IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 21 August 2023, accepted 14 September 2023, date of publication 18 September 2023,
date of current version 22 September 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3316602

RESEARCH ARTICLE

Top-k Distance Queries on Large Time-Evolving Graphs

ANDREA D'ASCENZO AND MATTIA D'EMIDIO
Department of Information Engineering, Computer Science, and Mathematics, University of L'Aquila, 67100 L'Aquila, Italy
Corresponding authors: Andrea D'ascenzo (andrea.dascenzo@graduate.univaq.it) and Mattia D'emidio (mattia.demidio@univaq.it)

This work was supported in part by the Italian National Group for Scientific Computation of Istituto Nazionale di Alta Matematica (GNCS-INdAM).

ABSTRACT Fast extraction of top-k distances from graph data is a primitive of paramount importance in the fields of data mining, network analytics and machine learning, where ranked distances are exploited for several purposes (e.g. link prediction or network classification). While investigation on computational methods to address this retrieval task for regularly sized, static inputs has been extensive, much less is known when managed graphs are *massive*, i.e. having millions of vertices/edges, and *time-evolving*, i.e. when their structure can grow over time, a scenario that introduces a number of scalability and effectiveness issues otherwise not arising. Since, nowadays, most real-world applications exploiting top-k distances have to handle inherently dynamic and rapidly growing graphs, in this paper we present the first *dynamic indexing scheme* that supports very fast queries on top-k distances when graphs are massive and incrementally time-evolving. We assess the scalability and effectiveness of our method through extensive experimentation on both real-world and artificial graph datasets.

INDEX TERMS Algorithm engineering, dynamic algorithms, k shortest distances, massive graph mining.

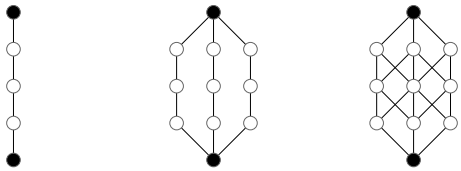
I. INTRODUCTION

Mining path-related properties (e.g. distances, communities, or centrality measures) is considered a fundamental operation to be performed on graph data, for several reasons. Chiefly, such model of data is one of the most used in computing systems, due to its effectiveness in capturing the inherent networked nature of many domains, and algorithms to quickly compute such properties represent indispensable tools in many prominent real-world scenarios where graph datasets have to be managed [37]. For instance, distances and centrality measures are largely exploited to accomplish artificial reasoning and machine learning tasks or for network optimization purposes, locally connected communities and eccentricities are employed to support many meaningful network analytics processes [3], [44], [47], [49], [59], [60]. Due to such applicability, computational problems related to the aforementioned properties have been deeply investigated in the literature, and for most of them efficient algorithms, with polynomially-bounded time complexities, are well-known since decades [28], [37], [57].

Nonetheless, a recent trend of research has been concerned with the scalability issues that most of such algorithms exhibit and, in particular, with the poor performance they show when applied to so-called *massive graphs*, i.e. graphs having millions of vertices and edges. In these cases, in fact, (even) polynomial-time algorithms can often yield unsustainable running times in practice, which are incompatible with the requirements of modern data-intensive information systems [4], [19], [39]. For this reason, and since massive graphs are pervading computing and data management applications, researchers and practitioners have been motivated to design innovative algorithmic strategies to achieve faster solutions to many computational problems of interest, at least from a practical viewpoint and/or for special graph classes [4], [8], [12], [19].

A particularly active area in this context is that dedicated to the so-called *k shortest distances (*K-SD*) problem* which asks to retrieve, upon a *query*, the *top-k distances* for a pair of vertices of a graph, i.e. the lengths of the k shortest paths connecting the pair. Fast computation of top-k distances

The associate editor coordinating the review of this manuscript and approving it for publication was Chong Leong Gan.



Graph	Top-1 Distance	Top-k Distances
(a)	4	[4, 6, 6, 6, 6, 8, 8, ...]
(b)	4	[4, 4, 4, 6, 6, 6, 6, ...]
(c)	4	[4, 4, 4, 4, 4, 4, 4, ...]

FIGURE 1. Top-1 distance versus Top-k distances for the two black vertices in three different graphs [3].

from graph data is a primitive of paramount importance in the fields of data mining, information retrieval, and machine learning, where ranked distances are exploited for diverse purposes [3], [22], including networks design/optimization, speech-recognition, hypertext classification, reconstruction of metabolic pathways, analysis of gene networks, similarity searching, link prediction. In general, the problem finds applications in all those analytics tasks where classic shortest-path distances are not enough informative to characterize the structural properties of a graph dataset. A paradigmatic example is given by all those real-world scenarios where, due to the well-known small-world phenomenon, the diameter of the managed graph is small. As a result, on the one hand many pairs of vertices are at the same (shortest path) distance. On the other hand, such pairs typically do not share the same set of top-k distances (see Fig. 1 for an example). Hence, based on the distance information alone, many pairs of vertices (or vertices) could be considered as equally relevant, which is by no means a realistic assumption; on the contrary, considering ranked, top-k distances is crucial to address mentioned analytics tasks with accuracy and effectiveness. We refer the interested reader to [22] and [44], and references therein, for a more thorough list of applications relying on effective computation of top-k distances, including the prominent setting of high-accuracy link prediction [3], [44].

The reference approach for solving the K-SD problem, in terms of worst-case time complexity, is Eppstein's algorithm [21] which takes $O(n+m+k)$ ($O(n\log n+m+k)$, respectively) time to compute the k shortest distances for a pair of vertices on an unweighted (weighted, respectively) n-vertex, m-edge graph. However, despite the polynomial time complexity, the approach is known to exhibit unsatisfactory performance, and to scale poorly with respect to the graph size and the value of k, as it can require up to tens of seconds to answer a query on the top-k distances for a single pair of vertices in large graphs [3].

Since in the above-mentioned applications this kind of queries must be interactively computed for many vertex pairs on large graphs, the authors of [3] introduce an algorithmic framework, called K-PLL in what follows, to allow the extraction of top-k distances from massive graphs with practical running times. The method divides the computational effort in two steps: (i) in an offline phase, a one-time preprocessing of the input is performed, with the aim of computing a compact data structure, named *k-2-Hop-Cover index* (K–2HC index or simply K–2HC, for short), that stores appropriately selected lengths of paths and cycles in the graph; (ii) at runtime, upon a query on the top-k distances for a pair of vertices, an appropriate *query algorithm*, that takes as input only the K–2HC data structure, is executed to answer such queries very quickly. Specifically, while in terms of worst-case query time and space complexities K-PLL is not better than Eppstein's algorithm, in [3] it is experimentally shown that K-PLL performs very well in practice and enables answering to top-k distance queries within few microseconds per vertex pair (thousands of times faster than Eppstein's algorithm), at the price of at most few thousands of seconds of preprocessing time and of storing some GBs of indexing data, even when graphs have millions of edges. Hence, K-PLL is considered the state-of-the-art for solving the K-SD problem at scale.

Unfortunately, the solution proposed in [3], as many similar methods for massive graph processing based on indexing techniques [17], [19], is not suited to be adopted in scenarios where the input graph is *time-evolving* (also known as *dynamic*), i.e. when the graph topology and/or edge weights can change over time. In fact, essentially all approaches that rely on preprocessing to obtain fast query answering, do not natively guarantee correctness when the managed graph grows over time since, even after few modifications on the input, precomputed data structures might become *obsolete* (i.e. no longer properly reflecting the underlying graph structure) and therefore lead to incorrect query results [5], [17], [24], [63]. This is the case of the framework of [3], as it is easy to see how, even after a single update to the topology of the input graph (e.g. an edge insertion), an arbitrary number of lengths of paths and cycles, stored in the index, can become obsolete and therefore potentially lead to incorrect top-k distances returned by the query algorithm.

To the best of our knowledge, the only possibility to use the K–2HC index on a time-evolving network (and hence to solve the K-SD problem very quickly at scale when graphs dynamically change) is to recompute the data structure from scratch after each update to the network occurs. The latter, however, cannot be considered a viable option in practice since the precomputation step, though effective, generally induces non-negligible time overheads, incompatible with data-intensive applications that rely on ranked distances.

Since, as well-documented in the literature [9], [15], [23], [32], [54], most real-world applications exploiting ranked distances deal with inherently dynamic and rapidly growing graphs, the availability of effective *dynamic algorithms*, able to identify and update only the part of the data structure that is compromised by some graph change faster than the preprocessing routine, is essential to enable the retrieval of top-k distances in temporal contexts. Again to the best of our

knowledge, no method of this kind exists for K-PLL and hence for the K-SD problem at scale, while similar investigations have been successfully conducted for preprocessing-based methods for extracting other path-related properties from large-scale graph datasets [5], [17], [30], [33], [34].

Our Contribution. In this paper, we design DYN-KPLL, an incremental dynamic algorithm that is able to keep k-2-Hop-Cover indexes updated when graphs can grow over time, i.e. when they are subject to incremental updates (vertex/edge insertions, or weight decreases). We prove its correctness, give its time complexity and present the results of extensive experimentation, involving both real and artificial time-evolving graphs, to demonstrate its scalability and effectiveness.

Specifically, we provide strong empirical evidences of DYN-KPLL: (i) being able to update K−2HC indexes very quickly, by running several orders of magnitude faster than the recomputation from scratch, even for massive graphs; (ii) being capable of preserving the compactness of the data structure and thus its competitive performance in terms of time to answer top-k distance queries. Thus, our method can be considered the first *dynamic indexing scheme* that natively supports very fast answers to top-k distance queries in large time-evolving graphs.

It is worth remarking here that the focus of this work is on incremental updates only for several reasons, well-motivated in the literature [5]. Specifically, such updates are the most frequent types of updates that occur in the real-world domains where ranked distances are exploited (e.g. co-authorship, co-occurrence, and interaction networks, are just few examples of graphs that can only grow over time, digital social networks are instead an example of graphs in which decremental updates - removal of edges or nodes or weight increases - are extremely rarer than incremental ones); second, no method is known to answer to top-k distance queries with the same excellent performance of K-PLL under dynamic conditions, without re-executing a preprocessing routine from scratch, hence designing an effective incremental algorithm represents a first step toward using K-PLL with time-evolving graphs; third, addressing incremental updates very often represent the first natural step to understand the inherent complexity of the problem of handling generic updates, and to drive the design of techniques to effectively attack it [17], [24], [43].

A. RELATED WORKS

The problem of computing ranked, top-k distances and paths has been largely investigated in the last decades, in several variants and flavors and within various domains of computer science and engineering. Perhaps the problem that is most closely related to the K-SD problem and that has been received similar attention in the literature is the so-called *k-Simple Shortest Paths problem* (or kSiSP) which asks to find, upon a query, the top-k shortest paths, in terms of length, that connect a given pair of vertices of a graph and are simple, i.e. that do not self-intersect or contain loops (differently w.r.t. the K-SD problem where loops must be taken into account, as part of the graph structure, while computing ranked distances). This version of the problem finds application in specific domains such as e.g. data routing for communication networks or journey planning in transit networks. Despite the evident similarities with the K-SD problem, the kSiSP is generally considered computationally harder. In particular, the best worst-case running time for this problem is that of Yen's algorithm [61], which requires $O(kn(m + n\log n))$ time to compute the top-k simple shortest paths for a vertex pair of a graph having n vertices and m edges. Such approach remains, to this day, the reference method to address the problem, even tough several attempts at improving its running time has been made in the last forty years. Specifically, Gotthilf et al. have managed to improve the algorithm to run in $(O(kn(m + n\log\log n)))$ worst-case time [29]) and there exists an algorithm that, for undirected graphs only, yields an $O(k(m + n\log n))$ worst-case running time, by Katoh et al. [38]. Nonetheless, both methods have been shown, experimentally, to exhibit a performance in practice that is similar to that of Yen's solution, with peak performance on moderate to large diameter graphs such as square grids or large road networks in the undirected variant [2], [26], [50]. Some heuristics, to achieve the computation of top-k simple shortest paths faster than above mentioned strategies, at least from a practical perspective, or for special graph classes or under assumptions on the computational model (e.g. in distributed settings), have been proposed in the recent past and are worth being mentioned to complete the overview on available solutions to the problem, see e.g. [11], [27], [36], [42], [55], [62], and [64].

However, none of them exhibit the same scalability properties, and query performance at scale, of methods based on preprocessing that have been proposed for other relevant problems of the graph mining domain, such as the method of Akiba et al. for the K-SD problem [3] or that by Delling et al. for plain shortest path distance queries [19]. In this sense, designing an algorithmic method to compute top-k simple shortest paths with small running times per query (within microseconds, compatible with interactive applications) when the managed graph is massive, is still an open problem and represents a very active area of research [64].

Most successes in the direction of designing scalable methods for performing graph mining operations on massive graphs have been obtained through consolidated algorithmic techniques, such as preprocessing [4], [19], [40], [64], sampling [6], [15], [53], approximation [7], [9], [14], [46] or core-level parallelization [18], [35], [50], [58].

Among strategies based on preprocessing, those that rely on computing indices in the form of vertex labelings have represented a significant portion of such progress, especially for path-related problems [63]. In particular, the hub-labeling technique, originally introduced for connectivity problems on large graphs by Cohen et al. [13], has been adapted to

solve several problems on large graph data: besides the aforementioned work of Delling et al. for shortest path distances [19], it is certainly worth mentioning the studies of: (i) Wang et al. [59], in which hub-labelings are exploited to accelerate the computation of best routes on timetable graphs; (ii) Abraham et al. where labelings are combined with hierarchical strategies to speed-up the computation of shortest paths in road networks [1]; (iii) Peng et al. in which a precomputed labeling is exploited to answer reachability queries up to 5 orders of magnitude faster than state-of-the-art [52]; (iv) Zhang et al. which adapt hub-labeling techniques to efficiently retrieve the number of shortest paths (i.e. the number of paths having the same, which is also the shortest, length) between any pair of vertices [63].

Essentially all studies on acceleration of algorithms for large graph mining by precomputation of suited data structures have been followed by investigations on corresponding *dynamic algorithms* to update/maintain such data structures under dynamic conditions, i.e. when the given input graph is time-evolving, in order to amortize the time necessary to the preprocessing (i.e. to avoid the recomputation from scratch of the data structure) whenever the graph is subject to some modification. The latter is universally considered a far more realistic setting with respect to static, non-changing graphs. Examples include the design and experimental evaluation of dynamic algorithms for shortest path trees [16], for transitive closures [30], for centrality measures [9], [58], or for graph-based timetable models [12].

Similar works have been concerned with the design and experimental evaluation of dynamic algorithms to update/maintain labeling based indices, such as e.g.: the work of Akiba et al. to update 2-hop-cover labelings when graphs are subject to incremental modifications (edge/vertex insertions) [5]; the work by D'Angelo et al., which extended the approach of [5] to handle the fully dynamic scenario (when the managed graph can undergo also edge/vertex deletions) [17]; and the work of [25] which improved the overall performance in terms of space and preprocessing/update time of [5] and [17] by considering an hybrid, landmark-based strategy that induce larger query times; the studies in [23] and [24], which have focused on the effect of batch of updates occurring simultaneously on mentioned hybrid labelings.

Note that, a thorough survey on recent advances in the field of dynamic graph algorithms has been drawn up by Hanauer et al. in [31].

II. PRELIMINARIES

In this study we focus on networks that are modeled as a graph $G = (V, E)$ with a vertex set V and an edge set E. We denote by $n = |V|$ ($m = |E|$, respectively) the number of vertices (edges, respectively) of G. To simplify our discussion, we consider only undirected, unweighted graphs first. Nonetheless, the method presented in this paper can be extended to weighted digraphs, as discussed in Section III-A. A *path* $p = (s = v_1, v_2, \ldots, t = v_r)$ in G, connecting a pair of vertices $s, t \in V$ (its *endpoints*), is a sequence of r vertices such that $\{v_i, v_{i+1}\} \in E$ for all $i \in [1, r-1]$. We call *cycle* any path whose endpoints coincide while we call *simple* a path with no self-intersections, i.e. without vertex repetitions. An *internal vertex* of a path p is a vertex in p different from its endpoints. The *length* $\ell(p)$ of a path p is the number of edges in p; note that, for non-simple paths, path length considers possible multiplicities of occurrences of edges. A *shortest path* $p(s, t)$, for a pair of vertices $s, t \in V$, is a path having minimum length among all those in G connecting s and t. The *distance* $d(s, t)$ between s and t is the length of a shortest path $p(s, t)$.

We assume vertices are uniquely represented by integers, so to enable natural comparisons for any pair $u, v \in V$ by expressions such as $u < v$ or $u \leq v$. Given any two vertices $s, t \in V$, we define: (i) $\mathcal{P}_{st}$ to be the set of paths connecting s and t in G; (ii) $\mathcal{P}_{st}^{>v}$ to be the set of paths in $\mathcal{P}_{st}$ whose internal vertices are all larger than v, for some $v \in V$; (iii) $\mathcal{P}_{st}^{\not>v}$ to be the set of paths in $\mathcal{P}_{st}$ 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, that is the i-th element in $\mathcal{P}_{st}$, sorted in non-decreasing order according to 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 the i-th shortest path in $\mathcal{P}_{st}$. Similarly, we use $d_i^{>v}(s, t)$ ($d_i^{\geq v}(s, t)$ and $d_i^{\not>v}(s, t)$, respectively) to refer to the i-th shortest distances when paths are restricted to consider only internal vertices that are larger (larger than or equal to and not greater, respectively) than some $v \in V$. Similarly, we use $p_i^{>v}(s, t)$ ($p_i^{\geq v}(s, t)$, respectively) to refer to the corresponding i-th shortest paths, and $d^{>v}(s, t) = d_1^{>v}(s, t)$ ($p^{>v}(s, t) = p_1^{>v}(s, t)$, respectively) to refer to the distance (a shortest path inducing such distance, respectively) subject to the same restrictions on vertices. Finally, we call $\deg_v$ the degree of a vertex $v \in V$, that is the number of *neighbors* $\{w : (v, w) \in E\}$ of v. We use $\deg_v^{>v}$ to denote the number of such neighbors $> v$.

A. K SHORTEST DISTANCES PROBLEM

Given a graph $G = (V, E)$, an integer $k \geq 1$, and a pair of vertices $s, t \in V$, the *k shortest distances* (K-SD) problem asks to compute the set $\mathcal{D}_{st}^k = \{d_1(s, t), d_2(s, t), \ldots, d_k(s, t)\}$ of the k shortest distances between s and t in G. The framework of [3] is the state-of-the-art approach to address the K-SD problem at scale. It is based on the computation of a data structure called *k-2-Hop Cover index*, which is a generalization of the 2-*Hop Cover index*, originally introduced in [4], and defined as follows.

Definition 1 (k-2-Hop Cover Index): Given a graph $G = (V, E)$, define, for each vertex $v \in V$: (i) a length label $L(v)$, containing pairs (u, δ_{uv}) where $u \in V$ and δ_{uv} is the length of a path from u to v; (ii) a loop label $C(v)$, storing a sequence of k integers $(\delta_1, \delta_2, \ldots, \delta_k)$ representing lengths of cycles in G that include vertex v. Then, the 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.

A k-2-Hop Cover index is often referred to as k-2-Hop Cover labeling or simply as k-2-Hop Cover (K–2HC for short); we use these notations interchangeably and refer to elements in the labels as *entries*. Depending on the entries stored in a K–2HC, such data structure can be used to solve the K-SD problem correctly or not. Specifically, this hold when the index satisfies the so-called *k-cover property*, which we define as follows:

Definition 2 (k-Cover Property): Given K–2HC $I = (L, C)$ *of a graph* $G = (V, E)$*, let* QUERY(I, s, t) *denote a query on* I *for a pair of vertices* $s, t \in V$*, that returns the smallest* k *elements from multiset* $\Delta(I, s, t) = \{\delta_{vs} + \delta_{vv} + \delta_{vt} | (v, \delta_{vs}) \in L(s), \delta_{vv} \in C(v), (v, \delta_{vt}) \in L(t)\}$*. Then,* I *satisfies the* k*-cover property if and only if, for any* $s, t \in V$*, we have* QUERY$(I, s, t) = \{d_1(s, t), d_2(s, t), \ldots, d_k(s, t))\}$.

In other words, an index satisfying the k-cover property for a graph G allows to retrieve the k shortest distances in G, for any pair of vertices $s, t \in V$, by a query on the index that selects the smallest summations in $\Delta(I, s, t)$, obtained by properly combining values of lengths of paths and cycles. Specifically, such combinations are obtained by summing the length of a path from s to some vertex v, the length of a cycle on v, and the length of a path from v to t. We say index $I covers G$ whenever I satisfies the k-cover property for G. Any vertex v that form one of the k smallest combinations in $\Delta(I, s, t)$ is called a *hub vertex* for pair s, t, for any $s, t \in V$. Clearly, whenever a pair s, t is disconnected in G then $d_i(s, t) = \infty \, \forall i \in [1, k]$ and QUERY(I, s, t) returns a single. default infinity value whenever there is no vertex $v \in V$ such that $(v, \delta_{sv}) \in L(s), \delta_{vv} \in C(v), (v, \delta_{tv}) \in L(t)$. An example of K–2HC $I = (L, C)$ covering a graph is shown in Figure 2. The *size* of the index is defined to be the total number of entries in all labels, both of length and loop type, and it can be easily shown that computing a K–2HC covering a graph and having minimum size is NP-hard: this follows from the hardness of computing a minimum sized 2-hop cover index [13]. Moreover, computing a K–2HC having size $O(kn^2)$ can be easily achieved by, e.g., $O(n^2)$ executions of the Eppstein's algorithm.

To the best of our knowledge, no algorithm is known for computing a K–2HC with a guarantee on the approximation on the size of the index. The method in [3] however achieves practical performance, in terms of trade-off between preprocessing time, index size and query time, and is currently considered the most effective framework to solve the K-SD problem for large graphs. Such method is based on precomputing a K–2HC that covers a given input graph by (i) sorting vertices according to some easy-to-compute centrality measure (e.g. degree); (ii) filling both loop and length labels progressively, by performing appropriately modified visits of the graph, each rooted at a different vertex of the graph, following the established sorting; (iii) incorporating a stopping criterion that prunes the searches whenever no length, shorter than those already stored, can be found. The preprocessing strategy to build a K–2HC is summarized in Algorithm 1 and consists of two main sub-routines, named MOD-BFS and PRUN-KSD, given in Algorithms 2 and 3, respectively.

Algorithm 1 Algorithm K-PLL

Input: Graph $G = (V, E)$, integer $k > 0$, vertex ordering $v_1, v_2, \ldots, v_n$
Output: K–2HC I covering graph G

for $i = 1 \ldots n$ **do** /* Computation of Cycle Labels */
  $C(v_i) \leftarrow$ MOD-BFS(v_i, k);
for $i = 1 \ldots n$ **do** /* Computation of Length Labels */
  $L(v_i) \leftarrow$ PRUN-KSD(v_i);
return $I = (\{C(v)_{v \in V}\}, \{L(v)_{v \in V}\})$;

Algorithm 2 Sub-Routine MOD-BFS of Algorithm 1

Input: Vertex $v \in V$, integer $k > 0$
Output: Loop label $C(v)$

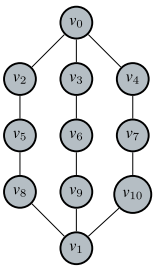
$C(v) \leftarrow \emptyset$;
foreach $t \in V$ **do** $visited[t] \leftarrow 0$;
$visited[v] \leftarrow 1$;
$Q \leftarrow \{(v, 0)\}$;
while $Q \neq \emptyset$ **do**
  Dequeue (x, δ) from Q;
  $visited[x] \leftarrow visited[x] + 1$;
  if $x = v$ **then** Add δ to $C(v)$;
  if $visited[x] < k$ **then**
    foreach $w \in V$ such that $(x, w) \in E \wedge w \geq v$ **do**
    Enqueue $(w, \delta + 1)$ into Q;
return $C(v)$;

Algorithm 3 Sub-Routine PRUN-KSD of Algorithm 1

Input: Vertex $v \in V$, integer $k > 0$
Output: Length label $L(v)$

$L(v) \leftarrow \emptyset$;
$Q \leftarrow \{(v, 0)\}$;
while $Q \neq \emptyset$ **do**
  Dequeue (x, δ) from Q;
  if $\delta < \max\{d : d \in$ QUERY$(I, v, x)\}$ **then**
    Add (v, δ) to $L(x)$;
    foreach $w \in V$ such that $(x, w) \in E \wedge w > v$ **do**
    Enqueue $(w, \delta + 1)$ into Q;
return $L(v)$;

Procedure MOD-BFS (PRUN-KSD, respectively) computes loop (length, respectively) labels by performing visits of the graph, each starting from a different vertex v_i, following the vertex sorting, that traverses only vertices larger than or equal to (larger to, respectively) v_i. The construction guarantees that: (i) lengths in $L(v)$, associated with a vertex u, form the sequence $(d_1^{>u}(u, v), d_2^{>u}(u, v), \ldots, d_l^{>u}(u, v))$



id	C	L
v_0	$\{2,2,2\}$	$\{(v_0,0)\}$
v_1	$\{2,2,2\}$	$\{(v_0,4),(v_0,4),(v_0,4),(v_1,0)\}$
v_2	$\{2,4,4\}$	$\{(v_0,1),(v_1,3),(v_2,0)\}$
v_3	$\{2,4,4\}$	$\{(v_0,1),(v_1,3),(v_3,0)\}$
v_4	$\{2,4,4\}$	$\{(v_0,1),(v_1,3),(v_4,0)\}$
v_5	$\{2,2,6\}$	$\{(v_0,2),(v_1,2),(v_2,1),(v_5,0)\}$
v_6	$\{2,2,6\}$	$\{(v_0,2),(v_1,2),(v_3,1),(v_6,0)\}$
v_7	$\{2,2,6\}$	$\{(v_0,2),(v_1,2),(v_4,1),(v_7,0)\}$
v_8	$\emptyset$	$\{(v_0,3),(v_1,1),(v_5,1),(v_8,0)\}$
v_9	$\emptyset$	$\{(v_0,3),(v_1,1),(v_6,1),(v_9,0)\}\}$
v_{10}	$\emptyset$	$\{(v_0,3),(v_1,1),(v_7,1),(v_{10},0)\}$

FIGURE 2. A K–2HC $I = (L, C)$ of a graph. Vertex IDs are assigned in non-increasing order of degree [19].

of the $1 \leq l \leq k$ shortest lengths induced by paths whose internal vertices are larger than u and that are shorter than $d_k^{\succ u}(u,v)$; (ii) lengths in $C(v)$ form the sequence $(d_1^{\geq v}(v,v), d_2^{\geq v}(v,v), \ldots, d_k^{\geq v}(v,v))$ of the lengths of k shortest cycles in G that include v and vertices larger than or equal to v. Properties (i) and (ii), combined, guarantee that the resulting K–2HC satisfies the k-cover property. It is easy to observe that the running time of Algorithm 1 is $O(nkl(n+m))$, if l is the maximum number of entries in any label [3]. Note that, in the reminder of the paper, for the sake of brevity, we use acronym K-PLL also to refer to the preprocessing routine of the framework, i.e. Algorithm 1.

B. TIME-EVOLVING SCENARIOS

We assume we are given an *initial* graph, say $G = (V, E)$, and that such graph can undergo *incremental* modifications (i.e. vertex/edge insertions) for G (i.e. the graph is *time-evolving*). We focus on the *incremental* K-SD *problem* which asks, given an incremental modification x (e.g. the insertion of an edge $e \notin E$) occurring on G, to compute the set $\mathcal{D}_{st}^k = \{d_1'(s,t), d_2'(s,t), \ldots, d_k'(s,t)\}$ of the k shortest distances 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 inserting e into E). Clearly, such problem can be solved, with the same complexity and practical performance, by any algorithm that solves the static counterpart of the problem without relying on preprocessed data (e.g. [21]), as it suffices to execute such algorithm on G', after a change, for the given pair. However, if preprocessed data are exploited to achieve superior query performance, as in the K-PLL framework, then solving the incremental K-SD problem requires updating such data in order to preserve the correctness of the approach, which translates into the definition of the following problem.

Definition 3 (Incremental K–2HC *Problem): Given a graph $G = (V, E)$ and a* K–2HC *I covering G. Let x be an incremental modification of G and let $G' = (V', E')$ be graph obtained by applying x to G. Then, the incremental* K–2HC *problem asks to compute a* K–2HC *I' that covers G'.*

To the best of our knowledge, the only known way to address the incremental K–2HC problem is to recompute from scratch a K–2HC I' covering G' via K-PLL. However, this induces large time overheads. Thus, in the next section we introduce a dynamic algorithm to cope with such problem without executing the preprocessing on each updated graph. Observe that, if the change to be managed is a vertex insertion, this can be modeled and handled as a sequence of edge insertions to a newly inserted vertex [17]. Therefore, in what follows, again for the simplicity of the description' sake, we focus on graphs subject to edge insertions only. In the reminder of the paper, we will use $d_j'(s,t) = \ell(p_i'(s,t))$ to denote the j-th *shortest distance*, $1 \leq j \leq k$, for a pair s, t in a graph G' whenever the meaning is clear from the context.

Algorithm 4 Algorithm DYN-KPLL

Input: Graph $G = (V, E)$, K–2HC $I = (L, C)$ covering G, integer $k > 0$, edge $(x, y) \notin E$, vertex ordering $v_1, v_2, \ldots, v_n$
Output: K–2HC $I = (L, C)$ covering $G' = (V, E')$, $(x, y) \in E'$

```
1 Compute AFF-SET as in Eq. 1;
2 foreach v ∈ AFF-SET do          /* Update Cycle Labels */
3 |  C(v) ← MOD-BFS(v, k)
4 foreach (v, δ_vx) ∈ L(x) do     /* Update Length Labels */
5 |  RESUME-PKSD(v, y, δ_vx)
6 foreach (v, δ_vy) ∈ L(y) do
7 |  RESUME-PKSD(v, x, δ_vy)
```

III. DYNAMIC ALGORITHM

In this section, we introduce our new method, called DYN-KPLL, to solve the incremental K–2HC problem.

The main routine (see Algorithm 4), takes as input a graph G, for which an index $I = (L, C)$ covering G is available, and an incremental update (the insertion of an edge $e = \{x, y\}$) for G. Let G' the graph obtained by inserting e into G; then, the algorithm updates I to obtain an index $I' = (L', C')$, which covers G', by separately performing the update of the loop labeling C and of the length labeling L. Specifically, first the update of C is performed. To this aim, the procedure

Algorithm 5 Sub-Routine RESUME-PKSD of Algorithm 4

Input: Vertex $v \in V$, endpoint $u \in \{x, y\}$ of inserted edge, length l_u of path from v to u induced by the insertion
Output: Updated $L(w)$ for any affected vertex $w \in V$

```
1 Q ← (u, l_u + 1);
2 while Q ≠ ∅ do
3     Dequeue (w, δ) from Q;
4     if δ < max{d : d ∈ QUERY(I, v, w)} then
5         L(w) ← L(w) ∪ (v, δ);
6         foreach z ∈ V such that (w, z) ∈ E ∧ z > v do
          Enqueue (z, δ + 1) into Q;
```

identifies any vertex v for which at least one length in $C(v)$ may be incorrect due to the change, i.e. any vertex such that $C(v)$ does not contain $(d_1'^{\geq v}(v, v), d_2'^{\geq v}(v, v), \ldots, d_k'^{\geq v}(v, v))$. This is done by computing a set AFF-SET, defined as:

$$\text{AFF-SET} = \{v \in V : d^{>v}(v, x) \leq k \vee d^{>v}(v, x) \leq k\} \cap \{v \in V : v \leq \min(x, y) \wedge \deg_v^{>v} < k\}. \quad (1)$$

Such set contains any vertex v: (i) which is connected to either x or y by a shortest path not longer than k and whose internal vertices are greater than v; (ii) whose set of neighbors greater than v has size smaller than k. For each of such vertices, previously found cycle lengths are removed from the loop labels and new ones are computed via procedure MOD-BFS.

Then, the algorithm continues with the update of L, which is achieved by a strategy inspired by the dynamic algorithm of [5]. Essentially, the underlying idea is to resume visits of the graph, rooted at specific vertices, and to prune such visits under certain conditions, in order to update only the length labels of vertices that are affected by the edge insertion. A vertex is said to be *affected* by an insertion if at least an entry must be added to the corresponding length label in order to guarantee that the resulting K–2HC I' covers the new graph G'. Such resumed visits are performed by procedure RESUME-PKSD, shown in Algorithm 5, and mimic those performed by routine PRUN-KSD. More specifically, the update procedure and its pruning mechanism are based on the following observation: if any of the k shortest distances between two vertices s and t changes, then any new value of distance that becomes part of the top-k shortest distances for the pair must be induced by paths from s to t passing through the new edge e in the new graph. Hence, the update procedure must process the graph, after the edge insertion, in order to find those vertices s for which the above condition holds toward some other vertex t, since its length label must be updated to store lengths of paths that induce new distances in the top-k set, and to limit the visit of the graph to such vertices only.

The above is done in two steps: first, we identify candidate pairs of vertices for which at least one value in the set of top-k shortest distances might change because of the new arc. This is achieved by scanning the length labels of the two endpoints of the newly inserted arc. Then, we start BFS-like visits, rooted at vertices that are in such length labels, from either of the two endpoints, and incorporate in such visits a pruning strategy that stops the traversing of the graph, at some vertex, once no more shortest distances induced by paths passing through said vertex can be found.

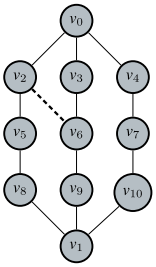
Now, w.l.o.g, we describe the details of the procedure for one endpoint only, say x, as it is symmetric for the other. The algorithm starts by scanning the length label of x and, for each pair $(v, \delta_{vx}) \in L(x)$, we execute procedure PRUN-KSD, that takes as inputs vertices v and y, and value δ_{vx}. Such routine "resumes" a visit, rooted at v, starting from vertex y and extending a path to x of length δ_{vx}. This is done by initializing a suited queue and by exploring the graph in a BFS fashion. Whenever a vertex w, together with its path length δ, is dequeued we test whether the maximum value returned by QUERY(I, v, w) is larger than δ, to evaluate whether any of the k shortest distances to w are shortened by the edge insertion. If this is the case, we add entry (v, δ) to $L(w)$, which corresponds to the shorter length induced by the new path, and continue the search towards neighbors of w having order greater than the root v. On the other hand, if δ is not less than the values returned by QUERY, than the visit from w is pruned. The procedure terminates when either all branches of the visit are pruned at some vertex or when the queue becomes empty. Figure 3 shows the result of the execution of algorithm DYN-KPLL on the K–2HC of Figure 2.

Observe that it can be shown that algorithm DYN-KPLL is able to correctly solve the incremental K–2HC problem, i.e. it is able to update a K–2HC index I, covering a graph G, to a K–2HC index I', covering graph G', which is the graph obtained by inserting an edge into G. Note that, I' satisfying the k-cover property on G' implies that I' can be used to correctly answer to top-k distance queries on G' (i.e. to solve the incremental K-SD problem). More specifically, we can prove the following result.

Theorem 1: Given a graph G and a K–2HC *index $I = (C, L)$ covering G, let G' be the graph obtained by inserting an edge $e \notin E$ into G. Call* K–2HC *$I' = (C', L')$ the updated* K–2HC *computed by Algorithm 4. Then, $I' = (C', L')$ satisfies the k-cover property for G'.*

Let $e = (x, y)$ the edge inserted into G. The proof is divided in two parts: (a) first, we show that $C'(v)$ is correct, i.e. contains lengths $(d_1'^{\geq v}(v, v), d_2'^{\geq v}(v, v), \ldots, d_k'^{\geq v}(v, v))$ for any $v \in V$; (b) then we prove that $I' = (C', L')$ satisfies the k-cover property for G' by showing that the length label $L'(t)$ of any vertex t contains the sequence $(d_1'^{>s}(s, t), d_2'^{>s}(s, t), \ldots, d_l'^{>s}(s, t))$ of the $1 \leq l \leq k$ shortest lengths induced by paths whose internal vertices are larger than s, for any $s \in V$ such that $s < t$, that are shorter than $d_k'^{\not>s}(s, t)$.

Concerning (a), observe that the only step of DYN-KPLL that alters loop labels is line 3. Since we employ the MOD-BFS sub-routine, which computes lengths $(d_1'^{\geq v}(v, v), d_2'^{\geq v}(v, v), \ldots, d_k'^{\geq v}(v, v))$ when invoked on a vertex v [3],



id	C	L
v_0	$\{2,2,2\}$	$\{(v_0,0)\}$
v_1	$\{2,2,2\}$	$\{(v_0,4),(v_0,4),(v_0,4),(v_1,0)\}$
v_2	$\{2,\underline{\mathbf{2}},4\}$	$\{(v_0,1),(v_1,3),\underline{(v_1,\mathbf{3})},(v_2,0)\}$
v_3	$\{2,4,4\}$	$\{(v_0,1),(v_1,3),\underline{(v_3,\mathbf{2})},(v_3,0)\}$
v_4	$\{2,4,4\}$	$\{(v_0,1),(v_1,3),(v_4,0)\}$
v_5	$\{2,2,6\}$	$\{(v_0,2),(v_1,2),(v_2,1),(v_5,0)\}$
v_6	$\{2,2,6\}$	$\{(v_0,2),\underline{(v_0,\mathbf{2})},(v_1,2),$ $\underline{(v_2,\mathbf{1})},(v_3,1),(v_6,0)\}$
v_7	$\{2,2,6\}$	$\{(v_0,2),(v_1,2),(v_4,1),(v_7,0)\}$
v_8	$\emptyset$	$\{(v_0,3),(v_1,1),(v_5,1),(v_8,0)\}$
v_9	$\emptyset$	$\{(v_0,3),\underline{(v_0,\mathbf{3})},(v_1,1),$ $\underline{(v_2,\mathbf{2})},(v_6,1),(v_9,0)\}\}$
v_{10}	$\emptyset$	$\{(v_0,3),(v_1,1),(v_7,1),(v_{10},0)\}$

FIGURE 3. Result of executing DYN-KPLL on the K−2HC of Figure 2 after inserting edge (v_2, v_6). New entries are underlined.

and since we execute such routine for each vertex in set AFF-SET, in order to prove (a) it is sufficient to show that $C'(w) = C(w)$ for vertices $w \notin$ AFF-SET. To this end, by contradiction assume that $C'(w)$ is not correct for some vertex $w \notin$ AFF-SET, i.e. if we take the k shortest values in $C'(w)$, say $l_1, l_2 \ldots, l_k$, then there exists one value l_i for some $i \in [1,k]$ that is longer than the length of one of the k shortest cycles in G', say c, that includes w and vertices larger than or equal to w. Clearly c must include edge e, as otherwise its length would already be in $C(w)$. Now, notice that any vertex $w \notin$ AFF-SET is such that either (i) both $d^{>w}(w,x) \geq k+1$ and $d^{>w}(w,y) \geq k+1$, for some shortest path $p^{>w}(w,x)$ and $p^{>w}(w,y)$, resp., since any vertex $w \in$ AFF-SET satisfies $w \leq \min(x,y)$, or (ii) $\deg_w^{>w} \geq k$ (see line 1). Consider (i): any cycle including edge e and vertex w, whose vertices are larger than or equal to w, must be at least $2k+3$ long. This value of length is higher than all lengths $2, 4, \ldots, 2k$ of the (at least) k cycles that are induced by the traversal, back and forth, in a BFS order and starting from w, of any sub-path of length at most k of either $p^{>w}(w,x)$ or $p^{>w}(w,y)$. Hence, we reach a contradiction. For case (ii) a similar argument can be applied. In fact, if a vertex w is such that $\deg_w^{>w} \geq k$, then the k shortest cycles on w are given by paths of length 2 obtained by traversing back and forth any k edges incident to w. We have thus reached a contradiction also here, since the new edge e does not contribute to the shortest cycles on w, i.e. $C'(w) = C(w)$, and this concludes the proof of (a).

We now focus on (b) and distinguish two cases: $s \notin L(x) \cup L(y)$ or $s \in L(x) \cup L(y)$. Assume that s and t are connected in G, as viceversa shortest distances are infinity in both G and G' and the claim trivially follows. In the first case, i.e. $s \notin L(x) \cup L(y)$, we have that s is not hub vertex in G for any of the k shortest distances from s to both x and y, and that either s is not connected to x and y, or any hub vertex for such pairs, say h, is such that $h < s$. In the former sub-case, no path in G from s to t passes through x and y, hence the same holds for G', since the only difference is the insertion of e, and the claim follows. In the latter sub-case, instead, we have that any hub vertex, say h, for pairs s, x and s, y, is such that $h < s$. This implies that all paths in G, inducing the k shortest distances from s to x, y, have an internal vertex that is smaller than s. Therefore none of the k shortest paths in G from s to t, whose internal vertices are larger than s (if any), passes through x or y. Thus, the corresponding shortest distances are not changed by the insertion and the claim again holds.

We now consider the second case, i.e. $s \in L(x) \cup L(y)$, and prove the statement for sub-case $s \in L(x)$ only, as the proof is symmetric for sub-case $s \in L(y)$. Suppose by contradiction that $s \in L(x)$ but lengths in $L'(t)$, associated to s, do not form the sequence $(d_1'^{>s}(s,t), d_2'^{>s}(s,t), \ldots, d_l'^{>s}(s,t))$ of the $1 \leq l \leq k$ shortest lengths induced by paths whose internal vertices are larger than s and that are shorter than $d_k^{\not>s}(s,t)$. Let $\gamma_1, \gamma_2, \ldots, \gamma_l$ be the sequence of the first $l \geq 0$ lengths associated to s in $L'(t)$ and let γ_i be i–th smallest value in this sequence such that $\gamma_i \neq d_i'^{>s}(s,t)$. Specifically, observe that since we are inserting an edge, we have that $d_i'^{>s}(s,t) < d_i^{>s}(s,t)$ hence $\gamma_i > d_i'^{>s}(s,t)$ can only be an overestimation of the true value $d_i'^{>s}(s,t)$ (and clearly this holds also if $s \notin L(t)$ as in that case $d_1^{>s}(s,t) = \infty$ by hypothesis). Moreover, the path inducing $d_i'^{>s}(s,t)$, say $p_i'^{>s}(s,t)$, must contain edge e, as otherwise we would have $\gamma_i = d_i'^{>s}(s,t)$. We can thus divide the path $p_i'^{>s}(s,t)$ as $p_i'^{>s}(s,x), \{x,y\}, p_i'^{>s}(y,t)$. Since $s \in L(x)$, we have that an execution of procedure RESUME-PKSD is started, rooted at s, by enqueueing $(y, \delta_{sx}+1)$ to Q for each length δ_{sx} in entries of $L(x)$ associated to s. Now, consider the value δ_{sx} induced by the path $p_i'^{>s}(s,x)$, which is such that $(s,\delta_{sx}) \in L(x)$, since I covers G. Then, it is easy to show, by induction on the lengths of paths induce by the visit, that procedure RESUME-PKSD rooted at s, with $(y, \delta_{sx}+1)$ enqueued into Q as initial step, will not be pruned neither in y nor in any of the vertices in $p_i'^{>s}(y,t)$, including t, leading to a contradiction. In fact, suppose that at some vertex w at distance δ_{sw} in the path $p_i'^{>s}(y,t)$ the visit is pruned. Since the path traverses only vertices whose order is higher than s, it must be the case

that δ_{sw} is larger than any of the top-k distances from s to w, provided by the current index. This implies that, in G', there exist k distances $(d_1'^{\not>s}(s,t), d_2'^{\not>s}(s,t), \ldots, d_k'^{\not>s}(s,t))$, induced by the k shortest paths from s to w concatenated to path $p_i'^{>s}(w,t)$, whose total length is less than $d_i'^{>s}(s,t)$, which is clearly a contradiction. Therefore, it follows that no vertex on $p_i'^{>s}(y,t)$ is pruned during routine RESUME-PKSD, which eventually adds entry $(s, d_i'^{>s}(s,t))$ to $L'(t)$. □

Concerning the time complexity of DYN-KPLL, we can prove the following result, expressed in an output bounded sense, a commonly done for dynamic algorithms in the literature [5], [17].

Theorem 2: Let $I = (L, C)$ be a K–2HC *covering a graph G and let l the maximum number of entries in any length label of I. Given an edge insertion x on G, let G' be the graph obtained by applying x to G. Then, algorithm* DYN-KPLL *takes $O(kl^2s + rkc)$ time to update I to a* K–2HC *I' covering G' where r, s and c denote the cardinality of* AFF-SET*, the maximum size of the subgraph visited during any execution of* RESUME-PKSD *and* MOD-BFS*, respectively.*

Observe that DYN-KPLL invokes RESUME-PKSD at most k times for any vertex $v \in L(x)$ ($v \in L(y)$, resp.). Notice also that, in any given execution of DYN-KPLL, the number of such vertices (hence calls to RESUME-PKSD) is $\beta = |L(x) \cup L(y)|$ and that, clearly, β is at most l. Each execution, moreover, in the worst-case takes $O(ls)$ time to perform QUERY on each visited vertex. Thus, the total running time required to update the length labeling by performing β times procedure RESUME-PKSD is $O(\beta(kls))$. Finally, line 3 is executed at most r times, each requiring $O(kc)$ time. Since set AFF-SET is computed in $O(r + c)$ time, the total running time is $O(\beta(kls) + r(kc)) = O(kl^2s + rkc)$, since $\beta = O(l)$. □

It is easy to notice that the time complexity of DYN-KPLL is in the worst-case, asymptotically speaking, larger than that of K-PLL, as l and r are $O(kn)$ and $O(n)$, respectively, while s and c are $O(m)$. However, our experimentation shows that, in practice, such values are by far smaller than the worst case. Moreover, since DYN-KPLL preserves the k-cover property, one can repeatedly solve the incremental K–2HC problem for sequences of modifications of arbitrary length σ in $O(\sigma(kl^2s + rkc))$ time, by updating σ times the index via DYN-KPLL.

A. GENERALIZATIONS

In what follows we briefly discuss on how both K-PLL and DYN-KPLL can be extended to handle general, possibly weighted, digraphs.

1) DIRECTED GRAPHS

In this case, a K–2HC stores three labels for each vertex $v \in V$, to consider edge orientations: (i) $C(v)$, storing lengths of (now oriented) cycles; (ii) $L_{in}(v)$, containing lengths of paths that terminate into v; (iii) $L_{out}(v)$, containing lengths of path emanating from v. The preprocessing phase is adapted to consider both directions and to run twice the preprocessing routine, one in G and one in the transpose graph of G. The visits are pruned by performing properly oriented queries, that combine lengths of paths emanating from s, cycles on hub vertices v, and lengths of paths terminating into t, for a pair s, t. Similarly, DYN-KPLL can be adapted to handle directed graphs by executing PRUN-KSD twice, once in G and once in the transpose graph of G, and by identifying vertices v in AFF-SET (line 1 in Algorithm 4) as in Eq. 1 that also have a path $p^{\geq x}(v,x)$ ($p^{\geq x}(v,y)$ resp.) of length at most k.

2) WEIGHTED GRAPHS

In this case, label entries store *weights* of cycles and paths, rather than lengths, which are the sums of the weights of the edges they are formed of. To build a K–2HC in this setting, one has to apply weighted versions of both MOD-BFS and PRUN-KSD, where the exploration is performed in a Dijkstra's algorithm fashion (i.e. by using a priority queue and by assigning priorities on the basis of path/cycle weights [20]). Similarly, DYN-KPLL can be adapted to handle general incremental changes to the graph, including weight decreases, by: (i) resuming weighted versions of MOD-BFS and PRUN-KSD from, resp., vertices in AFF-SET (line 1 in Algorithm 4) and vertices in the length labels of the updated edge endpoints (lines 4 and 6 in Algorithm 4). The upper bound used to identify vertices in AFF-SET (line 3 in Algorithm 4) is replaced by kW, where W is the largest edge weight in the graph.

IV. EXPERIMENTAL EVALUATION

In this section, we describe the experimental evaluation we conducted to assess the performance of DYN-KPLL.

A. EXPERIMENTAL SETUP

We implemented both K-PLL and DYN-KPLL; all our code is written in C++ and compiled with GCC 9.4.0 with optimization level $O3$.[1] 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. As inputs to our experiments, inspired by other experimental studies on graph algorithms [3], [19], [63], we consider a large collection of both real-world and artificial graphs. The former were taken from publicly available repositories [41], [45], [51], [56] and include graphs representing networks of various application domains of interest (e.g. web graphs) with heterogeneous densities and topologies. The latter were produced via well-established generation models, such as *Erdős-Rényi* and *Barabási-Albert* [10]. More details on used inputs, including number of vertices and edges, type (real-world or synthetically generated), average degree, and diameter (denoted by Γ), are reported in Table 1. Graphs are sorted from top to bottom according to $|V|$.

[1] Publicly available at https://github.com/D-hash/IncrementalK2HC

TABLE 1. Overview of input graphs.

Graph	Short	Type	Temporal	$\|V\|$	$\|E\|$	Avg. Deg.	Γ
ErodsRenyi	ERA	SYNTHETIC	○	5 000	1 250 002	500	2
MarkerCafe	MRC	REAL-WORLD	○	69 413	1 644 801	47	9
BarabasiAlbert10	BAA	SYNTHETIC	○	100 000	999 911	20	5
BarabasiAlbert50	BAB	SYNTHETIC	○	100 000	4 997 551	100	4
Livemocha	LIV	REAL-WORLD	○	104 103	2 193 083	42	6
Douban	DOU	REAL-WORLD	○	154 908	327 161	4	9
Twitch	TWI	REAL-WORLD	○	168 114	6 797 557	80	8
Gowalla	GOW	REAL-WORLD	○	196 591	950 327	10	16
Citeseer	CTS	REAL-WORLD	○	365 154	1 721 981	9	34
YouTube	YTB	REAL-WORLD	○	1 134 890	2 987 624	5	24
Yahoo	YAH	REAL-WORLD	●	100 001	587 964	12	40
DiggVote	DIG	REAL-WORLD	●	142 962	3 005 898	42	4
StackOverflow	SOF	REAL-WORLD	●	641 876	1 301 942	4	20
Epinions	EPI	REAL-WORLD	●	876 252	13 663 320	31	12
Wikipedia	WIK	REAL-WORLD	●	1 203 994	20 016 873	33	16

TABLE 2. Results of the RAND-INS experiment, $k = 2$ and $k = 4$.

G	$k=2$							$k=4$						
	CT (s)		Avg. Speed-up	IS (MB)		QT (μs)		CT (s)		Avg. Speed-up	IS (MB)		QT (μs)	
	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL
ERA	24.78	0.03	$7.8 \cdot 10^2$	126	126	18.3	18.3	39.08	0.07	$5.4 \cdot 10^2$	174	175	35.4	35.5
MRC	67.96	0.05	$1.3 \cdot 10^3$	1059	1071	9.6	9.4	134.48	0.15	$8.4 \cdot 10^3$	1715	1727	17.5	17.4
BAA	624.93	0.24	$2.5 \cdot 10^3$	6160	6303	37.6	38.4	1145.17	0.73	$1.5 \cdot 10^3$	8583	8776	71.1	73.5
BAB	3066.92	1.77	$1.7 \cdot 10^3$	15306	15364	132.4	134.4	3807.45	2.73	$1.3 \cdot 10^3$	16626	16686	167.1	169.5
LIV	386.55	0.13	$2.8 \cdot 10^3$	3506	3534	22.9	23.2	647.42	0.44	$1.4 \cdot 10^3$	4694	4733	34.9	36.2
DOU	277.14	0.07	$3.7 \cdot 10^3$	5379	5451	32.7	34.1	547.44	0.24	$2.2 \cdot 10^3$	7615	7704	55.9	58.3
TWI	1367.06	0.50	$2.7 \cdot 10^3$	8529	8563	34.5	36.2	2265.61	1.40	$1.6 \cdot 10^3$	11987	12038	56.7	60.4
GOW	82.99	0.02	$3.3 \cdot 10^3$	1990	2068	5.3	5.3	158.37	0.22	$7.0 \cdot 10^2$	2913	3022	8.8	8.8
CTS	1782.81	0.81	$2.1 \cdot 10^3$	15660	17603	36.0	37.5	3436.40	2.28	$1.5 \cdot 10^3$	21574	24194	55.7	57.9
YTB	804.13	0.05	$1.3 \cdot 10^4$	18757	18810	10.9	11.8	1539.02	1.02	$1.5 \cdot 10^3$	27246	27333	17.8	18.1

1) EXECUTED TESTS

For parameter k, our experimental trials use values as in [3], namely $k \in \{2, 4, 8, 16\}$, since (i) the range is relevant to the applications domains of interest; (ii) evaluating performance indicators across doubling values of the parameter magnifies the observed changes in the algorithms' behaviors [48]. For each input graph G and value of k, we perform three types of experiments, depending on how edges to insert are selected.

2) EXPERIMENT RAND-INS

In this experiment, we first execute K-PLL to compute a K–2HC index I covering G. Then, we select uniformly at random, for $\sigma > 0$ times, two vertices x, y that are not adjacent in the graph, add edge (x, y), obtain a graph G', and run DYN-KPLL to update index I to I' covering G'. Eventually, we compute from scratch a K–2HC index I'' covering the last snapshot of the graph via K-PLL. The purpose of this setting is to evaluate the algorithm's behavior regardless of the probability of an insertion to occur.

3) EXPERIMENT SEMI-REAL

In this experiment, we start by removing $\sigma > 0$ edges, selected uniformly at random, from a graph G to obtain a graph G_{init}. We compute a K–2HC index I covering G_{init} via K-PLL and then re-insert, one after the other, the σ sampled edges, until such removed edges are all re-inserted and the original graph G is restored. After each insertion we run DYN-KPLL to update the index to an index I' covering the graph G' comprising the insertion. Finally, we execute K-PLL to compute from scratch a K–2HC index I'' covering G. The purpose of this setting is to assess the algorithm's behavior in a semi-realistic context, where insertions are sampled to follow the distribution induced by edges that are actually in the graph at some point of its evolution.

4) EXPERIMENT TEMPORAL

In this experiment, we consider real-world graphs whose *historical evolution* is known in the form of *timestamps*, defining the order in which any edge has been added to the graph. For each dataset of this kind (identified by the flag *temporal* in Table 1), having a total of η edges, we start by considering the graph G to be a snapshot of the dataset with $\eta - \sigma > 0$ edges and by computing a K–2HC I covering G via K-PLL. Then, we proceed by adding the σ edges in the order dictated by the timestamps and by running DYN-KPLL, after each insertion, to update index I to I' covering the graph G' containing the insertion. Eventually we execute K-PLL to compute a K–2HC index I'' covering the final graph, as in the

TABLE 3. Results of the RAND-INS experiment, $k = 8$ and $k = 16$.

G	$k = 8$							$k = 16$						
	CT (s)		Avg. Speed-up	IS (MB)		QT (μs)		CT (s)		Avg. Speed-up	IS (MB)		QT (μs)	
	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL
ERA	65.90	0.38	$1.7 \cdot 10^2$	269	271	70.7	71.0	116.23	2.55	$4.5 \cdot 10^1$	447	451	127.6	129.0
MRC	259.58	0.32	$8.1 \cdot 10^2$	2565	2576	31.3	31.5	449.85	1.09	$4.0 \cdot 10^2$	3438	3466	46.5	46.8
BAA	1869.36	1.60	$1.1 \cdot 10^3$	10890	11121	110.7	111.3	3409.66	5.20	$6.5 \cdot 10^2$	14969	15301	155.1	158.3
BAB	4401.90	3.13	$1.4 \cdot 10^3$	16890	16949	173.5	176.2	7673.10	7.28	$1.0 \cdot 10^3$	19606	19660	194.8	198.2
LIV	955.55	0.94	$1.0 \cdot 10^3$	5957	6021	48.8	51.4	1387.15	3.07	$4.5 \cdot 10^2$	7753	7859	67.4	69.9
DOU	1046.47	0.66	$1.5 \cdot 10^3$	10458	10589	89.9	97.1	1815.70	1.14	$1.5 \cdot 10^3$	13810	13963	135.7	138.4
TWI	3280.26	4.08	$8.0 \cdot 10^2$	15527	15601	85.4	87.6	4889.45	16.48	$2.9 \cdot 10^2$	19054	19171	114.6	117.3
GOW	279.84	0.56	$4.9 \cdot 10^2$	4011	4161	12.1	12.2	462.30	1.17	$3.9 \cdot 10^2$	5264	5461	17.9	18.2
CTS	6260.67	6.80	$9.2 \cdot 10^2$	29319	32732	85.6	87.2	11333.71	15.60	$7.2 \cdot 10^2$	39301	43696	129.7	132.3
YTB	2723.76	3.29	$8.2 \cdot 10^2$	37083	37189	26.3	28.3	4853.06	6.14	$7.8 \cdot 10^2$	48319	48450	39.6	40.8

TABLE 4. Results of the SEMI-REAL experiment for $k = 2$ and $k = 4$.

G	$k = 2$							$k = 4$						
	CT (s)		Avg. Speed-up	IS (MB)		QT (μs)		CT (s)		Avg. Speed-up	IS (MB)		QT (μs)	
	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL
ERA	24.69	0.03	$7.8 \cdot 10^2$	126	126	19.2	19.2	39.28	0.07	$5.4 \cdot 10^2$	175	175	36.4	36.5
MRC	67.57	0.04	$1.4 \cdot 10^3$	1055	1059	9.4	9.4	133.92	0.10	$1.2 \cdot 10^3$	1712	1715	17.9	18.2
BAA	611.02	0.20	$3.0 \cdot 10^3$	6061	6159	38.6	38.6	1123.11	0.62	$1.7 \cdot 10^3$	8435	8582	73.1	73.5
BAB	3095.04	1.51	$2.0 \cdot 10^3$	15257	15306	136.9	138.0	3742.63	2.47	$1.5 \cdot 10^3$	16572	16626	170.7	174.1
LIV	388.76	0.12	$3.0 \cdot 10^3$	3506	3511	24.1	24.1	650.71	0.37	$1.7 \cdot 10^3$	4693	4705	37.2	35.9
DOU	266.97	0.08	$3.1 \cdot 10^3$	5228	5379	34.3	34.5	531.51	0.27	$1.9 \cdot 10^3$	7380	7613	58.7	60.0
TWI	1379.96	0.43	$3.1 \cdot 10^3$	8524	8529	35.3	36.3	2228.73	1.10	$2.0 \cdot 10^3$	11981	11986	58.3	60.6
GOW	79.19	0.02	$3.0 \cdot 10^3$	1978	1989	5.2	5.2	151.73	0.19	$7.8 \cdot 10^2$	2896	2910	8.4	8.4
CTS	1508.79	0.16	$8.8 \cdot 10^3$	15640	15680	31.9	32.7	2857.74	0.57	$4.9 \cdot 10^3$	21493	21528	47.5	49.4
YTB	814.93	0.09	$8.6 \cdot 10^3$	18729	18757	11.3	11.3	1558.53	0.96	$1.6 \cdot 10^3$	27211	27245	19.0	19.5

previous settings. The purpose of this setting is to evaluate the algorithm's performance in real-world scenarios.

In all mentioned experiments, at the end of the σ insertions (each followed by an execution of DYN-KPLL), and after the final execution of K-PLL, we perform 10^5 top-k distance queries on both I' and I'' and measure average query times. We also measure sizes of indexes I' and I'', preprocessing time to build I'' from scratch via K-PLL, and running time for obtaining each updated I' time via DYN-KPLL. In all trials vertex ordering is established according to non-increasing values of vertex degree, while σ is set to 10 000, since this induces significant changes to the topology of all considered input graphs, and solicits boundary conditions for the algorithm under study.

B. ANALYSIS

The results of our experimentation are summarized in Tables 2–3 (RAND-INS), Tables 4–5 (SEMI-REAL) and Tables 6–7 (TEMPORAL). For each input graph G, we report: (i) *computational time* (column CT, in seconds), that is running time of K-PLL to rebuild the index and average running time of DYN-KPLL to update the index after each insertion, resp.; (ii) *average speed-up*, that is average ratio of the running time of K-PLL to rebuild the index to the running time of DYN-KPLL to update the index, after each insertion; (iii) *index size*, that is size of the index recomputed via K-PLL and size of that updated via DYN-KPLL after all insertions, resp. (column IS, expressed in MBs); (iv) *average query time* for performing the 10^5 queries on the index recomputed via K-PLL and on that updated by DYN-KPLL after each insertion, resp. (column QT, in microseconds). Rows are sorted top-to-bottom according to graph order, i.e. $|V|$.

1) SPEED-UP AND SCALABILITY

Our data show that, despite the worst-case time complexity of Thm. 2, DYN-KPLL is extremely fast in updating indexes even for the largest inputs and values of k, and outperforms the recomputation from scratch by K-PLL in all experimental trials by orders of magnitude, regardless of graph size, density, diameter and k. More in details, the observed speed-up by DYN-KPLL is minimum when the value of k approaches the graph diameter, where DYN-KPLL is, on average, more than two orders of magnitude faster than K-PLL (see Tables 3–5, for $k = 16$), and it increases on large networks where DYN-KPLL is up to tens of thousands times faster than K-PLL (see e.g. graph YTB in Table 2 where K-PLL requires $\approx$ 13 minutes to build the index, while DYN-KPLL updates it in $\approx$ 5 hundredths of a second, or graph WIK in Table 7 where K-PLL requires $\approx$ 8 hours to build the index, while DYN-KPLL runs for $\approx$ 13 seconds on average). Indeed, we observe that speed-ups increase as the graph size increases, which suggests that our approach scales well with input size (see Figures 4–5 where graphs on the x-axis are sorted left-to-right according to $|V|$).

TABLE 5. Results of the SEMI-REAL experiment, $k = 8$ and $k = 16$.

G	$k = 8$							$k = 16$						
	CT (s)		Avg. Speed-up	IS (MB)		QT (μs)		CT (s)		Avg. Speed-up	IS (MB)		QT (μs)	
	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL
ERA	66.69	0.38	$1.7 \cdot 10^2$	271	273	72.7	73.3	118.19	2.67	$4.4 \cdot 10^1$	451	456	133.5	134.3
MRC	258.73	0.17	$1.5 \cdot 10^3$	2568	2568	30.9	31.0	448.12	0.47	$9.5 \cdot 10^2$	3437	3450	49.2	48.8
BAA	1832.77	1.34	$1.3 \cdot 10^3$	10729	10889	110.9	112.5	3379.99	3.94	$8.5 \cdot 10^2$	14816	14965	158.5	160.0
BAB	4445.33	2.89	$1.5 \cdot 10^3$	16839	16890	177.7	180.3	7261.38	4.97	$1.4 \cdot 10^3$	19593	19606	186.3	192.5
LIV	957.47	0.69	$1.3 \cdot 10^3$	5957	5970	51.0	50.5	1385.55	1.53	$9.0 \cdot 10^2$	7752	7759	73.4	71.1
DOU	998.17	0.65	$1.5 \cdot 10^3$	10114	10452	95.6	103.3	1740.62	1.36	$1.2 \cdot 10^3$	13351	13799	146.8	151.2
TWI	3289.66	2.51	$1.3 \cdot 10^3$	15513	15526	85.6	88.4	4563.28	6.95	$6.5 \cdot 10^2$	19039	19053	110.7	115.3
GOW	268.04	0.41	$6.4 \cdot 10^2$	3984	4004	12.2	12.2	440.31	0.82	$5.3 \cdot 10^2$	5230	5252	17.5	17.4
CTS	5223.56	2.25	$2.3 \cdot 10^3$	29134	29197	72.6	74.4	9339.01	4.73	$1.9 \cdot 10^3$	38994	39075	111.2	113.3
YTB	2759.14	2.21	$1.2 \cdot 10^3$	37024	37078	27.9	29.7	4629.99	3.29	$1.4 \cdot 10^3$	48245	48311	38.9	40.4

TABLE 6. Results of the TEMPORAL experiment for $k = 2$ and $k = 4$.

G	$k = 2$							$k = 4$						
	CT (s)		Avg. Speed-up	IS (MB)		QT (μs)		CT (s)		Avg. Speed-up	IS (MB)		QT (μs)	
	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL
YAH	622.53	0.23	$2.6 \cdot 10^3$	5562	5604	45.8	46.1	1211.10	0.63	$1.9 \cdot 10^3$	7622	7667	70.3	70.7
DIG	50.77	0.05	$9.3 \cdot 10^2$	1325	1332	8.5	8.5	473.71	0.54	$8.6 \cdot 10^2$	11181	11321	10.3	11.0
SOF	276.84	0.04	$6.1 \cdot 10^3$	8063	8165	7.3	7.6	68.28	0.15	$4.2 \cdot 10^2$	1731	1734	11.2	11.2
EPI	2216.64	0.76	$2.9 \cdot 10^3$	13645	13652	10.7	10.7	2213.54	2.18	$1.0 \cdot 10^3$	14654	14662	11.4	11.5
WIK	4674.56	0.98	$4.7 \cdot 10^3$	38661	38675	21.2	21.3	8011.85	3.53	$2.2 \cdot 10^3$	55646	55666	34.6	34.6

TABLE 7. Results of the TEMPORAL experiment for $k = 8$ and $k = 16$.

G	$k = 8$							$k = 16$						
	CT (s)		Avg. Speed-up	IS (MB)		QT (μs)		CT (s)		Avg. Speed-up	IS (MB)		QT (μs)	
	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL	K-PLL	DYN-KPLL		K-PLL	DYN-KPLL	K-PLL	DYN-KPLL
YAH	2172.34	1.67	$1.2 \cdot 10^3$	10170	10223	9.3	9.8	3928.84	3.78	$1.0 \cdot 10^3$	13223	13285	168.6	168.7
DIG	778.14	1.30	$5.9 \cdot 10^2$	14794	14976	15.5	15.9	178.12	0.31	$5.6 \cdot 10^2$	3073	3078	23.3	23.3
SOF	104.22	0.20	$5.1 \cdot 10^2$	2324	2326	16.4	16.4	1253.56	1.63	$7.6 \cdot 10^2$	19290	19523	22.5	22.7
EPI	358.16	3.21	$7.3 \cdot 10^2$	16243	16251	11.3	11.3	2609.97	4.68	$5.5 \cdot 10^2$	18805	18815	13.4	13.4
WIK	13108.47	5.47	$2.3 \cdot 10^3$	74921	74948	50.7	50.7	29478.43	13.42	$2.1 \cdot 10^3$	97533	97568	125.6	125.7

2) SIZE AND QUERY TIME

Measures collected during our experimentation also represent strong evidences of the fact that indexes updated via DYN-KPLL preserve the nice properties of k-2-Hop Covers in terms of compactness (i.e. index size), which is reflected into extremely small average query times, even for the largest inputs. In fact, essentially across all graphs, values of k, and settings, the sizes of indexes obtained through DYN-KPLL are comparable to those of indexes recomputed via K-PLL, to within few MBs of difference. Some exceptions are observed on graph CTS (e.g. when $k = 16$ in RAND-INS experiment), where the size of the index updated by DYN-KPLL grows to become around up to 11% larger than that of the index recomputed via K-PLL (see Table 3). This phenomenon is expected and most likely due to the lazy nature of the update strategy by DYN-KPLL which, similarly to other dynamic algorithms for labelings [5], avoids the removal, from the index, of so-called *obsolete* entries (i.e. entries that, due to incremental changes, are no longer necessary to cover any pair). This choice is done to keep update times low and, on the one hand, it is easy to observe that it does not affect the correctness: in fact, the k-cover property is preserved since the query algorithm always selects the smallest values in the multisets (and hence any path longer or having the same length as the k-th, if any, is not returned as part of a solution for any given pair). On the other hand, our data show that above mentioned deviations in sizes are in most of the cases negligible and in all cases do not affect average query times, which remain in the order of few microseconds after thousands of updates and also for instances having millions of edges. We remark that this is a desirable behavior to exhibit in a time-evolving environment, since non-preprocessing based methods can require tens of seconds to extract top-k distances from large graphs [3], while DYN-KPLL allows query answering in few microseconds at the price of few seconds of update time, and thus to exploit top-k distances in meaningful real-world scenarios (e.g. dynamic link prediction [3], [44]).

3) PLATFORM-INDEPENDENT METRICS

All above considerations on both effectiveness and scalability of DYN-KPLL are corroborated by the measures of values

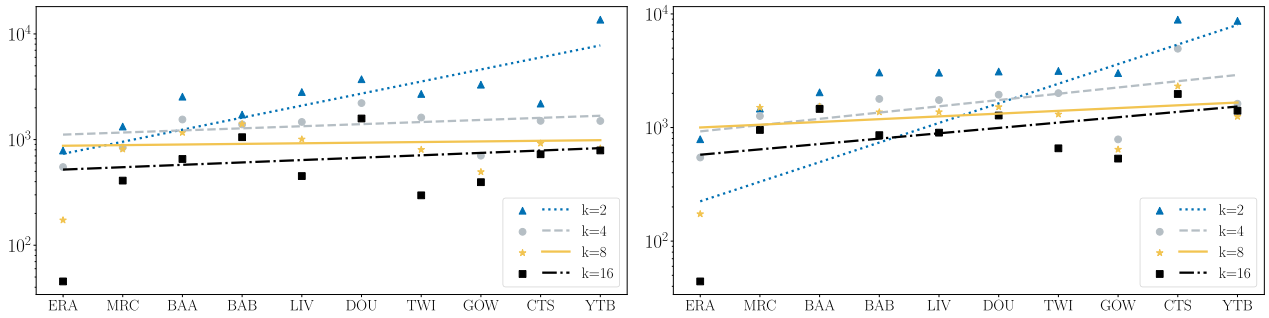


FIGURE 4. Speed-up by DYN-KPLL vs graph size in RAND-INS (left) and SEMI-REAL (right) experiments. Lines show linear regressions.

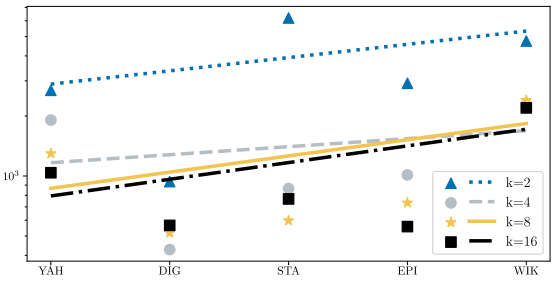


FIGURE 5. Speed-up by DYN-KPLL vs graph size in TEMPORAL experiments. Lines show linear regressions.

TABLE 8. Measures of platform-independent metrics for a subset of input graphs, RAND-INS experiment, $k = 2$ and $k = 16$.

G	$\lvert V\rvert$	$\lvert E\rvert$	β	r	s	c
$k = 2$						
DOU	154 908	327 161	917	1	260	1
BAA	100 000	999 910	1699	3	1 034	1
GOW	196 591	950 327	295	3	2 985	2
YTB	1 134 890	2 987 624	473	2	3 091	2
$k = 16$						
DOU	154 908	327 161	2013	1	154 908	49
BAA	100 000	999 911	3746	3	100 000	293
GOW	196 591	950 327	730	2	196 591	121
YTB	1 134 890	2 987 624	1153	2	1 134 890	73

β, r, s and c (as defined in Thm. 2) we collected in our tests. In almost all cases, in fact, β and r are orders of magnitude smaller than $|V|$ while c and s are orders of magnitude smaller than $|E|$ (except for large k, approaching the graph diameter; in such a case the measure of s tend to reach $|V|$). An excerpt of such measures is shown in Table 8 for the RAND-INS experiment with $k = 16$. Values are averaged, and rounded to the first integer digit, over the total number of graph updates. Results for other graphs and k are omitted as they are similar and lead to equivalent considerations.

4) IMPACT OF K AND TYPE OF EXPERIMENTS

Concerning the impact of k on the performance of DYN-KPLL, we observe that the provided speed-up tend to decrease as k increases (see e.g. column Avg. Speed-up in Tables 2–7 or trend lines in Figure 5), even if DYN-KPLL remains orders of magnitude faster than K-PLL. This might be due to the fact that the number of new cycles and paths, induced by a newly inserted edge and whose length is shorter than existing ones, tend to increase with k. This conclusion is supported by our measures of r, s and c which are such that $r \ll |V|$ and $s, c \ll |E|$ for low values of k but tend to increase with k itself, more evidently when k becomes larger than the graph diameter. Other performance indicators (e.g. query time) are weakly influenced by increases of k, and this witnesses for our method scaling well also against k. Finally, it is worth noticing that performance indicators observed for DYN-KPLL do not exhibit significant variation across experimental settings, which suggests that our method is robust against "adversarial" scenarios where edges to be inserted are not sampled from an empirical distribution and insertions do not follow typical network formation dynamics (e.g. *preferential attachment*).

To summarize, our experimentation provides strong evidences that maintaining k-2-Hop Covers via DYN-KPLL is the most practical framework to deal with top-k distance queries when large graphs subject to incremental updates have to be managed, and that hence DYN-KPLL improves over the state-of-the-art method in dynamic contexts.

V. CONCLUSION

We have studied methods to extract top-k distances from massively sized graphs. We have introduced DYN-KPLL, a new dynamic algorithm to update k-2-Hop Covers when the managed graph is time-evolving, and assessed its effectiveness and scalability through extensive experimentation, hence delivering the first scalable algorithmic framework for fast retrieval of top-k distances from massive time-evolving graphs.

Several future research directions can be identified. Perhaps the most relevant one concerns the consolidation of the experimental evaluation presented here to include weighted digraphs. Another interesting direction might be investigating whether and how DYN-KPLL can be generalized to handle any type of graph modification and hence to avoid the recomputation from scratch also when vertex/edge removals can occur, even though such modifications are much less frequent in the real-world domains where ranked distances are exploited [3], [5], [19].

REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck, "Hierarchical hub labelings for shortest paths," in *Proc. 20th Annu. Eur. Symp. (ESA)* (Lecture Notes in Computer Science), vol. 7501, L. Epstein and P. Ferragina, Eds. Ljubljana, Slovenia: Springer, Sep. 2012, pp. 24–35.

[2] D. Ajwani, E. Duriakova, N. Hurley, U. Meyer, and A. Schickedanz, "An empirical comparison of k-shortest simple path algorithms on multicores," in *Proc. 47th Int. Conf. Parallel Process. (ICPP)*. New York, NY, USA: Association for Computing Machinery, Aug. 2018.

[3] T. Akiba, T. Hayashi, N. Nori, Y. Iwata, and Y. Yoshida, "Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 2–8.

[4] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *Proc. ACM Int. Conf. Manag. Data (SIGMOD)*, 2013, pp. 349–360.

[5] T. Akiba, Y. Iwata, and Y. Yoshida, "Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling," in *Proc. 23rd ACM Int. Conf. World Wide Web*, Apr. 2014, pp. 237–248.

[6] B. Bandyopadhyay, D. Fuhry, A. Chakrabarti, and S. Parthasarathy, "Topological graph sketching for incremental and scalable analytics," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 1231–1240.

[7] A. Baroni, A. Conte, M. Patrignani, and S. Ruggieri, "Efficiently clustering very large attributed graphs," in *Proc. IEEE/ACM Int. Conf. Adv. Social Netw. Anal. Mining (ASONAM)*. New York, NY, USA: Association for Computing Machinery, Jul. 2017, pp. 369–376.

[8] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner, "Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm," *ACM J. Exp. Algorithmics*, vol. 15, pp. 1–26, Mar. 2010.

[9] E. Bergamini and H. Meyerhenke, "Approximating betweenness centrality in fully dynamic networks," *Internet Math.*, vol. 12, no. 5, pp. 281–314, Sep. 2016.

[10] B. Bollobás, *Random Graphs* (Cambridge Studies in Advanced Mathematics), vol. 73, 2nd ed. Cambridge, U.K.: Cambridge Univ. Press, 2011.

[11] L. Chang, X. Lin, L. Qin, J. X. Yu, and J. Pei, "Efficiently computing top-k shortest path join," in *Proc. 18th Int. Conf. Extending Database Technol. (EDBT)*, 2015, pp. 133–144.

[12] A. Cionini, G. D'Angelo, M. D'Emidio, D. Frigioni, K. Giannakopoulou, A. Paraskevopoulos, and C. D. Zaroliagis, "Engineering graph-based models for dynamic timetable information systems," *J. Discrete Algorithms*, vols. 46–47, pp. 40–58, Sep./Nov. 2017.

[13] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM J. Comput.*, vol. 32, no. 5, pp. 1338–1355, Jan. 2003.

[14] P. Crescenzi, C. Magnien, and A. Marino, "Approximating the temporal neighbourhood function of large temporal graphs," *Algorithms*, vol. 12, no. 10, p. 211, Oct. 2019.

[15] P. Crescenzi, C. Magnien, and A. Marino, "Finding top-k nodes for temporal closeness in large temporal graphs," *Algorithms*, vol. 13, no. 9, p. 211, Aug. 2020.

[16] A. D'Andrea, M. D'Emidio, D. Frigioni, S. Leucci, and G. Proietti, "Dynamic maintenance of a shortest-path tree on homogeneous batches of updates: New algorithms and experiments," *ACM J. Exp. Algorithmics*, vol. 20, pp. 1.5:1.1–1.5:1.33, Dec. 2015.

[17] G. D'angelo, M. D'emidio, and D. Frigioni, "Fully dynamic 2-hop cover labeling," *ACM J. Exp. Algorithmics*, vol. 24, pp. 1–36, Dec. 2019.

[18] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck, "PHAST: Hardware-accelerated shortest path trees," *J. Parallel Distrib. Comput.*, vol. 73, no. 7, pp. 940–952, Jul. 2013.

[19] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Robust distance queries on massive networks," in *Proc. 22th Eur. Symp. Algorithms (ESA)* (Lecture Notes in Computer Science), vol. 8737. Berlin, Germany: Springer, 2014, pp. 321–333.

[20] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[21] D. Eppstein, "Finding the k shortest paths," *SIAM J. Comput.*, vol. 28, no. 2, pp. 652–673, 1998.

[22] D. Eppstein, "k-best enumeration," in *Encyclopedia of Algorithms*. Boston, MA, USA: Springer, 2016, pp. 1003–1006.

[23] M. Farhan, H. Koehler, and Q. Wang, "BatchHL$^+$: Batch dynamic labelling for distance queries on large-scale networks," *VLDB J.*, pp. 1–29, May 2023.

[24] M. Farhan, Q. Wang, and H. Koehler, "BatchHL: Answering distance queries on batch-dynamic networks at scale," in *Proc. Int. Conf. Manag. Data (SIGMOD)*, Z. G. Ives, A. Bonifati, A. E. Abbadi, Eds. Philadelphia, PA, USA: ACM, Jun. 2022, pp. 2020–2033.

[25] M. Farhan, Q. Wang, Y. Lin, and B. McKay, "Fast fully dynamic labelling for distance queries," *VLDB J.*, vol. 31, pp. 483–506, May 2022.

[26] G. Feng, "Finding k shortest simple paths in directed graphs: A node classification algorithm," *Networks*, vol. 64, no. 1, pp. 6–17, Aug. 2014.

[27] J. Gao, H. Qiu, X. Jiang, T. Wang, and D. Yang, "Fast top-k simple shortest paths discovery in graphs," in *Proc. 19th ACM Int. Conf. Inf. Knowl. Manage.*, Oct. 2010, pp. 509–518.

[28] M. Girvan and M. E. J. Newman, "Community structure in social and biological networks," *Proc. Nat. Acad. Sci. USA*, vol. 99, no. 12, pp. 7821–7826, Jun. 2002.

[29] Z. Gotthilf and M. Lewenstein, "Improved algorithms for the k simple shortest paths and the replacement paths problems," *Inf. Process. Lett.*, vol. 109, no. 7, pp. 352–355, Mar. 2009.

[30] K. Hanauer, M. Henzinger, and C. Schulz, "Faster fully dynamic transitive closure in practice," in *Proc. 18th Int. Symp. Exp. Algorithms (SEA)* (Leibniz International Proceedings in Informatics), vol. 160, S. Faro and D. Cantone, Eds. Wadern, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 14:1–14:14.

[31] K. Hanauer, M. Henzinger, and C. Schulz, "Recent advances in fully dynamic graph algorithms—A quick reference guide," *ACM J. Exp. Algorithmics*, vol. 27, pp. 1–45, Dec. 2022.

[32] X. Hao, T. Lian, and L. Wang, "Dynamic link prediction by integrating node vector evolution and local neighborhood representation," in *Proc. 43rd Int. ACM SIGIR Conf. Res. Develop. Inf. Retr. (SIGIR)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1717–1720.

[33] M. S. Hassan, W. G. Aref, and A. M. Aly, "Graph indexing for shortest-path finding over dynamic sub-graphs," in *Proc. ACM Int. Conf. Manage. Data (SIGMOD)*, Jun. 2016, pp. 1183–1197.

[34] T. Hayashi, T. Akiba, and K.-I. Kawarabayashi, "Fully dynamic shortest-path distance query acceleration on massive networks," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, Oct. 2016, pp. 1533–1542.

[35] M. Henzinger, A. Noe, and C. Schulz, "Faster parallel multiterminal cuts," in *Proc. SIAM Conf. Appl. Comput. Discrete Algorithms (ACDA)*, M. Bender, J. Gilbert, B. Hendrickson, and B. D. Sullivan, Eds. Philadelphia, PA, USA: SIAM, Jul. 2021, pp. 100–110.

[36] J. Hershberger, M. Maxel, and S. Suri, "Finding the k shortest simple paths: A new algorithm and its implementation," *ACM Trans. Algorithms*, vol. 3, no. 4, p. 45, Nov. 2007.

[37] M.-Y. Kao, *Encyclopedia of Algorithms*. New York, NY, USA: Springer, 2016.

[38] N. Katoh, T. Ibaraki, and H. Mine, "An efficient algorithm for k shortest simple paths," *Networks*, vol. 12, no. 4, pp. 411–427, Winter 1982.

[39] D. Kocher and N. Augsten, "A scalable index for top-k subtree similarity queries," in *Proc. Int. Conf. Manage. Data (SIGMOD)*. New York, NY, USA: Association for Computing Machinery, Jun. 2019, pp. 1624–1641.

[40] S. Kontogiannis, D. Wagner, and C. Zaroliagis, "An axiomatic approach to time-dependent shortest path oracles," *Algorithmica*, vol. 84, no. 3, pp. 815–870, Mar. 2022.

[41] J. Kunegis, "KONECT—The Koblenz network collection," in *Proc. Int. Conf. World Wide Web Companion*, 2013, pp. 1343–1350.

[42] D. Kurz and P. Mutzel, "A sidetrack-based algorithm for finding the k shortest simple paths in a directed graph," in *Proc. 27th Int. Symp. Algorithms Comput. (ISAAC)* (Leibniz International Proceedings in Informatics), vol. 64, S.-H. Hong, Ed. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 49:1–49:13.

[43] J. Langguth, A. Tumanis, and A. Azad, "Incremental clustering algorithms for massive dynamic graphs," in *Proc. Int. Conf. Data Mining Workshops (ICDMW)*, Auckland, New Zealand, Dec. 2021, pp. 360–369.

[44] A. Lebedev, J. Lee, V. Rivera, and M. Mazzara, "Link prediction using top-k shortest distances," in *Proc. 31st Brit. Int. Conf. Databases (BICOD)* (Lecture Notes in Computer Science), vol. 10365, A. Calì, P. T. Wood, N. J. Martin, and A. Poulovassilis, Eds. Cham, Switzerland: Springer, 2017, pp. 101–105.

[45] J. Leskovec and R. Sosič, "SNAP: A general-purpose network analysis and graph-mining library," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 1, pp. 1–20, Jul. 2016.

[46] J. Li, Y. Cao, and X. Liu, "Approximating graph pattern queries using views," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 449–458.
[47] W. Li, M. Qiao, L. Qin, L. Chang, Y. Zhang, and X. Lin, "On scalable computation of graph eccentricities," in *Proc. Int. Conf. Manag. Data (SIGMOD)*, Z. G. Ives, A. Bonifati, and A. E. Abbadi, Eds. Philadelphia, PA, USA: ACM, Jun. 2022, pp. 904–916.
[48] C. C. McGeoch, *A Guide to Experimental Algorithmics*. Cambridge, U.K.: Cambridge Univ. Press, 2012.
[49] M. Newman, *Networks*. Oxford, U.K.: Oxford Univ. Press, 2018.
[50] M. Pascoal, "Implementations and empirical comparison of k shortest loopless path algorithms," 9th DIMACS Implement. Challenge—Shortest Paths, Sapienza Univ., Rome, Italy, Tech. Rep., 2006.
[51] T. P. Peixoto, "The Netzschleuder network catalogue and repository," Zenodo, Tech. Rep., 2020, doi: 10.5281/zenodo.7839981.
[52] Y. Peng, X. Lin, Y. Zhang, W. Zhang, and L. Qin, "Answering reachability and K-reach queries on large graphs with label constraints," *VLDB J.*, vol. 31, no. 1, pp. 101–127, Jan. 2022.
[53] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," in *Proc. 18th ACM Conf. Inf. Knowl. Manage.*, Nov. 2009, pp. 867–876.
[54] K. Qiu, J. Zhao, X. Wang, X. Fu, and S. Secci, "Efficient recovery path computation for fast reroute in large-scale software-defined networks," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 8, pp. 1755–1768, Aug. 2019.
[55] L. Roditty and U. Zwick, "Replacement paths and k simple shortest paths in unweighted directed graphs," *ACM Trans. Algorithms*, vol. 8, no. 4, pp. 1–11, Sep. 2012.
[56] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. 29th AAAI Conf. Artif. Intell.*, 2015, pp. 4292–4293.
[57] M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time," *J. ACM*, vol. 46, no. 3, pp. 362–394, May 1999.
[58] A. van der Grinten, G. Custers, D. L. Thanh, and H. Meyerhenke, "An MPI-parallel algorithm for static and dynamic top-k harmonic centrality," in *Proc. IEEE 34th Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, Bordeaux, France, Nov. 2022, pp. 100–109.
[59] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou, "Efficient route planning on public transportation networks: A labelling approach," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, May 2015, pp. 967–982.
[60] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida, "Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths," in *Proc. 22nd ACM Int. Conf. Conf. Inf. Knowl. Manage. (CIKM)*, 2013, pp. 1601–1606.
[61] J. Y. Yen, "An algorithm for finding shortest routes from all source nodes to a given destination in general networks," *Quart. Appl. Math.*, vol. 27, no. 4, pp. 526–530, 1970.
[62] Z. Yu, X. Yu, N. Koudas, Y. Liu, Y. Li, Y. Chen, and D. Yang, "Distributed processing of k shortest path queries over dynamic road networks," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 665–679.
[63] Y. Zhang and J. X. Yu, "Hub labeling for shortest path counting," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 1813–1828.
[64] A. A. Zoobi, D. Coudert, and N. Nisse, "Space and time trade-off for the k shortest simple paths problem," in *Proc. 18th Int. Symp. Exp. Algorithms (SEA)* (Leibniz International Proceedings in Informatics), vol. 160, S. Faro and D. Cantone, Eds. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 18:1–18:13.

ANDREA D'ASCENZO was born in L'Aquila, in 1996. He received the degree (cum laude) in computer science from the University of L'Aquila, where he is currently pursuing the Ph.D. degree in information and communication technology. His main research interests include experimental algorithms, in particular algorithms for massive (dynamic) graphs, combinatorial optimization, with a special attention to biology-related topics and algorithmic game theory.

MATTIA D'EMIDIO received the Ph.D. degree in information and electrical engineering from the University of L'Aquila, Italy, in 2014.

He is currently an Assistant Professor and a Researcher in computer engineering with the Department of Information Engineering, Computer Science and Mathematics (DISIM), University of L'Aquila. He is the author/coauthor of tens of papers selected for publication by international, peer-reviewed journals, and conferences. His primary research interests include the design, analysis, and implementation of efficient algorithms for combinatorial problems. His work follows an algorithm engineering approach and is mostly focused on algorithms for effective mining of massive (possibly dynamic) graphs, frameworks for parallel processing, big data analytics, and study of distributed systems of autonomous entities from an algorithmic perspective (including game theoretic aspects). He has been part of several program and organizing committees of international conferences. He has served as a reviewer for many international journals and conferences of the computer science/engineering.

Open Access funding provided by 'Università degli Studi dell'Aquila' within the CRUI CARE Agreement