Journal of Computer Engineering
ISSN 10003428

Original Research Article
Vol 12, Issue 11, 2023 | ISSN: 1000-3428

Efficient Computation of Shortest-Path Distances in Evolving Graphs

Alessandro Bianchi and Luca Rossi

Department of Computer Science and Engineering, University of Pisa, Via Caruso, I-56122 Pisa, Italy; Department of Mathematics and Statistics, University of Perugia, Via Pascoli, I-06123 Perugia, Italy

Abstract: Computing shortest-path distances is a fundamental primitive in the context of graph data mining, since this kind of information is essential in a broad range of prominent applications, which include social network analysis, data routing, web search optimization, database design and route planning. Standard algorithms for shortest paths (e.g., Dijkstra's) do not scale well with the graph size, as they take more than a second or huge memory overheads to answer a single *query on the distance* for large-scale graph datasets. Hence, they are not suited to mine distances from big graphs,

which are becoming the norm in most modern application contexts. Therefore, to achieve faster query answering, smarter and more scalable methods have been designed, the most effective of them based on precomputing and querying a compact representation of the transitive closure of the input graph, called the 2-HOP-COVER labeling. To use such approaches in realistic *time-evolving* scenarios, when the managed graph undergoes topological modifications over time, specific *dynamic algorithms*, carefully updating the labeling as the graph evolves, have been introduced. In fact, recomputing from scratch the 2-HOP-COVER structure every time the graph changes is not an option, as it induces unsustainable time overheads. While the state-of-the-art dynamic algorithm to update a 2-HOP-COVER labeling against *incremental* modifications (insertions of arcs/vertices, arc weights decreases) offers very fast update times, the only known solution for *decremental* modifications (deletions of arcs/vertices, arc weights increases) is still far from being considered practical, as it requires up to tens of seconds of processing per update in several prominent classes of real-world inputs, as experimentation shows. In this paper, we introduce a new dynamic algorithm to update 2-HOP-COVER labelings against decremental changes. We prove its correctness, formally analyze its worst-case performance, and assess its effectiveness through an experimental evaluation employing both real-world and synthetic inputs. Our results show that it improves, by up to several orders of magnitude, upon average update times of the only existing decremental algorithm, thus representing a step forward towards real-time distance mining in general, massive time-evolving graphs.

1. Introduction

Mining shortest-path distances is universally considered one of the most fundamental operation to be performed on graph data, due to the wide range of applications it finds. Prominent examples include social networks analysis [1], route planning [2,3], intelligent transport systems [4], context-aware search [5,6], web search optimization [7], graph

How to cite: Alessandro Bianchi and Luca Rossi (2023). Efficient Computation of Shortest-Path Distances in Evolving Graphs. Journal of Computer Engineering, 12(11), 160–196. DOI: https://doi.org/10.5281/zenodo.19350932

databases [8,9], decision making [10], analysis of biological networks [11], artificial intelligence [12] and network design [13].

Standard algorithms for solving the problem of computing answers to *queries on the distance*, i.e., to retrieve the weight of a shortest path between a pair of vertices of a given graph upon request, are well known and include the Breadth First Search (BFS) algorithm for unweighted graphs and

the Dijkstra's algorithm for weighted ones. Unfortunately, there exists vast, mostly experimental, algorithmic literature showing they are not suited for solving the problem at large-scale, i.e., when the graph to be managed is massively sized [14–17]. In fact, in these cases, the mentioned approaches do not scale well enough against the input' size, since their linear (or slightly superlinear for weighted graphs) worst-case time complexity yield impractical average time per query when graphs are huge (with up to millions of vertices and billions of arcs). The other extreme approach is to compute distances between all pairs of vertices beforehand and store them in an index [7]. Though in this way

we can answer distance queries almost instantly, this approach is also unacceptable since preprocessing time and index size are quadratic and unrealistically large.

For the above reasons, and since big graphs are becoming the norm in most modern application contexts that rely on distance mining, many smarter and more scalable techniques, between these two extreme solutions, have been proposed in the recent past to deal with massive graphs. Essentially all of them adopt the common strategy of preprocessing the graph to compute some data structure that is then exploited to accelerate the query algorithm [2,3,7,9,18–23]. Some of these methods have been designed to handle special classes of graphs, of interest of specific applications (e.g., speed-up techniques for route planning) and in road/transport networks (e.g., [2,14,20,21,24,25]) while some others are general and provide speedups over baseline strategies regardless of the structural properties of the graph to be managed (e.g., [3,15,19,23]). In this latter category we find 2-HOP-COVER-based labeling approaches that currently are considered state-of-the-art methods for distance mining in massive graphs. In fact, they have been shown to exhibit superior performance in terms of query times, allowing computation of shortest-path distances in microseconds even for billion-vertex graphs, at the price of a competitive preprocessing time and space overhead [7,8,21,26,27].

Specifically, such approaches rely on the idea of precomputing a compressed representation of the transitive closure of the input graph in terms of *concatenations of shortest paths*: for each pair of vertices of the input graph, such representation includes (at least) one intermediate vertex, called *hub vertex*, and (at least) two corresponding shortest paths, connecting the hub vertex to the two vertices of the pair, called *hops*. In this way, the shortest path connecting the pair of vertices can be reconstructed as the concatenation of the two hops at the hub vertex. In more details, a *label* at each vertex contains the length of a shortest path towards each hub and, to retrieve the distance between two vertices, it suffices to find a common hub in their labels minimizing the sum of the two distances. The difficult point here is to find a small set of hubs *covering* a shortest path for each pair of vertices in the graph, since in this way the resulting labeling is compact and, consequently the average query time is small. Indeed, it is known that finding a minimum size set of hubs is NP-hard [18,28] but both heuristics and approximation algorithms, for computing fairly compact labelings, providing excellent practical performance in terms of query times at the price of a reasonable preprocessing effort, are known [3,7,18]. Specifically, the two approaches of [3,7] currently are considered the best options in this direction.

A further level of complexity arises when the network to be handled is *time-evolving*, i.e., is when the underlying graph can evolve over time. In fact, in this which is universally considered the most realistic scenario, every time the network is subject to an update, i.e.,

when the corresponding graph undergoes some topological change (e.g., an arc removal or an arc weight decrease), the preprocessed data can become obsolete and hence must be updated to preserve query correctness (i.e., that exact distances are returned). A naive way to produce updated preprocessed data is to recompute everything from scratch. Nonetheless, this is not a practical strategy when inputs are massive, as it requires unsustainable computational overheads [8,27,29]. Observe that this issue is common to many preprocessing-based approaches for mining properties from big graphs (e.g., mining betweenness centralities [30]).

For these reasons, in the last few years, researchers have worked to adapt known methods, for mining (generic) properties from static graphs, to function in such time-evolving scenarios through the design of specific *dynamic algorithms*, i.e., algorithms that are able to update the preprocessed data to reflect graph changes without performing any full recomputation from scratch [8,14,17,24–27,30,31].

Such algorithms are typically divided in two broad categories, namely *incremental* and *decremental algorithms*, depending on whether they able to handle *incremental updates* only (i.e., insertion of vertices/arcs and arc weight decreases) or *decremental updates* only (deletion of vertices/arcs and arc weight increases).

Both types of algorithms find several prominent applications in the real-world, as well documented in the literature [8,17,26]. On the one hand, incremental algorithms are essential to handle scenarios where the graph to be managed tends to be subject to sequences of changes that correspond to a growth of the size of the underlying network. Examples include *citation networks*, where new references or new authors can be added dynamically, or *email/messaging networks*, where one can observe only new emails being sent and new addresses being added. On the other hand, decremental algorithms are useful in all those cases where the graph to be managed tends to be subject to sequences of changes that correspond to a reduction of the size/functionality of the underlying network. Examples include: *communication networks* where links can become unavailable, and arc weights representing latencies typically can deviate by increasing from the initial value; *railway/bus networks* where delays due to disruptions can occur, while no train is allowed to leave before the scheduled time; *data mining for linked data* (web link analysis, social network analysis, biological networks analysis, databases processing) where one's purpose is to determine which nodes/links are the most important by observing the effect of removals/congestions on the structure of the shortest paths (and/or diameter or components) of the corresponding graphs. Finally, it is well known that both kinds of solutions are necessary to build effective fully dynamic algorithms that are of fundamental importance for all those applications

where the structure of the sequences of modifications, affecting the graph, is unpredictable [26,32,33]. A typical example is *social network analysis* where users can join or leave, and connections between users can be established or removed.

For all above reasons, it is of utmost importance to have efficient algorithms to process both incremental and decremental operations when one aims at mining shortest-path properties from a time-evolving graph. Please note that the investigation on dynamic algorithms can also have a general graph theoretical interest since, often, studying how decremental changes affect shortest paths can also have applications to static graphs problems or other contexts (see, e.g., works on multi-commodity flows or cut problems [34]).

1.1. Our Contribution

For 2-HOP-COVER labelings, two dynamic algorithms that do not compromise on query performance are known, both employing the common strategy of identifying and updating only the part of the labeling that is compromised by a graph change. In details, the algorithm proposed in [8], named RESUME–2HC, is able to handle incremental updates only while the algorithm proposed in [26,27], named BIDIR–2HC, can manage decremental updates

only. Both have been shown, experimentally, to be rather effective and faster on average than the recomputation from scratch and to preserve the quality of the labeling in terms of query time and compactness (while there exists a dynamic approach that tolerates degradation on query time to achieve fast update [31]). Moreover, both have been combined and tested as a single fully dynamic algorithm in [26]. However, while the former exhibits extremely low update times that are compatible with real-time applications, the latter is slower, and sometimes only slightly better, on average, than the recomputation from scratch in some categories of input instances that are highly relevant to application domains, e.g., sparse weighted graphs.

In this paper, we focus on this latter issue and try to overcome the above limiting factor of BIDIR–2HC by introducing a new dynamic algorithm, called QUEUE–2HC, to update 2-HOP-COVER labelings when decremental update operations affect the input graph. We prove its correctness and formally analyze its worst-case performance. Since its worst-case time complexity cannot be directly compared with that of BIDIR–2HC, which depends on different parameters, we assess its effectiveness through an extensive experimental evaluation employing both real-world and synthetic inputs. Our results show that QUEUE–2HC improves, in some case up to orders of magnitude, upon the update times of BIDIR–2HC, especially in some input categories where such algorithm was shown to be not so effective. Our results also show QUEUE–2HC is always much faster than the recomputation from scratch, despite being worse in terms of worst-case complexity. This is a feature in common with BIDIR–2HC and RESUME–2HC, have been observed to be very effective in practice, despite their worst-case bound on the running time. The same holds for the preprocessing method, which is much faster in practice than what the worst-case bound suggests. This is a quite common behavior in the field of research dedicate to dynamic algorithms, typically due to two main reasons: either the worst-case essentially never or very rarely occurs in practical cases, or that the analysis is not tight. Nonetheless, having solutions like the one proposed in this paper is necessary to allow the use of the 2-HOP-COVER labeling method in the real-world. Thus, to summarize, QUEUE–2HC can be considered a step forward towards real-time distance mining for general, massive time-evolving graphs.

1.2. Structure of the Paper

The paper is organized as follows. In Section 2 we give the notation and nomenclature used throughout the manuscript, describe the basics of the 2-HOP-COVER labeling technique, and the algorithms known in the literature for handling the time-evolving scenario. In Section 3 we introduce our new algorithm, sketch its correctness proofs and computational complexity analysis while in Section 4 we present the experimental evaluation we conducted to assess the performance of the new approach. Finally, Section 5 concludes the paper and outlines possible future research directions.

2. Notation and Background

In this section, we provide the notation and the background notions that are necessary to introducing the contributions of the paper. In the remainder of the paper, for the sake of simplicity and generality, we consider the most general setting where the graph to be handled is a directed weighted graph $G = (V, A, w)$, with $n = |V|$ *vertices* and $m = |A|$ *arcs*, such that $w : A \to R^+$ is a weight function assigning a positive real $w(u, v)$ to any arc $(u, v) \in A$.

A *path* P_{st} in G, connecting two vertices s and t, is a sequence of arcs $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-2}, v_{k-1}), (v_{k-1}, v_k)$ where $v_0 = s$, $v_k = t$, and each consecutive pair of arcs in the sequence shares a common endpoint. We call k the *length* of the path itself, i.e.,

the number of arcs in the sequence. A path is *simple* if it does not contain self-loops, i.e., if any vertex in the sequence of arcs appears only once. Moreover, we denote by w(P) the *weight* of a path P, defined as the sum of the weights of its constituting arcs, i.e., $w(P) = \sum_{(u,v)\in P} w(u, v)$. A path P, connecting two vertices u and v, that has minimum weight among all paths in G between said two vertices is called a *shortest path* from u to v. We call d(u, v) = w(P) the *distance* from vertex u to vertex v in G, which is the weight of a shortest path from u to v in G. If such a path does not exist, i.e., vertices are disconnected, for all $(u, v) \in A$. Therefore, the weight of a shortest path equals its length in these graphs.∞ then we assume d(u, v) = . Observe that conventionally, in unweighted graphs we have w(u, v) = 1

We denote by $N_{out}(v)$ ($N_{in}(v)$, respectively) the set of the *outgoing* (*incoming*, respectively) *neighbors* of v in G, that is $N_{out}(v) = \{u \in V \mid \{v, u\} \in A\}$ ($N_{in}(v) = \{u \in V \mid \{u, v\} \in A\}$, respectively). Clearly, in undirected graphs we have $N_{out}(v) = N_{in}(v)$ for any $v \in V$. Furthermore, we denote by $G^T = (V, A^T, w)$ the *transpose* of G where an arc $(u, v) \in A^T$ if and only if $(v, u) \in A$, and $w(u, v) = w(v, u)$. Similarly, we use $N^T_{out}(v)$ ($N^T_{in}(v)$, respectively) to denote the set of the *outgoing* (*incoming*, respectively) *neighbors* of v in G^T. A *graph update* or *modification* can be either *incremental*, if it is an insertion of a new arc or a decrease in the weight of an existing arc, or *decremental*, if otherwise it is a deletion or an increase in the weight of an existing arc. Please note that insertions/deletions of arcs can insertions/deletions can be easily modeled as sets of modifications to the corresponding arcs, hence∞ be easily modeled as decreases of weights from to w(x, y) and vice versa. Notice also that vertex

the approaches for handling arc updates can be easily extended to manage vertex insertions/deletions by repeatedly executing them for the set of adjacent arcs. For the sake of clarity, we use d_G to represent the distance function in a given graph G but we omit subscripts from our notation whenever the meaning is clear from the context.

2.1. 2-HOP-COVER *Labeling*

In this section, we summarize the main characteristics of the 2-HOP-COVER labeling method. Our descriptions refer to the most general case of weighted directed graphs, which is the most complex to handle. Simpler versions of the approaches can be obtained for unweighted or undirected graphs,

we refer the reader to [26] for a thorough discussion on the differences.

For each vertex v of G, we call *incoming* and *outgoing labels*, respectively, $L_{in}(v)$ and $L_{out}(v)$ of v two sets of pairs (also known as *label entries*) of the form (u, δ_{uv}) and (u, δ_{vu}), respectively, where $u \in V$ while $\delta_{uv} = d_G(u, v)$ and $\delta_{vu} = d_G(v, u)$, respectively. The collection of all the labels $L = \{\{L_{out}(v)\}_{v\in V}, \{L_{in}(v)\}_{v\in V}\}$ is referred to as a *distance labeling* of G. For the sake of readability, we use $u \in L_{out}(v)$ ($u \in L_{in}(v)$, respectively) to denote that $(u, \delta_{vu}) \in L_{out}(v)$ ($(u, \delta_{uv}) \in L_{in}(v)$, respectively), whenever the meaning is clear from the context. Similarly, for any $u, v \in V$ we say

$L_{out}(u) \cap L_{in}(v) =6 \emptyset$ whenever there exists some $h \in V$ such that $(h, \delta_{uh}) \in L_{out}(u)$ and $(h, \delta_{hv}) \in L_{in}(v)$ (and vice versa). Labels can be used to retrieve the (exact or approximate) distance from vertex $s \in V$ to vertex $t \in V$ in G by performing a *query on the distance* from s to t, as in Equation (1).

$$Q(s, t, L) = \begin{cases} \min\{\delta_{sv} + \delta_{vt} \mid (v, \delta_{sv}) \in L_{out}(s) \wedge (v, \delta_{vt}) \in L_{in}(t)\} & \text{if } L_{out}(s) \cap L_{in}(t) \neq \emptyset \\ \infty & \text{otherwise.} \end{cases} \quad (1)$$

Please note that each query requires access to the labels of s and t only.

A distance labeling L is a 2-HOP-COVER *labeling* [18] of a graph G (often referred to simply as labeling of G in the remainder of the paper) if, for any pair of vertices $s, t \in V$:

- either $L_{out}(s) \cap L_{in}(t) \neq \emptyset$ contains (at least) a vertex $h \in V$ lying on a shortest path between s and t in G, and therefore $Q(s, t, L) = \delta_{sh} + \delta_{ht} = d_G(s, t)$ (s and t are connected in G);
- or $L_{out}(s) \cap L_{in}(t) = \emptyset$ and hence $Q(s,t,L) = d_G(s,t) = \infty$ (when s and t are not connected in G).

If a pair of vertices (s, t) is connected, then the above-mentioned vertex h $L_{out}(s)$ $L_{in}(t)$, lying on a shortest path from s to t, is called *hub vertex* of pair s, t (or simply *hub*). Vertex h is said to *cover* pair s, t. Symmetrically, the pair is said to be *covered* by h and, thus, by the labeling L. By extension, if the above conditions hold, the graph is said to be *covered* by the labeling and the labeling is said to *cover* the graph (or to satisfy the *cover property* for the considered graph). Notice that a given pair of vertices $u, v \in V$ can be covered by more than one hub vertex, depending on how the 2-HOP-COVER labeling is computed.

A naive approach to compute a 2-HOP-COVER labeling of a graph consists of executing twice a shortest-path algorithm that can be a Breadth First Search (BFS) in unweighted graphs or the Dijkstra's algorithm in weighted ones, using as root each vertex $v \in V$, once *forward* (that is, in G), and once *backward* (that is, in G^T). In the forward case, when a vertex u is extracted from the queue, used by the shortest-path algorithm, with a priority of d, that is when a shortest path with weight d from the root v to u in G is found, then entry $(u, \delta_{vu} = d)$ is added to $L_{in}(u)$. In the backward case, instead, entry $(u, \delta_{uv} = d)$ is added to $L_{out}(u)$ whenever vertex u is extracted from the queue and hence when a shortest path, with weight d, from the root v to u in G^T, is found.

It is easy to see how a labeling obtained as above covers the graph [7], since there is at least one hub vertex per pair of vertices (the root vertex of each visit). However, computing a labeling L as above costs $\Theta(n \cdot SP(n,m))$ worst-case time, where SP(n,m) is the worst-case computational time of a corresponding shortest-path algorithm. Moreover, L contains $\Theta(n^2)$ hub vertices and hence $\Theta(n^2)$ label entries, which is the same space complexity of storing all pairs distances. Furthermore, it induces a query algorithm with $\Omega(n)$ computational effort, since all labels contain $\Theta(n)$ entries. Thus, this basic version of 2-HOP-COVER labeling is far from being considered an effective approach for distance mining, since, as is, it is worse than standard methods for shortest paths. To make it practical, one must aim at minimizing both running times for computing a labeling (a.k.a. *preprocessing times*) and *labeling size* (i.e., total number of entries), which in turn hopefully induces small query times in practice.

To this end, unfortunately, it is known that finding a minimum-size labeling covering a graph G is an NP-hard problem [18]. However, few heuristic approximation algorithms for computing compact labelings are known [3,7,18].

In particular, the ones in [3,7] have been shown to achieve considerably better scalability, in terms of both preprocessing times and space requirements, than other methods in essentially all classes of graphs. Both these methods require $\Theta(n \cdot SP(n, m))$ worst-case running time to compute a labeling that can have $O(n^2)$ label entries. Nevertheless, extensive experimentation shows they behave very well in practice. Specifically, they exhibit preprocessing times in the order of hours even for massively sized inputs, and produce labelings that have a labeling size that on average, is much smaller than the worst-case estimation of a constant multiple of n^2 [3,7]. This leads to have average query times that in practice, are below milliseconds even for the largest instances.

The elements that are common to the two strategies are: (i) considering the vertices of V in order according to some importance criterion with respect to the shortest paths in the graph; (ii) performing a number n of shortest-path visits, each rooted at a vertex $v_i \in V$ in the above−mentioned order, say $\{v_1, v_2, \ldots, v_n\}$, that incrementally build the labeling starting from empty label sets (Observe that the ordering can be greedily changed in this phase to achieve better results); (iii) employing a so-called *pruning mechanism* that stops the branch of the graph that is currently being explored whenever a *pruning condition* is met. In particular, let us denote by $\{v_1, v_2, \ldots, v_n\}$ the vertex ordering and by

L^{k-1} the labeling computed after the execution of the two shortest-path visits, rooted at vertex v_{k-1} (i.e., containing only entries of the form $(v_i, *)$ for v_i 6 v_{k-1}).

If we consider a vertex u that is reached during the forward (backward, respectively) visit rooted at v_k and assume that δ is the currently discovered distance from v_k to u (from u to v_k, respectively). Then, the algorithm checks whether $Q(v_k, u, L^{k-1})$ 6 δ ($Q(u, v_k, L^{k-1})$ 6 δ, respectively). If the above condition holds, then L^{k-1} already contains a hub vertex for pair (v_k, u) ((u, v_k), respectively) and for all pairs (v_k, x) such that there exists a shortest path between v_k and x passing through vertex u (for all pairs (x, v_k) such that there exists a shortest path from x to v_k passing through vertex u, respectively). Therefore, the visit rooted v_k is *pruned* at u. Otherwise, the algorithm updates either $L_{in}(u)$ or $L_{out}(u)$) as described above and continues. Clearly, for each vertex u that is not reachable from v_k (that cannot reach v_k, respectively), we have $L_{in}(u)^k = L_{in}(u)^{k-1}$ ($L_{out}(u)^k = L_{out}(u)^{k-1}$, respectively). It can be proven that the obtained L_n is a 2−HOP−COVER labeling [7]. It can also easily seen that both methods guarantee that if v < u in the considered vertex ordering, then both u 6∈ L_{in}(v) and u ∈6 L_{out}(v), while v might be in $L_{in}(u)$ or in $L_{out}(u)$. This property, called *well-ordering* property [22], is quite important, as it can be exploited to prove that the computed labeling is *minimal* [7], in the sense that removing any single label entry from the labeling breaks (i.e., induces the violation of) the cover property for at least one pair of vertices of the graph. Formally speaking, the well-ordering property is as follows.

Property 1 (Well-ordered Labeling)**.** *Let* $L = \{\{L_{out}(v)\}_{v \in V}, \{L_{in}(v)\}_{v \in V}\}$ *be a* 2-HOP-COVER *labeling of a graph* G = (V, A). *Let* $\{v_1, v_2, \ldots, v_n\}$ *be an ordering on the vertices of* G. *Then* L *is well-ordered if, for any* $v_i, v_j \in V$ *such that* $v_i < v_j$ *we have that:*

- $Q(v_i, v_j, L) = d(v_i, v_j)$*;*
- $v_j \in$6 $L_{in}(v_i)$ *and* v_j 6∈ $L_{out}(v_i)$.

Please note that minimality is a highly desirable property since it has been empirically shown that 2−HOP−COVER labelings that are minimal and that are computed by considering certain vertex orderings yield excellent practical performance in terms of both preprocessing time, labeling size, and average query times. In particular, it is known that the performance of the methods above heavily depends on the vertex ordering and it has been experimentally observed that the average label size (and the resulting average running time of the query algorithm) decreases by several orders of magnitudes, with respect to random orderings, if vertices are sorted according to a centrality−related measure, like, e.g., vertex degree or (approximations of) betweenness centrality [3,7,26]. Moreover, for well−ordered (minimal) labelings, if distinct ids ranging from 0 to n − 1 are assigned to the vertices of V and, for each vertex v ∈ V, pairs in $L_{in}(v)$ and $L_{out}(v)$ are stored in the form of an array and sorted according to said ids, then Q(s, t, L) = d(s, t) can be effectively implemented as shown in

Algorithm 1 to take $O(|L_{out}(s)| + |L_{in}(t)|)$ worst-case running time ($|L_{out}(s)|$ and $|L_{in}(t)|$ denote the number of label entries in the two labels). Given a label, say $L_{out}(v)$, of a vertex v stored as above, in what follows we denote by $L_{out}(v)[i]$ it i-th element and by $L_{out}(v)$.size its size.

Algorithm 1: Query algorithm for minimal well-ordered labelings

---

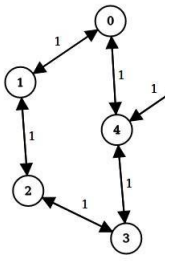
Input: Labeling L, queried vertices s, t ∈ V

Output: Distance d(s, t) in G

1 i ← 0;

In the remainder of the paper, we will call FS−2HC the generic preprocessing strategy, described above, that yields well-ordered minimal labelings.

Observe that all dynamic algorithms known can be used regardless of the specific preprocessing strategy used to compute the initial labeling of the graph [7,26]. We will show that the same holds for the new algorithm proposed in this work. More specifically, all dynamic algorithms considered in this paper only need an initial labeling that is well-ordered, and this is guaranteed if FS−2HC strategies are used. A sample directed, unweighted graph, with a corresponding well-ordered minimal 2-HOP-COVER labeling is shown in Figure 1.



	{(4,1),(0,0)}	{(4,1),(0,0)}
	{(4,2),(0,1),(3,2),(1,0)}	{(4,2),(0,1),(3,2),(1,0)}
	{(4,2),(0,2),(3,1),(1,1),(2,0)}	{(4,2),(0,2),(3,1),(1,1),(2,0)}
	{(4,1),(3,0)}	{(4,1),(3,0)}
	{(4,0)}	{(4,0)}
	{(4,1),(5,0)}	{(4,1),(5,0)}
	{(4,2),(5,1),(6,0)}	{(4,2),(5,1),(6,0)}

Figure 1. A sample graph and a corresponding 2-HOP-COVER labeling is shown. Vertex ordering is

{4, 0, 3, 1, 2, 5, 6}.

2.2. Dynamic Algorithms for Updating 2-HOP-COVER *Labelings*

Essentially, the only two known methods for updating 2-HOP-COVER labelings are those in [8,26], named RESUME–2HC and BIDIR–2HC in what follows, respectively, whose main features are briefly summarized below, since the new approach we propose in this paper borrows some features from both solutions. We refer the reader to [8,26] for a thorough description of the RESUME–2HC and BIDIR–2HC, respectively.

Algorithm RESUME–2HC**.** Given a graph G = (V, A) and a 2-HOP-COVER labeling L of G, algorithm RESUME–2HC is able to update L in order to reflect an incremental update (i.e., a newly added arc (a, b) 6∈ A or a decrease in the weight w(a, b) of an arc (a, b) ∈ A). In particular, if we denote by G a graph before a given incremental modification, by L a 2–HOP–COVER labeling L of G, and by G^o the graph resulting by the application of the change to G, then RESUME–2HC computes a 2-HOP-COVER labeling L^o of G^o by updating L as follows.

First of all notice that RESUME–2HC does not remove from L the so-called *outdated label entries* of L, i.e., entries in the label sets that correspond to shortest paths in G but not in G^o, due to the incremental update occurred on arc (a, b), and hence they are present also in L^o. Formally, an *outdated entry* is a pair (u, δ_{uv}) ∈ L_{in}(v) (or symmetrically (u, δ_{vu}) ∈ L_{out}(v)), for some u and v in G, such that δ_{uv} = d_G(u, v) =6 d_{G^o}(u, v) (or symmetrically δ_{vu} = d_G(v, u) =6 d_{G^o}(v, u)). The above choice is motivated by the fact that distances can only decrease as a consequence of incremental updates and hence, if one adds entries corresponding to new shortest paths, then queries return correct distances in G^o for all pairs of vertices even in the presence of such outdated entries (i.e., the cover property is preserved). This is because the query algorithm searches for minimum values in the label sets.

Therefore, RESUME–2HC simply adds new label entries or overwrite distances of existing ones. A drawback of this approach is that the minimality of the resulting labeling L^o as a whole is broken even after a single update and this is known to affect the performance over time (periodical reprocessing is necessary to restore minimality and avoid the labeling grows too large). Observe that the problem of designing an incremental algorithm that does not suffer from this issue is still open [29].

The strategy of RESUME–2HC, for adding label entries corresponding to new shortest paths, is based on the following two facts: (i) if the distance from a vertex v_k to a vertex u changes, then all new shortest paths from v_k to u pass through the updated arc (a, b); (ii) if a shortest path P from v_k to u =6 a, b changes, then the distance between v_k and w changes, where w is the penultimate vertex in P. Based on the above insights, for every vertex v_k, the idea is that it suffices to resume forward and backward, respectively, shortest-path visits (i.e., in G^o and G^{oT}, respectively) that: (i) are originally rooted at v_k; (ii) start at b and a, respectively; (iii) stop at unchanged vertices. That is, instead of inserting (v_k, 0) into the priority queue, the algorithm inserts pair (b, d_G(v_k, a) + w(a, b)) ((a, d_G(b, v_k) + w(a, b)), respectively). Then, the search proceeds by adopting a pruning mechanism similar to that of FS–2HC described above, modified to consider that outdated entries are not removed. In particular, it employs the notion of *prefixal query*, denoted as PQ, that, given two vertices s, t and an integer k and a labeling L, is computed as in Equation (2).

min {δ

$PQ(s, t, L, k) = \min_{v_i \in V, i \le k} \{ sv_i + \delta v_{it} \mid (v, \delta sv_i) \in L_{out}(s) \wedge (v, \delta v_{it}) \in L_{in}(t) \}$ **if** $L_{out}(s) \cap L_{in}(t) \neq \emptyset$ (2)

∞

otherwise.

In simpler terms, PQ(s, t, L, k) is the answer to a query from s to t computed from labeling L using distances to vertices $v_1, v_2, \ldots, v_k$ only. Suppose now a resumed forward (backward, respectively) visit, originally rooted at v_k, visits vertex u with distance δ, then it is possible to prune the search at u if $PQ(v_k, u, L, k) \le \delta$ ($PQ(u, v_k, L, k) \le \delta$, respectively). It can be shown that to obtain a 2-HOP-COVER labeling L^o of G^o by updating L with the above strategy, it suffices to resume forward and backward shortest-path searches rooted at vertices $\{w : w \in L_{out}(a) \vee w \in L_{in}(b)\}$ [8].

Algorithm BIDIR–2HC**.** First of all, notice that when handling a decremental graph update, outdated label entries must be removed from L to obtain a 2-HOP-COVER labeling L^o of G^o, as otherwise it is easy to show the cover property does not hold for G^o. Specifically, even after a single decremental operation, a query on such labeling might return an arbitrary underestimation of the true value of distance in G^o. For this reason, algorithm BIDIR–2HC works in three phases. If we assume that an arc (x, y) undergoes a decremental update (i.e., either it is removed or its weight is increased), in a first phase BIDIR–2HC performs two shortest-path-like searches of the graph G, one backward (i.e., on G^T) and one forward (i.e., on G itself), rooted at x and y respectively. By exploiting both L and the structure of the graph, the visits detect the so-called *affected vertices*, i.e., vertices that are candidate to contain at least one outdated label entry. In a second phase, the algorithm scans all such vertices and removes outdated label entries. For efficiency purposes, affected vertices are divided into two sets and are sorted according to the original vertex ordering, and the removal policy works as follows: (i) given an affected vertex v of the first set, entries (u, δ_{vu}) such that u is in the second set are removed from $L_{out}(v)$ (if any); (ii) given an affected vertex v of the second set, entries (u, δ_{uv}) such that u is in the second set are removed from $L_{in}(v)$ (if any). The result of the removal phase is that there might be pairs of affected vertices that are not covered by the resulting labeling, depending on the structure of G^o. Therefore, a third phase to restore the cover property for all such pairs is executed to obtain a labeling L^o that covers G^o. To this aim, a set of shortest-path searches is performed, namely one forward search in G^o, rooted at each affected vertex of the first set, and one backward search in G^{oT}, rooted at each affected vertex of the second set, following the vertex ordering to achieve both the well-ordered property and the minimality. Such visits essentially discover shortest paths connecting pairs of affected vertices and, accordingly, add entries to the corresponding labels. As well as both FS–2HC and RESUME–2HC, a pruning mechanism is incorporated in these visits, based on the ordering of vertices and on results of queries. However, the pruning is less “aggressive” and hence less effective in reducing the search space in practice. The motivation is essentially that shortest paths, connecting affected vertices in G^o or G^{oT}, can pass through non-affected vertices (see [27,29] for more details). Therefore a forward (backward, respectively) visit, rooted at affected vertex v_k of the second (first, respectively) set, can be stopped only if $Q(v_k, u, L^{k-1}) \le \delta$ (or $Q(u, v_k, L^{k-1}) \le \delta$, respectively) and u is an affected vertex of the first (second, respectively) set, where δ is the distance from v_k to u (from u to v_k, respectively) found by the visit and L^{k-1} is the labeling restricted to entries of the form $(v_i, *)$ for $v_i \le v_{k-1}$.

3. A New Algorithm: QUEUE–2HC

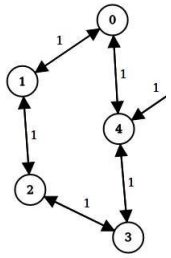
In this section, we describe our new decremental algorithm for updating 2-HOP-COVER labelings, named QUEUE–2HC. Our description refers to the most general case of weighted directed graphs,

which is the most complex to handle [3,8,26,27]. Simpler versions of the approaches can be easily obtained for unweighted or undirected graphs by considering the observations reported in the previous sections or in [8,26,27].

3.1. Profiling of Algorithm BIDIR–2HC

The new method is inspired by some empirical observations on the behavior of BIDIR–2HC, the only previously known solution for updating 2-HOP-COVER labelings against decremental graph changes. These observations come from a preliminary experimental study we conducted through profiling software on an implementation of BIDIR–2HC itself. Specifically, our data show that:

1. the first two phases require typically small fractions of the time taken by BIDIR–2HC, even in very large instances. However, some "computational redundancy" is observed. In particular, there can be some vertices whose label sets do not contain any outdated entry and yet such sets are scanned twice, once during the detection phase and once during the removal phase (within the computed set of *affected vertices*, see [26] or Section 2.2). This is clearly an undesired behavior that we aim at removing by designing a new algorithm for the purpose. An example of this scenario is shown in Figure 2.
2. the largest fraction of the entire computational effort to update a labeling, as a consequence of a decremental update, is spent by BIDIR–2HC on testing whether the cover property is satisfied and in restoring it when it is not, i.e., on the third phase. During this step, a number of shortest-path driven visits of the graph is performed, each rooted at one of the mentioned affected vertices and pruned according to a policy that is similar to that of FS–2HC. Unfortunately, however, the pruning conditions are less restrictive with respect to those of FS–2HC. Specifically, during the search started at v_k, whenever a vertex u is settled with a distance of δ, the algorithm checks whether $Q(v_k, u, L^{k-1}) \leq \delta$ and prunes the search in the affirmative case, since this implies that labeling L^{k-1} contains a hub vertex 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. On the contrary, BIDIR–2HC must consider the fact that new paths, connecting pairs of affected vertices, can pass through non-affected vertices. Hence, to guarantee that the cover property is restored for all pairs of vertices, the search is pruned only if $Q(v_k, u, L^{k-1}) \leq \delta$ *and* u is affected. A consequence of this strategy is that many vertices of the graph, whose label sets do not change, are visited and tested for pruning by the searches of the third phase. This is, again, an undesired behavior we aim at avoiding by our new solution.



	{(4,1),(0,0)}	{(4,1),(0,0)}
	{(4,2),(0,1)} {(3,2),(1,0)}	{(4,2),(0,1)} {(3,2),(1,0)}
	{(4,2),(0,2),(3,1)} {(1,1),(2,0)}	{(4,2),(0,2),(3,1)} {(1,1),(2,0)}
	{(4,1),(3,0)}	{(4,1),(3,0)}
	{(4,0)}	{(4,0)}

	{(4,1),(5,0)}	{(4,1),(5,0)}
	{(4,2),(5,1),(6,0)}	{(4,2),(5,1),(6,0)}

Figure 2. Consider the graph of Figure 1 (**left**) and the corresponding labeling (**right**). Assume arc (4, 0) is removed. If Algorithm BIDIR–2HC is executed, vertices 3, 5, 6 have their label sets scanned twice, once during the detection of affected vertices and once during the removal phase, since hub vertex for pairs (3, 0), (5, 0) and (6, 0) is vertex 4. However, no label entry is removed from $L_{in}(3)$ nor from $L_{out}(3)$ (the same hold for label sets of $L_{in}(5)$, $L_{out}(5)$, $L_{in}(3)$, $L_{out}(3)$). Our algorithm instead scans these label sets only once.

Beyond the Limitations of BIDIR–2HC**.** Our algorithm tries to overcome the above-mentioned limitations by attacking the problem in a different way with respect to BIDIR–2HC. In details, the algorithm works in two phases, named CLEAN and RECOVER, respectively, as summarized in Algorithm 2. The aim of the CLEAN phase is two-fold: (i) that of identifying and removing outdated label entries efficiently from the labeling (to produce an updated version of the labeling that contains **only entries** that are **correct** for the new graph); (ii) that of determining pairs of vertices that remain uncovered by the removal of outdated label entries without scanning their label sets twice (as done by BIDIR–2HC). The RECOVER phase's purpose, instead, is that of restoring the cover property for such pairs by suitably adding new label entries without visiting vertices of the graph whose label sets are unchanged by the graph update (unlike BIDIR–2HC). The two phases are described in the following sections.

Algorithm 2: Algorithm QUEUE–2HC

Input: Graph G, 2-HOP-COVER labeling L of G, arc (x, y) subject to decremental update.

Output: Resulting graph G^o, 2-HOP-COVER labeling L^o of G^o.

1 $(G^o, L^o, OUT, IN) \leftarrow CLEAN(G, L, x, y)$;

2 $L^o \leftarrow RECOVER(G^o, L^o, OUT, IN, x, y)$;

3 return (G^o, L^{oo});

3.2. CLEAN *Phase*

Given an arc $(x, y) \in A$ that undergoes a decremental update, the CLEAN phase aims at performing a more efficient regarding BIDIR–2HC, removal of outdated labels, by finding and removing all and only *outdated label entries* (from now on simply referred to as outdated entries) from the labeling, i.e., by avoiding to remove any correct entry. This is obtained by scanning the labels of vertices that are actually subject to entry removals and, contextually, by identifying pairs of vertices that might remain "uncovered" (i.e., without an hub in the labeling) after the removal of the mentioned incorrect label entries. This last identification step is necessary to test that the cover property for the new graph efficiently, in a next phase. The two purposes above are achieved, in detail, by:

1. determining two sets of vertices, named *forward invalid hubs* and *backward invalid hubs*, respectively, that are hubs for some pair in G but might stop being hubs in G^o, due to the operation on (x, y), so to remove from L the incorrect entries associated with such vertices;
2. identifying and exploring two (virtual) subgraphs of the input graph, named *forward cover graph* and *backward cover graph*, respectively, and denoted by $F^L_G(x, y)$ and $B^L_G(x, y)$, respectively In more details, to define our algorithm we give the following definitions.

Definition 1 (Forward Cover Graph)**.** *Given a graph* G = (V, A)*, a* 2-HOP-COVER *labeling* L *of* G*, an arc* $(x, y) \in A$*. The forward cover graph* $F^L_G(x, y)$ *is a subgraph of* G *with* V^o *and* A^o *as vertex and arc sets, respectively, as follows:*

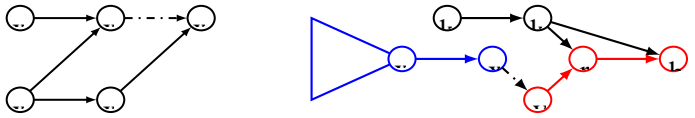
- *a vertex* u *is in* V^o *if* $u \in V$ *and there exist two pairs* $(h, \delta_{uh}) \in L_{out}(u)$ *and* $(h, \delta_{hy}) \in L_{in}(y)$ *such that* $\delta_{uh} + \delta_{hy} = d_G(u, y) = d_G(u, x) + w(x, y)$ *(that is a shortest path from* u *to* y *includes* (x, y) *and* h *is a hub vertex for pair* u, y*);*
- *an arc* (u, v) *is in* A^o *if* $(u, v) \in A$ *and* $u, v \in V^o$ *(hence* $d_G(u, y) = w(u, v) + d_G(v, x) + w(x, y)$*).*

In other words, vertices of the forward cover graph are those with, in G, a shortest path toward y that might contain arc (x, y) and hence whose distance toward y might change as a consequence of a decremental operation occurring on (x, y). Please note that whether the distance toward y changes or not depends on the number of shortest paths from the vertex to y that are in the graph, and on the vertex ordering (see Figure 3a for a graphical example of two different vertices of the forward cover graph). Note also that the arcs of such graph are the arcs of a shortest-path arborescence of G rooted at x plus arc (x, y). Symmetrically, we define a backward version of the cover graph as follows.

Definition 2 (Backward Cover Graph)**.** *Given a graph* G = (V, A)*, a* 2-HOP-COVER *labeling* L *of* G *and an arc* $(x, y) \in A$*. The backward cover graph* $B^L_G(x, y)$ *is a subgraph of* G *with* V^o *and* A^o *as vertex and arc sets, respectively, as follows:*

- *a vertex* u *is in* V^o *if* $u \in V$ *and there exist two pairs* $(h, \delta_{xh}) \in L_{out}(x)$ *and* $(h, \delta_{hv}) \in L_{in}(v)$ *such that* $\delta_{xh} + \delta_{hv} = d_G(x, v) = w(x, y) + d_G(y, v)$ *(that is a shortest path from* x *to* u *includes* (x, y) *and* h *is a hub vertex for pair* x, v*);*
- *an arc* (u, v) *is in* A^o *if* $(u, v) \in A$ *and* $u, v \in V^o$ *(hence* $d_G(x, v) = w(x, y) + d_G(y, u) + w(u, v)$*).*

In other words, analogously to the forward case, vertices of the backward cover graph are those having, in G, a shortest path from x that might contain arc (x, y) and hence whose distance from x might change as a consequence of a decremental operation occurring on (x, y). Please note that again whether the distance from x changes or not depends on the number of shortest paths from the vertex from x that are in the graph, and on the vertex ordering. Note also that the arcs of such graph are the arcs of a shortest-path arborescence of G^T rooted at y plus arc (x, y) (a toy example of the structure of cover graphs for a given input graph is shown in Figure 3b).



(**a**) (**b**)

Figure 3. (**a**): assume all arcs in this example have unitary weight and the vertex ordering is $\{x_o,x_1,\ldots,x_4\}$. Both x_4 and x_3 are in $F^L_G(x_1, x_o)$ since the hub vertex for pairs x_4, x_o and x_3, x_o, in the labeling L induced by the above ordering, is vertex x_o. However, if arc (x_1, x_o) is removed then $d_G(x_4, x_o)$ changes while $d_G(x_3, x_o)$ does not, since x_3 has two shortest paths of the same weight to x_o. (**b**): In the toy graph G shown here the only shortest path from x to h passes through arc (x, y). Therefore, if (x, y) undergoes a decremental update, we have that x, x_1, and all vertices that are connected to x_1, are in the forward cover graph (in blue). Symmetrically, r and h are in the backward cover graph (in red) while k_1 and k_2 are in neither of the two cover graphs.

Finally, we define the set of *forward invalid hubs* $H_f(x, y) \subseteq V$ (*backward invalid hubs* $H_b(x, y) \subseteq V$, respectively) of (x, y) as follows.

Definition 3 (Forward Invalid Hubs)**.** *Given a graph* G = (V, A)*, a* 2-HOP-COVER *labeling* L *of* G *and an arc* $(x, y) \in A$*. Then, the set of forward invalid hubs* $H_f(x, y)$ *of* (x, y) *is defined as:*

$$H_f(x, y) = \{h \in V : (h, \delta_{xh}) \in L_{out}(x) \wedge (h, \delta_{yh}) \in L_{out}(y) \wedge w(x, y) + \delta_{yh} = \delta_{xh}\}.$$

Similarly, we give a backward version.

Definition 4 (Backward Invalid Hubs)**.** *Given a graph* G = (V, A)*, a* 2-HOP-COVER *labeling* L *of* G *and an arc* $(x, y) \in A$*. Then, the set of backward invalid hubs* $H_b(x, y)$ *of* (x, y) *is defined as:*

$$H_b(x, y) = \{h \in V : (h, \delta_{hx}) \in L_{in}(x) \wedge (h, \delta_{hy}) \in L_{in}(y) \wedge w(x, y) + \delta_{hx} = \delta_{hy}\}.$$

Given the above definitions, algorithm CLEAN's strategy is based on the following properties. Notice that for the sake of simplicity, in what follows we use $v \in F^L_G(x, y)$ or $v \in B^L_G(x, y)$ to denote a vertex v belonging to the vertex set of either the forward or the backward cover graph.

Property 2. *Let* L *be a* 2-HOP-COVER *labeling of* G*. Assume arc* (x, y) *is subject to a decremental operation and let* G^o *be the graph obtained by applying to* G *such modification. Let* $F^L_G(x, y)$ *(*$B^L_G(x, y)$*, resp.) be the forward (backward, resp.) cover graph. Then, for any vertex* $u \in V$ *such that* $u \notin F^L_G(x, y)$ *(*$v \notin B^L_G(x, y)$*, resp.) we have that* $L_{out}(u)$ *(*$L_{in}(u)$*, resp.) contains only correct entries for* G^o*, that is for any* $(h, \delta_{uh}) \in L_{out}(u)$ *we have* $\delta_{uh} = d_G(u, h) = d'_G(u, h)$ *(for any* $(h, \delta_{hu}) \in L_{in}(u)$ *we have* $\delta_{hu} = d_G(h, u) = d'_G(h, u)$*, resp.).*

Proof. By definition of forward cover graph, we know $u \notin F^L_G(x, y)$ implies there does not exist two pairs $(h, \delta_{uh}) \in L_{out}(u)$ and $(h, \delta_{hy}) \in L_{in}(y)$ such that $\delta_{uh} + \delta_{hy} = d_G(u, y) = d_G(u, x) + w(x, y)$ for some $h \in V$. Observe that for any 2−HOP−COVER labeling, $(h, \delta_{uh}) \in L_{out}(u)$, for some $h \in V$, implies $(h, \delta_{hh} = 0) \in L_{in}(h)$. Now, by contradiction, assume an entry $(h, \delta_{uh}) \in L_{out}(u)$ is outdated,

that is $\delta_{uh} = d_G(u, h) \neq d'_G(u, h)$. This implies that the only shortest path from u to y in G includes arc

(x, y) (since the update is decremental we have that w(x, y) increases) and h is a hub vertex for pair u, h.

Hence we would have u is in the forward cover graph since there exist pairs $(h, \delta_{uh}) \in L_{out}(u)$ and $(h, \delta_{hh} = 0) \in L_{in}(h)$ such that $\delta_{uh} + \delta_{hh} = d_G(u, h) = d_G(u, x) + w(x, y)$, which is a contradiction.

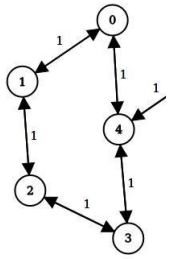
The proof for the backward version is symmetric. □

Corollary 1. *For any pair* $u, v \in V$ *such that* $u \notin F^L_G(x, y)$ *and* $v \notin B^L_G(x, y)$ *the pair also for* G^o*, that is* $Q(u, v, L) = d_G(u, v) = d_{G^o}(u, v)$.

Proof. Consider that for any pair $u, v \in V$, since L is a 2-HOP-COVER labeling, we have $Q(u, v, L) = d_G(u, v)$. Moreover, having $u \notin F^L_G(x, y)$ and $v \notin B^L_G(x, y)$ implies both $L_{out}(u)$ and $L_{in}(v)$ contain only correct entries for G^o. Therefore $Q(u, v, L) = d_G(u, v) = d_{G^o}(u, v)$. □

Hence, regardless of the magnitude of the change on (x, y), we are sure that the cover property is satisfied by L also in G^o, for pairs u, v such that $u \notin F^L_G(x, y)$ and $v \notin B^L_G(x, y)$ and we do not need to search for outdated entries in outgoing labels of vertices of the forward cover graph nor in incoming labels of vertices of the backward cover graph.

Now, we establish how to determine which entries are outdated in the labels of the vertices of the two cover graphs. Please note that there can be labels of vertices of the two cover graphs that do not contain outdated entries. Still, we will show it is necessary to identify such vertices since they might be interested by violations of the cover property due to the removals of the above-mentioned outdated entries. This heavily depends on the structure of the shortest paths in the graph and on the effect of the decremental update on such structure, as shown in the example of Figure 4.



	{(4,1),(0,0)}	{(4,1),(0,0)}
	{(4,2),(0,1)} {(3,2),(1,0)}	{(4,2),(0,1)} {(3,2),(1,0)}
	{(4,2),(0,2),(3,1)} {(1,1),(2,0)}	{(4,2),(0,2),(3,1)} {(1,1),(2,0)}
	{(4,1),(3,0)}	{(4,1),(3,0)}
	{(4,0)}	{(4,0)}
	{(4,1),(5,0)}	{(4,1),(5,0)}
	{(4,2),(5,1),(6,0)}	{(4,2),(5,1),(6,0)}

Figure 4. Consider again the graph of Figure 1 and a corresponding 2-HOP-COVER labeling. Assume arc (4, 0) is removed. We have that vertex 3 is in $F^L_G(0, 1)$ and vertex 0 is in $B^L_G(0, 1)$. However, there is no outdated label entry in $L_{out}(3)$ but the decremental operation on arc (4, 0) causes the cover property to be broken for pair (3, 0).

Property 3. *Let* $F^L_G(x, y)$ *and* $B^L_G(x, y)$ *be the forward and backward cover graphs, respectively. Let* L *be a well-ordered* 2-HOP-COVER *labeling of* G. *Assume arc* (x, y) *is subject to a decremental operation and let* G^o *be the graph obtained by applying to* G *such modification. Let* $H_f(x, y) \subseteq V$ *and* $H_b(x, y) \subseteq V$ *be the sets of forward and backward invalid hubs. Then:*

1. *a label entry* (h, δ_{vh}) *in the outgoing label* $L_{out}(v)$ *of a vertex* $v \in F^L_G(x, y)$ *is outdated if and only if* $h \in H_b(x, y)$ *and there is no vertex* $u \in N_{out}(v)$ *such that: (i)* $(h, \delta_{uh}) \in L_{out}(u)$*; (ii)* $w(v, u) + \delta_{uh} = \delta_{vh}$*; (iii)* $u \notin F^L_G(x, y)$*;*
2. *a label entry* (h, δ_{hv}) *in the incoming label* $L_{in}(v)$ *of a vertex* $v \in B^L_G(x, y)$ *is outdated if and only if* $h \in$

   $H_f(x, y)$ *and there is no vertex* $u \in N_{in}(v)$ *such that: (i)* $(h, \delta_{hu}) \in L_{in}(u)$*; (ii)* $w(u, v) + \delta_{hu} = \delta_{hv}$*; (iii)* $u \in 6\ B^L_G(x, y)$.

Proof. We focus on case 1. The proof for case 2 is symmetric.

(⇒) First of all observe that if $L_{out}(v)$ contains an outdated label entry (h, δ_{vh}) then, by Lemma 2, we must have $v \in F^L_G(x, y)$ and hence we know that $d_G(v, h) = d_G(v, x) + w(x, y) + d_G(y, h)$ and $\delta_{vh} = d_G(v, h) = 6\ d_G(v, h)$, i.e., v is connected in G to y via a shortest path that includes (x, y). By contradiction, we now assume that (h, δ_{vh}) is outdated but one of the two following conditions is true:

1. $h \in 6\ H_b(x, y)$;
2. $h \in H_b(x, y)$ and there is a vertex $u \in N_{out}(v)$ such that: (i) $(h, \delta_{uh}) \in L_{out}(u)$; (ii) $w(v, u) + \delta_{uh} = \delta_{vh}$; (iii) $u \in 6\ F^L_G(x, y)$.

Case 1. If $h \in 6\ H_b(x, y)$, we have that h is not a hub vertex for pair x, h in L. This implies that there exists another hub vertex h^o that covers pair x, h which in turn implies that:

- either $d_G(x, h) = d_G(x, h^o) + d_G(h^o, h) < w(x, y) + d_G(y, h)$;
- or $d_G(x, h) = d_G(x, h^o) + d_G(h^o, h) = w(x, y) + d_G(y, h)$ but h is not hub vertex (that is h does not belong to $L_{out}(x)$) since h^o precedes both h and x in the vertex ordering ($h^o < h$ and $h^o < x$) and since L is well-ordered.

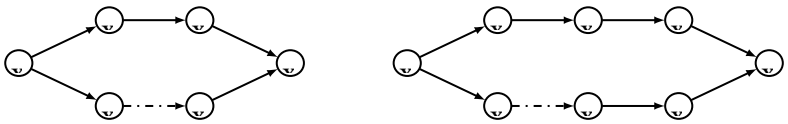
In both sub-cases we obtain a contradiction, as (h, δ_{vh}) is not outdated (the decremental operation does not change the value of δ_{vh}).

Case 2. We have that there exists a neighbor of v, namely u, that is not in $F^L_G(x, y)$. Hence, its outgoing label, by Property 2, does not contain any incorrect entry and we have $\delta_{uh} = d_G(u, h) = d_{G^o}(u, h)$ for any $(h, \delta_{uh}) \in L_{out}(u)$. Therefore, as $(v, u) \in A$ and the weight of (v, u) does not change in G^o, we have that $d_G(v, h) = w(v, u) + d_G(u, h) = d_{G^o}(v, h)$ and thus the label entry δ_{vh} is not outdated, which is a contradiction.

(⇐) Again by contradiction we assume that there is a non-outdated (correct) entry $(h, \delta_{vh}) \in L_{out}(v)$ while $h \in H_b(x, y)$ and there is no vertex $u \in N_{out}(v)$ such that: (i) $(h, \delta_{uh} \in L_{out}(u)$; (ii) $w(v, u) + \delta_{uh} = \delta_{vh}$; (iii) $u \notin F^L_G(x, y)$. Since $v \in F_G{}^L(x, y)$ and $h \in H_b(x, y)$ we know that $d_G(v, h) = d_G(v, x) + d_G(x, h) = d_G(v, x) + w(x, y) + d_G(y, h)$. Now, since the entry is correct for G^o, there must exist another shortest path, say P, in G that: (i) does not contain (x, y) and such that $w(P) = d_G(v, h)$. Call u^o the neighbor of v on this second shortest path.

Since there is no neighbor u such that $(h, \delta_{uh}) \in L_{out}(u)$ and $w(v, u) + \delta_{uh} = \delta_{vh}$, it follows that h 6 (h, δ_{uh}) 6$\in L_{out}(u^o)$.

It follows that pair u^o, h is covered by some other vertex, say h^o, such that $h^o < h$ and $h^o < u^o$. Since $(h, \delta_{vh}) \in L_{out}(v)$ we have also that h must be preceding h^o in the vertex ordering (i.e., $h < h^o$) as otherwise the visit rooted at h would have not reached v (it would have been pruned thanks to h^o). This is a contradiction since for the pruning on P to happen h^o must precede h. An explanatory example of this scenario is shown in Figure 5a. □



(**a**) (**b**)

Figure 5. (**a**): an example of the contradiction reached in the proof of Property 3. Suppose vertex ordering is $\{x_0, x_1, x_2, \dots, x_5\}$ and all arcs weight 1 for the sake of the example. Therefore, in any minimal well-ordered 2-HOP-COVER labeling entry $(x_1, \delta_{x_4x_1})$ cannot belong to $L_{out}(x_4)$. This allows
Algorithm CLEAN to avoid removing correct entries. (**b**): a case where a correct entry is preserved by Algorithm CLEAN. Suppose the vertex ordering is $\{x_0, x_1, x_2, \dots, x_7\}$ and all arcs weight 1 for the sake of the example. Therefore in any minimal well-ordered 2-HOP-COVER labeling, $L_{out}(x_4)$ will contain
$(x_0, \delta_{x_4x_0})$ with $\delta_{x_4x_0} = d(x_4, x_0) = 4$. Clearly, also $L_{out}(x_1)$ and $L_{out}(x_2)$ will contain. similarly, entries $(x_0, 3)$ and $(x_0, 3)$, respectively. If arc (x_2, x_5) undergoes some decremental update, the (correct) entry in $L_{out}(x_4)$ is not removed thanks to the presence of the alternative shortest path through x_1
with the same weight as that through x_2.

To summarize, Properties 2–3 provide a theoretical basis upon which we design an effective algorithm for removing outdated entries, and for detecting pairs of uncovered vertices in G^o, as we know that: (i) to find and remove (all and only) outdated entries it suffices to scan labels $L_{out}(v)$ ($L_{in}(v)$, resp.) of vertices $v \in F^L_G(x, y)$ ($v \in B^L_G(x, y)$, resp.) and to search for entries in the form (h, δ_{vh}) such that $h \in H_b(x, y)$ ((h, δ_{hv}) such that $h \in H_f(x, y)$, resp.); (ii) to find pairs of vertices u, v whose cover property can be broken in G^o due to removals of outdated entries, it suffices to determine vertices of $F^L_G(x, y)$ and $B^L_G(x, y)$,

Therefore, the strategy of algorithm CLEAN is divided in two steps. First, it computes sets $H_f(x, y)$ and $H_b(x, y)$, according to the definition (line 1 of Algorithm 3), and then it performs two suited visits of the input graph G (Algorithms 5 and 6) with the two-fold aim of identifying the vertices of the two cover graphs and removing outdated entries. These two sets are stored to evaluate, in a subsequent phase, whether the cover property in the new graph is satisfied or not.

Algorithm 3: Algorithm CLEAN

Input: Graph G, 2-HOP-COVER labeling L of G, arc (x, y) subject to decremental update.

Output: Resulting graph G^o, a labeling L^o containing only correct label entries for G^o (but possibly not covering it).

1 Compute $H_f(x, y)$ and $H_b(x, y)$;

2 (o_removal, OUT) ← FCLEAN(G, L, x, y, $H_b(x, y)$);

3 (i_removal, IN) ← BCLEAN(G, L, x, y, $H_f(x, y)$);

4 Apply update to arc (x, y) to obtain G ; **5** L^o ← L;

6 **foreach** v ∈ OUT **do** /* Remove outdated entries from outgoing labels */

7 **foreach** h ∈ *o_removal* **do** Remove (h, ∗) from L_{out}^o (v);

8 **foreach** v ∈ IN **do** /* Remove outdated entries from incoming labels */

9 **foreach** h ∈ *i_removal* **do** Remove (h, ∗) from L_{in}^o (v);

10 return (G^o, L^o, OUT, IN)

Specifically, each of the two visits start from the endpoints of the arc that is interested by the update (x and y, respectively), as shown in Algorithms 5 and 6, and proceed by determining vertices of the cover graphs on the basis of the corresponding conditions (see Definitions 1 and 2).

To limit the search to such vertices only, each visit employs a priority queue to drive the exploration in a shortest-path first fashion, starting from either x or y, and performs corresponding relax operations (see procedure 7 used in both Algorithms 5 and 6). This guarantees that vertices of the cover graphs are analyzed in order of distance from x or y and hence that entries are removed only if outdated.

In fact, whenever a vertex v of the forward (backward, resp.) cover graph is found, its outgoing (incoming, resp.) label is searched for entries in the form (h, ∗) for every h ∈ H_b(x, y) (h ∈ H_f(x, y), resp.). If any entry of this kind is present, then the algorithm determines whether the entry is correct or not for G^o by looking at the outgoing (incoming, resp.) neighbors' labels to establish whether there exists a second shortest path in G from v to h (from h to v, resp.), which is sufficient for the label entry to be correct in G^o (see Figure 5b).

If the search fails, then the entry is outdated and hence it is selected for removal (see line 12 of

Algorithm 5 and line 12 of Algorithm 6) by adding the hub to a corresponding list (one per vertex, see line 17 of Algorithm 5 and line 17 of Algorithm 6). Otherwise, it is correct and hence it is preserved (see, e.g., line 13 of Algorithm 5). Once all vertices of the cover graphs have been found, and their label sets searched to select outdated entries, the visit terminates and both the lists of hubs corresponding to outdated entries and sets of vertices of the cover graphs are returned.

Algorithm 4: Algorithm RECOVER

Input: Graph G^o, Labeling L^o, sets OUT and IN

Output: 2-HOP-COVER labeling L^{oo} of G^o

```
2   Q ← ∅;
3   L'' ← L';
4   if h ∈ IN then
5       foreach s ∈ OUT : s > h do        /* s < h ⟹ h cannot be added to L_out(s) */
6           δ_min ← ∞;
7           foreach v_i ∈ N_out(s) : v_i ∉ OUT do
8               if h < v_i and (h, δ_{v_i t}) ∈ L''_out(v_i) then
9                   δ_min ← min{δ_min, δ_{v_i t}};
10          mark[s] ← true;
11          Q.Insert(⟨s, δ_min⟩);
12      while Q ≠ ∅ do
13          (v, δ) ← Q.deleteMin();
14          if δ < PQ(v, h, L'', IDX(h)) then
15              L''_out(v) ← L''_out(v) ∪ {(h, δ)};
16              PRUNEDRELAX(v, N_in(u), true, OUT);
17  if h ∈ OUT then
18      foreach t ∈ IN : t > h do         /* t < h ⟹ h cannot be added to L_in(t) */
19          δ_min ← ∞;
20          foreach v_i ∈ N_in(t) : v_i ∉ IN do
21              if h < v_i and (h, δ_{v_i h}) ∈ L''_in(v_i) then
22                  δ_min ← min{δ_min, δ_{v_i h}};
23          mark[t] ← true;
24          Q.Insert(⟨t, δ_min⟩);
25      while Q ≠ ∅ do
26          (v, δ) ← Q.deleteMin();
27          if δ < PQ(h, v, L'', IDX(h)) then
28              L''_in(v) ← L''_in(v) ∪ (h, δ);
29              PRUNEDRELAX(v, N_out(u), false, IN);
```

1 foreach h ∈ {IN ∪ OUT} *in ascending order of* IDX(h) **do**

The actual removal of outdated entries is performed then at this point, once the two visits are concluded, by sequentially scanning only the labels of vertices of the cover graphs that contain at least one outdated label entry (see lines 7–9 of Algorithm 3).

This is done for the sake of the efficiency, so to be able to test the membership of vertices of the cover graph and the presence of outdated entries in linear time, by exploiting the content of the 2-HOP-COVER labeling L of G (see, e.g., line 18 of Algorithm 5 or line 18 of Algorithm 6). The naive alternative to achieve the same purpose would be executing a traditional shortest-path algorithm (with possibly superlinear worst-case running time, e.g., Dijkstra's).

We conclude the section by proving some results concerned with the correctness of Algorithm CLEAN. In particular, first we show the two sets OUT and IN, computed by the algorithm, are exactly the two vertex sets of $F^{L}_{G}(x, y)$ and $B^{L}_{G}(x, y)$.

Algorithm 5 ProcedureECLEAN

Input: Graph G, 2-HOP-COVER labeling L of G, arc (x, y) subject to update, set $H_b(x, y)$. **Output:** Set o_removal of entries to remove from outgoing labels, set OUT.

1 $Q \leftarrow \emptyset$;

2 foreach $v \in V$ **do**

3 mark[v] $\leftarrow$ false;

4 o_removal[v] $\leftarrow \emptyset$;

5 mark[x] $\leftarrow$ true;

```
6  Q.Insert(⟨x, w(x,y)⟩);
7  while Q ≠ ∅ do
8      (v, δ) ← Q.deleteMin();
9      foreach h ∈ H_b(x,y) : h ∈ L_out(v) do OUTDATED[h] ← true;      /* Removal Test */
10     foreach u ∈ N_out(v) : u ∉ OUT do
11         foreach h ∈ H_b(x,y) : h ∈ L_out(v) do
12             if h ∈ L_out(u) ∧ δ_vh = w(v,u) + δ_uh then
13                 OUTDATED[h] ← false;
14                 break;
15     foreach h ∈ H_b(x,y) : h ∈ L_out(v) do
16         if OUTDATED[h] = true then                    /* Selected for Removal */
17             o_removal[v] ← o_removal[v] ∪ {h};
18     if {h' ∈ V : h' ∈ L_out(v) ∧ h' ∈ L_in(y), δ_vh' + δ_h'y = δ} ≠ ∅ then   /* Membership to
        Forward Cover Graph */
19         OUT ← OUT ∪ {v};
20         RELAX(δ, v, N_in(u), false);
21 return (o_removal, OUT);
```

Algorithm 6: Procedure BCLEAN

Input: Graph G, 2-HOP-COVER labeling L of G, arc (x, y) subject to update, set H_f(x, y). **Output:** Set i_removal of entries to remove from incoming labels, set IN.

1 $Q \leftarrow \emptyset$;

2 foreach $v \in V$ **do**

3 mark[v] $\leftarrow$ false;

4 i_removal[v] $\leftarrow \emptyset$;

5 mark[y] $\leftarrow$ true;

```
Q.Insert(⟨y, w(x,y)⟩);
while Q ≠ ∅ do
    (v, δ) ← Q.deleteMin();
    foreach h ∈ H_f(x,y) : h ∈ L_in(v) do OUTDATED[h] ← true;        /* Removal Test */
    foreach u ∈ N_in(v) : u ∉ IN do
        foreach h ∈ H_f(x,y) : h ∈ L_in(v) do
            if h ∈ L_in(u) ∧ δ_hv = δ_hu + w(v,u) then
                OUTDATED[h] ← false;
                break;
    foreach h ∈ H_f(x,y) : h ∈ L_in(v) do
        if OUTDATED[h] = true then                                  /* Selected for Removal */
            i_removal[v] ← i_removal[v] ∪ {h};
    if {h' ∈ V : h' ∈ L_out(x) ∧ h' ∈ L_in(v), δ_xh' + δ_h'v = δ} ≠ ∅ then    /* Membership to
     Backward Cover Graph */
        IN ← IN ∪ {v};
        RELAX(δ, v, N_out(u), true);
return (i_removal, IN);
```

Algorithm 7: Sub-Procedure RELAX used in Procedures FCLEAN and BCLEAN

```
Input: Value δ extracted from the queue, Vertex v, neighbor set N, Boolean OUTGOING
if OUTGOING = true then
    foreach u ∈ N do
        if ¬mark[u] then
            QInsert (⟨u, δ + w(v,u)⟩);
            mark[u] ← true ;
        else if Qkey(u) > δ + w(v,u) then
            QdecreaseKey(⟨u, δ + w(v,u)⟩);
else
    foreach u ∈ N do
        if ¬mark[u] then
            QInsert (⟨u, δ + w(u,v)⟩);
            mark[u] ← true ;
        else if Qkey(u) > δ + w(u,v) then
            QdecreaseKey(⟨u, δ + w(u,v)⟩);
```

Lemma 1. *Let* L *be a minimal well-ordered* 2-HOP-COVER *labeling of a graph* G. *Let us assume that an arc* (x, y) *of* G *undergoes a decremental update. Let* F^L_G(x, y) (B^L_G(x, y), *resp.) be the forward (backward, resp.) cover graph. Then, if a vertex* v *is in* F^L_G(x, y) *(in* B^L_G(x, y)*, resp.), we have that vertex* v *is added to set* OUT *by Algorithm 5 (to set* IN *by Algorithm 6, resp.).*

Proof. The proof is by induction on the number |Q| of deleteMin operations on the queue Q. We first focus on Algorithm 5. Consider that at least one vertex is always inserted in the queue, namely x, with a priority of w(x, y), the weight of the arc in G. Observe that |Q| is a monotonically increasing value.

Base case (|Q| = 1*).* The only time when we have |Q| = 1 is when Q = {x}. Hence v = x and this shows the inductive basis since, if x is in F^L_G(x, y) we have that $\{h^o \in V : h^o \in L_{out}(v) \wedge h^o \in L_{in}(y), \delta_{vh^o} + \delta_{h^o y} = \delta\} = 6 \emptyset$ with $\delta = \delta_{xx} + \delta_{xy} = d_G(v, y) = d_G(v, x) + w(x, y) = d_G(x, x) + w(x, y) = w(x, y)$.

Therefore, the vertex is added to OUT by the algorithm. Please note that x is always added to Q. *Inductive Hypothesis*. We assume the claim holds when |Q| = k, i.e., we have k extracted vertices that are in $F^{L}_{G}(x, y)$ and have been visited by Algorithm 5.

Inductive Step. We consider the case when |Q| = k + 1. Observe that the next extracted vertex v must be a neighbor of some vertex, say v^o, processed in one of the previous k iterations, as otherwise v would not be in Q. Moreover, $d_G(v, y) = d_G(v, v^o) + d_G(v^o, x) + w(x, y)$ as v is the one with minimum priority. Therefore, we have that, if vertex v is in $F^{L}_{G}(x, y)$, in line 18, set $\{h^o \in V : h^o \in L_{out}(v) \wedge h^o \in$

$L_{in}(y), \delta_{vh}o + \delta_{h}o_{y} = \delta\}$ must be not empty, with $\delta = \delta_{vh} + \delta_{hy}$ for some $h \in \{h^o \in V : h^o \in L_{out}(v) \wedge h^o \in L_{in}(y), \delta_{vh}o + \delta_{h}o_{y} = \delta\}$ (since any pair of vertices is covered by some hub h in L for G) and $d_G(v, y) = d_G(v, v^o) + d_G(v^o, x) + w(x, y) = d_G(v, h) + d_G(h, y)$. Thus, the claim holds. □

Then, we prove that all entries left in the labeling by Algorithm CLEAN are correct for G^o.

Lemma 2. *Let* L *be a minimal well-ordered* 2-HOP-COVER *labeling of a graph* G. *Let us assume that an arc*

(x, y) *of* G *undergoes a decremental update and let* G^o *be the resulting graph. Let* L^o *the labeling returned by Algorithm 3. Then, for any* $v \in V$, *all entries* $(h, \delta_{vh}) \in L'_{out}(v)$ *are correct for* G^o, *that is* $\delta_{vh} = d_G o(v, h)$. *Symmetrically, for any* $v \in V$, *all entries* $(h, \delta_{hv}) \in L'_{in}(v)$ *are correct for* G^o, *that is* $\delta_{hv} = d_G o(h, v)$.

Proof. By contradiction, assume that there exists some entry $(h, \delta_{vh}) \in L_{out}(v)$, for some $h \in V$, that does not correspond to a shortest path in G^o after executing Algorithm 3, i.e., such that $\delta_{vh} =6 d_G o(v, h)$. Since {x, y} undergoes a decremental update, we must have that $\delta_{vh} < d_G o(v, h)$, as otherwise a first contradiction would be reached (either a decremental update is not increasing the distance or $\delta_{vh} = d_G o(v, h)$).Moreover, we know $\delta_{vh} = d_G(v, h)$ since L is a 2-HOP-COVER labeling therefore v must be a descendant of x in a shortest-path arborescence of G rooted at h, as otherwise (h, δ_{vh}) would not be in $L_{out}(v)$. Hence, by Lemma 1, vertex v must be in the forward cover graph $F^{L}_{G}(x, y)$ and hence is processed by Algorithm 3 in line 19. Thus, two cases can occur: either $h \in H_f(x, y)$ or not. In the former case, the label entry is removed by Algorithm CLEAN, which is again a contradiction. In the latter case, instead, we have that the shortest path from x to h in G did not contain (x, y), hence $d_G(x, h) = d_G o(x, h)$. Now, on the one hand, if the shortest path from v to h in G included (x, y), we reach a contradiction. In fact, (h, δ_{vh}) is removed, since $d_G(v, h) = d_G(v, x) + w(x, y) + d_G(y, h)$ and hence $v \in F^{L}_{G}(x, y)$ and $h \in H_f(x, y)$. On the other hand, if the shortest path from v to h in G did not include (x, y), we obtain the contradiction of δ_{vh} being correct, since (x, y) not being on the shortest path from v to h implies $d_G(v, h) = d_G o(v, h)$. A symmetric argument can be used to show that all entries $(h, \delta_{hv}) \in L_{in}(v)$ are correct, for any $h \in V$, by considering the shortest−path arborescence of G^T and set $H_b(x, y)$. □

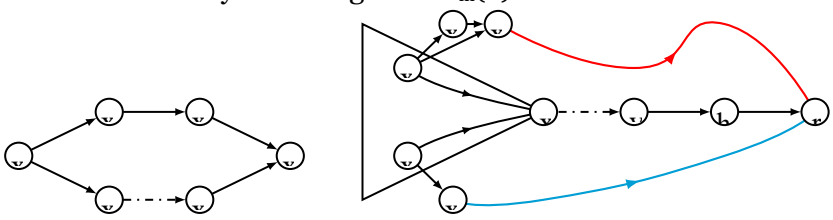
Finally, we prove a slightly stronger property, which is the minimality of removals. Specifically, we can prove that no correct label entry is removed by Algorithm 3, i.e., the number of removals is minimal to remove all entries that do not correspond to shortest paths

in G^o. In what follows, given two labelings L and L^o we use $L \setminus L^o$ to denote the set of label entries that are in L but not in L^o.

Lemma 3. *Let* L *be a minimal well-ordered* 2-HOP-COVER *labeling of a graph* G. *Let us assume that an arc*
(x, y) *of* G *undergoes a decremental update and let* G^o *be the resulting graph. Let* L^o *the labeling returned by Algorithm 3. Then,* $L \setminus L^o$ *does not contain any label entry that is correct for* G^o.

Proof. Assume by contradiction a correct label entry (h, δ_{vh}) is removed from $L_{out}(v)$, that is $L \setminus L^o$ `contains, for some h` $\in$ `V, pair (h,` δ_{vh}) and $\delta_{vh} = d_G o(v, h)$. Call u the neighbor of v that induces the presence of such label entry in L which is well-ordered. Clearly we have that $\delta_{vh} = d_G(v, h) = w(v, u) + \delta_{uh}$ and $d_G(v, h) = w(v, u) + d_G(u, h)$ since there must exist a neighbor on the shortest path from v to h (which is traversed, e.g., by FS−2HC).

`Now, either the shortest path P from u to h contains arc (x, y) or not. In the latter case, we have that h` $\in$ $L_{out}(u)$ and $d_G(v, h) = \delta_{vh} = w(v, u) + \delta_{uh} = w(v, u) + d_G(u, h) = w(v, u) + d_G o(u, h) = d_G o(v, h)$. Therefore, in line 13 of Algorithm 5 the label entry is not selected for removal, which is a contradiction. In the former case, as the entry is correct for G^o, there must exist another shortest path, say P^o, that does not contain arc (x, y) and such that $w(P) = w(P^o) = d_G(u, h)$. Call u^o the neighbor of v on such path, i.e., $u^o \in N_{out}(v)$ and $d_G(v, h) = w(v, u^o) + d_G(u^o, h)$. Since the label entry is removed, it follows that Algorithm 5 in line 12 does not find any neighbor of u^o satisfying $\delta_{vh} = w(v, u^o) + \delta_u o_h$, i.e., u^o cannot have $(h, \delta_u o_h)$ in $L_{out}(u^o)$. Therefore, the visit rooted at h has not reached u^o due to some pruning test. It follows that pair $u^o h$ is covered (on P^o) by some other vertex, say h^o, such that $h^o < h$ and $h^o < u^o$. Since $(h, \delta_{vh}) \in L_{out}(v)$ we have also that h must be preceding h^o in the vertex ordering (i.e., $h < h^o$) as otherwise the visit rooted at h would have not reached v (it would have been pruned thanks to h^o). This is a contradiction since, for the pruning on P^o to happen, vertex h^o must precede h (an explanatory example of the scenario discussed in this proof is shown in Figure 6a). A symmetric argument can be used to prove that no correct label entry is removed from any incoming label $L_{in}(v)$. □



(a) **(b)**

Figure 6. (**a**): an example of the contradiction reached in the proof of Lemma 3. Suppose vertex ordering is $\{x_0, x_1, x_2, \ldots, x_5\}$ and all arcs weight 1. Therefore, in any minimal well-ordered 2-HOP-COVER labeling, entry $(x_1, \delta_{x4}x_1$ cannot belong to $L_{out}(x_4)$, due to the pruning test succeeding because of x_0. Also, in this case no correct entry is removed by Algorithm CLEAN. (**b**): restore phase for vertex r.

3.3. RECOVER *Phase*

The second phase of Algorithm QUEUE−2HC, named RECOVER, takes place after Algorithm CLEAN is concluded and considers sets OUT and IN with the aim of restoring the

(possibly broken) cover property for the new graph G^o. The procedure, shown in Algorithm 4, takes as input also G^o and the labeling returned by Algorithm CLEAN, say L^o, and works by resuming a series of forward and backward shortest-path visits, each considering as root one of the vertices in sets OUT and IN (i.e., vertices of forward and backward cover graphs), in the order imposed by the vertex ordering. Please note that in the remainder of the paper, given a vertex $h \in V$, we denote by IDX(h) the index of h in said ordering. Each visit, rooted at some vertex $h \in$ OUT, has the purpose of checking whether the cover property is satisfied for pairs h, v for any other $v \in$ IN and, in case it is not, to restore it. This is done by visiting only the vertices and the arcs of the forward and backward cover graphs, as discussed in the following. The above is possible thanks to the following property on the labeling, holding after the execution of Algorithm CLEAN.

Property 4. *Let* $F^L_G(x, y)$ *and* $B^L_G(x, y)$ *the forward and backward cover graphs, respectively. Let* L^o *be the labeling returned by Algorithm 3 when applied to* G*, to its* 2-HOP-COVER *labeling* L*, as a consequence of a decremental update on arc* $(x, y) \in A$. *Then, for any pair* $u, v \in V$ *such that* $u \notin F^L_G(x, y)$ *or* $v \notin B^L_G(x, y)$*, we have that labeling* L^o *covers pair* u, v *also for* G^o*, that is* $Q(u, v, L^o) = d_{G^o}(u, v)$.

Proof. By contradiction assume for a pair $u, v \in V$ such that $u \not\in F^L_G(x, y)$ (while v can be of any kind) we have that labeling L^o does not cover the pair for G^o, i.e., $Q(u, v, L^o) \neq d_{G^o}(u, v)$ (the case when u can be of any kind while $v \notin B^L_G(x, y)$ is symmetric). Notice that by Lemma 2 we know that entries in L^o are all correct for G^o. Now, only three cases can occur: (i) $v \in F^L_G(x, y)$; (ii) $v \in B^L_G(x, y)$; (iii) $v \not\in F^L_G(x, y)$ and $v \not\in B^L_G(x, y)$. Notice in fact that by the definition of cover graphs, we cannot have that $v \in F^L_G(x, y)$ and $v \in B^L_G(x, y)$.

Case (i). Both u and v are in $F^L_G(x, y)$ hence we know that the shortest path in G between u and v does not contain (x, y). Furthermore, it is straightforward to see that the shortest path from u to v is made of only vertices in $F^L_G(x, y)$ (by contradiction, otherwise u cannot be in $F^L_G(x, y)$). Hence the hub, say h covering the pair must be in $F^L_G(x, y)$ as well. Therefore h is not in $H_b(x, y)$ thus it cannot be removed from $L_{out}(u)$ by Algorithm 5. Moreover, $L_{in}(v)$ is not changed by Algorithm 5. Hence, we reach the contradiction of $Q(u, v, L) = Q(u, v, L^o) = d_G(u, v) = d_{G^o}(u, v)$.

Case (ii). We know the shortest path from x to v in G included arc (x, y). We can distinguish two

sub-cases: either the hub h covering pair u, v in G belongs to such path or not. In the former case, we have that $h \in H_f(x, y)$ and hence also the shortest path from u to h in G must include (x, y). This implies that $u \in F^L_G(x, y)$ since there must exist two pairs $(h', \delta_{uh'}) \in L'_{out}(u)$ and $(h', \delta_{h'y}) \in L'_{in}(y)$ such that $\delta_{uh^o} + \delta_{h^o y} = d_G(u, y)$ and $d_G(u, y) = d_G(u, x) + w(x, y)$ (that is there must exist an h that is a hub vertex for pair u, y) which is a contradiction. In the latter case, the shortest

path from u to h in G does not include (x, y) and neither the shortest path from h to v in G. Hence, by Lemma 3, correct entries (h, δ_{uh}) and (h, δ_{hv}) must be in L_{out}^{o} (u) and L_{in}^{o} (v), respectively, which implies the pair is covered and the contradiction is reached.

Case (iii). As with case (i), we have that the shortest path in G between u and v does not contain (x, y) and that both labels $L_{out}(u)$ and $L_{in}(v)$ are not changed by Algorithm 5. Hence, again we have the contradiction of with $Q(u, v, L) = Q(u, v, L^{o}) = d_G(u, v) = d_G o(u, v)$. □

By Property 4 it emerges that to guarantee the property holding for G^{o}, it suffices to restart some shortest−path visits as follows. Given a root vertex h ∈ IN (h ∈ OUT), all vertices v ∈ OUT (v ∈ IN, respectively) are first analyzed in order to find *viable neighbors* for h. A viable neighbor u ∈ $N_{out}(v)$ (u ∈ $N_{in}(v)$, respectively) is a neighbor such that: (a) h ∈ $L_{out}(u)$ (h ∈ $L_{in}(u)$, respectively) and (b) u 6∈ OUT (u 6∈ IN, respectively). For all such neighbors, we know that after the execution of Algorithm CLEAN, there exists at least one shortest path from u to h (from h to u, $\min_{\notin OUT,(h,\delta_{uh})\in L_{out}(u)}$ respectively) that did not contain (x, y) in G. Clearly, there can be many viable neighbors (see Figure 6b). Therefore, to correctly discover distances corresponding to shortest paths, the procedure selects the one with a minimum distance encoded in the label entry. In particular, if we call dmin ={δ_{uh} + w(v, u)} then the algorithm inserts v into a queue

u∈Nout(v),u

Q with priority dmin (see Algorithm 4). If no viable neighbor exists in G^{o}, then the vertex is not inserted in the queue. Observe that the search for viable neighbors can be accelerated by skipping vertices according to their ordering and to the well−ordered property (see lines 5–8 or lines 18–21) that guarantees that if v < u then u 6∈ $L_{in}(v)$ and u 6∈ $L_{out}(v)$.

Algorithm 8: Procedure PRUNEDRELAX used by Algorithm RECOVER

Input: Vertex v, neighbor set N, Boolean OUTGOING, vertex set T.

```
1  if OUTGOING = true then
2      foreach v_i ∈ N_in(v) : v_i ∈ T do
3          if ¬mark[v_i] then
4              mark[v_i] ← true;
5              Q.Insert(⟨v_i, δ + w(v_i, v)⟩);
6          else if δ + w(v_i, v) < Q.key(v_i) then
7              Q.decreaseKey(⟨v_i, δ + w(v_i, v)⟩);
8  else
9      foreach v_i ∈ N_out(v) : v_i ∈ T do
10         if ¬mark[v_i] then
11             mark[v_i] ← true;
12             Q.Insert(⟨v_i, δ + w(v, v_i)⟩);
13         else if δ + w(v, v_i) < Q.key(v_i) then
14             Q.decreaseKey(⟨v_i, δ + w(v, v_i)⟩);
```

Once all vertices in OUT (IN, respectively) has been treated as above, we have that Q contains vertices in OUT (IN, respectively) with at least one viable neighbor, i.e., connected to the vertex h under consideration through a path that is not including (x, y). If Q is empty, it follows that no such vertices exist, and hence it is easy to see that the decremental operation must be an arc removal that has increased the number of strongly connected components of G (otherwise we would have that at least vertex x has vertex y as viable neighbor). Therefore, vertices in OUT (IN, respectively) are not connected to (are not reachable from, respectively) h and hence the visit for h can terminate. If Q is not empty instead, the vertex v with minimum priority, say δ, is extracted and, if δ < PQ(v, h, L, IDX(h))

(or δ < PQ(h, v, L, IDX(h)), respectively) then new label entry (h, δ) is added to $L_{out}(v)$ ($L_{in}(v)$, respectively). In particular, if the value of δ is smaller than the distance returned by PQ, it follows that the shortest path being discovered is not encoded in the labeling, i.e., that pair v, h is not covered by

L. Please note that the pruning test considers vertices up to index IDX(h) in the ordering, as done by RESUME−2HC to preserve minimality of the labeling (see [8] for more details on *prefixal correctness*). Now, if a new label entry is added, then only the neighbors of v that are in OUT (IN, resp.) are scanned in a relaxation step (see Algorithm 8) to add them to the queue or to decrease their key. In fact, vertices outside OUT (IN, resp.) by Lemma 2 already satisfy the cover property to h. The algorithm terminates when the queue is empty for all vertices in OUT and IN. We are now ready to conclude the proof of correctness of Algorithm QUEUE−2HC.

Theorem 1. *Let* $F^{L}_{G}(x, y)$ *and* $B^{L}_{G}(x, y)$ *the forward and backward cover graphs, respectively. Let* L^{oo} *be the labeling returned by Algorithm 4 when applied to graph* G^{o}*, labeling* L^{o} *and set* OUT *and* IN*, respectively.*
Then, L^{oo} *is a minimal well-ordered* 2-HOP-COVER *labeling that covers* G^{o}*, i.e., for any pair* u, v ∈ V *we have* $Q(u, v, L^{oo}) = d_{G}o(u, v)$.

Proof. First of all, observe that any pair that can have the cover property not satisfied for G^{o} (see Property 4) is considered by Algorithm RECOVER, either in line 5 or in line 8.

We focus on proving the claim for all pairs h, t for a given vertex h ∈ OUT (the proof for the case h ∈ IN is symmetric). The proof is by induction on the number |Q| of deleteMin operations on the queue Q. Observe that if |Q| = 0 we have that no vertex t ∈ IN has at least one viable neighbor. It is easy to see that this implies none of such vertices are reachable, in G^{o}, from h in G^{o}. In fact, any viable neighbor, say w, is outside IN hence pair h, w is covered by L^{o} by Lemma 4. Moreover, all entries in L^{o} are correct. Thus, $d_{G}o(u, v) = Q(u, v, L^{oo})$ = and the claim holds.

Base case (|Q| = 1*).* In this case, we have that one vertex, say∞ w, has a viable neighbor and w > h. By Lemma 2 it follows that δ = $d_{G^{o}}$(h, w) and hence the cover property is tested in line 27. If δ is smaller than PQ(h, w, L^{oo}, IDX(h), a label entry (h, δ) is added to $L_{out}(w)$ thus the claim follows. Otherwise, we have that PQ(h, w, L^{oo}, IDX $(h) = d'_{G}(h, w)$ which, in turn, implies that $Q(h, w, L'') = d'_{G}(h, w)$ by the definition of PQ.

Inductive Hypothesis. We assume the claim holds when |Q| = k, i.e., we have Q(h, w, L^{oo}) = $d_{G^{o}}$(h, w) and w < h for any vertex t among the k vertices that have been extracted up to this point. *Inductive Step*. We consider the case when |Q| = k + 1. Observe that the next

extracted vertex v must be a neighbor of some vertex, say v^o, processed in one of the previous k iterations, as otherwise v would not be in Q. Therefore, we have that $\delta = d_{G^o}(h, v)$, as otherwise either v would not be extracted to be the minimum, or v^o would not be among the previously extracted vertices. Therefore, two cases can occur: either δ is smaller than $PQ(h, v, L^{oo}, IDX(h))$ or not. In the former case, a label entry (h, δ) is added to $L_{out}(w)$ thus the claim follows. In the latter case, again we have that $PQ(h, v, L^{oo}, IDX(h)) = d_{G^o}(h, v)$ which, in turn, implies that $Q(h, v, L^{oo}) = d_{G^o}(h, v)$ by the definition of PQ and this concludes the proof. □

3.4. Complexity of QUEUE–2HC

In this section, we provide the complexity analysis of QUEUE–2HC. We start by bounding the worst-case running time of Algorithm CLEAN. All results refer to a graph with n vertices and m arcs. **Theorem 2.** *Algorithm 3 runs in* $O(mn + n^2 \log n)$ *worst-case time per graph update.*

Proof. First of all, it is easy to observe line 1 takes linear time in the size of the two involved label sets
(that of x and that of y). In fact, computing $H_f(x, y)$ and $H_b(x, y)$ requires several comparisons, each taking O(1) time, that is equal to the size of the two label sets (that can be scanned in order regarding vertex ordering, as in Algorithm 1). Therefore, if we call |L| the size of the largest label (containing the largest number of pairs), then this step takes O(|L|) worst cast time, which is O(n).

Now, we bound the computational time of procedure FCLEAN, which is equal to that procedure BCLEAN (which operates symmetrically on the graph and on the labeling). Specifically, it is easy to see that the initialization phase of mark and o_removal costs O(n), while the cycle of lines 7–20 is simply a shortest-path driven visit of the forward cover graph, which mimics Dijkstra's algorithm (hence this would require $O(m + n \log n)$). The only difference consists of the tests of lines 9–17, whose purpose is to determine where entries are actually outdated, and in the condition for continuing the visit (membership to the cover graph) that is checked in line 18. Each execution of the former requires $O(|H_b(x, y)| \cdot |N_{out}(v)| = O(n)$ worst-case time every time a vertex v is dequeued while the latter costs O(|L|) per arc relax operation. Thus, the cost of this visit is upperbounded by $O(mn + n^2 \log n)$. Finally, removal of outdated entries (lines 6–9) requires, in the worst case, to execute two label set scans (once for the outgoing label and once for the incoming label) per vertex of the graph. Each scan costs $O(|L|) = O(n)$ thus the running time of this part is upperbounded by $O(n|L|) = O(n^2)$. □

Theorem 3. *Algorithm 4 requires* $O(n^3 \log n)$ *worst-case time per graph update.*

Proof. We bound the cost of lines 4–16, i.e., for handling the case of $h \in IN$. In fact, note that the running time of the procedure for handling the case of $h \in OUT$ is asymptotically the same. Now, observe that for each $h \in IN$, we first scan the vertices in OUT to search for viable neighbors. This costs $|N_{out}(v)||L| = O(n^2)$ for a vertex v, if again we call |L| the size of the largest label. Moreover, again for each $h \in IN$, we perform a shortest–path driven visit of the graph, whose worst–case running time is $O(m + n^2 \log n)$, since we need to add an extra $O(|L|) = O(n)$ factor whenever we

dequeue a vertex, for testing the cover property and possibly adding a label. Thus, for each vertex $h \in IN$ the worst−case running time is $O(m + n^2 \log n)$. Hence, since the number of vertices in IN is O(n), we obtain a total running time for the procedure of $O(mn + n^3 \log n) = O(n^3 \log n)$. □

Therefore, we can summarize the complexity of Algorithm QUEUE−2HC as follows.

Theorem 4. *Algorithm* QUEUE−2HC *requires* $O(n^3 \log n)$ *worst-case time per graph update.*

Proof. The claim follows by the proofs of Theorems 2−3. □

By the above, it is easy to notice that given a graph update, QUEUE−2HC in the worst-case is slower than recomputing the labeling from scratch (which takes $O(mn + n^2 \log n)$ worst-case time). However, we show in the next section that experimentally QUEUE−2HC is much faster in all tested instances. Regarding BIDIR−2HC, it is not trivial to understand, on a theoretical basis, whether asymptotically speaking its performance is better or worse than that of QUEUE−2HC, since the two time complexities depend on different parameters. In the experimental evaluation that follows, we hence also consider BIDIR−2HC and show that experimentally QUEUE−2HC is faster also than BIDIR−2HC, in all tested instances.

4. Experimentation

In this section, we present the experimental study we conducted to assess the performance of

QUEUE−2HC. In particular, we implemented both QUEUE−2HC, BIDIR−2HC and FS−2HC, and designed a test environment to evaluate all considered algorithms on given input graphs. More in details, among the general approaches that go under the name of FS−2HC, we implemented the version proposed in [7] where the ordering is fixed during the computation of the labeling, while for BIDIR−2HC, we implemented both its two versions, given in [26], and selected the fastest for comparisons. Regarding the vertex ordering, we consider either the degree or an approximation of the betweenness, as in [26]. Specifically, for each graph, for the sake of fairness, we selected the ordering yielding faster preprocessing (and hence the most compact labeling) among the two.

Setup and Inputs. Our entire framework is based on NetworKit [35], a widely adopted open-source toolkit for implementing graph algorithms and performing various types of network analyses. All code has been written in C++ and compiled with GNU g++ v.7.4.0 (O3 opt. level) under Linux (Kernel 5.3.0-53). Implementations of BIDIR−2HC and QUEUE−2HC are both sequential, while FS−2HC can exploit core-level parallelism for unweighted instances [8]. All tests have been executed on a workstation equipped with an Intel ® Xeon ® CPU E5-2643 v3 3.40GHz, 128 GB of RAM of type DIMM Synchronous 2133 MHz (0.5 ns), and three levels of cache (384KiB L1 cache, 1536KiB L2 cache, 20MiB L3 cache). As input to our tests, inspired by other studies on the subject [4,7,8,14,17,26,27,36], we used a large set of both real-world datasets, taken from known publicly available repositories, such as SNAP [37], NetworkRepository [38] and KONECT [39], and synthetic graphs, randomly generated via well-known algorithms for random graph generation, such as the *Erdo˝s-Rényi* model [40] and the Barabási−Albert model [41]. We considered both directed/undirected graphs

and weighted/unweighted ones. All details of the tested inputs can be found at the corresponding URLs, while sizes and main characteristics are given in Table 1.

Table 1. Overview of the used input graphs. The first and second columns reports name of the dataset and type of the corresponding network. Third and fourth columns show number of vertices and arcs of the graph, while the fifth column reports the average vertex degree. The last three columns, namely **S**, **D**, and **W**, respectively, indicate whether the graph is synthetic or real-world and whether it is directed/weighted or not, respectively (● = true, ○ = false). Graphs are ordered according to their number of arcs, non-decreasing.

Dataset	Network Type	|V|	|E|	avg deg	S	D	W
CAIDA (CAI)	ETHERNET	3.20×10^4	4.01×10^4	2.51	○	○	●
LUXEMBOURG (LUX)	ROAD	3.06×10^4	7.55×10^4	4.11	○	●	●
WGTGNUTELLA (GNU)	PEER2PEER	6.26×10^4	1.48×10^5	4.73	○	●	●
BRIGHTKITE (BKT)	LOCATION-BASED	5.82×10^4	2.14×10^5	7.35	○	○	○
EFZ (EFZ)	RAILWAY	1.25×10^5	4.02×10^5	6.43	○	●	●
EU-ALL (EUA)	EMAIL	2.65×10^5	4.19×10^5	2.77	○	●	○
EPINIONS (EPN)	SOCIAL	1.32×10^5	8.41×10^5	12.76	○	●	○
BARABÁSI-A. (BAA)	SYNTHETIC (Power-Law)	6.32×10^5	1.00×10^6	3.17	●	○	●
WEB-NOTREDAME (NTR)	HYPERLINKS	3.26×10^5	1.09×10^6	6.69	○	○	○
NETHERLANDS (NLD)	ROAD	8.92×10^5	2.28×10^6	5.11	○	●	●
YOUTUBE (YTB)	SOCIAL	1.13×10^6	2.99×10^6	5.26	○	○	○
WIKITALK (WTK)	COMMUNICATION	2.39×10^6	5.02×10^6	4.19	○	●	○
HUMAN-GENOME (BIO)	BIOLOGICAL	1.43×10^4	9.03×10^6	1262.94	○	○	●
AS-SKITTER (SKI)	COMPUTER	1.70×10^6	1.11×10^7	13.08	○	○	○
DBPEDIA (DBP)	KNOWLEDGE	3.97×10^6	1.29×10^7	6.97	○	●	○
ERDŐS-RÉNYI (ERD)	SYNTHETIC (Uniform)	1.00×10^4	2.50×10^7	2499.11	●	○	●

Experiments. To assess the performance of QUEUE–2HC against BIDIR–2HC and FS–2HC, we performed a set of experimental trials that aim at modeling the real-world scenarios where the studied algorithms are supposed to be employed. Specifically, we assume we are given a time-evolving network, subject to decremental modifications, and we need to mine distances between vertices of the graph. Hence, our experimental setup is as follows: for each input graph we start by computing an initial 2-HOP-COVER labeling via FS–2HC. Then, the graph undergoes a sequence of $k = 200$ randomly selected decremental updates (either arc removals or arc weight increases). After each graph modification, we update the labeling (either via BIDIR–2HC or QUEUE–2HC) and measure the computational time required by the dynamic algorithm. Furthermore, we also execute FS–2HC on the modified graph to recompute a 2-HOP-COVER labeling from scratch and again measure the required running time. For each of the obtained labelings we measure the *size*, that is the number of label entries, and the *average query time*, i.e., the average time for answering a query via the labeling for $q = 10^6$ randomly selected pairs of vertices of the graph (in microseconds). This latter step is done both for the sake of correctness, to test that the labelings return the same

values of distances, and for evaluating the quality of the labelings updated via the dynamic algorithms, against that of the labeling recomputed via FS–2HC, as in [26]. In fact, it is known that compact labelings yield better performance in terms of average query times [3].

At the end of the k updates, statistic measures of central tendency and dispersion (e.g., mean, median, interquartile range, standard deviation) are computed for all observed performance indicators. The results of our experimentation are summarized in Table 2, where we compare the running time of

QUEUE–2HC against that of BIDIR–2HC and FS–2HC, respectively, by showing both mean and median values, as central tendency measures. Dispersion of (some of) the samples is shown through the scatterplots of Figure 7 where we report all observed values of running time for a meaningful subset of the tested inputs. Please note that we omit the measures of average labeling size and average query times, since they are essentially identical for the three computed labelings (that recomputed from scratch and the two obtained via dynamic algorithms). This is expected by the theoretical results, since both the dynamic algorithms and FS–2HC produce minimal labelings (small, negligible variations are observed due to ties in the vertex ordering, which are broken arbitrarily). Notice also that our choice of considering randomly selected updates is dictated by the fact that unfortunately, datasets containing real-world graphs with real-world sequences of decremental updates are not so easy to find in publicly available repositories. However, we employed quite large sequences of updates and this is considered rather effective in capturing the typical performance of this kind of algorithms, as well documented in the literature [42,43]. Clearly, the larger the size of the sequences, the better is with respect to variance in the observed results. To the purpose, in our case, the value of k is selected so to obtained observed average running times that are quite stable between different runs considering different sequences.

Table 2. Experimental results: average update time of QUEUE–2HC against that of BIDIR–2HC and

FS–2HC. For each network, we report both mean and median values. The column named PARALLEL indicates whether FS–2HC is accelerated by executing part of its code in parallel among the cores (for unweighted instances [8], ● = true, ○ = false).

Dataset		AVERAGE UPDATE TIME PER GRAPH MOD (S)					
		FS–2HC		BIDIR–2HC		QUEUE–2HC	
	PARALLEL	MEAN	MEDIAN	MEAN	MEDIAN	MEAN	MEDIAN
CAI	○	4.89×10^{-1}	4.75×10^{-1}	2.14×10^{-1}	2.46×10^{-1}	**3.77×10^{-2}**	**4.18×10^{-2}**
LUX	○	4.99×10^{0}	4.99×10^{0}	9.46×10^{-1}	1.05×10^{0}	**2.24×10^{-1}**	**9.36×10^{-2}**
GNU	○	7.83×10^{1}	7.84×10^{1}	1.30×10^{1}	7.82×10^{0}	**4.64×10^{0}**	**1.43×10^{-1}**
BKT	●	1.23×10^{1}	1.23×10^{1}	2.28×10^{0}	3.06×10^{0}	**3.90×10^{-1}**	**2.68×10^{-1}**
EFZ	○	3.97×10^{2}	3.96×10^{2}	4.83×10^{1}	3.65×10^{1}	**3.30×10^{1}**	**1.58×10^{0}**
EUA	●	1.25×10^{1}	1.25×10^{1}	1.18×10^{0}	1.56×10^{0}	**1.51×10^{-1}**	**9.34×10^{-2}**
EPN	●	8.28×10^{1}	8.28×10^{1}	7.67×10^{0}	9.30×10^{0}	**3.34×10^{0}**	**5.69×10^{-1}**
BAA	○	1.56×10^{2}	1.56×10^{2}	9.42×10^{1}	1.28×10^{2}	**4.11×10^{0}**	**3.27×10^{0}**
NTR	●	1.23×10^{2}	1.23×10^{2}	1.37×10^{0}	1.93×10^{0}	**1.33×10^{0}**	**5.16×10^{-1}**

Table 2. *Cont.*

Dataset		AVERAGE UPDATE TIME PER GRAPH MOD (S)						
		FS–2HC	BIDIR–2HC			QUEUE–2HC		
	PARALLEL	MEAN	MEDIAN	MEAN	MEDIAN	MEAN	MEDIAN	
NI	○	7.73×	7.76×	1.55×	1.81×	**3.06×**	**7.22×**	
VT	●	6.94×	6.94×	1.43×	1.57×	**1.58×**	**7.19×**	
WT	●	7.48×	7.48×	3.61×	2.84×	**2.14×**	**7.60×**	
BI	○	3.05×	3.05×	1.60×	5.28× –	**7.36×** –	**1.59×** –	
SK	●	2.21×	2.21×	3.67×	4.01×	**1.23×**	**3.02×**	
DB	●	2.44×	2.44×	7.65×	6.18×	**3.19×**	**1.56×**	
ER	○	9.48×	9.47×	**1.83×** –	**1.02×** –	5.08× –	5.36× –	

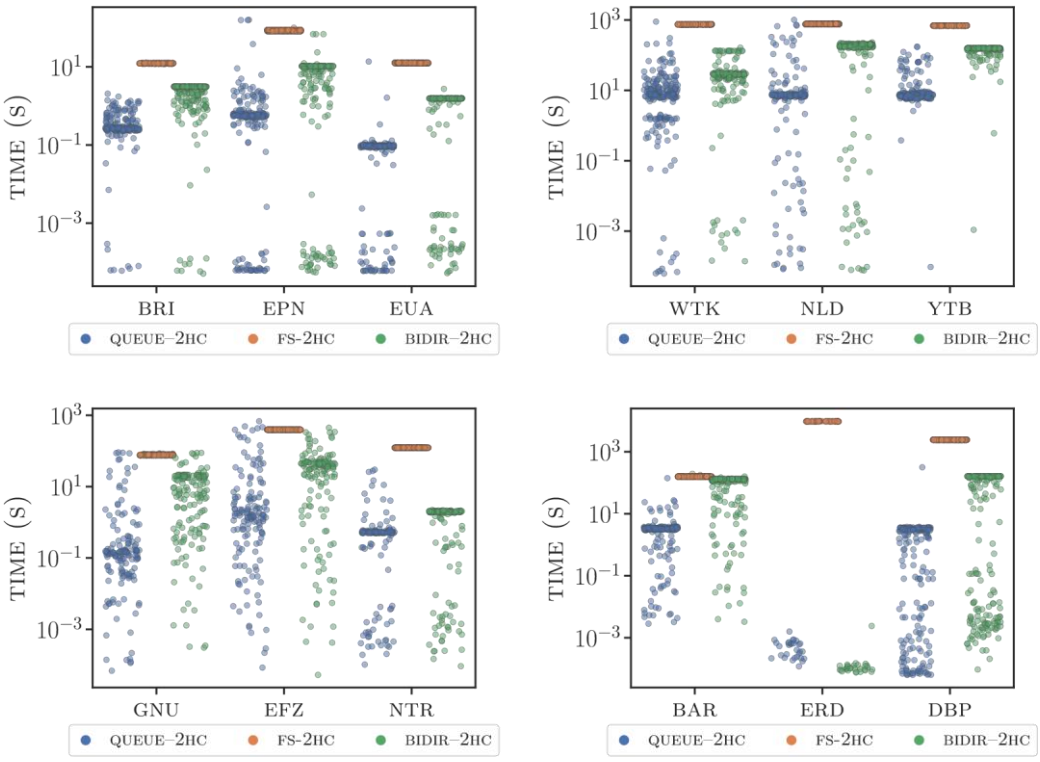


Figure 7. Running times of FS–2HC, BIDIR–2HC and QUEUE–2HC on some of the considered inputs. The y-axis is log-scaled to magnify the differences.

Analysis. Our experiments' main outcomes are essentially two. First and foremost, we observe that QUEUE–2HC worst-case analysis deviates from the performance in practice. In fact, despite the super-cubic worst-case time per update, QUEUE–2HC is always much faster than FS–2HC in our experiments, up to several orders of magnitude faster. Moreover, it is also faster than BIDIR–2HC, on average. Of particular interest are the improvements

observed in weighted instances, where BIDIR–2HC is not particularly effective and slow enough to be close to FS–2HC (see instances NLD or
LUX). More specifically, QUEUE–2HC achieves superior performance with respect to BIDIR–2HC update times, up to several orders of magnitude smaller update times (see, e.g., BAA or BIOinstances). The only exception is graph ERD, where the two solutions behave similarly and both exhibit extremely low update times, compared to the time spent by FS–2HC. This might be due to the fact that a small number of operations suffice to update the labeling in this instance (as the graph is quite dense). Hence both solutions are very effective, likely close to an optimal behavior, and the difference is just induced by the determination of the cover graphs. For the sake of fairness, similar behavior is observed in instance NTR, where however QUEUE–2HC is slightly faster.
Thus, on the whole, the performance of QUEUE–2HC is much closer to that of the very efficient
RESUME–2HC for incremental updates [8], which is considered suited to be applied to mine distances from massive time-evolving graphs in real-time contexts [27]. This suggests QUEUE–2HC as well can be considered a reasonable solution for real-time contexts, and that in any case should the preferred option to be adopted in time-evolving scenarios when decremental graph operations must be handled.

5. Conclusions and Future Work

In this paper, we have introduced a new dynamic algorithm, named QUEUE–2HC, to update 2-HOP-COVER labelings, as a consequence of decremental updates, in time-evolving graphs. We have shown its correctness and analyzed its computational complexity in the worst case. Not surprisingly, the worst-case analysis does not capture well the true performance of the new algorithm. In fact, extensive experimentation we conducted, to assess the performance of QUEUE–2HC, shows QUEUE–2HC is very effective in practice, and outperforms by far the only other known solutions for the problem, i.e., the recomputation from scratch [7] and algorithm BIDIR–2HC [26]. Therefore,
we provide strong evidence that QUEUE–2HC can considered the reference solution to allow distance mining when decremental graph operations can occur in time-evolving scenarios.

To achieve a complete framework for handling efficiently distance queries in massive time-evolving graphs, future works following this study should concentrate on removing the main limitation of the only known algorithm able to handle incremental operations that is RESUME–2HC [8]. In fact, this algorithm ignores outdated entries that are induced by incremental modifications: this does not affect correctness but leads to degradation in query performance and to the need for periodic reprocessing (see [8,29] for more details).

Another interesting direction should be to extend the experimental evaluation to: (i) consider the greedy ordering of [3]; (ii) include larger inputs or real-world sequences of graph modifications; (iii) investigate the existence of relationships of correlation between graph features (e.g., density, planarity or regularity) and the performance of both 2-HOP-COVER labelings and the corresponding algorithms, namely FS–2HC, RESUME–2HC and QUEUE–2HC. This could lead to a more refined analysis explaining why dynamic approaches are so effective in practice, despite bad worst-case bounds (as done in the past for other algorithmic frameworks, such as speed-up techniques for Dijkstra's algorithm in road networks).

Finally, it would be remarkable to develop an algorithm even faster than QUEUE–2HC, in practice. An idea in this sense could be that of adopting a sort of memorization approach, by precomputing and maintaining, during the evolution of the graph, some additional (compact) data structure storing information about central vertices or critical arcs, to faster identify or update outdated label entries. The benefits of this strategy, however, could be limited due to overhead that would be required to also update the additional data structure itself. An attractive alternative could be to find a lower bound on the computational effort that is necessary to update a 2-HOP-COVER labeling under decremental operations.

References

1. Vieira, M.V.; Fonseca, B.M.; Damazio, R.; Golgher, P.B.; Reis, D.C.; Ribeiro-Neto, B.A. Efficient search ranking in social networks. In Proceedings of the 16th ACM Conference on Conference on Information and Knowledge Management (CIKM 2007), Lisbon, Portugal, 6–10 November 2007; pp. 563–572.
2. Abraham, I.; Delling, D.; Goldberg, A.V.; Werneck, R.F. Hierarchical Hub Labelings for Shortest Paths. In *European Symposium on Algorithms*; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7501, pp. 24–35. 3. Delling, D.; Goldberg, A.V.; Pajor, T.; Werneck, R.F. Robust Distance Queries on Massive Networks. In Proceedings of the 22st European Symposium on Algorithms (ESA 2004), Wroclaw, Poland, 8–10 September 2014; Springer: Berlin/Heidelberg, Germany, 2014; Volume 8737, pp. 321–333.
4. Cionini, A.; D'Angelo, G.; D'Emidio, M.; Frigioni, D.; Giannakopoulou, K.; Paraskevopoulos, A.; Zaroliagis, C. Engineering Graph-Based Models for Dynamic Timetable Information Systems. *J. Discret. Algorithms* **2017**, *46–47*, 40–58. [CrossRef]
5. Bonchi, F. Distance-Based Community Search. In Proceedings of the 45th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), Novy Smokovec, Slovakia, 27–30
   January 2019; Catania, B., Královic, R., Nawrocki, J.R., Pighizzini, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2019; Volume 11376, pp. 21–27. [CrossRef]
6. Wei, H.; Yu, J.X.; Lu, C.; Jin, R. Reachability Querying: An Independent Permutation Labeling Approach. *VLDB J.* **2018**, *27*, 1–26. [CrossRef]
7. Akiba, T.; Iwata, Y.; Yoshida, Y. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In Proceedings of the International Conference on Management of Data (SIGMOD 2013), New York, NY, USA, 22–27 June 2013; pp. 349–360.
8. Akiba, T.; Iwata, Y.; Yoshida, Y. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In Proceedings of the 23rd Int. World Wide Web Conference (WWW 2014), Seoul, Korea, 7–11 April 2014; pp. 237–248.
9. Cheng, J.; Yu, J.X. On-line exact shortest distance query processing. In Proceedings of the 12th International Conference on Extending Database Technology (EDBT 2009), Saint-Petersburg, Russia, 23–26 March 2009; pp. 481–492.
10. Jiang, M.; Fu, A.W.C.; Wong, R.C.W. Exact Top-k Nearest Keyword Search in Large Networks. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD 2015), Melbourne, Australia, 31 May–4 June 2015; ACM: New York, NY, USA, 2015; pp. 393–404. [CrossRef]
11. Borassi, M.; Crescenzi, P.; Habib, M.; Kosters, W.A.; Marino, A.; Takes, F.W. Fast diameter and radius BFS-based computation in (weakly connected) real-world graphs:

With an application to the six degrees of separation games. *Theor. Comput. Sci.* **2015**, *586*, 59–80. [CrossRef]
12. Uras, T.; Koenig, S. Subgoal graphs for fast optimal pathfinding. *Game AI Pro* **2015**, *2*, 145–159.
13. D'Emidio, M.; Forlizzi, L.; Frigioni, D.; Leucci, S.; Proietti, G. Hardness, approximability, and fixed-parameter tractability of the clustered shortest-path tree problem. *J. Comb. Optim.* **2019**, *38*, 165–184. [CrossRef]
14. D'Angelo, G.; D'Emidio, M.; Frigioni, D. Fully dynamic update of arc-flags. *Networks* **2014**, *63*, 243–259. [CrossRef]
15. D'Angelo, G.; D'Emidio, M.; Frigioni, D.; Vitale, C. Fully Dynamic Maintenance of Arc-Flags in Road
Networks. In Proceedings of the 11th International Symposium on Experimental Algorithms (SEA 2012), Bordeaux, France, 7–9 June 2012; Springer: Berlin/Heidelberg, Germany, 2012; Volume 7276, pp. 135–147.
16. Bauer, R.; Wagner, D. Batch Dynamic Single-Source Shortest-Path Algorithms: An Experimental Study. In Proceedings of the 8th International Symposium on Experimental Algorithms (SEA 2009), Dortmund, Germany, 4–6 June 2009; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5526, pp. 51–62.
17. Bast, H.; Delling, D.; Goldberg, A.V.; Müller-Hannemann, M.; Pajor, T.; Sanders, P.; Wagner, D.; Werneck, R.F. Route Planning in Transportation Networks. In *Algorithm Engineering*; Springer: Cham, Switzerland, 2016; Volume 9220, pp. 19–80.
18. Cohen, E.; Halperin, E.; Kaplan, H.; Zwick, U. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* **2003**, *32*, 1338–1355. [CrossRef]
19. Fu, A.W.C.; Wu, H.; Cheng, J.; Wong, R.C.W. IS-Label: An Independent-set Based Labeling Scheme for
Point-to-point Distance Querying. *Proc. VLDB 2013* **2013**, *6*, 457–468. [CrossRef]
20. Jin, R.; Ruan, N.; Xiang, Y.; Lee, V.E. A highway-centric labeling approach for answering distance queries on large sparse graphs. In Proceedings of the International Conference on Management of Data (SIGMOD 2012), Scottsdale, AZ, USA, 20–24 May 2012; pp. 445–456.
21. Delling, D.; Dibbelt, J.; Pajor, T.; Werneck, R.F. Public Transit Labeling. In *Experimental Algorithms*; Bampis, E., Ed.; Springer International Publishing: Cham, Switzerland, 2015; pp. 273–285.
22. Qin, Y.; Sheng, Q.Z.; Zhang, W.E. SIEF: Efficiently Answering Distance Queries for Failure Prone Graphs. In Proceedings of the 18th Int. Conference on Extending Database Technology (EDBT 2015), Brussels, Belgium, 23–27 March 2015; pp. 145–156.
23. Wei, F. TEDI: Efficient Shortest Path Query Answering on Graphs. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD 2010), Indianapolis, IN, USA, 6–11 June 2010; pp. 99–110.
24. Delling, D.; Goldberg, A.V.; Pajor, T.; Werneck, R.F. Customizable route planning. In Proceedings of the 10th Symposium on Experimental Algorithms (SEA 2011), Kolimpari, Chania, Crete, Greece, 5–7 May 2011; Springer: Berlin/Heidelberg, Germany, 2011; Volume 6630, pp. 376–387.
25. D'Emidio, M.; Khan, I. Dynamic Public Transit Labeling. In Proceedings of the Computational Science and Its Applications (ICCSA 2019), 19th International

Conference, Saint Petersburg, Russia, 1–4 July 2019; Misra, S., Gervasi, O., Murgante, B., Stankova, E.N., Korkhov, V., Torre, C.M., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O., Tarantino, E., Eds.; Proceedings, Part I; Springer: Cham, Switzerland, 2019; Volume 11619, pp. 103–117. [CrossRef]

26. D'Angelo, G.; D'Emidio, M.; Frigioni, D. Fully Dynamic 2-Hop Cover Labeling. *J. Exp. Algorithmics* **2019**, *24*, 1.6:1–1.6:36. [CrossRef]
27. D'Angelo, G.; D'Emidio, M.; Frigioni, D. Distance Queries in Large-Scale Fully Dynamic Complex Networks. In Proceedings of the 27th International Workshop on Combinatorial Algorithms (IWOCA 2016), Helsinki, Finland, 17–19 August 2016; Springer: Cham, Switzerland, 2016; Volume 9843, pp. 109–121.
28. Weller, M. Optimal Hub Labeling is NP-complete. *arXiv* **2014**, arXiv:1407.8373.
29. Cicerone, S.; D'Emidio, M.; Frigioni, D. On Mining Distances in Large-Scale Dynamic Graphs. In Proceedings of the 19th Italian Conference on Theoretical Computer Science, Urbino, Italy, 18–20 September 2018; Volume 2243, pp. 77–81.
30. Bergamini, E.; Meyerhenke, H. Fully-dynamic Approximation of Betweenness Centrality. In *Algorithms-ESA*; Springer: Berlin/Heidelberg, Germany, 2015.
31. Hayashi, T.; Akiba, T.; Kawarabayashi, K. Fully Dynamic Shortest-Path Distance Query Acceleration on Massive Networks. In Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (CIKM 2016), Indianapolis, IN, USA, 24–28 October 2016; pp. 1533–1542.
32. D'Andrea, A.; D'Emidio, M.; Frigioni, D.; Leucci, S.; Proietti, G. Experimental Evaluation of Dynamic Shortest Path Tree Algorithms on Homogeneous Batches. In Proceedings of the 13th International Symposium on Experimental Algorithms (SEA 2014), Copenhagen, Denmark, 29 June–1 July 2014; Springer: Cham, Switzerland, 2014; Volume 8504, pp. 283–294.
33. Henzinger, M.; Krinninger, S.; Nanongkai, D. Decremental Single-Source Shortest Paths on Undirected Graphs in Near-Linear Total Update Time. *J. ACM* **2018**, *65*. [CrossRef]
34. Chuzhoy, J.; Khanna, S. A New Algorithm for Decremental Single-Source Shortest Paths with Applications to Vertex-Capacitated Flow and Cut Problems. In Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, Phoenix, AZ, USA, 26 June 2019; Association for Computing Machinery: New York, NY, USA, 2019; pp. 389–400. [CrossRef]
35. Staudt, C.L.; Sazonovs, A.; Meyerhenke, H. NetworKit: A tool suite for large-scale complex network analysis. *Netw. Sci.* **2016**, *4*, 508–530. [CrossRef]
36. Delling, D.; Goldberg, A.V.; Savchenko, R.; Werneck, R.F. Hub Labels: Theory and Practice. In Proceedings of the 13th Symposium on Experimental Algorithms (SEA 2014), Copenhagen, Denmark, 29 June–1 July 2014; Springer: Cham, Switzerland, 2014; Volume 8504, pp. 259–270.
37. Leskovec, J.; Krevl, A. SNAP Datasets: Stanford Large Network Dataset Collection. 2014. Available online: http://snap.stanford.edu/data (accessed on 11 February 2020).
38. Rossi, R.A.; Ahmed, N.K. The Network Data Repository with Interactive Graph Analytics and Visualization. In Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015), Austin, TX, USA, 25–30 January 2015.
39. Kunegis, J. KONECT: The Koblenz Network Collection. 2013. Available online: http://konect.uni-koblenz.

de/ (accessed on 11 February 2020).
40. Bollobás, B. *Random Graphs*; Cambridge University Press: Cambridge, UK, 2001.
41. Albert, R.; Barabási, A.L. Statistical mechanics of complex network. *Rev. Mod. Phys.* **2002**, *74*, 47–97. [CrossRef]
42. D'Andrea, A.; D'Emidio, M.; Frigioni, D.; Leucci, S.; Proietti, G. Dynamic Maintenance of a Shortest-Path Tree on Homogeneous Batches of Updates: New Algorithms and Experiments. *ACM J. Exp. Algorithmics* **2015**, *20*, 1.5:1.1–1.5:1.33. [CrossRef]
43. Jamour, F.; Skiadopoulos, S.; Kalnis, P. Parallel Algorithm for Incremental Betweenness Centrality on Large Graphs. *IEEE Trans. Parallel Distrib. Syst.* **2018**, *29*, 659–672. [CrossRef]