in Algorithm 5. The main difference with respect to Algorithm 1 is in lines 11 through 18. In particular, we heuristically avoid computing the set of hubs of pairs (u, y) in many cases by checking whether $d_i(u, y) \neq d_{i-1}(u, y)$. If such condition holds, then u is inserted in $\texttt{A}(x)$ thanks to the property that, if $d_i(u, y) \neq d_{i-1}(u, y)$, for some $u \in V$, then $u \in \overline{\text{AFF}_x}$ (see line 11).

ALGORITHM 5: Alternative computation of $\texttt{A}(x)$

```
1  A(x) ← ∅;
2  foreach v ∈ V do
3  |  mark[v] ← false;
4  Q ← ∅;
5  mark[x] ← true;
6  Q.Enqueue(x);
7  while Q ≠ ∅ do
8  |  v ← Q.Dequeue();
9  |  A(x) ← A(x) ∪ {v};
10 |  foreach u ∈ N_i(v) such that ¬mark[u] do
11 |  |  if d_i(u, y) ≠ d_{i−1}(u, y) then
12 |  |  |  mark[u] ← true;
13 |  |  |  Q.Enqueue(u);
14 |  |  else
15 |  |  |  Let h be the smallest hub in the set of hubs of pair (u, y) in L;
16 |  |  |  if h ∈ A(x) ∨ ((h = u ∨ h = y) ∧ d_{i−1}(u, y) = d_{i−1}(u, x) + 1) then
17 |  |  |  |  mark[u] ← true;
18 |  |  |  |  Q.Enqueue(u);
```

To implement such heuristic, distances $d_{i-1}(x, v)$ and $d_i(x, v)$ need to be computed. This is done by means of two BFS rooted at x in G_{i-1} and G_i, respectively. In the case that $d_i(u, y) = d_{i-1}(u, y)$, then u might still be affected. However, we do not need to scan the entire set H of all hubs between pairs (u, y), but we only check the smallest hub (see lines 15 and 16) thanks to the definition of $\overline{\text{AFF}_x}$. The proof of the next lemma is similar to that of Lemma 3.5.

Lemma 3.18. *At the end of Algorithm 5,* $\texttt{A}(x) = \overline{\text{AFF}_x}$.

Clearly, the computational complexity of Algorithm 5, in the worst case, is the same as that of Algorithm 1. However, in practice, it allows one to reduce by far the computational effort required for the computation of the sets of affected nodes. In our experimental study, we conducted a set of preliminary experiments to support this claim. Notice that analogous modifications can be done to the symmetric version of Algorithm 1, rooted at node y, to compute $\overline{\text{AFF}_y}$.

As a final remark, it is worth observing that sets $\overline{\text{AFF}_x}$ and $\overline{\text{AFF}_y}$ are such that $|\overline{\text{AFF}_x}| \leq |\text{AFF}_x|$ and $|\overline{\text{AFF}_y}| \leq |\text{AFF}_y|$. This is a desirable outcome, since the computational complexity of the third phase of DecPLL, whose aim is that of restoring the cover property, directly depends on the cardinality of the above sets of affected nodes, as we have shown in Section 3.3.

4 EXTENSIONS

In this section, we discuss how to extend all algorithms given in the previous sections to the cases of *directed* and *weighted* graphs. The two extensions can be combined in order to handle *directed weighted* graphs.

4.1 Directed Graphs

In a directed graph, we denote by $d(s,t)$ the distance *from* node s *to* node t. To adapt the 2-hop Cover labeling method to a directed graph $G = (V,E)$, it is enough to define two label sets L_{in} and L_{out} (Akiba et al. 2013). For each v, $L_{in}(v)$ stores a set of pairs (u, δ_{uv}), while $L_{out}(v)$ stores pairs (u, δ_{vu}), where $\delta_{uv} = d(u,v)$ and $\delta_{vu} = d(v,u)$. In this case, labels can be used to answer a *query* on the distance from node s to node t as

$$\textsc{Query}(s,t,L) = \begin{cases} \min\limits_{v \in V} \{\delta_{sv} + \delta_{vt} \mid v \in L_{out}(s) \wedge v \in L_{in}(t)\} & \text{if } L_{out}(s) \cap L_{in}(t) \neq \emptyset \\ \infty & \text{otherwise.} \end{cases}$$

A labeling $L = \{L_{out}, L_{in}\}$ is called a *2-hop Cover labeling* of a directed graph G if, for each pair $s,t \in V$, $L_{out}(s) \cap L_{in}(t)$ contains at least a node u in a shortest path *from* s *to* t (or it is empty if there is no path from s to t). The definitions of *hub*, *covered pair* and *graph*, and *cover property* follow accordingly.

It can be shown (see Akiba et al. (2013)) that $\textsc{Query}(s,t,L) = d(s,t)$ if the two label sets L_{in} and L_{out} are computed by using bidirectional versions of either the *naive* method or PLL that are suited for being used in directed graphs. Note that, unlike undirected graphs, $\textsc{Query}(s,t,L)$ is not necessarily equal to $\textsc{Query}(t,s,L)$. Both adaptations perform two BFS searches (which can be possibly pruned in the case of PLL only) for each node $v \in V$: one in G and one in the transpose graph $\overline{G}$, which is defined as $\overline{G} = (V, \overline{E})$, where $\overline{E} = \{(u,v) : (v,u) \in E\}$. Therefore, for each node $v \in V$, the first search essentially finds all shortest paths from v, while the second one finds all shortest paths to v. Notice that the pruning mechanism of the directed version of PLL is basically identical to that of the undirected version, with the only exception of considering the orientation of shortest paths and hence of queries on the labeling, i.e., that, in general, $\textsc{Query}(s,t,L)$ can differ from $\textsc{Query}(t,s,L)$.

As well as the approaches for computing L_{in} and L_{out} from scratch, the dynamic algorithms discussed in Sections 2.3 and 3 can be extended to be used in directed graphs in order to update both L_{in} and L_{out} in case of graph update operations. The common underlying idea of all modifications is that of considering the orientation of edges, and therefore of shortest paths, when updating the data structure.

In particular, regarding IncPLL, the key modification consists of taking into account the fact that a newly inserted edge (a,b) potentially induces changes in all those shortest paths passing through (a,b); that is, (1) if the distance *from* a node v_k *to* a node u changes, then all new shortest paths from v_k to u pass through the new directed edge (a,b), and (2) if the shortest path P from v_k to $u \neq a,b$ changes, then the distance from v_k to w changes, where w is the penultimate node in P. Therefore, it can be easily shown that, in order to update the labeling, it suffices to resume, for each $v_k \in L_{in}(a) \cup L_{out}(b)$:

- the BFS in $\overline{G}$, rooted at v_k, *from* b, by inserting pair $(b, d_{\overline{G}}(b, v_k)))$ to the initial queue of the BFS (here $d_{\overline{G}}(b, v_k)$ denotes the distance in $\overline{G}$ from b to v_k before the insertion), and
- the BFS in G, rooted at v_k, *from* a, by inserting pair $(a, d_G(a, v_k))$ to the initial queue of the BFS (here $d_G(a, v_k)$ denotes the distance in G from a to v_k before the insertion).

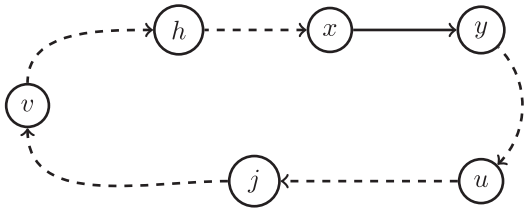

Fig. 2. A shortest path from node v to node u. The solid line represents edge (x, y), while dashed lines represent shortest paths. Assume that $h \in L_{out}(v) \cap L_{in}(u)$; then h is a hub for pair (v, u). If (x, y) is removed, then $v \in \text{AFF}_x$ and $u \in \text{AFF}_y$, and label entry (h, δ_{hu}) in $L_{in}(u)$ is not correct and must be updated.

Regarding DecPLL, its adaptation to directed graphs is slightly more technically involved, as it affects each of the three phases of the algorithm, which we discuss separately. To this aim, we first need to review some of the definitions given in Section 3 in order to be suitable for directed graphs. In particular, while the general notion of affected nodes remains essentially unchanged (it suffices to consider orientation in paths *induced* by L), its partition into two sets, to be exploited to determine outdated label entries, is significantly different in the case of directed graphs. In fact, given a removed edge (x, y), for each affected node v we have that either $d_{i-1}(v, x) < d_{i-1}(v, y)$ or $d_{i-1}(x, v) > d_{i-1}(y, v)$ or *both* (see node u in Figure 2 for a graphical example of this latter case).

Therefore, we can still logically divide set AFF into two subsets AFF_x and AFF_y, but in this case such sets might be not disjoint. Indeed, affected nodes satisfy one (or both) of the following two conditions:

- —There exists a shortest path *induced* by L from v to any other node u, i.e., a path in $\text{PATH}(v, u, L)$ that passes through edge (x, y) in G_{i-1}.
- —There exists a shortest path *induced* by L from any other node u to v, i.e., a path in $\text{PATH}(u, v, L)$ that passes through edge (x, y) in G_{i-1}.

We consider nodes of the first kind to belong to AFF_x and nodes of the second kind to belong to AFF_y. Nodes satisfying both conditions belong to both sets. Now, we observe that if u and v are two affected nodes such that $u \in \text{AFF}_x$ and $v \in \text{AFF}_y$ and a shortest path from u to v passes through (x, y), then a hub h of pair (u, v) is also affected and either $h \in \text{AFF}_x$ or $h \in \text{AFF}_y$ (see Figure 2). The next lemma gives us a way to exploit the set of affected nodes, in the case of directed graphs, to identify the pairs in L that must be updated. The proof easily follows by the proof of Lemma 3.3.

Lemma 4.1. *Let L be a 2-hop Cover labeling of a directed graph G_{i-1}. Let us assume that an edge (x, y) is removed from G_{i-1} and that G_i is the resulting graph for $i \geq 1$. Then, for each $u, v \in V$, we have that $\text{QUERY}(u, v, L) \neq d_i(u, v)$ only if $u \in \text{AFF}_x$ and $v \in \text{AFF}_y$. Moreover, we have that $\text{QUERY}(v, u, L) \neq d_i(v, u)$ only if $v \in \text{AFF}_x$ and $u \in \text{AFF}_y$.*

Given the above considerations, the statement of Lemma 3.4 changes as follows for directed graphs. The proof directly follows by the proof of Lemma 3.4.

Lemma 4.2. *Let L be a 2-hop Cover labeling of a directed graph G_{i-1}. Let us assume that an edge (x, y) is removed from G_{i-1} and that G_i is the resulting graph for $i \geq 1$. Then, for all nodes v and u such that $v \notin \text{AFF}_x \vee u \notin \text{AFF}_y$, all entries $(u, \delta_{vu}) \in L_{out}(v)$ and $(v, \delta_{vu}) \in L_{in}(u)$ are correct for G_i, i.e., $\delta_{vu} = d_i(v, u)$.*

Given the two above lemmas, we can now define a modification of Algorithm 1 for performing the first phase of DecPLL in directed graphs, i.e., for computing sets AFF_x and AFF_y as defined above. The main difference underlying the modified algorithm is that orientation in both shortest

paths and queries has to be considered while trying to identify the two sets $\textsc{Aff}_x$ and $\textsc{Aff}_y$. This, unlike the case of undirected graphs, leads to the definition of two different procedures for computing $\textsc{Aff}_x$ and $\textsc{Aff}_y$ that have to be performed (in any order) to complete the first phase.

In more detail, the procedure for computing $\mathtt{A}(y)$ is pretty similar to Algorithm 1. The main differences, besides the trivial one of being a BFS rooted at y that considers $\mathtt{A}(y)$ instead of $\mathtt{A}(x)$ (see lines 1, 9, and 13), are the following: (1) in lines 5 and 6, we consider y instead of x; (2) in line 11, we let H be the set of hubs of pair (x, u) in L; and (3) in line 13, we replace the original test with the following one: $h \in \mathtt{A}(y) \vee ((h = u \vee h = x) \wedge d_{i-1}(x, u) = d_{i-1}(y, u) + 1)$—i.e., we check whether either h belongs to $\mathtt{A}(y)$ or $h \in \{x, u\}$ and the shortest path *from x toward u* in G_{i-1} was possibly using edge (x, y). The procedure for the computation of $\mathtt{A}(x)$ in directed graphs, instead, is slightly more involved. The main difference with respect to Algorithm 1 is that it performs a BFS-like visit rooted at x in the transpose graph $\overline{G}$; i.e., at line 10, edges (u, v) incoming into v have to be considered, rather than outgoing ones. In addition, in line 11, we let H be the set of hubs of pair (u, y) in L. Finally, in line 13, we replace the original test with the following one: $h \in \mathtt{A}(x) \vee ((h = u \vee h = y) \wedge d_{i-1}(u, y) = d_{i-1}(u, x) + 1)$; i.e., we check whether either h belongs to $\mathtt{A}(x)$ or $h \in \{u, y\}$ and the shortest path *from u toward y* in G_{i-1} was possibly using edge (x, y). By elaborating on the proof of Lemma 3.5, it is easy to show that the above-described modifications of Algorithm 1 correctly compute the sets $\textsc{Aff}_x$ and $\textsc{Aff}_y$, respectively; i.e., at the end of their execution, we have $\mathtt{A}(x) = \textsc{Aff}_x$ and $\mathtt{A}(y) = \textsc{Aff}_y$, respectively.

Regarding the second phase of DecPLL, i.e., the removal of outdated labels, we can derive a corresponding routine for directed graphs as well. In detail, given the two sets of affected nodes defined as above, the routine is a slight modification of that shown in Algorithm 2 that suitably considers L_{in} and L_{out} when scanning affected nodes, as follows: for all affected nodes u and v that satisfy the condition of Lemma 4.1 (i.e., for each $(v, u) : (v \in \mathtt{A}(x) \wedge u \in \mathtt{A}(y))$), it removes (u, δ_{uv}) from $L_{out}(v)$ if $u \in L_{out}(v)$ and removes (v, δ_{uv}) from $L_{in}(u)$ if $u \in L_{out}(v)$. By elaborating on the proof of Lemma 3.7, it is easy to prove that at the end of the above modification of Algorithm 2, L does not contain any pair that satisfies the condition of Lemma 4.1.

Finally, regarding the third phase of DecPLL, also the two strategies given in Section 3 (see Algorithms 3 and 4) for restoring the cover property can be modified in order to be used in directed graphs, as we summarize in the following.

In particular, Algorithm 3 can be adapted to be used in directed graphs by considering the following changes. First of all, the selection of the largest set, among $\mathtt{A}(x)$ and $\mathtt{A}(y)$, induces also an orientation of the BFS visits. In particular, if $|\mathtt{A}(x)| < |\mathtt{A}(y)|$, i.e., $\mathtt{SA} = \mathtt{A}(x)$, then the visits are performed on G; otherwise, $\overline{G}$ is used (i.e., in line 19 only edges in the form (u, v), incoming into v, are considered). In the first case, in line 16, we test whether $\mathtt{dist}[v] < \textsc{Query}(a, v, L)$, while in the second case, we check whether $\mathtt{dist}[v] < \textsc{Query}(v, a, L)$. Accordingly, if the test succeeds, i.e., we are discovering a path shorter than those induced currently by the labeling, and $v \in \mathtt{LA}$, then (1) in the case when $\mathtt{SA} = \mathtt{A}(x)$, we add $(v, \mathtt{dist}[v])$ to $L_{out}(a)$ ($(a, \mathtt{dist}[v])$ to $L_{in}(v)$, respectively) if $v < a$ ($v \geq a$, respectively), and (2) in the case when $\mathtt{SA} = \mathtt{A}(y)$, we add $(v, \mathtt{dist}[v])$ to $L_{in}(a)$ ($(a, \mathtt{dist}[v])$ in $L_{out}(v)$, respectively) if $v < a$ ($v \geq a$, respectively).

Similarly, Algorithm 4 also can be modified to be suitable for directed graphs. In this adaptation, the orientation of the visits is selected after line 3 where, if $a \in \mathtt{A}(x)$, then $\overline{G}$ is used, while if $a \in \mathtt{A}(y)$, then G is considered. The main differences induced by the orientation are essentially two: (1) In line 15, where if $a \in \mathtt{A}(x)$, then only edges (u, v) incoming into v have to be considered, while otherwise only edges (v, u) outgoing v are scanned. Note that the information about a being in $\mathtt{A}(x)$ or in $\mathtt{A}(y)$ can be easily stored in an auxiliary vector, to be accessed in constant time. (2) After the extraction of line 11, if $a \in \mathtt{A}(x)$, then we need to test whether the discovered distance is smaller than that encoded in the labeling from a to v, i.e., to test whether $\mathtt{dist}[v] <$

$\textsc{Query}(a, v, L)$, while if $a \in \mathtt{A}(y)$, we have to verify whether $\mathtt{dist}[v] < \textsc{Query}(v, a, L)$. In both cases, if the test succeeds, i.e., the labeling is inducing a longer shortest path as compared to the one being considered, then if $a \in \mathtt{A}(x) \wedge v \in \mathtt{A}(y)$, we add $(a, \mathtt{dist}[v])$ to $L_{in}(v)$, while if $a \in \mathtt{A}(y) \wedge v \in \mathtt{A}(x)$, we insert $(a, \mathtt{dist}[v])$ into $L_{out}(v)$, as v at this point is larger than or equal to a according to the node ordering, given the test of line 12.

By slightly modifying the proofs of Theorems 3.9 and 3.10, it is possible to show that at the end of the two above-described adaptations of Algorithms 3 and 4, respectively, L is a minimal 2-hop Cover labeling that covers directed graph G_i and satisfies the well-ordered property.

Moreover, note that the complexity of the two above approaches for directed graphs is exactly the same as that stated in Theorems 3.12 and 3.15, respectively.

As a final observation, it is worth noticing that, also for directed graphs, an alternative way of computing the two sets $\mathtt{A}(x)$ and $\mathtt{A}(y)$ that avoids the explicit computation of H can be defined for reducing, at least from the practical viewpoint, the required computational effort.

All results discussed in Section 3.4 can be extended to directed graphs by considering orientation of shortest paths and of queries on L. In particular, we can define alternative sets $\overline{\textsc{Aff}_x}$ and $\overline{\textsc{Aff}_y}$ as done for undirected graphs and exploit their relationship with $\textsc{Aff}_x$ and $\textsc{Aff}_y$ to reduce the computational effort required by the algorithm. Similarly to procedures for computing sets $\textsc{Aff}_x$ and $\textsc{Aff}_y$, also for sets $\overline{\textsc{Aff}_x}$ and $\overline{\textsc{Aff}_y}$ we need two separate (asymmetrical) routines to handle orientation of paths, distances, and queries.

In more detail, we can define two procedures, by modifying Algorithm 5, for computing the two sets $\mathtt{A}(x)$ and $\mathtt{A}(y)$ that then can be shown to coincide with sets $\overline{\textsc{Aff}_x}$ and $\overline{\textsc{Aff}_y}$. Again, the procedure for computing $\mathtt{A}(y)$ is pretty similar to the one given in Algorithm 5. The main differences, besides the trivial one of being a BFS rooted at y that considers $\mathtt{A}(y)$ instead of $\mathtt{A}(x)$ (see lines 1, 9, and 16), are the following: (1) in lines 5 and 6, we consider y instead of x; (2) in line 11, we replace the original test with the following one: $d_i(x, u) \neq d_{i-1}(x, u)$—i.e., we check whether the distance from x to u has changed (clearly because of the removal of (x, y)); (3) in line 15, we let h be the smallest hub in the set of hubs of pair (x, u) in L; and (4) in line 16, we replace the original test with the following one: $h \in \mathtt{A}(y) \vee ((h = u \vee h = x) \wedge d_{i-1}(x, u) = d_{i-1}(y, u) + 1)$—i.e., we verify whether either h belongs to $\mathtt{A}(y)$ or $h \in \{x, u\}$ and the shortest path *from x toward u* in G_{i-1} was possibly using edge (x, y).

The adaptation of Algorithm 5 for the heuristically improved computation of $\mathtt{A}(x)$ in directed graphs, instead, is slightly more involved. The main difference with respect to the original routine, again, is that it performs a BFS-like visit rooted at x in the transpose graph $\overline{G}$; i.e., at line 10, edges (u, v) incoming into v have to be considered, rather than outgoing ones. In addition, (1) we replace the condition of the test in line 11 with $d_i(u, y) \neq d_{i-1}(u, y)$—i.e., we check whether the distance from u to y has changed (clearly because of the removal of (x, y)); (2) in line 15, we let h be the smallest hub in the set of hubs of pair (u, y) in L; and (3) in line 16, we replace the original test with the following one: $h \in \mathtt{A}(x) \vee ((h = u \vee h = y) \wedge d_{i-1}(u, y) = d_{i-1}(u, x) + 1)$—i.e., we verify whether either h belongs to $\mathtt{A}(x)$ or $h \in \{u, y\}$ and the shortest path *from u toward y* in G_{i-1} was possibly using edge (x, y).

By elaborating on the proof of Lemma 3.18, it is possible to show that the two adaptations described above are able to correctly compute the sets $\overline{\textsc{Aff}_x}$ and $\overline{\textsc{Aff}_y}$, respectively; i.e., at the end of their execution, we have $\mathtt{A}(x) = \overline{\textsc{Aff}_x}$ and $\mathtt{A}(y) = \overline{\textsc{Aff}_y}$, respectively.

4.2 Weighted Graphs

In a weighted undirected graph $G = (V, E, w)$, each edge $\{u, v\}$ has an associated real nonnegative weight, which is denoted by $w(u, v)$. In this case, we denote by $d(s, t)$ the weighted distance

between s and t, which corresponds to the sum of the weights of the edges belonging to a shortest path between s and t. More formally, given a shortest path $P = \{v_1 = s, v_2, \ldots, v_k = t\}$, between s and t, of length k, we denote its weight by $w(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$. Clearly, $w(P)$ equals $d(s,t)$ whenever P is a shortest path.

In weighted graphs, the notion of *modification* (or *update*) needs to be extended in order to include also changes in the weights, thus generalizing the concept presented in Section 2. Therefore, graph update operations can be of two types: (1) an *incremental update operation*, i.e. either an *insertion of a new edge* or a *decrease of the weight* of an existing edge, and (2) a *decremental update operation*, i.e., either a *removal* or an *increase of the weight* of an existing edge. Node insertions and removals can be modeled as modifications of multiple edges incident to the same node as well.

To adapt the 2-hop Cover labeling method to a weighted graph $G = (V, E, w)$, it is enough to consider slightly different label entries that store weighted distances instead of hop distances, i.e., number of edges within shortest paths. This essentially does not induce any change in the baseline framework since the following hold:

— As well as in the previous cases, labels can be used to answer a *query* on the distance between two nodes s and t as

$$\textsc{Query}(s,t,L) = \begin{cases} \min_{v \in V}\{\delta_{sv} + \delta_{vt} \mid v \in L(s) \wedge v \in L(t)\} & \text{If } L(s) \cap L(t) \neq \emptyset \\ \infty & \text{otherwise.} \end{cases}$$

— It can be shown (see Akiba et al. (2013)) that $\textsc{Query}(s,t,L) = d(s,t)$ in a weighted graph G if the label set is computed by using suitable modifications of either the *naive* method or PLL. In this regard, Dijkstra-like visits take the place of BFS searches, and priority queues, instead of simple FIFO queues, are used.

The main differences instead arise in the implementation phase. In fact, notice that unweighted graphs of our interest are small-world networks, and hence 8-bit integers are enough to represent hop distances. In weighted graphs, this assumption clearly fails, thus requiring storing a rather larger number of bits per value of distance (e.g., 64 bits; see Akiba et al. (2013)).

Similar considerations can be done for the dynamic algorithms discussed in the previous sections. In fact, as well as approaches for computing L from scratch, algorithms IncPLL and DecPLL can be extended to be used in weighted graphs by considering Dijkstra-like visits in place of BFS searches and by using priority queues instead of simple FIFO queues.

In more detail, regarding algorithm IncPLL, the adaptation is rather simple and can be summarized as follows. Given an incremental graph update operation occurring on an edge (a,b), IncPLL resumes Dijkstra-like executions, instead of BFS searches, originally rooted at nodes v_k if $v_k \in L(a) \cup L(b)$ by inserting to the initial queue pair $(b, d(v_k, a) + w(a,b))$, where $w(a,b)$ is the new weight of edge $\{a,b\}$.

Regarding algorithm DecPLL, the adaptation is slightly more technically involved. We highlight in the following the main changes with respect to each phase of the algorithm. In particular, for the computation of the affected nodes, in Algorithm 1, it is enough to change the test of line 13 to include the weight of edge $\{u,v\}$, i.e., to replace 1 with $w(u,v)$. The same holds for the alternative version, i.e., Algorithm 5 (see line 16) and for the adaptations for directed graphs. In particular, we need to replace 1 with $w(u,v)$ whenever an edge is scanned and 1 with $w(x,y)$ whenever we perform a test on the distance to check that it is increased by changes occurring in edge (x,y) (e.g., a test on $d_{i-1}(x,u) = d_{i-1}(y,u) + 1$ becomes a test on $d_{i-1}(x,u) = d_{i-1}(y,u) + w(x,y)$). Concerning the removal phase, essentially nothing changes. Finally, with respect to Greedy–Restore and Order–Restore, the following changes are required. In both algorithms, Q is a min-based priority queue, and thus any extract operation, like that of line 14 of Algorithm 3 is essentially an

extract–min operation, while adding operations, like that of line 18 of Algorithm 4, is essentially an insert operation with key $dist[u]$. Moreover, unlike BFS-like searches, shortest paths are not necessarily discovered in a frontier–expansion manner (i.e., a shortest path of weight k is found after all shortest paths of weight $k-1$ have been found). Therefore, a decrease-key step needs to be introduced. In particular, in every foreach loop where an edge (u, v) is scanned, like that of line 19 of Algorithm 3, if the node is already marked, then we need to check whether $dist[v] + w(u, v)$ is smaller than the current key of v and, in the affirmative case, to perform a corresponding decrease-key operation on Q. Note that proofs of correctness and complexity analyses of the above routines can be easily derived by those proposed in Sections 3 and 4.1.

5 THE FULLY DYNAMIC ALGORITHM

In this section, we briefly describe how it is possible to suitably combine IncPLL and DecPLL to obtain a fully dynamic algorithm, named from now on as FulPLL, which is able to deal with both incremental and decremental updates.

The strategy underlying FulPLL can be briefly summarized as follows. Whenever a change in the graph is detected, the algorithm invokes, according to the type of change (either decremental or incremental), the corresponding routine to handle it, namely, either DecPLL (in case of decremental updates) or IncPLL (in case of incremental updates). Notice that the worst-case computational complexity of FulPLL is upper bounded by the maximum of the worst-case complexities of its two dynamic components, i.e., IncPLL and DecPLL. In particular, for unweighted graphs, FulPLL requires $O(\max\{nm + n^2, m_{\mathsf{LA}}\Lambda \log |\mathsf{LA}| + |\mathsf{SA}|(m + n \log |\mathsf{LA}| + n\Lambda)\})$ computational time in the worst case (see Sections 2.3 and 3 for more details). In the case of weighted graphs, both terms increase by a logarithmic factor due to the use of Dijkstra's algorithm instead of BFS.

6 EXPERIMENTAL EVALUATION

In this section, we present the experimental study we conducted to assess and compare the performance of all presented approaches.

6.1 Implemented Algorithms and Setup

Our experimental framework is organized as follows. We first implemented the basic version for undirected graphs of all considered algorithms for 2-hop Cover labelings. In more detail, we implemented

- —the static algorithm PLL for the from-scratch computation of 2-hop Cover labelings,
- —the incremental algorithm IncPLL,
- —the decremental algorithm DecPLL, and
- —the fully dynamic algorithm FulPLL.

In order to implement all the above algorithms, we took inspiration from the existing implementation of IncPLL and PLL for undirected graphs, freely available on the website of one of the authors of Akiba et al. (2014).[3] In more detail, we directly incorporated in our experimental framework the code of both PLL and IncPLL for undirected unweighted graphs and a basic version of the algorithm for querying the labeling. Then, we implemented ex novo DecPLL, FulPLL, a more finely tuned version of the query algorithm, and adaptations of all considered algorithms able to deal with both directed and weighted graphs, according to the descriptions given in Section 4.

Note that, in the case of weighted graphs, DecPLL (IncPLL, respectively) is modified in order to be able to manage any kind of decremental (incremental, respectively) update operation

[3]We thank the authors for kindly providing the code.

occurring on the graph, including weight increases (weight decreases, respectively). The same holds for FulPLL. Notice also that, as shown in Akiba et al. (2014), in case of unweighted (either undirected or directed) graphs, PLL can take advantage of multicore architectures. Therefore, we implemented its fastest (parallel) version, while all other algorithms are sequential.

Concerning the computation of the set of affected nodes, we incorporated within DecPLL both Algorithms 1 and 5. Preliminary experiments show that the latter one is always the fastest of the two. Therefore, in the following, we omit results regarding Algorithm 1.

Regarding the third phase of DecPLL, we implemented both Greedy–Restore and Order–Restore. We have conducted a set of preliminary experiments in order to determine, empirically, under which circumstances Order–Restore is faster than Greedy–Restore. Such tests show that the use of the former is advised as long as the size of the smaller set, between $\mathtt{A}(x)$ and $\mathtt{A}(y)$, does not exceed, roughly, $n/\log n$, while the use of the latter is more convenient otherwise. Therefore, we included in the implementation of DecPLL a test before the beginning of the third phase that chooses one of the two strategies according to the above condition.

As in Akiba et al. (2014), we chose to sort nodes in nonincreasing order of degree since, beyond its simplicity, this methodology has been shown to be robust, i.e., to deliver labelings that exhibit practical tradeoffs between preprocessing time, average query time, and space requirements, in a wide variety of networks (Delling et al. 2014a). For those cases where this is not true, such as rather regular ones (e.g., road networks or Erdős-Rényi graphs), we implemented an ordering based on the computation of (approximated) betweenness centrality of the nodes proposed in Geisberger et al. (2008). We conducted some preliminary experiments that suggest that such an approach delivers better-quality labelings on such instances, with respect to preprocessing time, average query time, and space requirements.

All our software has been developed in C++ within NetworKit, an open-source toolkit for large-scale network analysis,[4] compiled with GNU g++ v. 5.5 (03 optimization level) under Linux (Kernel 4.4.0-47). All our tests have been executed on a workstation equipped with an Intel Xeon© CPU, with 128GB of main memory.

6.2 Input Graphs

As input, we used various network instances of both real-world and synthetic-generated type. In particular, regarding the former, we considered datasets similar to those used in other studies on static and dynamic 2-hop Cover-based methods (Akiba et al. 2013, 2014; Delling et al. 2014a; Hayashi et al. 2016), which are freely available on known repositories for this kind of data, such as SNAP, PTV, and Konect. Regarding the latter, instead, as done in other experimental studies of this kind (Bauer and Wagner 2009; D'Andrea et al. 2015), we selected established graph generation models that have been shown to produce instances of practical interest, like the Barabási-Albert model (Albert and Barabási 2002).

For both real-world and synthetic networks, in order to test all versions of the proposed algorithms, we either considered or generated graphs belonging to each of the following categories: (1) unweighted undirected graphs, (2) weighted undirected graphs, (3) unweighted directed graphs, and (4) weighted directed graphs. Graph details of all considered networks are reported in Table 1, while a brief description of each dataset is given in the following:

— Caida (cai): weighted undirected graphs corresponding to snapshots of mutually connected autonomous systems of the Internet, collected by the CAIDA project (Hyun et al. 2014), where nodes are autonomous systems, and edges denote communication links; weights

[4] networkit.iti.kit.edu.

Table 1. Overview of Input Graphs

Dataset	Network	|V|	|E|	AvgDeg	S	D	W
Caida (cai)	Communication	32,000	40,204	2.51	○	○	●
Gnutella (gnu)	p2p	36,682	88,328	4.82	○	●	○
Brightkite (bkt)	Location-based	58,228	214,078	7.35	○	○	○
Eu-All (eua)	email	265,214	365,570	2.77	○	●	○
SimpWiki-En (swe)	Hyper-link	100,312	826,491	16.5	○	○	○
Twitter (twi)	social	465,017	834,797	3.59	○	●	○
Epinions (epn)	Social	131,828	841,372	12.76	○	●	○
Barabási-A. (baa)	Synthetic	631,912	1,000,772	3.17	●	○	●
Netherlands (nld)	Road	892,027	2,278,290	5.11	○	●	●
FlickrImg (fli)	Meta-data	105,938	2,316,948	43.74	○	○	○
YouTube (ytb)	Social	1,134,890	2,987,624	5.26	○	○	○
Google (goo)	Web	875,713	4,322,051	9.87	○	●	○
WikiTalk (wtk)	Communication	2,394,385	4,659,565	4.19	○	●	○
Erdős-Rényi (erd)	Synthetic	50,000	6,252,811	250.11	●	○	●
BerkStan (wbs)	Web	685,230	7,600,595	22.18	○	●	○
AS-Skitter (ski)	Computer	1,696,415	11,095,298	13.08	○	○	○
DBPedia (dbp)	Miscellaneous	3,966,924	13,820,853	6.97	○	●	○
ForestFire-U (ffu)	Synthetic	2,000,000	14,908,267	14.91	●	○	○
FlickrLinks (fll)	Social	1,715,255	15,550,782	18.13	○	○	○
ForestFire-D (ffd)	Synthetic	2,100,000	16,044,834	15.28	●	●	○
Wiki-It (itw)	Hyper-link	1,203,995	21,639,725	36.9	○	●	○

The first column reports the name of the dataset while the second column shows the type of considered network. The third and fourth columns show number of nodes and edges of the corresponding graph, while the fifth column reports the average node degree. The sixth column (**S**) indicates whether the graph is synthetic or obtained by parsing real-world datasets (● = synthetic, ○ = real world). Finally, the seventh and eighth columns (**D** and **W**, respectively) show whether the considered graph is directed (● = directed, ○ = undirected) and/or weighted (● = weighted, ○ = unweighted), respectively. Graphs are ordered according to their number of edges.

are assigned to the edges according to the round-trip time associated to the corresponding link (D'Angelo et al. 2014b, 2011)

— Gnutella (gnu): a snapshot of the Gnutella peer-to-peer (p2p) file-sharing network where nodes represent hosts and edges represent connections between them;

— Brightkite (bkt): a location-based social network where users share their locations by checking in; nodes are users and edges represent friendships between users

— Eu-All (eua): the email communication network of a large, undisclosed European institution, where nodes represent individuals and there is an edge between two people if at least one email has been sent from one person to the other,

— SimpWiki-En (swe): the network of hyperlinks between articles of the Simple English Wikipedia where nodes represent articles, while an edge indicates that there is an hyperlink between them

— Twitter (twi): a network containing information about who follows whom on Twitter, where nodes represent users and an edge shows that a user follows another one

— Epinions (epn): the who-trusts-whom online social network of a general consumer review site[5] where nodes are users and edges represent trust links between users (Akiba et al. 2014)

[5] www.epinions.com.

—Barabási-A. (baa): random scale-free weighted undirected graphs generated by the well-known Barabási-Albert algorithm, which incorporates a preferential attachment mechanism; scale-free networks are widely observed in natural and human-made systems, including the Internet, the World Wide Web, citation networks, and some social networks; weights are assigned to edges uniformly at random within a predefined range (e.g., [100, 100, 000] (D'Angelo et al. 2014b, 2011))
—Netherlands (nld): the road networks of the Netherlands, where nodes are points on a road map and directed edges are road segments connecting them; weights are assigned to the edges according representations of an estimate of the travel time needed to traverse road segments
—FlickrImg (fli): the network of Flickr images sharing common metadata such as tags, groups, and locations; a node represents an image, and an edge indicates that two images share the same metadata
—YouTube (ytb): a snapshot of the Youtube social network, where nodes are users that form friendships (edges) with each other (Akiba et al. 2014)
—Google (goo): a snapshot of the network induced by the web pages indexed by Google, released in 2002 by Google as a part of the Google Programming Contest, where nodes represent web pages and edges represent hyperlinks between them
—WikiTalk (wtk): the network contains all the users and the discussions from the inception of Wikipedia till January 2008; nodes represent Wikipedia users and a directed edge from node i to node j represents that user i at least once edited a talk page of user j
—Erdős-Rényi (erd): synthetic undirected weighted graphs generated via the Erdős-Rényi model; given a fixed probability p, an Erdős-Rényi graph is constructed by connecting nodes randomly with probability p and by assigning them a weight uniformly chosen at random within a predefined range (e.g., [100, 100, 000] (D'Angelo et al. 2014b))
—BerkStan (wbs): the Berkeley-Stanford web network; nodes represent pages from berkely.edu and stanford.edu domains and directed edges represent hyperlinks between them
—AS-Skitter (ski): an Internet topology network, obtained via traceroute, where nodes represent routers and edges represent communication links between them
—DBPedia (dbp): the complete DBpedia network (v. 3.6), where nodes are structured contents included in DBpedia and each edge represents the existence of a semantic relationship between two contents
—ForestFire (ffu and ffd): synthetic instances generated via the ForestFire model-based network generator, which is available on SNAP (Akiba et al. 2014); the model produces directed graphs only; undirected ones can be obtained by removing orientations from edges
—FlickrLinks (fll): the social network of Flickr users and their connections; the network is undirected
—Wiki-It (itw): the network of hyperlinks between articles of the Italian Wikipedia

6.3 Executed Tests

We conducted a wide set of experiments, which can be logically divided in two categories: *synthetic updates* and *real-world updates*.

6.3.1 Synthetic Updates. In the synthetic case, for each graph G, we first construct a labeling $L(G)$ by PLL. Then, we generate and perform three different types of sequences of updates (a.k.a. *workloads*), called inc, dec, and ful, respectively. In the inc (dec, respectively) case, we select and perform k randomly chosen incremental (decremental, respectively) graph update operations. Each

incremental (decremental, respectively) operation consists of inserting an edge between two nodes that are not connected (removing an existing edge, respectively). In the case of weighted graphs only, an incremental (decremental, respectively) graph update operation can be also a decrease (an increase, respectively) of the weight of an existing edge. The weight variation is randomly chosen and ranges in [1%, 99%] of the original weight of the edge. Notice that, for insertions in weighted graphs, the newly created edge is assigned a weight that is randomly chosen among the weights of edges that are already present in the graph.

By applying the ith operation on G_i, we obtain G_{i+1}. Then, we proceed as follows: we first execute a corresponding dynamic algorithm, i.e., IncPLL (DecPLL, respectively), to reflect the change on $L(G_i)$, thus computing $L(G_{i+1})$. In parallel, we execute PLL directly on G_{i+1} to obtain $L(G_{i+1})$ from scratch.

In the FUL workload case, we select and perform k operations of mixed randomly chosen type; i.e., for each operation we either remove or add an edge (the choice is done uniformly, by flipping a coin). Then, we run FulPLL and compare it with PLL, as above.

In all the above settings, k is chosen in $\{50, 100, 10,000\}$ depending on the size of the considered input graph. In particular, k is set to the lowest value for those networks of very large size, such as, AS-Skitter, since in this case PLL can require hours per operation,[6] while it is set to the highest value for networks where PLL takes an order of a few seconds, e.g., Caida or Eu-All. An intermediate value of k is considered for networks whose size is something in between, like Epinions or Gnutella.

In this regard, for the sake of completeness, we have also performed an additional set of experiments, called *scalability oriented experiments*, where we consider $k = \{1{,}000, 10{,}000\}$ and execute the dynamic algorithms only, without executing PLL. This is done in order to evaluate how the dynamic algorithms scale against the number of graph updates, in particular for those large networks for which we were forced to choose $k = 50$ in both the above-mentioned DEC and FUL workloads (due to the time-consuming PLL). For these latter instances, we considered $k = 1{,}000$, while we selected $k = 10{,}000$ for all others.

6.3.2 Real-World Updates. In the real-world case, we used two dynamic datasets available on Konect, namely, SimpWiki-En and Wiki-It, representing real unweighted networks evolving over time (an undirected and a directed one, respectively). In this scenario, the FUL workload consists of a realistic sequence of updates, i.e., insertions and removals. Note that, in such datasets, the number of insertions is around 75% of the total number of updates within the dataset (networks tend to grow).

To our knowledge, real-world *weighted* networks with real weight changes are not available to the research community. Thus, we leave open the evaluation of such a setting.

6.4 Performance Indicators

As primary performance metrics, after each graph update, we measure (1) the computational time (CT), in seconds, spent by the considered dynamic algorithm (i.e., either IncPLL or DecPLL or FulPLL) for updating the labeling, and (2) the computational time (CT), in seconds, spent by PLL for building the labeling from scratch. Then, for each graph update, we compute the *speedup*, i.e., the ratio between the latter and the former, which clearly is a proxy for how much faster a dynamic algorithm is as compared to PLL. As further performance metrics, we considered, as in Akiba et al. (2014), the space occupancy required for storing the labeling and the average time needed

[6] It is worth remarking that the preprocessing time of all considered dynamic algorithms is upper bounded by the running time of PLL, which requires $O(nm + n^2)$ ($O(n(m + n \log n))$, respectively) time in the worst case for unweighted (weighted, respectively) graphs.

Table 2. Experimental Results: DEC Workload

Dataset	DEC workload						O
	CT (s)		LS (MB)		QT (μs)		
	PLL	DecPLL	PLL	DecPLL	PLL	DecPLL	
CAI	1.1	0.497	24	23	39.5	40.1	D
GNU	102	21.1	322	322	62.7	61.3	D
BKT	98.7	0.328	81	81	23.1	25.7	D
EUA	19.8	0.073	217	217	7.2	7.5	D
TWI	25.4	0.018	390	390	7.3	7.3	D
EPN	71.8	0.630	372	372	13.5	14.6	D
BAA	143	48.4	954	954	41.8	48.1	B
NLD	1,280	371	7,410	7,553	63.9	70.9	B
FLL	17,900	9.92	12,970	12,970	77.8	102.0	D
YTB	2,720	104.0	2,899	2,899	43.9	60.4	D
GOO	3,950	4.27	3,862	3,862	39.9	57.2	D
WTK	3,920	5.15	5,035	5,035	35.2	37.9	D
ERD	2,530	4.37	881	879	123	119	B
WBS	2,510	0.639	1,659	1,659	31.4	28.9	D
SKI	15,800	17.2	11,826	11,826	70.8	110.0	D
DBP	20,600	2.61	14,877	14,877	45.9	48.7	D
FFU	35,300	14.3	23,556	23,556	110	153	D
FLI	1,770	48.4	836	836	81.5	82.4	D
FFD	29,200	18.7	16,499	16,499	57.3	90.8	D

for answering a query, since these two parameters are universally considered measures of the practical effectiveness of the labeling. In particular, after each graph update, for each considered labeling L (either computed via PLL or by dynamic algorithms), we measure (1) the *labeling size* (LS), i.e., the space required to store L, in bytes, on average, and (2) the *average query time* (QT), i.e., the time needed to answer a distance query, in microseconds, on average over 10,000 random queries. For the sake of correctness, for each query we also verify that it returns the same distance in the labeling computed from scratch by PLL and in that updated by the dynamic algorithm.

The results of all conducted experiments are summarized in Tables 2 and 3, where we report, for each network, average values, over a set of k graph update operations, for all mentioned performance indicators, except the speedup, for which we give a more self-explanatory graphical representation later on in the section.

In particular, in such tables, the first column shows the name of the dataset, while columns 2, 4, and 6 show performance indicators of PLL (i.e., its average CT, LS, and QT, respectively). Columns 3, 5, and 7, instead, report the performance indicators of DecPLL (Table 2) and FulPLL (Table 3), i.e., its CT, LS, and QT, respectively. Finally, the eighth column reports which type of node ordering produced the best results in terms of CT for PLL (D = ordering on degree, B = ordering on approximated betweenness centrality). Note that this is done for the sake of fairness, since in this way we compare DecPLL and FulPLL with the best possible executions of PLL.

Regarding the speedup, an overview of our results is given in Figures 3 and 4, where we report the distribution of the speedup of DecPLL and FulPLL, respectively, via box-plot charts. We highlight, for each set of the obtained speedups, minimum, first quartile, median, third quartile, and maximum values. We provide the results for a meaningful subset of the considered networks. Other datasets produce similar outcomes, and hence we omit them. As a final remark, note that

Table 3. Experimental Results: FUL Workload

Dataset	FUL workload						O
	CT (s)		LS (MB)		QT (μs)		
	PLL	FulPLL	PLL	FulPLL	PLL	FulPLL	
CAI	1.21	0.331	24	25	39.9	41.5	D
GNU	113	7.2	322	322	62.7	61.4	D
BKT	95.3	0.217	81	81	22.7	26.4	D
EUA	19.6	0.032	217	217	7.2	7.0	D
SWE	78	0.232	181	180	47.4	49.2	D
TWI	25.6	0.007	390	390	7.3	7.3	D
EPN	71.8	0.121	372	372	13.5	13.6	D
BAA	141	6.97	954	954	41.1	45.9	B
NLD	1,350	350	7,057	6,990	60.5	54.9	B
FLI	1,740	33.9	836	836	80.1	81.0	D
YTB	2,650	79.3	2,899	2,899	40.6	55.6	D
GOO	3,950	0.566	3,862	3,872	39.4	52.3	D
WTK	3,920	2.03	5,035	5,035	34.0	49.9	D
ERD	2,520	2.42	882	880	122	119	B
WBS	2,480	0.176	1,659	1,659	30.9	27.3	D
SKI	17,000	3.95	11,826	11,826	72.8	120.1	D
DBP	20,700	0.583	14,877	14,877	43.8	57.9	D
FFU	35,300	10.1	23,555	23,555	112	143	D
FLL	17,300	7.29	12,970	12,970	75.8	125.0	D
FFD	25,500	9.46	16,499	16,499	53.7	64.6	D
ITW	17,400	16.8	11,253	11,253	54.5	80.9	D

Note that, for SWE and ITW networks, the workload consists of a realistic sequence of updates.

we omit the results of the INC workload case since they are aligned with those presented in Akiba et al. (2014).

6.5 Analysis

The main outcome of our experimental evaluation is that the new proposed approach for handling decremental updates, i.e., DecPLL, is very effective in practice. In fact, despite the worst-case bound on its computational complexity (see Section 3), we observe (see Table 2) that the CT taken by DecPLL for updating a labeling, as a consequence of a graph change, on average, is always by far smaller than that required by PLL, even several orders of magnitude in many cases (see, e.g., networks SKI, FLL, or DBP in Table 2, just to mention a few examples).

Another interesting conclusion that can be drawn from our data is that FulPLL, obtained by combining DecPLL and IncPLL, is also very effective in practice. In fact, we observe (see Table 3) that the CT taken by FulPLL for updating a labeling, as a consequence of a graph change, on average, is even smaller than that required by DecPLL (and of course transitively by far smaller than that taken by PLL). The orders of magnitude of difference, in this case, are even more (see, e.g., networks TWI, GOO, or EUA in Table 3). This is surely due to the fact that FulPLL manages incremental updates by the very fast IncPLL (see Akiba et al. (2014)), whose performance is essentially better than that of DecPLL mainly because of the inherent nature of the incremental problem, which is universally considered easier than the decremental one. This opinion has been confirmed by several theoretical and experimental studies on dynamic algorithms (e.g., Akiba et al. (2014), D'Angelo

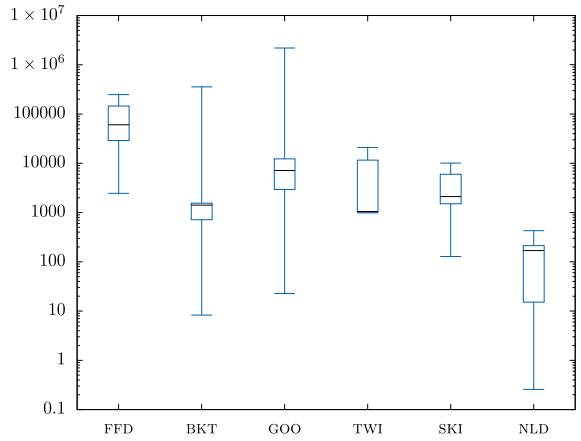

Fig. 3. Speedup distribution of DecPLL against PLL: box-plot overview.

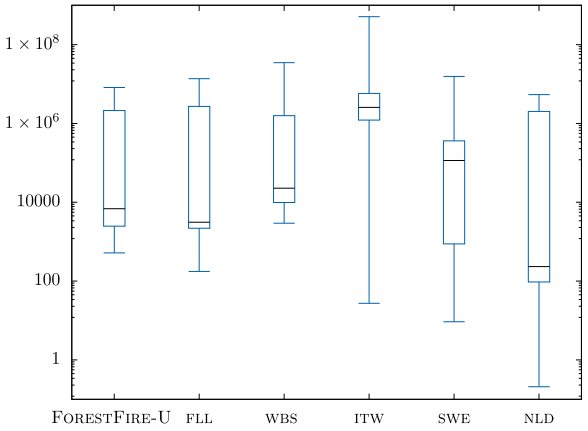

Fig. 4. Speedup distribution of FulPLL against PLL: box-plot overview.

et al. (2012), and Georgiadis et al. (2012)). The above considerations are strongly supported by the data we provide in Figures 5 and 6, where we show the distribution of the running times of DecPLL and FulPLL only, obtained in the scalability-oriented experiments described above (k = 1,000). In particular, on the x-axis we report input networks, while on the log-scaled y-axis we report the running time of DecPLL and FulPLL, respectively, in the form of jittered scatterplots (where a small random offset is added to the x component to avoid overplotting). We show data for most of the considered datasets, while some networks exhibiting similar behavior are omitted from the plots for readability reasons. These results, combined with the data provided in Tables 2 and 3, clearly represent strong evidence of the effectiveness of the two dynamic approaches.

In particular, we can observe that the median value of their computational time is always by far below that of PLL (shown by the blue line). Moreover, the distribution of the running times is symmetric, with a quite low dispersion around the median and very few outliers (see, e.g., DBP) that are probably due to some pathological topological scenario. As a side remark, it is worth noticing that most of the data reported in Figure 6 exhibit a bimodal behavior, which is quite expected, due to IncPLL being faster than DecPLL. Few exceptions can be observed (e.g., instances FFU or FLL) where data show a third mode. These exceptions might be due to decremental updates that cause

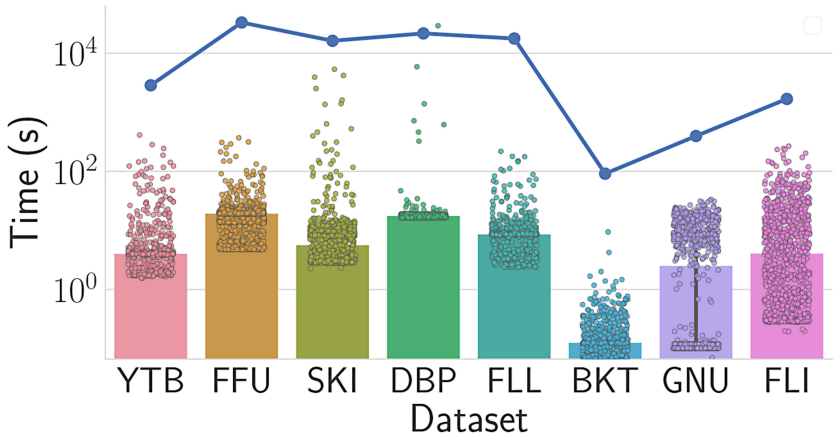

Fig. 5. Distribution of the running times of DecPLL: the height of the bar from the bottom highlights median values. The blue line shows the running time of PLL for computing the labeling from scratch at the beginning of the experiment.

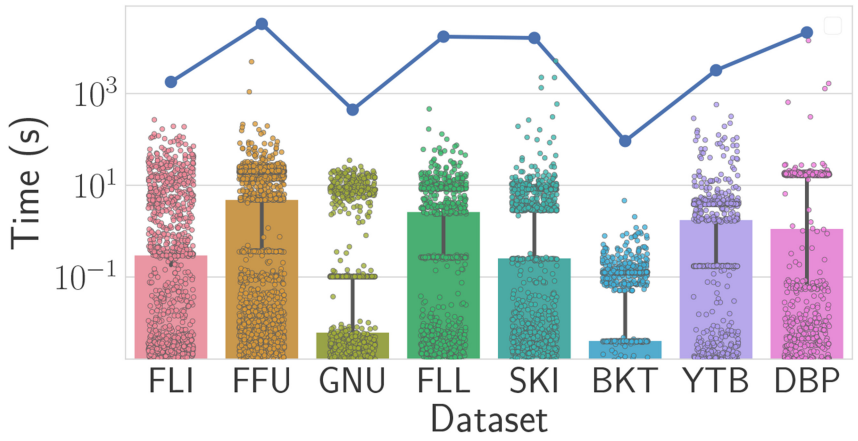

Fig. 6. Distribution of the running times of FulPLL: the height of the bar from the bottom highlights median values. The blue line shows the running time of PLL for computing the labeling from scratch at the beginning of the experiment.

a component of the graph to be disconnected. In such cases, in fact, DecPLL does not perform the third phase and can be marginally slower than IncPLL.

The practical effectiveness of the dynamic algorithms introduced in this article is even clearer if we focus on Figures 3 to 4 and 7 to 8. For the sake of clarity, notice that the latter refer to scalability-oriented experiments. We can observe that, in the reported instances, DecPLL and FulPLL are always faster than PLL (speedups are always above 1). The only exceptions to this good performance can be noticed in the case of instance NLD. This is probably due to the fact that, in directed, sparse, weighted graphs, topological updates induce heavy changes in the structure of the labeling, and hence the computational effort required by DecPLL and FulPLL increases up to the point of being slower than PLL. Nevertheless, pathological cases inducing the worst-case bound of Section 3 can be assumed to happen very rarely in practice. The above considerations are also supported by the average values of CT shown in Tables 2 and 3. Furthermore, the median value of the speedup looks

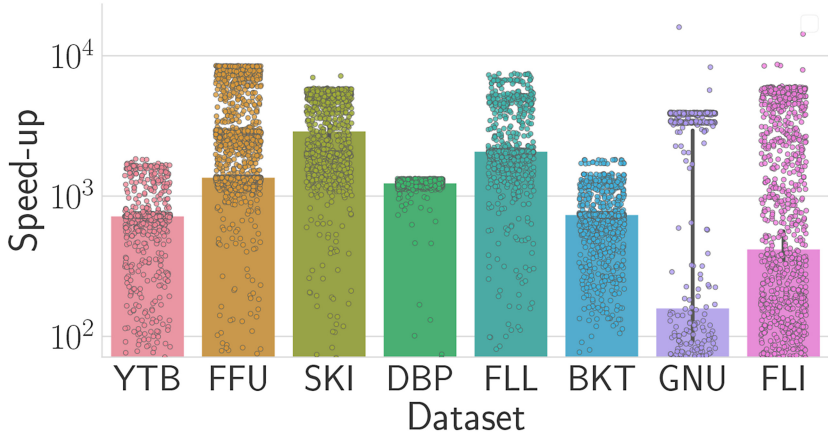

Fig. 7. Speedup distribution of DecPLL against PLL: full overview.

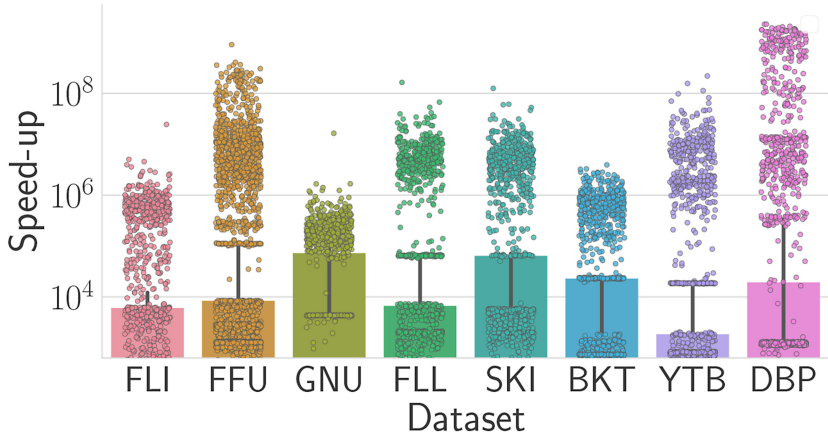

Fig. 8. Speedup distribution of FulPLL against PLL: full overview.

rather high, often larger than 10^4, and there are cases in which FulPLL (DecPLL, respectively) is more than 10^8 (10^5, respectively) times faster than PLL.

Regarding the space occupancy performance indicator, our results highlight that the use of dynamic algorithms does not induce an increase in labeling size. This confirms the outcome of our theoretical analysis of Section 3 about DecPLL, and that of the experimental evaluation of IncPLL proposed in Akiba et al. (2014), where the authors show that, even giving up on the minimality property, the algorithm behaves well in practice with respect to space occupancy. Concerning the average query time, our experiments confirm that dynamic algorithms deliver labelings that exhibit an average query time comparable to that of labelings computed from scratch. This is in agreement with the considerations done with respect to the average labeling size, which is directly related to query time (Delling et al. 2014a). Moreover, we observe that labelings of rather differently sized networks have similar average query times (e.g., GNU vs. SKI or BAA vs. DBP in Tables 2 and 3). This supports the claim that the average query time is weakly correlated to the network size and rather more dependent on density and node ordering, as shown in other works on 2-hop Cover labelings, e.g., Akiba et al. (2014) and Delling et al. (2014a).

In summary, DecPLL can be considered as the first algorithm for updating 2-hop Cover labelings that is able to efficiently handle decremental updates. In fact, it reflects graph changes on the labeling extremely faster than the fastest available approach, i.e., PLL, and at the same time efficiently answers queries without increasing the labeling size over time. This is clearly a highly desirable behavior in real-world dynamic scenarios. In fact, relying on PLL would imply having wrong answers to distance queries for large periods of time.

Regarding FulPLL, its performance with respect to update time is even better than DecPLL, since it manages insertions by the very fast IncPLL. This is even clearer if we focus on Figures 5 and 6. Moreover, our data show that this is achieved without any degradation with respect to labeling size and query time. In fact, even though IncPLL does not remove outdated labels, and then if used alone it might require the periodic execution of PLL to restore minimality and avoid query time increases, in the FUL experiments we can observe that the average labeling size and query time are always comparable to those of both DecPLL and PLL. This is due to the very effective Algorithm 2, which deletes outdated label entries that are not removed by IncPLL. For the above reasons, FulPLL can be considered the first fully dynamic algorithm for updating 2-hop Cover labelings, able to deal with both incremental and decremental updates in all kinds of graphs, and to answer distance queries without any compromises on performance, thus constituting a step forward in the literature on the matter.

6.6 On the Definition of Affected Nodes

For the sake of completeness, in this section we discuss definitions of affected nodes simpler than those given in Sections 3 and 3.4. In particular, it is worth remarking that more intuitive definitions of affected nodes are clearly possible for DecPLL. For instance, a rather standard technique in dynamic algorithms for shortest paths that we have considered during a preliminary phase of this work consists of selecting, as "affected" by a decremental operation on an edge, say, (x, y), all those nodes u, v such that a shortest path from u to v passes through edge (x, y).

By adopting this definition, the correctness of our algorithms would still be preserved. In particular, we could divide said pairs of nodes according to the structure of shortest paths into two sets, namely, Simpl_x (node $u \in \text{Simpl}_x$ if $d(u, y) = d(u, x) + 1$) and Simpl_y (node $v \in \text{Simpl}_y$ if $d(v, x) = d(v, y) + 1$), and use them in place of Aff_x and $\overline{\text{Aff}_x}$, respectively (or in place of Aff_y and $\overline{\text{Aff}_y}$, respectively).

The correctness of our algorithms would still be preserved, since Simpl_x and Simpl_y are trivially super-sets of sets of affected nodes obtained by Algorithms 1 and 5 (and by the corresponding symmetric versions rooted at y). This would result in a less computationally intensive phase for the detection of affected nodes, since it would be enough to run a visit and to test that $d(u, y) = d(u, x) + 1$ ($d(v, x) = d(v, y) + 1$, respectively) or not for any node able to reach y (x, respectively) in the graph before the change, instead of also having to query the labeling, as done by Algorithms 1 and 5.

However, this choice is not advisable for several reasons. First and foremost, adopting the simplified definition of affected nodes in our algorithm induces the removal of correct label entries from the labeling, and hence the need for employing computational effort to recompute and store them again from scratch. This is trivially an undesirable behavior. Also, the above-mentioned strategy would be counterintuitive with respect to the spirit of dynamic algorithms, whose purpose, in general, is that of detecting exactly (or at least in the best possible way) the part of the data structure that has to be updated as a consequence of a modification to the input.

Second, the complexity of the two remaining phases of DecPLL heavily depends on the cardinalities of the sets of affected nodes. Therefore, having larger sets of affected nodes induces a computational overhead that is most likely not balanced by the time saved during the less

Table 4. Comparison of the Sizes of the Sets of Affected Nodes

Graph	SIMPL_x	AFF_x	$\overline{\text{AFF}_x}$	SIMPL_y	AFF_y	$\overline{\text{AFF}_y}$
GNU	1,738	626	593	1	1	1
BKT	611	1	1	24,683	1,825	486
EPN	5,825	94	73	19,856	1,087	997
YTB	7,605	1,322	38	787,317	272,355	28,202
FFU	178,716	115,450	3	2	1	1
FLL	1,142	1	1	8	1	1
FFD	3,231	230	50	3	3	3

time-consuming first phase. Both of these undesired effects would increase with graphs having many multiple shortest paths (and these are expected to be in large number in many networks considered in this study).

To confirm this consideration, we conducted a series of preliminary experiments, where we observed the following: (1) the majority of the time taken by DecPLL is spent on the second and third phases (more than 90% of the time in all cases), while the detection of affected nodes is already very effective, and (2) the use of the simplified definition of affected nodes induces, overall, an increase in running time of DecPLL since the time saved during the first phase is negligible, while the overhead on the two remaining phases is huge. In Table 4, we provide an overview of the differences in the cardinalities of the sets of affected nodes for some of the networks considered in this work.

7 CONCLUSION AND FUTURE WORK

In this article, we have tackled the problem of updating the 2-hop Cover labeling of a graph in a dynamic setting. In detail, we have introduced a new *decremental algorithm* able to update 2-hop Cover labelings under edge/node removals and edge weight increases. To the best of our knowledge, no algorithm of this kind was previously known. We have shown the new algorithm to be (1) correct—i.e., queries on the distance, on the updated labeling, are guaranteed to return exact values after each graph change; (2) efficient in terms of the number of nodes that change their distance toward some other node of the network, as a consequence of a decremental graph update; and (3) able to preserve the *minimality* of the labeling, which is a desirable property that has impact on both query times and space occupancy.

We have then combined the new decremental algorithm with the unique *incremental approach* known in the literature (Akiba et al. 2014), thus obtaining the first *fully dynamic* algorithm for updating 2-hop Cover labelings under general graph updates.

We have provided a large set of experimental evidences, involving both real and synthetic inputs, showing that the new approaches are very effective in practice, thus allowing one to employ the 2-hop Cover labeling approach in dynamic networks with practical performance. Prior to this work, this was not known to be possible.

There are several research directions that deserve further investigation. The first one is to adapt our new method to deal with *historical queries*, which have important applications in evolving networks analysis (Akiba et al. 2014). Another interesting direction is that of extending the proposed experimental study in order to (1) consider different and more robust orderings for the nodes of the network, e.g., SamPG (Delling et al. 2014a); (2) investigate the impact of compressing the labeling on the performance of the proposed algorithms (see Delling et al. (2014a)); and (3) find out how well the new approaches scale against input sizes and graph densities. Moreover, it would be interesting to investigate the possibility of devising new efficient algorithms able to (1) handle incremental

updates while at the same time guaranteeing the minimality of the labeling, thus avoiding the increase in the size of the labeling and hence of the query time, and (2) process *batches* of updates, i.e., sets of updates occurring on the network simultaneously, since this is a quite frequent graph update operation in practice (see, e.g., D'Andrea et al. (2013, 2014, 2015)).

Finally, given the not outstanding performance of both DecPLL and FulPLL in directed, sparse, weighted graphs, and given the importance of such input instances in practical applications, it would be worth studying, more deeply, the behavior of the proposed solutions in such graph instances and to find out whether it is possible to devise dedicated algorithms, perhaps exploiting structural properties specific to such networks.

REFERENCES

I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. 2012. Hierarchical hub labelings for shortest paths. In *20th European Symposium on Algorithms (ESA'12) (Lecture Notes in Computer Science)*, Vol. 7501. Springer, 24–35.

T. Akiba, Y. Iwata, and Y. Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *International Conference on Management of Data (SIGMOD'13)*. ACM, 349–360.

T. Akiba, Y. Iwata, and Y. Yoshida. 2014. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *23rd International World Wide Web Conference (WWW'14)*. ACM, 237–248.

R. Albert and A.-L. Barabási. 2002. Statistical mechanics of complex network. *Reviews of Modern Physics* 74 (2002), 47–97.

R. Bauer and D. Wagner. 2009. Batch dynamic single-source shortest-path algorithms: An experimental study. In *8th International Symposium on Experimental Algorithms (SEA'09) (Lecture Notes in Computer Science)*, Vol. 5526. Springer, 51–62.

S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang. 2006. Complex networks: Structure and dynamics. *Physics Reports* 424, 4–5 (2006), 175–308.

F. Bruera, S. Cicerone, G. D'Angelo, G. Di Stefano, and D. Frigioni. 2008. Dynamic multi-level overlay graphs for shortest paths. *Mathematics in Computer Science* 1, 4 (2008), 709–736.

I. Chabini and S. Lan. 2002. Adaptations of the A* algorithm for the computation of fastest paths in deterministic discrete-time dynamic networks. *IEEE Transactions on Intelligent Transportation Systems* 3, 1 (2002), 60–74.

J. Cheng and J. X. Yu. 2009. On-line exact shortest distance query processing. In *12th International Conference on Extending Database Technology (EDBT'09)*. ACM, 481–492.

A. Cionini, G. D'Angelo, M. D'Emidio, D. Frigioni, K. Giannakopoulou, A. Paraskevopoulos, and C. D. Zaroliagis. 2014. Engineering graph-based models for dynamic timetable information systems. In *14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'14) (OASICS)*, Vol. 42. Schloss Dagstuhl, 46–61.

A. Cionini, G. D'Angelo, M. D'Emidio, D. Frigioni, K. Giannakopoulou, A. Paraskevopoulos, and C. D. Zaroliagis. 2017. Engineering graph-based models for dynamic timetable information systems. *Journal of Discrete Algorithms* 46–47 (2017), 40–58.

E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. 2002. Reachability and distance queries via 2-hop labels. In *13th ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*. ACM/SIAM, 937–946.

A. D'Andrea, M. D'Emidio, D. Frigioni, S. Leucci, and G. Proietti. 2013. Dynamically maintaining shortest path trees under batches of updates. In *20th International Colloquium on Structural Information and Communication Complexity (SIROCCO'13) (Lecture Notes in Computer Science)*, Vol. 8179. Springer, 286–297.

A. D'Andrea, M. D'Emidio, D. Frigioni, S. Leucci, and G. Proietti. 2014. Experimental evaluation of dynamic shortest path tree algorithms on homogeneous batches. In *13th International Symposium on Experimental Algorithms (SEA'14) (Lecture Notes in Computer Science)*, Vol. 8504. Springer, 283–294.

A. D'Andrea, M. D'Emidio, D. Frigioni, S. Leucci, and G. Proietti. 2015. Dynamic maintenance of a shortest-path tree on homogeneous batches of updates: New algorithms and experiments. *ACM Journal of Experimental Algorithmics* 20 (2015), Article 1.5.

G. D'Angelo, M. D'Emidio, and D. Frigioni. 2014a. Fully dynamic update of arc-flags. *Networks* 63, 3 (2014), 243–259.

G. D'Angelo, M. D'Emidio, and D. Frigioni. 2014b. A loop-free shortest-path routing algorithm for dynamic networks. *Theoretical Computer Science* 516 (2014), 1–19.

G. D'Angelo, M. D'Emidio, and D. Frigioni. 2016. Distance queries in large-scale fully dynamic complex networks. In *27th International Workshop on Combinatorial Algorithms (IWOCA'16) (Lecture Notes in Computer Science)*, Vol. 9843. Springer, 109–121.

G. D'Angelo, M. D'Emidio, D. Frigioni, and V. Maurizio. 2011. A speed-up technique for distributed shortest paths computation. In *Computational Science and Its Applications (ICCSA'11) - International Conference. Proceedings, Part II (Lecture Notes in Computer Science)*, Vol. 6783. Springer, 578–593.

G. D'Angelo, M. D'Emidio, D. Frigioni, and C. Vitale. 2012. Fully dynamic maintenance of arc-flags in road networks. In *11th International Symposium on Experimental Algorithms (SEA'12) (Lecture Notes in Computer Science)*, Vol. 7276. Springer, 135–147.

D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. 2011. Customizable route planning. In *10th International Symposium on Experimental Algorithms (SEA'11) (Lecture Notes in Computer Science)*, Vol. 6630. Springer, 376–387.

D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. 2014a. Robust distance queries on massive networks. In *22th European Symposium on Algorithms (ESA'14) (Lecture Notes in Computer Science)*, Vol. 8737. Springer, 321–333.

D. Delling, A. V. Goldberg, R. Savchenko, and R. F. Werneck. 2014b. Hub labels: Theory and practice. In *13th International Symposium on Experimental Algorithms (SEA'14) (Lecture Notes in Computer Science)*, Vol. 8504. Springer, 259–270.

D. Delling, G. F. Italiano, T. Pajor, and F. Santaroni. 2014c. Better transit routing by exploiting vehicle GPS data. In *7th SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'14)*. ACM, 31–40.

A. W.-C. Fu, H. Wu, J. Cheng, and R. C.-W. Wong. 2013. IS-label: An independent-set based labeling scheme for point-to-point distance querying. In *Proceedings 39th International Conference on Very Large Databases (VLDB'13)*, 457–468.

R. Geisberger, P. Sanders, and D. Schultes. 2008. Better approximation of betweenness centrality. In *10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*. Society for Industrial and Applied Mathematics, 90–100.

L. Georgiadis, G. F. Italiano, L. Laura, and F. Santaroni. 2012. An experimental study of dynamic dominators. In *20th Annual European Symposium on Algorithms (ESA'12) (Lecture Notes in Computer Science)*, Vol. 7501. Springer, 491–502.

T. Hayashi, T. Akiba, and K. Kawarabayashi. 2016. Fully dynamic shortest-path distance query acceleration on massive networks. In *25th ACM International on Conference on Information and Knowledge Management (CIKM'16)*. ACM, 1533–1542.

Y. Hyun, B. Huffaker, D. Andersen, E. Aben, C. Shannon, M. Luckie, and K. C. Claffy. 2014. The CAIDA IPv4 Routed/24 Topology Dataset. Retrieved from http://www.caida.org/data/active/ipv4_routed_24_topology_dataset.xml.

R. Jin, N. Ruan, Y. Xiang, and V. E. Lee. 2012. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *International Conference on Management of Data (SIGMOD'12)*. ACM, 445–456.

M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. 2009. Fast shortest path distance estimation in large networks. In *18th ACM Conference on Information and Knowledge Management (CIKM'09)*. ACM, 867–876.

Y. Qin, Q. Z. Sheng, and W. E. Zhang. 2015. SIEF: Efficiently answering distance queries for failure prone graphs. In *18th International Conference on Extending Database Technology (EDBT'15)*. OpenProceedings.org, 145–156.

M. V. Vieira, B. M. Fonseca, R. Damazio, P. Braz Golgher, D. Castro Reis, and B. A. Ribeiro-Neto. 2007. Efficient search ranking in social networks. In *16th Conference on Information and Knowledge Management (CIKM'07)*. ACM, 563–572.

F. Wei. 2010. TEDI: Efficient shortest path query answering on graphs. In *International Conference on Management of Data (SIGMOD'10)*. ACM, 99–110.

Received March 2017; revised October 2018; accepted October 2018