On Mining Dynamic Graphs for *k* Shortest Paths

Andrea D'Ascenzo[1][0000−0001−5612−0798] and Mattia D'Emidio[2][0000−0001−7833−9520]

[1] Luiss University, Rome, Italy, adascenzo@luiss.it
[2] University of L'Aquila, L'Aquila, Italy, mattia.demidio@univaq.it

Abstract. Mining graphs, upon query, for k shortest paths between vertex pairs is a prominent primitive to support several analytics tasks on complex networked datasets. The state-of-the-art method to implement this primitive is kPll, a framework that provides very fast query answering, even for large inputs and volumes of queries, by pre-computing and exploiting an appropriate *index* of the graph. However, if the graph's topology undergoes changes over time, such index might become obsolete and thus yield incorrect query results. Re-building the index from scratch, upon every modification, induces unsustainable time overheads, incompatible with applications using k shortest paths for analytics purposes. Motivated by this limitation, in this paper, we introduce decKpll, the first dynamic algorithm to maintain a kPll index under *decremental* modifications. We assess the effectiveness and scalability of our algorithm through extensive experimentation and show it updates kPll indices orders of magnitude faster than the re-computation from scratch, while preserving its compactness and query performance. We also combine decKpll with incKpll, the only known dynamic algorithm to maintain a kPll index under *incremental* modifications, and hence showcase, on real-world datasets, the first method to support fast extraction of k shortest paths from graphs that evolve by arbitrary topological changes.

Keywords: Graph Algorithms · Dynamic Networks · Algorithm Engineering · Experimental Algorithmics

1 Introduction

Computing, upon a *query*, a set of k *shortest paths* for a pair of vertices of a graph is considered a primitive of high importance in the context of graph analytics. Such paths, in fact, are often used to determine relations of closeness or similarity between vertices, based on the graph topological structure [4,9,29], and hence to support network classification tasks where graphs can be compared by sampling pairs of vertices and by analyzing their topological relationships [1, 28]. In particular, k shortest paths (ranked by their respective lengths, called *k shortest distances*), are known to represent a natural, informative and robust measure of distance/similarity between vertices (and hence graphs) [1,12,20].

Methods to answer to queries on k shortest paths can be broadly divided into two groups, namely *without indexing* and *with indexing* [1]. In the former group, we find algorithms that directly solve, upon query, the single pair

k shortest paths problem, i.e. that compute k shortest paths between a pair of vertices in a graph in non-descending order of their costs. Chiefly, reference algorithms in this group are: (i) the kBFS algorithm, a variant of the Breadth First Search (BFS) algorithm that visits a same vertex up to k times [1]; (ii) the Eppstein's algorithm which builds a shortest path tree, rooted at a vertex, and form paths to the other vertex by selecting edges that are outside the tree and connect different sub-trees [12]. For an n-vertex m-edge graph, the former runs in $\mathcal{O}((n+m)k)$ time while the latter reduces the time complexity to $\mathcal{O}(n+m+k)$. In the latter group, instead, we have strategies that divide the computational effort in two steps: in an offline phase, a one-time *pre-processing* of the input graph is performed to construct an *index* data structure which is then, at run-time, exploited to dramatically reduce the time necessary to answer queries on k shortest paths [1, 19]. Methods without indexing are considered efficient in terms of space occupancy, and preferred in practical scenarios where memory is a constrained resource (e.g. embedded systems), while methods with indexing are superior in terms of query times and therefore are adopted in all real-world applications where query performance becomes is crucial, e.g. when large graphs are managed or several (possibly millions) queries have to be answered [1].

Among indexing-based methods, the kPll framework, proposed in [1], has emerged as the one performing best in practice. Such framework consists of: (i) a pre-processing algorithm that computes, once, a so-called *k-2-Hop-Cover index* of the input graph storing, for each vertex, in the form of *labels*, selected information about paths and cycles passing through said vertex in the graph; (ii) a query algorithm, that retrieves, upon request, the k shortest paths by only accessing the labels of two queried vertices. Despite having high time and space complexities in the worst case, extensive experimental work has shown that carefully engineered implementations of kPll, incorporating effective pruning strategies, outperform all other methods in the literature for queries on k shortest paths, since: (i) the time (space, resp.) overhead for pre-computing (storing, resp.) the index, is affordable and compatible with the requirements of applications of interest (at most few thousands of seconds to run the pre-processing and few GBs to store the index); (ii) using the index ensures extremely small query times, orders of magnitude smaller ($\approx \mu s$ per query) than any other known solution [1, 9].

The main drawback of kPll, and in general of indexing-based methods for graph mining, is that pre-computed information can easily become *obsolete* and yield incorrect query results when the managed network changes over time, i.e. if the underlying graph is *dynamic* [9, 10]. In fact, most methods for fast query answering through pre-processing store topological data in the index that can be heavily altered even by very limited modifications occurring to the input (e.g. an edge deletion) [3, 8, 16, 18]. One possibility to use indexing-based techniques with dynamic graphs is to re-compute the index from scratch after an change occurs on the network. However, this is considered impractical, since the pre-computation phase, though effective, induces large time overheads if repeated too often, e.g. with the frequency typically networks undergo updates in practice [3, 17]. For this reason, and since most real-world graphs of interest for analytical purposes are inherently dynamic (e.g. social/road graphs) [3, 17, 24] researchers have worked on designing effective *dynamic algorithms* that are able to identify and update only the part of the pre-computed data structure that is altered by a graph change, faster than the pre-processing routine, while preserving the correctness of queries and the compactness of the index [3, 15]. Investigations of this kind have concerned most indexing-based frameworks for retrieving non-trivial graph structures [3, 8, 13, 14, 16]. For KPLL indices, currently, the only known solution of this kind is algorithm INCKPLL [9], which however works only for graphs that change by *incremental updates* (vertex/edge insertions). To the best of our knowledge, no dynamic algorithm is known to update a KPLL index when the graph is *fully dynamic*, i.e. changes in an unrestricted, arbitrary fashion (including *decremental updates*, i.e. vertex/edge deletions).

Our Contribution. In this paper, we remove such limitation and present DECKPLL, the first algorithm to efficiently maintain a KPLL index under *decremental* changes. We prove the correctness of our method and assess its effectiveness through extensive experimental work. In particular, we provide empirical evidences on DECKPLL being able to update a KPLL index, on average, orders of magnitude faster than the re-computation from scratch, while at the same time preserving its compactness and query performance. We also experimentally show that DECKPLL can be used in combination with INCKPLL, the only known dynamic algorithm to maintain a KPLL index under *incremental* modifications hence showcasing the first method to support very fast, interactive extraction of k shortest paths from graphs that evolve by arbitrary topological changes.

Related Works. The problem of computing k shortest paths and distances has been widely investigated in the last decade, in several variants. The *k Shortest Simple Paths (*K-SSP*) problem*, for instance, requires that the sought k paths have to be *simple*, i.e. must not self-intersect. This version is generally considered computationally harder and, indeed, no asymptotic improvement is known to Yen's algorithm [27], which runs in $O(kn(m+n\log n))$ time in an n-vertex, m-edge graph. Some heuristics have been proposed to empirically accelerate Yen's algorithm (see, e.g. [30]), however none of them shows query performance comparable to that achieved by indexing-based methods for similar graph analytics problems (see, e.g., [1, 2, 10]). In this sense, designing an algorithmic method to solve the K-SSP problem with small query times is an open problem and represents an active area of research [30]. Among frameworks known in the literature to support graph mining processes at scale, those relying on computing compact indices via preprocessing have been observed to exhibit the best performance and scalability properties [10, 13]. In particular, the hub-labeling technique, introduced in [6], has been successfully modified to handle, effectively, several graph analytics tasks, such as, e.g., path counting [29], best connections in transport systems [11, 26] or constrained connectivity [23]. Due to the inherent time-evolving nature of real-world networks, most studies on pre-processing-based frameworks have been followed by investigations on corresponding *dynamic* algorithms to efficiently reflect topological changes, occurring on the underlying graph, on pre-computed data structures. Examples of such investigations include the design and experimental evaluation of dynamic algorithms for maintaining, over time: shortest path trees [7], transitive closure [16], centrality measures [21], or timetable graphs [5].

2 Preliminaries

We are given a graph $G = (V, E)$, with a vertex set V and a edge set E. We call $n = |V|$ ($m = |E|$, resp.) its number of vertices (edges, resp.). We denote by $(x, y) \in E$ an edge connecting two vertices $x, y \in V$ in G. For the sake of simplicity, we describe all methods for undirected, unweighted graphs. All presented techniques and algorithms can be extended to weighted digraphs.

A *path* $p = (v_1, v_2, \ldots, v_r)$ in G, connecting two vertices $s, t \in V$, is a sequence of r vertices such that $(v_i, v_{i+1}) \in E$ for all $i \in [1, r-1]$. We say edge (v_i, v_{i+1}) belongs to the path or $(v_i, v_{i+1}) \in p$ for each pair of vertices in p. We call: $v_1 = s$ and $v_r = t$ the *endpoints* of the path; and any $v_i, 1 < i < r$ an *internal vertex* of the path. Any path whose endpoints coincide is called a *cycle*, while a path is *simple* if the sequence has no vertex repetitions. The *length* $\ell(p)$ of a path p is the number of edges in p. Multiplicities of edge repetitions are counted if the path is not simple. A *shortest path* $p(s,t)$, for two vertices $s, t \in V$, is a path having minimum length among all those in G connecting s and t. The *distance* $d(s,t)$ for a pair $s, t \in V$ is the length $\ell(p(s,t))$ of a shortest path $p(s,t)$. Given a path $p = (v_1, \ldots, v_r)$, we use $p[v_i, v_j]$ to identify the *subpath* of p having v_i and v_j as endpoints, and whose internal vertices are those between v_i and v_j, for $1 \leq i < j \leq r$. If vertices v_i and v_j appear multiple times, the first (last, resp.) occurrence of v_i (v_j, resp.) is considered. Subpath $p[v_1, v_j]$ ($p[v_i, v_r]$, resp.) is called *prefix* (*suffix*, resp.) path of p. Additionally, given two paths $p = (v_1, \ldots, v_r)$ and $q = (w_1, \ldots, w_l)$ such that $v_r = w_1$, we denote by $p \oplus q = (v_1, \ldots, v_r = w_1, \ldots, w_l)$ the *concatenation* of p and q.

We assume vertices are associated to unique integer identifiers to enable natural comparisons for any pair $u, v \in V$ by expressions such as $u < v$ or $u \leq v$. Given two vertices $s, t \in V$, we call: (i) $\mathcal{P}_{st}$ the set of paths in G connecting s and t; (ii) $\mathcal{P}_{st}^{>v} \subseteq \mathcal{P}_{st}$ the subset of $\mathcal{P}_{st}$ containing paths whose internal vertices are all larger than some $v \in V$; (iii) $\mathcal{P}_{st}^{\not> v}$ the subset of $\mathcal{P}_{st}$ containing paths such that at least one internal vertex is smaller than or equal to v. Furthermore, we call $p_i(s,t)$ the *i-th shortest path* between s and t, i.e. the i-th element in $\mathcal{P}_{st}$, sorted in non-decreasing order of path lengths, and use $d_i(s,t) = \ell(p_i(s,t))$ to refer to the *i-th shortest distance* for pair s, t, i.e. the length of $p_i(s,t)$. Similarly, we use: (i) $d_i^{>v}(s,t)$ ($d_i^{\geq v}(s,t)$ and $d_i^{\not> v}(s,t)$, resp.) to refer to the i-th shortest distance when paths are restricted to have internal vertices that all are larger (larger than or equal to and not greater, resp.) than some $v \in V$; (ii) $p_i^{>v}(s,t)$ ($p_i^{\geq v}(s,t)$, resp.) to identify the corresponding i-th shortest path; (iii) $d^{>v}(s,t) = d_1^{>v}(s,t)$ ($p^{>v}(s,t) = p_1^{>v}(s,t)$, resp.) to denote the distance (a shortest path inducing the distance, resp.) subject to the same restrictions on vertices. Finally, we call $\mathcal{P}_{st}^{k}$ a set of k shortest paths between s and t in G, i.e. a set containing an i-th shortest path for any $i \in [0, k-1]$, and by $\mathcal{P}_{st}^{k,>u}$ a set of k shortest paths between s and t in G having internal vertices larger than some $u \in V$.

Given a graph $G = (V, E)$, an integer $k > 1$, and a pair of vertices $s, t \in V$, the k *shortest paths* (K-SP) problem asks to compute a set $\mathcal{P}_{st}^{k}$ of k shortest paths between s and t in G, i.e. to answer a *query* on the k shortest paths for s and t. The KPLL framework [1] is the reference approach to address the K-SP problem in terms of computational time per query. The method is based on the pre-computation, for a given graph, of a data structure called *k-2-Hop-Cover* (k2HC, for short) *index*, a generalization of the 2-*Hop-Cover index*, introduced in [6]. Such index can be used to solve both the K-SP problem and its natural specialization, named k *shortest distances* problem, that asks to retrieve only the lengths of k shortest paths, and is defined as follows. Given a graph $G = (V, E)$, define, for each vertex $v \in V$: (i) a *length label* $L(v)$, containing pairs in the form $(u, \mathcal{P}_{u,v}^{k,>u})$ where $u \in V$; (ii) a *loop label* $C(v)$, storing a sequence of k cycles $(c_1(v), c_2(v), \ldots, c_k(v))$ in G that include vertex v. Then, pair $I = (L, C)$, where $L = \{L(v)_{v \in V}\}$ and $C = \{C(v)\}_{v \in V}$, is called a k-2-Hop Cover index of G.

Note that the k2HC index is often referred to as k2HC labeling or simply k2HC. We use these notations interchangeably. Elements in the labels are called *entries.* A k2HC index is said to *cover* a graph G, or equivalently to satisfy the *k-cover property* for G, whenever it can be exploited to correctly solve the K-SP problem on G. Specifically, given a k2HC index $I = (L, C)$ of a graph $G = (V, E)$, let QUERY(I, s, t) identify the result of executing a *query* on I for a pair of vertices $s, t \in V$ and let such result consist of the shortest k paths, in terms of length, in multi-set $\Delta(I, s, t) = \{p_{vs} \oplus c_i(v) \oplus p_{vt} | (v, \mathcal{P}_{vs}) \in L(s), p_{vs} \in \mathcal{P}_{vs}, c_i(v) \in C(v), (v, \mathcal{P}_{vt}) \in L(t), p_{vt} \in \mathcal{P}_{vt}\}$. Then, I covers G, or equivalently satisfies the k-cover property for G, if and only if, for any pair $(s, t) \in V \times V$, we have QUERY$(I, s, t) = \mathcal{P}_{st}^{k}$. In other words, a k2HC covering a graph G allows to retrieve the k shortest paths in G for any pair of vertices $s, t \in V$, by a query on the index that selects shortest combinations in $\Delta(I, s, t)$, obtained by appending a cycle, originating at some vertex $v \in V$, to a path from s to v, and by concatenating to it a path from v to t. Any vertex v that forms one of the k smallest combinations of $\Delta(I, s, t)$ is called a *hub vertex* for pair s, t. Whenever a pair s, t is disconnected in G then $d_i(s, t) = \infty \, \forall i \in [1, k]$. Correspondingly, QUERY$(I, s, t)$ returns a single, default `null` value whenever there is no vertex $v \in V$ such that $(v, \mathcal{P}_{sv}) \in L(s), (v, \mathcal{P}_{tv}) \in L(t)$.

We call *size* of the index the total number of entries in length and loop labels. A naive k2HC of size $\Omega(kn^2)$ can be computed by $\mathcal{O}(n^2)$ executions of Eppstein's algorithm. However, to obtain superior query performance, a k2HC of minimum size is desirable. Unfortunately, by a reduction to the computation of a minimum 2-Hop-Cover index, the problem of computing a minimum-sized k2HC is $\mathcal{NP}$-hard [6]. Nevertheless, the heuristic of [1] has been shown to compute reasonably compact k2HC indices, which results in superior performance in terms of trade-off between preprocessing time, index size and query time. Specifically, the construction guarantees that, for any $v \in V$: (i) paths in $L(v)$, associated with a vertex $u \in V$, belong to $\mathcal{P}_{uv}^{k,>u}$; (ii) cycles in $C(v)$ belong to $\mathcal{P}_{vv}^{k>v}$. Properties

(i) and (ii), combined, guarantee that the resulting k2HC satisfies the k-cover property. Note that, in the reminder of the paper, for the sake of brevity, we use acronym KPLL also to refer to the preprocessing routine of the framework.

Dynamic Scenario. In a *dynamic* scenario we assume we are given an *initial* graph, say $G = (V, E)$, that can undergo either *incremental* (i.e. vertex/edge insertions) or *decremental* modifications (i.e. vertex/edge removals) at arbitrary times. In such scenario, pre-computed data structures, exploited to address the K-SP problem (e.g. the k2HC), might not reflect properly the graph structure as the topology changes over time, and thus the task of maintaining stored information updated emerges. Given a decremental modification x occurring on a graph G (e.g. the deletion of an edge $e \in E$), the *decremental* K-SP *problem* asks to compute the set $\mathcal{P}^k_{st} = \{p'_1(s,t), p'_2(s,t), \dots, p'_k(s,t)\}$ of the k shortest paths between s and t in G', for some $s, t \in V'$, where $G' = (V', E')$ is the graph obtained by applying x to G (e.g. by removing e from E). If one relies on a k2HC to solve the K-SP problem, the decremental K-SP problem translates to a corresponding *decremental* k2HC *problem* which asks, given a graph $G = (V, E)$, a k2HC index I covering G, and a decremental modification x on G, to compute a k2HC I' that covers G' where G' is obtained by applying x to G. Note that, the incremental counterpart of the K-SP problem has been defined and addressed in [9]. In said work, the authors introduce INCKPLL, a *partially dynamic* (specifically *incremental*) algorithm that, given a graph G, a k2HC index $I = (L, C)$ covering G, and an incremental update x on G, updates I to an index $I' = (L', C')$ which covers graph G', obtained by applying x to G.

3 Decremental Algorithm

In this section we introduce DECKPLL, a dynamic algorithm to handle the decremental k2HC problem. Similarly to the incremental problem, if the change to be managed is a vertex update (insertion or deletion), this can be modeled as an appropriate sequence of edge operations (insertions or deletions) incident to the interested vertex [8]. Therefore, in what follows, for the sake of simplicity, we focus on handling edge deletions only. As in [1], we consider vertices are sorted by an easy-to-compute ordering, e.g. non-increasingly by degree, and indices reflect such ordering, i.e. $v_i \leq v_j \implies |N(v_i)| \leq |N(v_j)|$.

Let $G = (V, E)$ be a graph and let $I = (L, C)$ be a k2HC covering G. Let us be given a decremental update for G, namely the removal of an edge $e = (x, y) \in E$. Differently from the incremental problem, in the decremental problem entries of the index might become *obsolete* (i.e. might not correspond to a path or a cycle in the graph) as a consequence of a decremental update. Such entries, therefore, must be removed to guarantee the correctness of queries. To this aim, our update procedure (see Algorithm 1), works in three phases: (i) detection of vertices *affected* by the edge removal, i.e. whose length label contains at least one obsolete entry; (ii) removal of obsolete entries, i.e. an entry associated to a path that traverses the removed edge; (iii) restoration of k-cover property, which might be broken by the removal of obsolete entries. In the detection of affected

Algorithm 1: Algorithm decKpll.

```
Input: A k2hc $I = (L, C)$ covering graph $G = (V, E)$, edge $(x, y) \in E$
Output: A k2hc $I' = (L', C')$ covering graph $G' = (V, E \setminus \{(x, y)\})$
1 aff-set $\leftarrow \emptyset$;
2 foreach $u \in \{x, y\}$ do findaff($u$, aff-set);
3 aff-hubs $\leftarrow$ removal($I$, aff-set);
4 $I' \leftarrow$ restore($I$, aff-hubs);
5 return $I'$;
```

vertices (see Procedure 2), we search for vertices whose length label contains at least one obsolete entry. To this aim, we execute a BFS-like visit, on the new graph G', rooted at the two endpoints of the removed edge x and y. For each visited vertex v, the procedure executes QUERY to retrieve the k shortest paths from one of the endpoints to v encoded in the index: if none of such paths contains the removed edge, then we prune (i.e. stop) the visit at v. Vice versa, we add v to the set AFF-SET of affected vertices (initially empty). The algorithm is symmetric for x and y hence it is executed twice.

Procedure 2: Sub-routine findaff of Algorithm 1.

```
Input: Endpoint $u$ of removed edge, set aff-set
1  $E' \leftarrow E \setminus \{e\}$;
2  $G' \leftarrow (V, E')$;
3  foreach $t \in V$ do $visited[t] \leftarrow false$;
4  $visited[u] \leftarrow true$;
5  $Q \leftarrow \{u\}$;
6  while $Q \neq \emptyset$ do
7      Dequeue $v$ from $Q$;
8      $visited[v] \leftarrow true$;
9      foreach $path \in$ query($I, v, y$) do
10         if $(x, y) \in path$ then
11             aff-set $\leftarrow$ aff-set $\cup \{v\}$
12     foreach $(v, w) \in E' : \neg visited[w]$ do
13         Enqueue $w$ into $Q$;
```

After the detection of affected vertices, the algorithm removes obsolete entries by scanning length labels of each $v \in$ AFF-SET and by removing entries such that associated paths contain edge (x, y). This is done by sub-routine REMOVAL (see Procedure 3). During the removal, the algorithm traces and stores in a suited set, named AFF-HUBS, the hub vertices associated to removed entries. Finally, in the third and last phase (sub-routine RESTORE, see Procedure 4), the algorithm restores the k-cover property. To this purpose, both loop labels and length labels are updated. Specifically, for the former, we compute a set D_{cy} that contains any vertex v for which at least one loop label may be obsolete due to the change. In particular, such set, for a given loop labeling C, contains any vertex v which is: (i) connected either to x or y by a shortest path whose length is at most k and whose vertices are all larger than v; (ii) larger than or equal to the minimum between x and y (according to the vertex ordering); (iii) such that there exists a cycle in $C(v)$ containing the removed edge. These conditions can be summarized

Procedure 3: Sub-routine REMOVAL of Algorithm 1.

```
Input: A $k$2HC $I = (L, C)$ covering graph $G$, set AFF-SET
Output: Set AFF-HUBS.
AFF-HUBS $\leftarrow \emptyset$;
foreach $v \in$ AFF-SET do
    foreach $(h, \mathcal{P}) \in L'(v)$ do
        foreach $path \in \mathcal{P}$ do
            if $(x, y) \in path$ then
                Remove $path$ from $\mathcal{P}$;
                AFF-HUBS $\leftarrow$ AFF-HUBS $\cup \{h\}$;
return AFF-HUBS;
```

by Eq. 1:

$$\mathrm{D}_{cy}(C) = \{v \in V : (d^{>v}(v,x) \leq k \vee d^{>v}(v,y) \leq k) \wedge v \leq \min(x,y) \wedge \exists c_i \in C(v) \ : \ (x,y) \in c_i\} \tag{1}$$

For each vertex w of $\mathrm{D}_{cy}(C)$, loop label entries are removed from $C(w)$ and new ones are computed in lines 1-13 of Algorithm 4. This part of the algorithm mimics the method in [1] for computing a k2HC from scratch, but only for a selected subset of vertices. Specifically, it starts a visit of the graph from each $v_i \in \mathrm{D}_{cy}$, following the vertex ordering and traversing only vertices not smaller than v_i, and storing newly discovered cycles in the loop labels.

Procedure 4: Sub-routine RESTORE of Algorithm 1.

```
    Input: A $k$2HC $I = (L, C)$ not guaranteed to cover $G'$, set AFF-HUBS
    Output: A $k$2HC $I' = (L', C')$ covering $G'$
1   Compute $\mathrm{D}_{cy}(C)$ as in Eq. 1;
2   foreach $w \in \mathrm{D}_{cy}(C)$ do
3       $C(v) \leftarrow \emptyset$;
4       foreach $t \in V$ do $visited[t] \leftarrow 0$;
5       $visited[w] \leftarrow 1$;
6       $Q \leftarrow \{(w, \{w\})\}$;
7       while $Q \neq \emptyset$ do
8           Dequeue $(x, p)$ from $Q$;
9           $visited[x] \leftarrow visited[x] + 1$;
10          if $x = w$ then Add $p$ to $C(w)$;
11          if $visited[x] < k$ then
12              foreach $(x, v) \in E : v \geq w$ do
13                  Enqueue $(v, p \oplus \{v\})$ into $Q$;
14  foreach $h \in$ AFF-HUBS do
15      $L(h) \leftarrow \emptyset$;
16      $Q \leftarrow \{(h, 0)\}$;
17      while $Q \neq \emptyset$ do
18          Dequeue $(x, p)$ from $Q$;
19          if $\ell(p) < \max\{\ell(p) : p \in$ QUERY$(I, h, x)\}$ then
20              Add $(h, p)$ to $L(x)$;
21              foreach $(x, w) \in E : w > v$ do
22                  Enqueue $(w, p \oplus \{w\})$ into $Q$;
23  return $I' = (\{C(v)_{v \in V}\}, \{L(v)_{v \in V}\})$;
```

Finally, the algorithm searches the graph for new length label entries that must be added to the index to ensure that the k-cover property holds for the modified graph G'. By definition, set AFF-HUBS contains any vertex h such that the construction of the kPll index, which performs a shortest-path like visit rooted at each vertex h [1], traversed the removed edge (x, y). Thus, to compute a k2hc index covering the modified graph, we execute lines 14-22 of Procedure 4 where we perform a shortest-path like visit, rooted at each vertex $h \in$ aff-hubs, that traverses only vertices larger than h and exploits information stored in the index to prune the visit when newly discovered paths are longer than those encoded in the current index. When new paths are discovered, a corresponding length label entry is added to the k2hc. We now state the correctness and provide a complexity analysis of decKpll. Due to space limitations, all proofs are deferred to a full version of this paper.

Lemma 1. *Let $I = (L, C)$ be a k2hc covering a graph $G = (V, E)$. Assume $e = (x, y) \in E$ is removed from G. Then, set* aff-set*, computed by Algorithm 2, contains vertices affected by the removal, i.e. any vertex $v \in V$ whose length label $L(v)$ contains an entry $(h, \mathcal{P}_{hv}^{k,>h})$ such that edge e belongs to a path in $\mathcal{P}_{hv}^{k,>h}$.*

By Lemma 1, it follows that, after the execution of removal, we obtain: (i) an index that does not contain any obsolete length entry; (ii) set aff-hubs containing any vertex associated to at least one removal of a length entry. We can then prove the following.

Theorem 1. *Let $I = (L, C)$ be a k2hc covering a graph $G = (V, E)$ and let $G' = (V, E \setminus \{(x, y)\})$ for some $e = (x, y) \in E$. Call k2hc $I' = (L', C')$ the k2hc updated by Algorithm 4. Then, I' satisfies the k-cover property for G'.*

We now give the time complexity of decKpll in an output bounded sense [3,8].

Theorem 2. *Let $I = (L, C)$ be a k2hc covering a graph $G = (V, E)$. Let $l = \max_{v \in V} |L(v)|$, and ω be the maximum length of any path in G. Given an edge $e = \{x, y\} \in E$, let $G' = (V, E' = E \setminus \{e\})$. Then, algorithm* decKpll *takes $\mathcal{O}(n(nk\omega l + m))$ time to update I to a k2hc I' covering G'.*

Note that the time complexity of decKpll is larger than that of the kPll pre-processing since: $l = \mathcal{O}(kn)$, $r = \mathcal{O}(n)$, $s = \mathcal{O}(m)$ and $c = \mathcal{O}(m)$. However, our experiments show that, on average, such values are smaller than the worst case.

4 Experimental Evaluation

In this section, we describe the experimental study we conducted to evaluate the performance of decKpll. To this aim, we implemented: (i) the pre-processing routine of kPll; (ii) algorithm kBFS, the modified version of the BFS algorithm that is considered the state-of-the-art to retrieve k shortest paths when no index is available [1]; (iii) our new decremental algorithm decKpll. Moreover, for the sake of completeness, we implemented incremental algorithm incKpll and combined it with decKpll to form algorithm fulKpll, a fully dynamic algorithm that maintains a k2hc index under any type of update, by applying incKpll or decKpll, resp., depending on whether the update to be handled is, resp., incremental or decremental. This is done with the purpose of assessing the performance of DECKPLL and INCKPLL when applied in combination on graphs that change by arbitrary modifications. In fact, the behavior of the resulting combination, referred to FULKPLL in what follows, might be influenced by the lazy strategy of INCKPLL, since the latter does not remove obsolete entries after incremental updates and tends to increase the index size, which in turn might increment the computational effort of DECKPLL to update it. Nonetheless, further in this section we provide empirical evidences confirming that the two dynamic algorithms are very effective also when combined.

As inputs to experiments we consider a collection of real-world graphs representing networks of various application domains of interest (e.g. web or social graphs) with heterogeneous topologies [22, 25] (see Table 1, where graphs are sorted top-to-bottom by size $|V| + |E|$). Concerning parameter k, we use values of k in $\{2, 4, 8, 16\}$, as other studies on the K-SP problem [1, 9].

Graph	**Short**	$\lvert V\rvert$	$\lvert E\rvert$	avg. deg.	diameter
EatRS	ERS	23 219	328 105	28	5
Arxiv	ARX	34 401	420 828	24	14
PPIHuman	PPH	16 820	449 036	53	7
PPIMouse	PPM	17 349	733 037	84	6
Gowalla	GOW	196 591	950 327	10	16
NotreDame	NDM	325 729	1 090 108	6	46
MarkerCafe	MRC	69 413	1 644 801	47	9
Arabic	ARB	163 598	1 747 269	21	76
Stanford	SFD	255 265	1 941 926	15	164
FlyHemibrain	FHB	21 739	2 897 925	266	5

Table 1: Overview of Input Graphs.

For each input graph G and value of k, we perform two types of experiments, named DECR and FULL, resp., depending on the types of selected graph changes. In experiment DECR, we first execute KPLL to compute a k2HC index I covering G. Then, we select uniformly at random, for $\sigma > 0$ times, an edge (x, y) in the graph, remove it to obtain a graph G', and run DECKPLL to update index I to I' covering G'. We repeat the process for G' and, at the end, we compute from scratch a k2HC index I'' covering the last snapshot of the graph via KPLL. The purpose of this setting is to evaluate scenarios where deletions occur with uniform probability. In experiment FULL, instead, we uniformly select at random $\sigma > 0$ edges from the input graph G and remove them to obtain a graph G_{init}. We compute a k2HC index I covering G_{init} via KPLL and then re-insert the σ sampled edges, one after the other, until all removed edges are added back. In between such insertions, at fixed intervals (every 5 insertions), we apply randomly selected edge removals to the graph, for a total of $\frac{\sigma}{5}$ decremental update operations. Such fraction is selected following common distributions of updates in real-world graphs [7]. After each modification, we execute FULKPLL to update index I to I'. Eventually, we run KPLL to compute a k2HC index I'' covering the last snapshot of graph. This second experiment is designed with a twofold purpose, namely evaluating update times of DECKPLL and INCKPLL, when their executions are interleaved into FULKPLL, and assessing whether the space occupancy to store the k2HC changes when FULKPLL algorithm is applied, and specifically if it is impacted by the lazy strategy of INCKPLL which does not remove obsolete entries.

In both experiments, after each execution of the dynamic algorithms (after the final execution of KPLL, resp.) we perform 10^5 queries on I' (on I'', resp.) and measure: (i) median query times; (ii) size of index I' (I'', resp.); (iii) running time to update index I' via the dynamic algorithms after each change (to recompute I'' from scratch via KPLL, resp.). In all trials, vertex ordering follows non-increasing values of vertex degree, as in [1,9], while σ is set to 500. All our code is written in C++ and compiled with GCC 9.4.0 with opt. level $O3$. All tests have been executed on a workstation equipped with an Intel Xeon© CPU E5-2643 3.40 GHz and 128 GB of RAM, running Ubuntu Linux.

Analysis. A excerpt of our experimental results, for $k = 8$ and for experiment DECR and FULL is given in Table 2. Results for other values of k lead to similar

Graph	CT (s)		SPEED-UP	IS (MB)		QT (μs)	
	KPLL	DEC KPLL		KPLL	DEC KPLL	KPLL	DEC KPLL
ERS	3 699	368.31	$1.0 \cdot 10^1$	554	554	2 775	2 776
ARX	3 186	108.65	$2.9 \cdot 10^1$	1 130	1 130	2 020	2 023
PPH	1 092	19.16	$5.7 \cdot 10^1$	264	264	1 315	1 316
PPM	1 402	25.57	$5.4 \cdot 10^1$	360	360	1 604	1 605
GOW	2 681	193.58	$1.3 \cdot 10^1$	2 052	2 052	580	580
NDM	1 120	15.27	$7.3 \cdot 10^1$	2 945	2 945	324	324
MRC	3 474	90.44	$3.8 \cdot 10^1$	1 035	1 035	1 204	1 204
ARB	1 668	0.43	$3.8 \cdot 10^3$	2 713	2 713	1 495	1 495
SFD	1 240	4.69	$2.6 \cdot 10^2$	1 925	1 925	272	272
FHB	13 247	28.97	$4.5 \cdot 10^2$	772	772	3 782	3 782

Graph	CT (s)		SPEED-UP	IS (MB)		QT (μs)	
	KPLL	FUL KPLL		KPLL	FUL KPLL	KPLL	FUL KPLL
ERS	3 520	0.31	$1.1 \cdot 10^4$	554	555	2 597	2 598
ARX	3 235	0.32	$1.0 \cdot 10^4$	1 130	1 132	1 958	1 963
PPH	1 228	0.16	$7.2 \cdot 10^3$	264	264	1 390	1 391
PPM	1 429	0.10	$1.4 \cdot 10^4$	360	360	2 113	2 114
GOW	2 735	0.06	$4.0 \cdot 10^4$	2 052	2 053	592	594
NDM	1 119	0.01	$9.4 \cdot 10^4$	2 945	2 949	453	455
MRC	3 568	0.05	$6.2 \cdot 10^4$	1 035	1 035	1 196	1 199
ARB	1 688	0.02	$7.4 \cdot 10^4$	2 713	2 713	1 463	1 463
SFD	1 322	0.01	$1.1 \cdot 10^5$	1 922	1 936	360	366
FHB	13 268	1.27	$1.0 \cdot 10^4$	772	772	4 303	4 303

Table 2: Results of the DECR (left) and FULL (right) experiment for $k = 8$.

interpretation, hence are omitted due to space limitations and will be given in a full version of this paper. For each input graph, we report: (i) the running time of KPLL to build the index and the median running time of DECKPLL or FULKPLL to update it after each modification, resp. (see column *computational time* – CT, in seconds); (ii) the median of the ratios of the time to re-build the index by KPLL to that to update it by DECKPLL or FULKPLL after each modification (see column SPEED-UP); (iii) the size of the index recomputed via KPLL and the size of the index updated via DECKPLL or FULKPLL, after all modifications (see column *index size* – IS, in MBs)); (iv) the median execution time to perform 10^5 queries on the index recomputed via KPLL and on that updated by DECKPLL or FULKPLL, resp., after each modification (see column *query time* – QT, in μs).

The first conclusion that can be drawn from our experimental data is that the analysis of Thm. 2 is pessimistic, since the measured running time of DECKPLL is always at least an order of magnitude smaller than the time to re-compute from scratch the index, regardless of graph size, diameter, and k. On top of that, we notice that DECKPLL is always significantly faster than the recomputation from scratch, with a speed-up against KPLL that ranges from one order of magnitude up to 5 orders of magnitude (see Fig. 1, where we report the measured speed-up, for all values of k and graphs, as a function of the graph size). Even more interestingly, we observe that the speed-up increases as $|V|+|E|$ increases, which suggests that DECKPLL scales well with the input size (see, again, Fig. 1, left). Concerning experiment FULL, the combination of INCKPLL and DECKPLL yields very small update times for any type of modification, with a speed-up against KPLL that spans from 3 to 5 orders of magnitude. Remarkably, the median execution time of FULKPLL is less than 2 seconds even in the largest input.In terms of space occupancy, our experimentation highlights that the size of indices, updated via DECKPLL and FULKPLL, is always comparable to that obtained by the re-computation from scratch through KPLL (see column IS in Table 2). This is a highly desirable observed behavior for our dynamic algorithms, since it suggests they are able to preserve the compactness of the index, which is known to be directly related to small average query time [1,9]. The latter aspect is confirmed by our measures of such query times which do not change, even after many graph updates, and remain comparable (few microseconds to few milliseconds per query) to those obtained by querying indices recomputed via KPLL, even for the largest graphs and values of k (see column QT in Table 2).

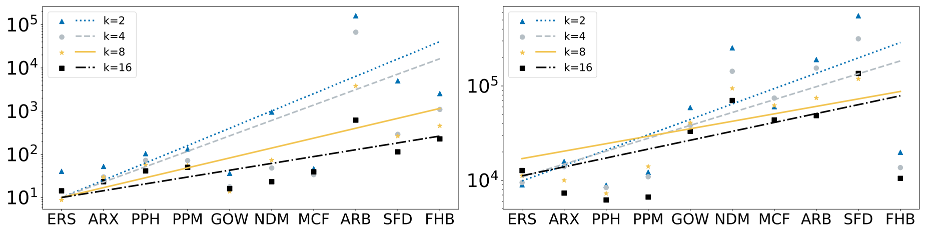

Fig. 1: Speed-up of DECKPLL and FULKPLL against KPLL as a function of graph size in experiments DECR (left) and FULL (right). Lines show linear regressions.

Concerning the impact of parameter k on the performance of considered algorithms, our data show that the speed-up tends to decrease as k increases. This is expected, since the number of affected vertices depends on the number of paths encoded in the index. In particular, the value of k directly impacts on the size of sub-graph that must be visited by Procedure 2 to identify vertices whose loop label entries need to be recomputed (see Eq. 1). Nevertheless, machine-independent measures we collected in our experiments showcase that the cardinality of both sets D_{cy} and AFF-HUBS is always around an order of magnitude smaller than $|V|$, which explains the large speed-ups shown by both DECKPLL and FULKPLL in all tests (see Table 3 for an excerpt of such measures). Finally, we note that in experiment FULL, which is the closest to real-world applications (real networks tend to undergo many incremental updates interleaved by a fraction of decremental ones) algorithm fulKpll is extremely fast at updating the k2hc while query times on the k2hc remain very low.

Experiment amortized. To consolidate the above observations on the effectiveness of decKpll and fulKpll, and for the sake of fairness in the comparison, in what follows we present the results of an experiment, called amortized, whose aim is evaluate the average running time to perform k shortest paths queries by available solutions for dynamic graphs, regardless of whether they rely on indexing or not, in an amortized sense. In this direction, a common methodology, adopted in the literature for approaches that support queries on the graph with or without indexing [16], is to consider performance indicator *cumulative time* to execute a set of q queries. Such indicator includes preprocessing, query and update times for methods that rely on an initialization phase and pre-computed data (such as the kPll index with decKpll or fulKpll) while it includes only the running time of the query algorithm for methods that do not use any auxiliary data for query acceleration (e.g. kBFS or Eppstein's). To apply such methodology to our context, we executed the tests described in

Graph	$\|V\|$	$\|E\|$	k	$\|$ D$_{cy}$ $\|$	$\|$ aff-hubs $\|$	**Graph**	$\|V\|$	$\|E\|$	k	$\|$ D$_{cy}$ $\|$	$\|$ aff-hubs $\|$
pph	16 820	449 036	2	25	153	arb	163 598	1 747 269	2	96	10 491
			4	47	425				4	567	11 165
			8	63	1 123				8	808	36 151
			16	99	799				16	3 341	43 628

Table 3: Excerpt of machine-independent measures.

what follows. For each input graph, we randomly generate a number $\eta = 110$ of graph updates and a set of $q = 10^3$ vertex pairs. We pre-process the graph to compute a k2hc by kPll and measure the time spent by the pre-processing routine. Then, for η times, we repeat the following process: (i) we apply one of the η graph updates; (ii) on the one hand, we execute and measure the total running time to determine a set of k shortest paths, for the q selected vertex pairs, by the kBFS algorithm (identified as the best option in practice to extract k shortest paths from real-world graphs without indexing [1]); (ii) on the other hand, we update the k2hc by dynamic algorithms, and compute answers to the same set of q queries, while measuring both the time for updating the k2hc and for q executions of the query routine on the k2hc. We execute twice per graph the above experiment, by fixing the η operations to be all decremental or of any type, and by therefore applying either decKpll or fulKpll, accordingly (again, distribution of updates is inspired to real-world scenarios [7, 9]).

An exceprt of the results of experiment amortized is shown in Fig. 2. We report, for one of the largest instances considered in this study, namely graph arb: (i) the cumulative running time of decKpll or fulKpll, i.e. the time taken to build the initial k2hc index, summed to the time spent (by decKpll or fulKpll) to update it, for all η graph modifications (decremental or of arbitrary type), and to the time to answer the q queries, after each graph change, via the index; (ii) the cumulative running time of kBFS which is simply the total execution time of kBFS for extracting the same set of k shortest paths. Results for other graphs are similar, lead to equivalent conclusions, and hence are omitted due to space limitations. Our experiments show how dynamic indexing-based approaches outperform the kBFS method by up to more than an order of magnitude (even for the largest inputs and values of k), and regardless of updates being decremental only or arbitrary. In fact, we observe (see Fig. 2) the cumulative running time of decKpll in graph arb to be more than an order of magnitude smaller than that of kBFS. Similarly, fulKpll is from around 40 (80, resp.) times faster than kBFS to answer queries when $k = 16$ ($k = 8$, resp.). To summarize, our experimental work represents a strong empirical evidence of the fact that maintaining a k2hc index via fulKpll is the most effective strategy to support the retrieval of k shortest paths in dynamic, possibly large graphs.

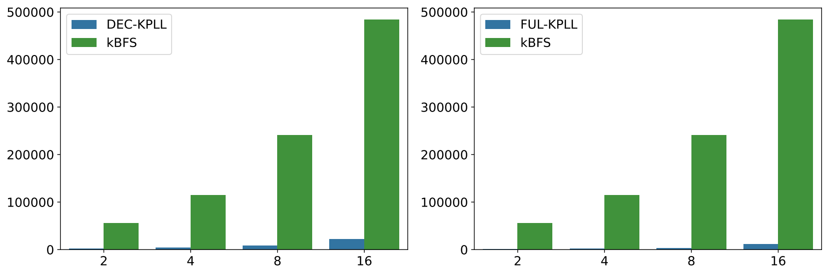

Fig. 2: Cumulative running time of decKpll (left) and fulKpll (right) compared to that of kBFS on graph arb (right), for all values of k.

5 Conclusion

In this paper, we have advanced the state-of-the-art in methods for mining graphs for k shortest paths in dynamic graphs. We have provided a full dynamization of framework kPll by introducing decKpll, the first algorithm to efficiently maintain a k2hc under *decremental* modifications. We have assessed its practical effectiveness through extensive experimental work and have shown that, on average, it updates kPll indices orders of magnitude faster than the re-computation from scratch while preserving its compactness and competitive query performance. We have experimentally shown that decKpll can be used in combination with incKpll, the only known algorithm to maintain a k2hc under *incremental* modifications hence showcasing the first method that supports very fast, interactive retrieval of k shortest paths from fully dynamic graphs.

Acknowledgments. Work partially supported by: Italian Ministry of University and Research through Project "EXPAND: scalable algorithms for EXPloratory Analyses of heterogeneous and dynamic Networked Data" (PRIN grant n. 2022TS4Y3N), funded by the European Union - Next Generation EU; Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM); Project EMERGE, innovation agreement between MiSE, Abruzzo Region, Radiolabs, Elital, Leonardo, Telespazio, University of L'Aquila and Centre of EXcellence EX-Emerge (Grant n. 70/2017); PNRR MUR Project SCHEDULE "Smart interseCtions witH connEcteD and aUtonomous vehicLEs" (PE00000001, CUP J33C22002880001).

References

1. Akiba, T., Hayashi, T., Nori, N., Iwata, Y., Yoshida, Y.: Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling. In: Proc. of 29th Int'l Conference on Artificial Intelligence. pp. 2–8. AAAI Press (2015)
2. Akiba, T., Iwata, Y., Yoshida, Y.: Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In: Proceedings of 2013 ACM International Conference on Management of Data (SIGMOD 2013). pp. 349–360. ACM (2013)
3. Akiba, T., Iwata, Y., Yoshida, Y.: Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In: Proceedings of 23rd Int'l World Wide Web Conference (WWW 2014). pp. 237–248. ACM (2014)
4. Chondrogiannis, T., Bouros, P., Gamper, J., Leser, U., Blumenthal, D.B.: Finding k-shortest paths with limited overlap. The VLDB Journal **29**(5), 1023–1047 (2020)
5. Cionini, A., D'Angelo, G., D'Emidio, M., Frigioni, D., Giannakopoulou, K., Paraskevopoulos, A., Zaroliagis, C.D.: Engineering graph-based models for dynamic timetable information systems. J. Discrete Algorithms **46-47**, 40–58 (2017)
6. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM Journal on Computing **32**(5), 1338–1355 (2003)
7. 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 **20**, 1.5:1.1–1.5:1.33 (2015)
8. D'Angelo, G., D'Emidio, M., Frigioni, D.: Fully dynamic 2-hop cover labeling. ACM J. Exp. Algorithmics **24** (2019)
9. D'Ascenzo, A., D'Emidio, M.: Top-k distance queries on large time-evolving graphs. IEEE Access **11**, 102228–102242 (2023)
10. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Robust distance queries on massive networks. In: Proc. of 22th European Symposium on Algorithms (ESA 2014). Lecture Notes in Computer Science, vol. 8737, pp. 321–333. Springer (2014)
11. D'Emidio, M., Khan, I.: Dynamic public transit labeling. In: Proceedings of 19th International Conference on Computational Science and Its Applications (ICCSA 2019). Lecture Notes in Computer Science, vol. 11619, pp. 103–117. Springer (2019)
12. Eppstein, D.: k-best enumeration. In: Encyclopedia of Algorithms, pp. 1003–1006. Springer (2016)
13. Farhan, M., Koehler, H., Wang, Q.: Batchhl+: batch dynamic labelling for distance queries on large-scale networks. VLDB J. **33**, 101–129 (2024)
14. Farhan, M., Wang, Q.: Efficient maintenance of highway cover labelling for distance queries on large dynamic graphs. World Wide Web (WWW) **26**(5), 2427–2452 (2023)
15. Feng, Q., Peng, Y., Zhang, W., Lin, X., Zhang, Y.: DSPC: efficiently answering shortest path counting on dynamic graphs. In: Proceedings of 27th International Conference on Extending Database Technology (EDBT 2024). pp. 116–128. OpenProceedings.org (2024)

16. Hanauer, K., Henzinger, M., Schulz, C.: Faster fully dynamic transitive closure in practice. In: Proceedings of 18th International Symposium on Experimental Algorithms (SEA 2020). LIPIcs, vol. 160, pp. 14:1–14:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
17. Hao, X., Lian, T., Wang, L.: Dynamic link prediction by integrating node vector evolution and local neighborhood representation. In: Proceedings of 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval. p. 1717–1720. SIGIR '20, ACM, New York, NY, USA (2020)
18. Hassan, M.S., Aref, W.G., Aly, A.M.: Graph indexing for shortest-path finding over dynamic sub-graphs. In: Proc. of 2016 ACM Int'l Conference on Management of Data (SIGMOD 2016). pp. 1183–1197. ACM (2016)
19. Jamil, H.M.: Efficient top-k shortest path query processing in sparse graph databases. In: Proceedings of 7th International Conference on Web Intelligence, Mining and Semantics (WIMS 2017). ACM (2017)
20. Lebedev, A., Lee, J., Rivera, V., Mazzara, M.: Link prediction using top-k shortest distances. In: Proceedings of 31st British Int'l Conference on Databases (BICOD 2017). Lecture Notes in Computer Science, vol. 10365, pp. 101–105. Springer (2017)
21. Ni, P., Hanai, M., Tan, W.J., Cai, W.: Efficient closeness centrality computation in time-evolving graphs. In: Proc. of 2019 IEEE/ACM Int'l Conference on Advances in Social Networks Analysis and Mining (ASONAM). pp. 378–385. ACM (2019)
22. Peixoto, T.P.: The netzschleuder network catalogue and repository (2020), https://networks.skewed.de/
23. Peng, Y., Lin, X., Zhang, Y., Zhang, W., Qin, L.: Answering reachability and k-reach queries on large graphs with label constraints. VLDB J. **31**(1), 101–127 (2022). https://doi.org/10.1007/s00778-021-00695-0
24. Qiu, K., Zhao, J., Wang, X., Fu, X., Secci, S.: Efficient recovery path computation for fast reroute in large-scale software-defined networks. IEEE Journal on Selected Areas in Communications **37**(8), 1755–1768 (2019)
25. Rossi, R.A., Ahmed, N.K.: The network data repository with interactive graph analytics and visualization. In: Proceedings of 29th AAAI Conference on Artificial Intelligence (AAAI 2015). p. 4292–4293. AAAI Press (2015)
26. Wang, S., Lin, W., Yang, Y., Xiao, X., Zhou, S.: Efficient route planning on public transportation networks: A labelling approach. In: Proceedings of 2015 ACM Int'l Conference on Management of Data (SIGMOD 2015). p. 967–982. ACM (2015)
27. Yen, J.Y.: An algorithm for finding shortest routes from all source nodes to a given destination in general networks. Quarterly of applied mathematics **27**(4), 526–530 (1970)
28. Yue, H., Guan, Q., Pan, Y., Chen, L., Lv, J., Yao, Y.: Detecting clusters over intercity transportation networks using k-shortest paths and hierarchical clustering: a case study of mainland china. International Journal of Geographical Information Science **33**(5), 1082–1105 (2019)
29. Zhang, Y., Yu, J.X.: Hub labeling for shortest path counting. In: Proceedings of 2020 ACM Int'l Conference on Management of Data (SIGMOD 2020). p. 1813–1828. Association for Computing Machinery (2020)
30. Zoobi, A.A., Coudert, D., Nisse, N.: Space and Time Trade-Off for the k Shortest Simple Paths Problem. In: Proceedings of 18th Int'l Symposium on Experimental Algorithms (SEA 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 160, pp. 18:1–18:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2020)