2023 IEEE 39th International Conference on Data Engineering (ICDE)

Hop-Constrained s-t Simple Path Enumeration on Large Dynamic Graphs

Jiujing Zhang[†], Shiyu Yang[†*] , Dian Ouyang[†], Fan Zhang[†], Xuemin Lin[§], Long Yuan[‡]
[†]*Guangzhou University;* [§]*Antai College of Economics and Management, Shanghai Jiao Tong University;*
[‡]*Nanjing University of Science and Technology*
jiujing.zhang@outlook.com; {syyang, dian.ouyang, zhangf}@gzhu.edu.cn;
xuemin.lin@sjtu.edu.cn; longyuan@njust.edu.cn

***Abstract*—Hop-constrained s-t simple path (k-st path) enumeration is a fundamental problem in graph databases and plays an important role in many real-world applications. Given a dynamic graph G, a source-target pair s-t, and a hop constraint k, we aim to efficiently compute k-st paths: list all simple paths within length k from s to t, and then continuously maintain the results against edge updates. Although the k-st path enumeration has been well studied in static setting, the existing works on static graphs cannot be applied or adapted to handle dynamic graphs efficiently. To address the challenges on dynamic computation, we propose a partial path-based index structure and an efficient enumeration algorithm based on the index. We also propose several well-designed techniques to efficiently maintain the index and locate the affected results with graph updates. Comprehensive experiments verify that our proposed *CPE$_{update}$* algorithm outperforms the state-of-the-art methods by up to 4 orders of magnitude on dynamic graphs. The experiment results also show that the time cost of our initialization step *CPE$_{startup}$* (including index construction) is similar to the state-of-the-art static method.**

I. Introduction

Given a source vertex s, a target vertex t and a hop constraint k, the hop-constrained s-t simple path enumeration aims to list every simple path from s to t with length not exceeding k, where a simple path is a path in which all vertices are distinct. In this paper, we use k-st path enumeration for short to denote the hop-constrained s-t simple path enumeration.

Due to the importance of k-st path enumeration, a series of solutions [1]–[6] have been proposed. However, all of the existing works are designed for static graphs. In real-world applications, graphs are usually dynamic and can be updated frequently [7], [8], e.g., an update in a social network can be triggered by a simple "Like" click and new transactions are continuously submitted to E-commerce networks. Efficiently maintaining the path enumeration results on dynamic graphs is critical for various applications. We demonstrate the importance from the following three examples.

- *Financial Crimes Detection.* Financial networks are usually modeled as directed graphs and the transaction from user A to user B can be represented by an edge from A to B. Some known Red Flag Indicators of money laundering are reported in [9]–[12], such as the use of multiple bank accounts as well as that of intermediaries without appropriate reasons. As they reported, many cases of money laundering are observed along transaction cycle and short flow paths, which can be detected by enumerating k-st paths between suspected accounts. With the help of k-st paths, we can compute a value to reflect the total risk level, which could be a crucial measure in crime detection. As reported in [8], the graph of the Alibaba E-commerce platform is being updated at an average rate of 3,000 edges per second, and over 20,000 new edges are added per second at the peak. When the edge update happens, it is necessary to update the risk level value by accessing the new paths so that the property loss can be avoided to the most extent.
- *Social Network.* In social networks [13], a path that connects two users reflects that there is a relationship between these two users, and all paths between these users can reflect the strength of such relationships. For social network applications, evaluation of the relationship between users is one of the most important tasks [13]. It relies on materialized k-st paths to measure the relationship between vertex pairs.Social network platforms can achieve such goals by issuing the k-st path enumeration queries. Online social networks are changing every single second, instead of using the complete set of k-st paths to recompute the result, the evaluation result can also be updated by querying the new/deleted k-st paths, which can avoid unnecessary computation and improve the efficiency significantly.
- *Communication Network.* The communication network can be seen as a large dynamic graph, where each edge indicates the communication between two terminals on the Internet. With the movement of communication nodes (e.g. smart phones and IoT devices join or leave the network) and the changing environmental conditions (e.g. DDos attack caused network disconnection), the graph changes rapidly and constantly. Path enumeration has been used for communication network analysis for a long history. For instance, enumeration of all simple paths between a terminal pair is used while computing the terminal reliability in communication networks [14]. Besides, path enumeration can also be adopted to measure the robustness of the communication network.

*Corresponding author

For many applications, such as fraud detection, we usually have a list of suspects/candidates, and the k-st path enumeration algorithm on dynamic graphs aims to monitor the suspect/candidate pairs and access the new/deleted paths in a real time manner. On dynamic graphs, when the graph is updated, instead of recomputing the complete set, the downstream applications prefer to access the new/deleted paths, which not only is enough to support downstream applications but also can save the query processing time. But the existing works focus on enumerating the complete set of paths on a static setting. Even if the existing approaches can be used to processing dynamic graphs, we have to either compare the complete set of the latest result with the previous to find the changed part or feed the complete set of paths to the downstream applications. Therefore, directly adopting the existing approaches to solve the k-st path enumeration problem on dynamic graphs will be costly and introducing redundant computations. For instance, on dense or large-scale dynamic graphs, it may take a few hours to update the result against a slight graph update.

We also examine two of the most representative existing solutions for static k-st path enumeration, i.e., *PathEnum* [5] and *BC-JOIN* [3]. *PathEnum* is the state-of-the-art method on static graphs and there is no intermediate result to list paths. Thus, it is infeasible to utilize existing intermediate computations and we have to recompute the result from scratch when the graph changes, which is cost-prohibitive. *BC-JOIN* is an algorithm with a bidirectional search-based join paradigm, in which the intermediate results can be regarded as an index and maintained. However, compared to *PathEnum*, it is much more costly in path enumeration and the index cannot be efficiently maintained to handle graph updates.

Besides, since k-st path enumeration problem can be considered as a special case of subgraph matching problem, which enumerate all instances of a given pattern in a data graph, the problem of enumerating k-st paths on dynamic graphs can be transformed into the problem of continuous subgraph matching (CSM) [15]–[20] (set the query graph as paths with length up to k). Although these methods performs well on enumerating general patterns, it does not consider the possible pruning opportunities to reduce unnecessary computation specific to k-st path, thus it is inefficient and unscalable to the k-st path enumeration problem.

Motivated by this, we design a partial path-based index structure and associated techniques to efficiently maintain k-st paths on dynamic graphs. With the help of the proposed index, we can easily enumerate the proportion of paths that are affected by the graph update without redundant computation. Moreover, we carefully design corresponding maintenance methods to update the index on the basis of previous computation. For the index construction, we adapt the bidirectional search-based join paradigm and design powerful pruning techniques to build the index efficiently. According to the experiment results on dynamic graphs, our proposed method can compute the updated results instantly (often less than 1 ms) given graph updates while the existing methods have to re-compute the results. Although our algorithm is designed for dynamic graphs, our initialization step (including the index construction) can answer the k-st path enumeration query as efficient as the state-of-the-art method *PathEnum* on static graphs.

Contributions. The contributions are summarized as follows.

- To the best of our knowledge, this is the first work to study the k-st path enumeration problem on dynamic graphs.
- We design a novel partial path-based index structure and the corresponding techniques to efficiently maintain the k-st paths.
- We also propose efficient bidirectional search-based index construction algorithm and index maintenance methods with several powerful pruning techniques. Comprehensive complexity analysis is also provided.
- Extensive experiments are conducted with a variety of workloads. The experimental results demonstrate that our proposed algorithms significantly outperform the state-of-the-art methods by up to 4 orders of magnitude on dynamic graphs. We also show that, even on static graphs, benefiting from our powerful pruning techniques, the performance of our initialization (bidirectional search-based method) is as efficient as the state-of-the-art method.

Organization. The rest of the paper is organized as follows. In Section II, we introduce the preliminaries, formulate k-st path enumeration problem on dynamic graphs, and give an overview of our solution. Section III proposes a partial path-based index and corresponding methods to enumerate the complete set and the new/deleted part of k-st paths, respectively. Section IV illustrates the index construction algorithm and index maintenance algorithm against edge insertion and deletion. After that, we present the comprehensive experimental results in Section V. Finally, we survey related works in Section VI and conclude the paper in Section VII.

II. BACKGROUND

A. Preliminaries

$G = (V, E, U)$ denoted as a dynamic directed graph where (1) $V(G)$ is a set of n vertices; (2) $E(G) \subseteq V(G) \times V(G)$ is a set of m static edges that exists in G before the update of dynamic edges, in which $e(u, v)$ denotes a directed edge from the vertex u to the vertex v; (3) U denotes the set of updated edges. Each edge update is denoted using $e(u, v, +)$ or $e(u, v, -)$ where $+$ $(-)$ means it's an inserted (deleted) edge from vertex u to v. Note that we only consider edge updates in this paper. This is because insertions or deletions of the graph vertices can also be expressed using edge updates. In this paper, the dynamic graph G is initialized by an initial static graph with all vertices and static edges. Then it is continuously updated upon the arrival and expiration of edges.

Let $N_{in}(v) = \{u|e(u, v) \in E\}(N_{out}(v) = \{u|e(v, u) \in E\})$ represents the in-going (out-going) neighbors of v. If the context is clear, we use "neighbor" to refer to the "out-going neighbor". By $G^r = (V(G), E^r(G))$, we denote the reverse graph of G, where $V(G^r) = V(G)$ and for each directed

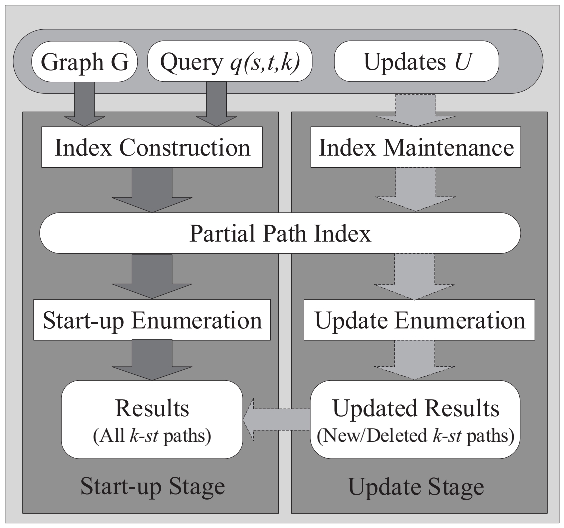


Fig. 1. An Overview of Our Solution

edge $e(u, v) \in E(G)$, there is a corresponding edge $e(v, u) \in E^r(G)$. $Dist_s[v]$ ($Dist_t[v]$) denotes the shortest distance from s to v(from v to t). We say a path p is a k-hop constrained path if $len(p) \leq k$ where $len(p)$ is the number of edges in p and k is the hop constraint. For presentation simplicity, we denote k-hop constrained s-t simple path by k-st path.

Problem Statements. In this paper, we study the k-st path enumeration problem on dynamic graphs. Given a dynamic directed graph $G = (V, E, U)$, a specific source-target pair s-t and a hop constraint k, a k-st path enumeration will continuously report all k-st paths resulting from each edge update $e(u, v, +/-)$ on the dynamic graph G.

B. Solution Overview

Figure 1 gives an overview of our solution for k-st path enumeration problem on dynamic graphs which is naturally divided into start-up and update stages.

Start-up stage (Initialization): Given a graph G and a k-st query $q(s, t, k)$, we first initialize the partial path-based index on the original graph using a bidirectional search-based method. After the index construction, we can conduct the start-up enumeration on the index to list all k-st paths.

Update stage: When each edge $e \in U$ is continuously inserted to or deleted from the graph, index maintenance is conducted to update the index. Therefore, the update enumeration can continuously report the new/deleted paths resulting from the edge update. After that, users are free to use the updated results individually or merge them with all results of the given query.

III. A Partial Path-based Index

In Section III-A, we propose a partial path-based index to solve the k-st path enumeration problem on dynamic graphs. In Section III-B, we propose index-based algorithms to efficiently enumerate k-st paths on the initial graph and update the result against edge updates.

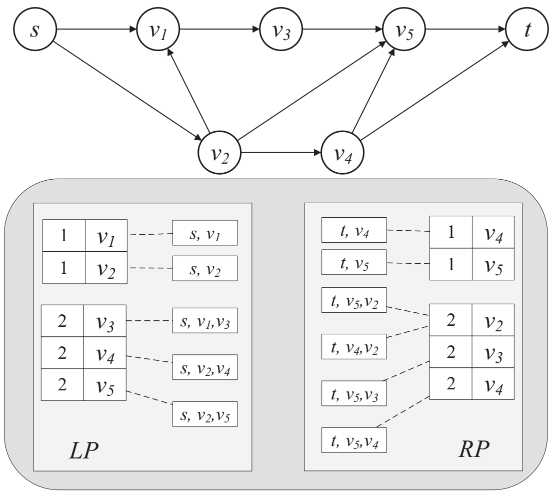


Fig. 2. An Example of Partial Path Index with $q(s, t, 4)$ and $l = r = 2$

A. Index Structure

In order to store the previous intermediate result and use it to get the new/deleted paths quickly, we proposed a partial path-based index, which is inspired by the middle vertices cut.

Definition 1: **Middle vertices cut (V_c).** Given a path $p = \{v_1, ..., v_k\}$, its middle vertex is the $\lceil k/2 \rceil$-th vertex along the path, denoted by v_c. The middle vertices set of all k-st paths are the middle vertices cut and denoted by V_c.

Following the bidirectional search-based join paradigm [3], we can find the middle vertices cut for any query $q(s, t, k)$ so that each k-st path contains at least one vertex in the middle vertices cut. Then each path $p = \{s, ..., v_c, ..., t\}$ can be divided into a left partial path $lp = \{s, ..., v_c\}$ and a right partial path $rp = \{t, ..., v_c\}$. Due to the density of real-world graphs, if we can find out all left and right partial paths, all k-st paths can be easily enumerated by joining each left partial path with each right partial path in each middle vertex. More importantly, the size of all partial paths can be much smaller than the size of all k-st paths.

Motivated by this, a naive solution is to utilize partial paths to build a index, then we can get all k-st paths from the index for a query $q(s, t, k)$. Specifically, we store all left partial paths and right partial paths in the index LP and RP, respectively. In detail, we store each left partial path $lp = \{s, ..., v_c\}$ that satisfied $len(lp) + Dist_t[v_c] \leq k$ and with length up to $l = \lceil k/2 \rceil$ in index LP, and store each right partial path $rp = \{t, ..., v_c\}$ satisfied $len(rp) + Dist_s[v_c] \leq k$ and with length up to $r = \lfloor k/2 \rfloor$ in index RP. By storing the partial paths with the above constraint, we can make sure that all partial paths in the index can be joined to a path from s to t. Moreover, we use $LP_i(v_c)$ to store paths from s to v_c with length i, and use $RP_j(v_c)$ to store paths from v_c to t with length j. Thus $LP = \cup_{i=1}^{\lceil k/2 \rceil} \cup_{v_c \in V_c} LP_i(v_c)$, $RP = \cup_{j=1}^{\lfloor k/2 \rfloor} \cup_{v_c \in V_c} RP_j(v_c)$. By joining $LP_i(v_c)$ and $RP_j(v_c)$ for each i, j pair that satisfied $i = j$ or $i = j + 1$ for each middle vertex v_c, we can make sure v_c is the middle vertex and output all k-st paths with length from 2 to k.

An example of the partial path index is given in Figure 2. Given a query $q(s,t,4)$ and a graph G, we store each left partial path $lp = \{s, ..., v_c\}$ with length up to $l = 2$ and satisfied $len(lp) + Dist_t[v_c] \leq k$ and each right partial path $rp = \{t, ..., v_c\}$ with length up to $r = 2$ and satisfied $len(rp)+ Dist_s[v_c] \leq k$ in index LP and RP, respectively. Note that we don't store every simple path from s to v_c. That's why the partial path $\{s, v_2, v_1\}$ is not in our index in Figure 2, the distance from $v1$ to t is 3 and the hop-constraint is 4, thus there is no s-t path contains $\{s, v_2, v_1\}$.

According to our experimental result in Section V, using the proposed partial path index to deal with k-st path enumeration query on dynamic graphs can not only speed up the result updating by only listing the changed part of paths, but also help us to save a large proportion of storage space.

B. Enumeration on Index

*1) **Start-up Enumeration**:* According to Definition 1, the dynamic graph G is initialized by a static graph G_{static}. Given the partial path index on the G_{static}, we use Algorithm 1 to enumerate all k-st paths by joining the partial paths.

We use a list $Plan$ to store all (i, j) pairs, in which the sum of i and j is from 2 to k. Given a query $q(s, t, k)$, the list $Plan$ is generated in the process of index construction (Section IV-A). For example, consider the case in Figure 2 and use the $\lceil k/2 \rceil$-th vertex as the middle vertex, there are $(1, 1), (2, 1), (2, 2)$ in $Plan$. And then we join the partial paths according to the order in $Plan$ (line 1-2). Given LP, RP and the join length pair (i, j), for each middle vertex v_c, we join paths from s to v_c with length i with paths from t to v_c with length j and checking the repetitions of vertices (line 3-7).

The correctness is proved as follows:

***Theorem** 1:* Given a query $q(s, t, k)$ and corresponding partial path index LP with paths length from 1 to l and RP with paths length from 1 to r where (l, r) is the biggest (i, j) pair in $Plan$ and satisfies $l + r = k$, all the k-st paths can be listed by Algorithm 1.

***Proof** 1:* For each path p in the result of query $q(s, t, k)$, we have $p = \{s, ..., v_c, ..., t\}$ that can be decomposed to $p_1 = \{s, ..., v_c\}$ and $p_2 = \{t, ..., v_c\}$ where $pair((len(p_1), len(p_2)) \in Plan$. Recall that all left-side paths end at v_c with length $len(p_1)$ and all right-side paths end at v_c with length $len(p_2)$ are in the index. After joining by Algorithm 1, $p = \{s, ..., v_c, ..., t\}$ will be listed. Thus, the correctness of Algorithm 1 follows.

Moreover, there is no duplicate detection cost in our join-based enumeration algorithm as proved in the following theorem.

***Theorem** 2:* Given partial path index LP with paths that length from 1 to l and RP with paths that length from 1 to r where $l + r = k$, Algorithm 1 only list each k-st path once.

***Proof** 2:* For the (i, j) pairs list $Plan$, we have $\forall (i, j) \in Plan$, $\not\exists (i', j') \in Plan$ satisfied $i' \neq i$, $j' \neq j$ and $i' + j' = i + j$. And there is no duplicate path in our partial path index LP and RP. Thus it's impossible for Algorithm 1 to list duplicate paths.

Algorithm 1: Enumeration on Index

```
Input: The join list $Plan$, the partial path index $LP$
       and $RP$
Output: all $k$-$st$ paths
1 foreach $pair(i, j) \in Plan$ do
2 |   $Join(LP, RP, i, j)$;
3 Procedure $Join(LP, RP, i, j)$:
4 |   foreach $v_c \in V_c$ do
5 |   |   foreach $lp \in LP_i(v_c)$ do
6 |   |   |   foreach $rp \in RP_j(v_c)$ do
7 |   |   |   |   if there's no repeated vertex in $lp$ and
  |   |   |   |     $rp$ (except $v_c$) then  list $lp \oplus rp$ ;
```

When enumerating k-st paths, each left-side path $lp \in LP(v_c)$ can match at least one right-side path $rp \in RP(v_c)$ to form a path. So the time complexity of the start-up enumeration is bounded by $O(k \times |P|)$, where the $|P|$ is the number of s-t paths with length up to k.

*2) **Update Enumeration**:* After the arrival or expiration of edges, we can access the latest result of k-st paths with the help of the start-up enumeration. Nonetheless, the time cost of re-querying the complete set of results is expensive, especially for queries that have a big hop constraint k on relatively dense graphs. Hence, a better strategy is to enumerate the changed part of result instead of all the results, which we call it the update enumeration. Benefit by our partial path index, we can easily enumerate all new/deleted paths on the basis of the following theorem:

***Theorem** 3:* After the insertion or deletion of edge, for each changed path p' which is new or deleted, it can be divided into two parts, left-side path lp' and right-side path rp'. After maintaining the index, at least one of them is in LP' or RP', where LP' (RP') is the changed part of LP (RP) after edge update.

***Proof** 3:* Assume there is a new (deleted) path p' and its left partial path $lp' \notin LP'$ and right partial path $rp' \notin RP'$, then the path p' can be enumerated before (after) the edge update, which contradicts the fact that p' is new (deleted), thus the theorem holds.
According to Theorem 3, we can specifically enumerate all new/ deleted k-st paths by Algorithm 1 with a constraint that ensures at least one side of partial path is new or deleted. Note that V_c in the update enumeration are easily got from LP' and RP'.

As a result, the time and space complexity of joining and enumerating the new/deleted paths are both bounded by $O(k \times \Delta|P|)$, where $\Delta|P|$ is the number of new/deleted hop-constrained s-t paths with length up to k resulting from the edge update.

IV. Index Construction and Maintenance

In this section, we propose algorithms to efficiently construct the index on the initial graph and maintain it against edge updates.

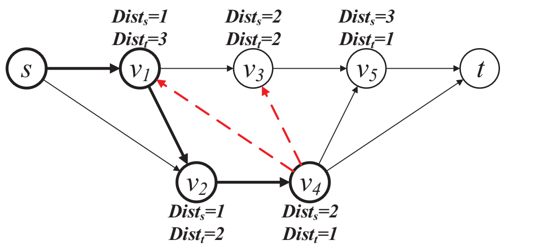


Fig. 3. An Example About Distance Pruning

A. *Index Construction*

By using the partial path index, we can store intermediate results with reasonable space and speed up the result updating for graph updates. However, the only existing solution [3] that can be directly adopted to compute all partial paths is time consuming. Therefore, we propose a well-designed algorithm with powerful pruning techniques to construct the index efficiently.

*1) **Preprocessing**:* Straightforwardly searching partial paths and building the index in the original graph is costly in time, because there are some vertices that will never appear in the index. In other words, we can find an induced subgraph G_{sub} which is equivalent to the original graph G for index construction but can reduce the search space. In the preprocessing step, we use the following theorem to find an induced graph G_{sub}.

***Theorem** 4:* For the graph G, we have the shortest distance map $Dist_s$ ($Dist_t$) that records the $Dist_s[v]$ ($Dist_t[v]$) for each $v \in V(G)$. Performing k-st path query on graph G is equivalent to doing it on a subgraph G_{sub}, where G_{sub} is induced by the vertex set $V_{sub} = \{u|u \in V(G) \wedge Dist_s[u] + Dist_t[u] \leq k\}$.

***Proof** 4:* Assume there exists a k-st path $p = \{s, ..., v, ..., t\}$, where u is a vertex that $u \notin V(G_{sub})$. Thus $Dist_s(v) + Dist_t(v) > k$ holds, which contradicts the premise $len(p) \leq k$. Hence the theorem is tenable.

According to this, we can eliminate vertices that are not in G_{sub}. The distance map for vertices within k-1 hops can be computed by a bidirectional (k-1)-hop BFS. After computing the shortest distance map, we can find all vertices in V_{sub} and build the G_{sub}.

The worst-case time and space complexity are both $O(|V|+|E|)$ of the above operation because we conduct bidirectional BFS to compute distance map and then build the induced subgraph, which is at most the same size as the original graph G.

*2) **Bidirectional Search-based Index Construction**:* According to the above discussion, we can construct the partial path index by bidirectional search. The bidirectional search follows the BFS paradigm and two optimization techniques are proposed to further reduce the search space.

After the preprocessing, we prune the search space by eliminating those vertices that will never appear in the query result, but there still are some edges that may lead to invalid results. Consider the following observation.

Observation: For example, given a induced subgraph G_{sub} and a query $q(s, t, 5)$ as shown in Figure 3, we can see that all the vertices in G_{sub} is already satisfied $Dist_s[v] + Dist_t[v] \leq k$ so that most of them will lead to at least one k-st path. However, we can see those edges drawn in red dashed line, which will lead to some invalid paths, such as $\{s, v_1, v_2, v_4, v_3, v_5\}$ and $\{s, v_2, v_4, v_1, v_3, v_5\}$. If these edges can be eliminated in advance, we can avoid considerable invalid searching cost.

Motivated by this, we introduce the pruning rule as follows.

***Optimization** 1:* (Distance Pruning) In the process of the BFS-based search in G_{sub}, assume we have arrived at vertex u passed by path p. Then in the next step, we can prune the search space by eliminating vertices in $\{v|v \in N_{out}(u)$ satisfied $len(p) + 1 + Dist_t[v] > k\}$.

When performing a single directional BFS starting from s in Figure 3 and we have arrived at v_4 pass by v_1, v_2. The existing path length is 3. Then the out-neighbors are v_1, v_3, v_5, t, we can remove v_1 and v_3 from candidates by Optimization 1.

In addition to pruning techniques, there's another optimization technique in our solution. The basic reason why we adopt a bidirectional search paradigm is that the size of partial paths cut by the middle vertices cut is smaller than the final results to a great extent. In [3], the cut position of the middle vertices cut is the $\lceil k/2 \rceil$-th vertex along the path. Due to the uneven density of graphs, the exact middle position of a query may not be the optimal position to minimize the size of partial paths. Benefit from our BFS-based method, we can dynamically find a optimal position instead of the $\lceil k/2 \rceil$-th position which is used by *BC-JOIN* for each query q to store the index.

***Optimization** 2:* (Dynamic Mechanism) In each time of BFS search, we compare the number of paths in both forward and reverse direction and greedily continue the search in the direction with fewer paths. When the sum number of search levels in two directions reaches k, we use the meet position as the cut position of the partial path index.

By introducing the dynamic mechanism, we can significantly reduce the size of the partial path index. More importantly, this optimization can be applied to our BFS-based algorithm without any overheads.

Algorithm Description. Following the above ideas, our bidirectional BFS-based index construction algorithm, which traverses the search tree in level order, is shown in Algorithm 2. For each direction of the search, we use queues to store vertices and paths in the current level, and search for the next level based on them. Specifically, l (r) means left(right) and $v(p)$ means vertex (path). Then we start two level search from s and t respectively (line 4-5). In the process of the level search (line 11-18), we eliminate some edges by Optimization 1, and if the path we passed by is simple, we store it in the index and push the vertex and corresponding path to queues for subsequent path search (line 16-18). After the first-time level search from both directions, we adopt Optimization 2 that compares the number of paths in two directions and continues the search in the direction that have fewer paths (line 7-10). After each BFS search, we record the (i, j) pairs in $Plan$.

Algorithm 2: Bidirectional Index Construction

```
Input: The induced subgraph $G_{sub}$ and reverse of
       induced subgraph $G^r_{sub}$, source vertex $s$, target
       vertex $t$, current level $i$, hop constraint $k$
Output: Partial path index $LP$ and $RP$ and the join
        plan $Plan$
1  $i \leftarrow 0$, $j \leftarrow 0$;
2  $lQ^0_v \leftarrow \{s\}$, $rQ^0_v \leftarrow \{t\}$;
3  $lQ^0_p \leftarrow \{p(s)\}$, $rQ^0_p \leftarrow \{p(t)\}$;
4  level_search($G$, $t$, $k$, ++$i$, $lQ^i_v$, $lQ^i_p$, $LP$, $Dist_t$);
5  level_search($G^r$, $s$, $k$, ++$j$, $rQ^j_v$, $rQ^j_p$, $RP$, $Dist_s$);
6  Add $pair(i, j)$ in $Plan$;
7  while $i + j < k$ do
8      if $|rQ^j_p| \leq |lQ^i_p|$ then level_search($G$, $t$, $k$, ++$i$,
         $lQ^i_v$, $lQ^i_p$, $LP$, $Dist_t$);
9      else level_search($G^r$, $s$, $k$, ++$j$, $rQ^j_v$, $rQ^j_p$, $RP$,
         $Dist_s$);
10     Add $pair(i, j)$ to $Plan$;
11 Procedure level_search($G, t, k, i, Q^{i-1}_v, Q^{i-1}_p$, $P$,
     $Dist$):
12     $Q^i_v, Q^i_p \leftarrow \emptyset$;
13     while $Q^{i-1}_v \neq \emptyset$ do
14         $u \leftarrow Q^{i-1}_v.pop()$;
15         $p \leftarrow Q^{i-1}_p.pop()$;
16         foreach $v \in \{v | v \in N(u) \wedge v \neq$
             $t \wedge Dist[v] + i \leq k \wedge v \notin p\}$ do
17             Push $v$ to $Q^i_v$;
18             Push $p \oplus v$ to $Q^i_p$ and $P_i(v)$;
```

3) Complexity Analysis: The time cost of index construction is $(l \times |P_l| + r \times |P_r|)$, where $|P_l|$ and $|P_r|$ is the number of l-hop paths starting from s and the number of r-hop paths starting from t, respectively, where l and r are parameters generated by Algorithm 2 and $l + r = k$. The space cost is $(l \times |LP| + r \times |RP|)$, where $|LP|$ and $|RP|$ is the number of paths in LP and RP, respectively. Note that paths in $LP(RP)$ are all simple paths, but paths in P_l (P_r) may not be simple. And the time and space complexity of index construction are both $O(k \times |P|)$, where $|P|$ is the number of k-hop st paths from s to t.

B. Index Maintenance

Another challenging task is how to correctly and efficiently maintain the index against graph updates. More specifically, after the insertion of an edge, for each newly generated path, it should be ensured that all new left partial paths are in LP and right partial paths are in RP after the index maintenance. Consequently, we can quickly report the updated results from index without accessing the original graph and querying from scratch.

Similar to most of problems on dynamic graphs, only a part of updates will affect the query result. For the k-st path enumeration problem, we can check whether an edge update may affect the result by a simple inequality: For each edge update $e(u, v, +/-)$, if $Dist_s[u] + Dist_t[v] + 1 \leq k$ holds, then it may affect the result. In this paper, we only consider the situation that this inequality holds. Otherwise, the problem is trivial. And for simplicity, we only discuss the maintenance corresponding to the change of $Dist_s$. Because the maintenance corresponding to $Dist_t$ is the same but in the reverse graph. Specially, we denote the changed part of LP (RP) that are newly inserted or deleted after the edge update as LP' (RP'). Similarly, we denote the distance map after the edge update as $Dist'_s$ and $Dist'_t$.

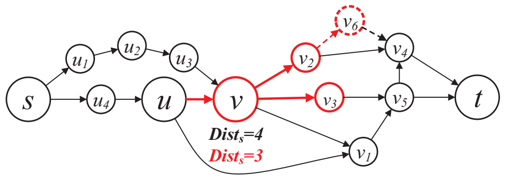


Fig. 4. Example of Edge Insertion

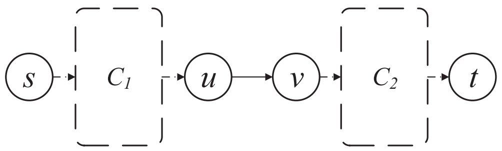


Fig. 5. Search Space of Index Maintenance

When an edge is inserted into (deleted from) the graph G, both the distance map and the index might be influenced. Firstly, it may lead to changes for both $Dist_s$ and $Dist_t$. Specifically, for a vertex v' that $Dist_s[v']$ decreases (increases) after the edge update, we call it is relaxed (tightened). Secondly, for a vertex v' that $Dist_s[v']$ $(Dist_t[v'])$ changes, both $LP(v')$ and $RP(v')$ can be affected, where $LP(v')$ and $RP(v')$ denote all paths end at v' in LP and RP, respectively. Consider the following example.

Example 1: As shown in Figure 4, consider a query $q(s, t, k = 7)$ and $l = 4, r = 3$, an edge $e(u, v)$ is inserted to the graph, then $Dist_s[v]$ is decreased from 4 to 3 because of the insertion. As a result of the relaxation of v, the successors of v may also be relaxed. In this example, v_2, v_3 are relaxed by $Dist_s[v]$, and v_6 is further relaxed by v_2. Particularly, v_6 is not in the induced graph initially, but it satisfies $Dist_s[v_6] + Dist_t[v_6] \leq k$ after the relaxation, thus v_6 should be added to G_{sub}. Then to maintain the index corresponding to v, paths from s passed by u to v with length 3 and paths from v to t with length 3 should be added to $LP(v)$ and $RP(v)$, respectively.

In the above example, we only discuss how to maintain the index for a single vertex, but the index should also be maintained for each vertex v' for which $Dist_s[v']$ changes and similar maintenance should also be considered in the reverse direction that $Dist_t[v']$ changes. Hence, the changes brought by an edge update can be very complicated, and it's quite challenging to maintain both RP and LP for every changed vertex efficiently.

To address the above challenge, we propose efficient techniques to maintain the shortest distance map and index. We first introduce the key idea of our techniques. In Figure 5, $e(u, v)$ is the edge update, and we divided vertices in all k-

st paths excluding s, t, u, v into two parts. Let $V(C_1)$ and $V(C_2)$ denote vertices between s and u and between v and t in all k-st paths, respectively. Due to the complex influence and subsequent chain reaction brought by the edge update, we should find an appropriate order that can make sure that the distance map and index are maintained correctly and we only maintain the LP and RP for each vertex once. It means that the priority is to maintain the part of the index that won't be affected by the subsequent. According to this, we process maintenance in the following order. (1) updating distance map $Dist_s$ and RP for vertices in $V(C_2) \cup \{v\}$; (2) updating distance map $Dist_t$ and LP for vertices in $V(C_1) \cup \{u\}$; (3) updating RP for vertices in $V(C_1) \cup \{u\}$; (4) updating LP for vertices in $V(C_2) \cup \{v\}$.

*1) **Edge Insertion**:* We explain how to maintain the distance map and partial path index against edge insertion in detail.

Distance Map Update. As we mentioned, the distance map may be changed after the insertion of edges. For example in Figure 5, an edge insertion $e(u, v, +)$ may reduce the distance from s to vertices in $V(C_2)$. Based on the above observation, we introduce the following theorem.

***Theorem** 5:* If a vertex v' is relaxed and $v' \neq v$, then $\exists u' \in N_{in}(v')$ is relaxed.

***Proof** 5:* Suppose on the contrary there is a relaxed vertex v' such that $\forall u' \in N_{in}(v')$, u' is unrelaxed, and we have $Dist_s[v'] = \min_{u' \in N_{in}(v')} Dist_s[u'] + 1$ before the insertion of $e(u, v)$, then we have $Dist'_s[v'] < \min_{u' \in N_{in}(v')} Dist_s[u'] + 1$ after the insertion. And $Dist_s[u'] = Dist'_s[u']$ holds because u' is unrelaxed. It contradicts the fact that $Dist'_s[v'] = \min_{u' \in N_{in}(v')} Dist'_s[u'] + 1$, thus the theorem holds.

According to Theorem 5, we can find that the relaxation of $Dist_s$ is spread from v in a tree form. Only when $Dist_s[u] + 1 < Dist_s[v]$ holds, $Dist_s[v]$ should be updated to $Dist_s[u] + 1$ and we continue to check the relaxation of follow-up vertices. After traversing all vertices following the inequality, the maintenance of the distance map is finished.

Algorithm Description. Algorithm 3 illustrates the method of finding relaxed vertices and updating the shortest distance map in detail. In the process of BFS, we use a set S_{relax} to store relaxed vertices. Also, we use a set S_{edge} to store vertices that are unrelaxed but at least one of their in-neighbors are relaxed. If v is relaxed by u, we add vertex v into set S_{nxt} and S_{relax}, and we update the $Dist'_s$ (line 3-5). Then we start the level search and check the relaxation of v's successors. If there are no subsequent relaxed vertices, we add v's successors into S_{edge} (line 13). If a successor v' is relaxed, it's necessary to continue the BFS by adding the relaxed successor to S_{nxt} and S_{relax}, and then we update the distance map $Dist'_s$ (line 14-16). Note that for each vertex v' that is not in the induced subgraph before and satisfied $Dist_s[v] + Dist_t[v'] \leq k$ after Algorithm 3, we add it to G_{sub}.

Index Update. According to the index structure proposed in Section III-A, for each relaxed vertex v that $Dist_s[v]$ decreases, right-side paths from v to t with length from $k - Dist_s[v]$ to $k - Dist'_s[v]$ and left-side paths from s passed by u to v with length up to l and should be added to $RP(v)$ and $LP(v)$, respectively. Similar to the traversing order of index construction, we naturally use a hop-constrained BFS to maintain the LP. And to maintain the RP, we design a DFS-based method to maintain RP starting from unrelaxed vertices. The details are shown in Algorithm 4.

Algorithm 3: Update the distance map after insertion

```
Input: Graph $G$, the inserted edge $e(u, v)$, distance
       map $Dist_s$ and $Dist_t$, hop constraint $k$
Output: the updated distance map $Dist_s$, relaxed
        vertices set $S_{relax}$, edge vertices set $S_{edge}$
1  foreach $v' \in G$ do $visited[v'] \leftarrow false$;
2  $S_{edge}$, $S_{relax}$, $S_{cur} \leftarrow \emptyset$;
3  if $Dist_s[u] + 1 < Dist_s[v]$ then
4      $S_{cur} \leftarrow \{v\}$;
5      $Dist'_s[v] \leftarrow Dist_s[u] + 1$;
6  level $i \leftarrow Dist_s[u] + 1$;
7  while $S_{cur} \neq \emptyset \wedge i < k$ do
8      $S_{nxt} \leftarrow \emptyset$;
9      $i$++;
10     foreach $u' \in S_{cur}$ do
11         foreach $v' \in \{v | v \in N_{out}(u') \wedge i + 1 \leq$
             $k \wedge visited[v] = false\}$ do
12             $visited[v'] \leftarrow true$;
13             if $Dist_s[v'] + Dist_t[v'] \leq k$ then add $v'$ to
                 $S_{edge}$;
14             if $i < Dist_s[v']$ then
15                 add $v'$ to $S_{nxt}$ and $S_{relax}$;
16                 $Dist'_s[v'] = \min(Dist_s[v'], i)$;
17     $S_{cur} \leftarrow S_{nxt}$;
```

Algorithm Description. After updating the distance map and getting relaxed vertices S_{relax} and edge vertices S_{edge}, we conduct a hop-constrained DFS search named *UDFS*. The *UDFS* starts from vertices in S_{edge} and search in vertices S_{relax} to maintain the RP' for vertices in $V(C_2) \cup \{v\}$ (line 1-6 in Algorithm 4). In detail, *UDFS* searches each vertex $v'' \in \{v | v \in N_{in}(v') \wedge i + 1 + Dist'_s[v] \leq k \wedge v \in S_{relax} \wedge Dist_s[v] + i + 1 > k\}$ and add $p \oplus v' \oplus v''$ to the index $RP'_{i+1}(v'')$, and it terminates when arriving r hops away from t. Note that $\oplus$ is a path concatenating operation.

To update the left partial path index LP for vertices in $V(C_2)$, we perform a hop-constrained BFS from v, traversing the vertices within l hops from s, and updating the LP for them with the paths in their predecessors' index (line 9-18).

After all these updates, all changes of the index for vertices in $V(C_2)$ are well maintained. Similar to above process, if $Dist_t[v] + 1 < Dist_t[u]$ holds, we also need to update the distance map $Dist_t$ and index for vertices in C_1 as the order we mentioned before.

*2) **Edge Deletion**:* Different from the strategy against edge insertion, it is hard to maintain the distance map after edge deletion. To maintain the distance map against insertion, we can easily find relaxed vertices and directly update the shortest distance from s and $v' \in V(C_2)$ with its predecessors respectively. But after the edge deletion, the increase of $Dist_s[v']$

Algorithm 4: Maintain the index after insertion

```
Input: Induced subgraph $G_{sub}$, the inserted edge
       $e(u,v)$, edge vertices set $S_{edge}$, relaxed
       vertices set $S_{relax}$, distance map $Dist_s$ and
       $Dist_t$, hop constraint $k$
Output: the updated index $LP', RP'$
1  foreach $u' \in S_{edge}$ do
2      foreach $\{i | Dist_t[u'] \le i < r\}$ do
3          foreach
           $v' \in \{v | v \in \{N_{in}(u') \wedge i+1+Dist'_s[v] \le$
           $k \wedge v \in S_{relax} \wedge Dist_s[v]+i+1 > k\}$ do
4              foreach $p \in RP_i(u')$ do
5                  if $v' \notin p$ then add $p \oplus v'$ to $RP'_{i+1}(v')$;
6                  if $v' \neq v \wedge i+1 < r$ then
                       UDFS$(i+1, p \oplus v', v')$;
7  foreach $\{i | Dist_t[v] \le i < r\}$ do
8      foreach $p \in RP'_i(v)$ do add $p \oplus u$ to $RP'_{i+1}(u)$;
9  $S_{cur} \leftarrow \{v\}$;
10 foreach $\{i | Dist_s[u]+1 \le i < l\}$ do
11     $S_{nxt} \leftarrow \emptyset$;
12     foreach $u' \in S_{cur}$ do
13         foreach
           $v' \in \{v | v \in N_{out}(u') \wedge i+1+Dist'_t[v'] \le k\}$
           do
14             add $v'$ to $S_{nxt}$;
15             foreach $p \in LP'_i(u')$ do
16                 if $v' \notin p$ then add $p \oplus v'$ to $LP'_{i+1}(v')$;
17     if $LP'_{i+1}(v) \neq \emptyset$ then add $v$ to $S_{nxt}$;
18     $S_{cur} \leftarrow S_{nxt}$;
```

only happens in the situation that the shortest path from s to v' passed by the expired edge $e(u,v)$, which is hard to be directly computed. Thus the maintenance strategy we used against edge insertion is invalid because we can't straightforwardly know the shortest distance between s and v' after edge expiration. It is obviously inefficient to re-compute the distance map for each update.

Distance Map Update. To overcome this challenge, we design a BFS-based algorithm to update the distance map without re-computation. Firstly, although we can't directly find the tightened vertices like Algorithm 3, we can find all the possible tightened vertices by their $Dist_s$. After the expiration of $e(u,v)$, $Dist_s[v]$ may be increased. If $Dist_s[v] \le Dist_s[u]$, then the expiration of $e(u,v)$ makes no difference to $Dist_s[v]$, because all the shortest paths from s to v don't pass by $e(u,v)$. Otherwise, v is a possible tightened vertex, and we can find a possible tightened vertices set $S_{tighten}$ according to the inequality $Dist_s[v'] = Dist_s[u'] + 1$.

After that, an observation is that for each vertex $v' \in V(C_2) \cup \{v\}$, $Dist_s[v']$ should be equal to $Dist_s[u'_{min}] + 1$, where $u'_{min} = \arg\min_{u' \in N_{in}(v')} Dist_s[u']$. But only when u'_{min} is not a tightened vertex, we can use $Dist_s[u'_{min}] + 1$ to update $Dist_s[v']$. If u'_{min} is a tightened vertex after the edge deletion, we can't update $Dist_s[v']$ until $Dist_s[u'_{min}]$ is updated. In a worse case, for example, when all tightened vertices are in a loop, then the above check will fall in the

Algorithm 5: Update distance map after deletion

```
Input: Graph $G$, the deleted edge $e(u,v)$, distance
       map $Dist_s$ and $Dist_t$, hop constraint $k$
Output: the updated distance map $Dist'_s, Dist'_t$
1  if $Dist_s[u]+1 \neq Dist_s[v]$ then return;
2  foreach $v' \in G$ do $visited[v'] \leftarrow 0$;
3  $S_{edge} \leftarrow \emptyset$, $S_{tighten} \leftarrow \emptyset$, $S_{cur} \leftarrow \{v\}$;
4  level $i \leftarrow Dist_s[u]+1$;
5  while $S_{cur} \neq \emptyset \wedge i < k$ do
6      $S_{nxt} \leftarrow \emptyset$;
7      $i$++;
8      foreach $u' \in S_{cur}$ do
9          foreach $v' \in \{v | v \in N_{out}(u') \wedge i+1 \le$
           $k \wedge visited[v] = false\}$ do
10             $visited[v'] \leftarrow true$;
11             if $Dist_s[u']+1 = Dist_s[v']$ then
12                 add $v'$ in $S_{tighten}$;
13                 foreach
                   $u'' \in \{v | v \in N_{in}(v') \wedge v \notin S_{tighten}\}$
                   do
14                     add $u''$ in $S_{edge}$;
15     $S_{cur} \leftarrow S_{nxt}$;
16 Initialize $k$-1 buckets $B_{0,\ldots,k-2}$ with vertices in $S_{edge}$;
17 while $S_{tighten} \neq \emptyset$ do
18     $u' \leftarrow$ the vertex with minimal $Dist_s$ in $B$;
19     Remove $u'$ from $B$;
20     foreach $v' \in N_{out}(u')$ do
21         if $v' \in S_{tighten}$ then
22             $Dist'_s[v'] \leftarrow Dist'_s[u']+1$;
23             add $v'$ into $B$;
24             $S_{tighten} \leftarrow S_{tighten} \setminus \{v'\}$;
```

loop and makes it hard to update the distance map. To solve this, we find all the predecessors of possible tightened vertices that are not in $S_{tighten}$ and add them to a set S_{edge}. We straightforwardly start a one-hop BFS from vertices in S_{edge}, to search for the possible tightened vertices and update their distance map.

Algorithm Description. In Algorithm 5, we use a set $S_{tighten}$ to store the vertices that will probably be tightened. If $Dist_s[v] = Dist_s[u] + 1$ holds, we conduct a level search starting from v to find possible tightened vertices (line 2-15). In the level search, for each v's out-going neighbor v', if v' is a possible tightened vertex, we add v' to the set $S_{tighten}$ and S_{nxt} (line 11-12). Then we add all unrelaxed vertices in v''s in-going neighbors to S_{edge} (line13-14). The process repeats until the level search arrives k-1 hops away from s.

We assign vertices in S_{edge} to k-1 buckets according to $Dist_s[v']$ (line 16). For example, we insert the vertex v' with $Dist_s[v'] = 1$ into B_1. Then we traverse buckets in increasing order of $Dist_s$ (line 18-19). For each vertex v', if its out-neighbor $u' \in S_{tighten}$, the shortest distance from s to u' passes by v'. So we update the $Dist_s[u']$ and move u' from $S_{tighten}$ to $B_{Dist'_s[u']}$ (line 20-24). The process terminates when $S_{tighten}$ is empty.

Index Update. After updating the distance map, we can maintain the index accordingly. For each tightened vertex v', we remove all paths that in $RP_i(v'), k - Dist'_s[v'] < i \leq r$ and remove v' from G_{sub} if $Dist'_s[v'] + Dist'_t[v'] > k$. After that, we have maintained RP for vertices in $\{v\} \cup V(C_2)$. To maintain LP for vertices in $\{v\} \cup V(C_2)$, we start a (k-$Dist_s[v]$)-hop BFS from v, and check all vertices we visited in this BFS search. For each vertex v' we visited, we check $RP(v')$ and remove all the paths that passed by $e(u, v)$.

After all these updates, all the changes of the index in $V(C_2)$ are well maintained. Similar to the above process, we also should update the distance map $Dist_t$ and index for vertices in C_1. Note that to enumerate deleted paths after edge deletion, we keep the paths that should be removed and delete them after finishing the update enumeration.

*3) **Complexity Analysis**:* To maintain the distance map against insertion, we visit each out-going edge of each relax vertices at most once, which costs $O(|E|)$ time in the worst case. And when searching in relaxed vertices S_{relax} to maintain the index, we visit each path that is new/deleted at most once and add it to our index if it is simple. Thus the time cost is k times the number of new/deleted partial paths in LP and RP, which is less than $2 \times \Delta|P|$, where $\Delta|P|$ is the number of new/deleted hop-constrained s-t paths with length up to k resulting from the edge update. Hence the time cost of maintaining the index against insertion is $O(k \times \Delta|P|)$,

To maintain the distance map against deletion, we visit each (in-going and out-going) edge of each vertices in $S_{tighten}$ at most once, which costs $O(|E|)$ time in the worst case. After that, it takes $O(k+|V|)$ to add vertices in S_{edge} to k-1 buckets. The rest cost of updating distance map against deletion is to traverse all vertices in buckets B, which costs at most $O(|E|)$. Thus the time cost of maintaining the distance map against deletion is $O(|E|+k)$ and index update for deletion is $O((k+d_{avg}) \times \Delta|P|)$.As removing a path in our index passed by $e(u, v)$, we need to remove paths passed by $e(u, v)$ in its successors, which costs $d_{avg} \times \Delta|P|$.

The space cost of maintenance is $O(k \times \Delta|P|)$, which is the size of new/deleted parts in the index. Thus after maintenance, the index still follows the overall $O(k \times |P|)$ space complexity.

V. Evaluations

A. Experimental Setting

*1) **Datasets**:* 14 real-world graphs are utilized to evaluate the effectiveness and efficiency of our proposed techniques. All datasets are downloaded from three public websites: Konect [21], NetworkRepository [22] and SNAP [23]. TABLE I demonstrates the details of the datasets. Note that d_{avg} denotes the average degree, D denotes the diameter, and D_{90} denotes the 90-percentile effective diameter of the graph.

*2) **Settings**:* We obtain the source code of *BC-JOIN* from the original authors, *PathEnum* [24], CSM^* [25] and proposed algorithms [26] are open-source. All programs are implemented in C++ and compiled by g++ 8.3.1 with -O3 enabled. All experiments are performed on a machine with 32 Intel Xeon 2.1GHz and 256GB main memory running Linux (CentOS 7.0).

TABLE I
Datasets Used in Experiments

Dataset	$Name$	$\|V\|$	$\|E\|$	d_{avg}	D	D_{90}
Reactome	RT	6.3K	294K	46.64	24	5.39
soc-Epinions1	EP	75K	1.01M	13.42	14	5
Slashdot0922	SD	82K	1.89M	23.08	11	4.7
Amazon	AM	334K	2.26M	6.76	44	15
twitter-social	TS	465K	1.79M	3.86	8	4.96
Baidu	BD	425K	6.72M	15.8	32	8.54
BerkStan	BS	685K	15.2M	22.18	208	9.79
web-google	WG	875K	10.2M	11.6	24	7.95
Skitter	SK	1.6M	20.8M	13.08	31	5.85
WikiTalk	WK	2M	8.4M	4.2	9	4
soc-pokec	PK	1.6M	30M	18.4	11	5.2
LiveJournal	LJ	4M	113.6M	28.4	16	6.5
DBpedia	DP	18M	339M	18.85	12	4.98
Twitter (WWW)	TW	42M	2.96B	70.51	23	3.97

*3) **Comparisons**:* We study the following algorithms in comparison with our algorithms.

- *BC-JOIN* [3]: The most competitive bidirectional search-based solution for k-st path enumeration on static graphs.
- *PathEnum* [5]: The state-of-the-art method for k-st path enumeration on static graphs.
- *CSM**: The state-of-the-art solutions on dynamic graphs as we mentioned in Section I. According to [27], there is no absolute winner in CSM problem, so we report the most efficient algorithm (denoted by *CSM**) among SJ-Tree [15], Graphflow [19], IEDyn [17], TurboFlux [16] and SymBi [20] in each experiment. Since *CSM** only support undirected graph, so we only report its performance on undirected datasets AM, SK and LJ.
- *CPE$_{startup}$*: Our solution including the *index construction* and *start-up enumeration* algorithms.
- *CPE$_{update}$*: Our solution including the *index maintenance* and *update enumeration* algorithms.

B. ***Efficiency of Start-up Stage.***

In Section III-B1 and Section IV-A, we introduce how we build the index and enumerate k-st paths on the initial graph, which can be adopted to deal with the traditional static k-st path enumeration problem [3]. Therefore, we evaluate the running time of our *CPE$_{startup}$* algorithm to answer the traditional k-st path enumeration problem on different datasets and compare it with two existing SoTA algorithms, *PathEnum* and *BC-JOIN*. We set $k = 6$ and randomly generate 1,000 queries for each dataset. As shown in Figure 6, the query time on different graphs varies greatly, which ranges from less than one millisecond to hundreds of seconds.

Although our solution needs to store the partial paths to support subsequent maintenance on dynamic graphs and *PathEnum* is an online algorithm that focuses on static graphs and doesn't store the intermediate results, our start-up stage algorithm still can achieve similar performance to the state-of-the-art algorithm *PathEnum*. And compared to *BC-JOIN*, we achieve up to three orders of magnitude speedup. Although the barrier-based pruning of *BC-JOIN* can provide a theoretical guarantee, the maintenance of barrier is quite expensive in practice. Furthermore, our pruning technique can prune the majority of invalid paths which further improve the efficiency of the proposed *CPE$_{startup}$*. In addition, we also can see that

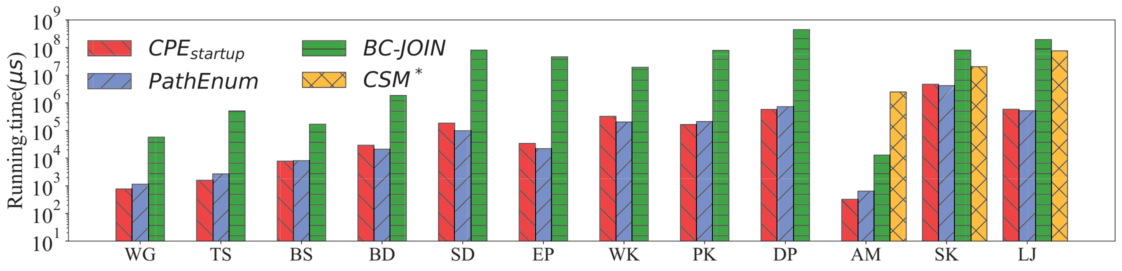
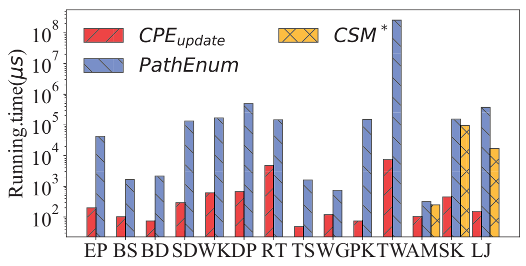


Fig. 6. Efficiency of Start-up Stage on different datasets.



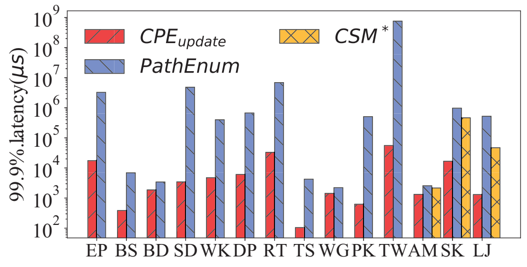


Fig. 7. Efficiency of Update Stage on different datasets.

the general proposed continuous subgraph matching method CSM^* is less efficient on the start-up stage. Note that the running time of index construction is included in both *CPE*$_{startup}$ and *CSM**.

C. Efficiency of Update Stage.

In this subsection, we evaluate the efficiency of our update stage algorithm *CPE*$_{update}$ (includes both index maintenance and update enumeration) against edge updates on 14 different datasets with $k = 6$.

In our experiments, we continuously process the query, maintain the index and update the result instead of adopting sliding window model or batch strategy to deal with the update of edges. Because of graph sparsity, when more than one edge arrives at the same time, there is few computation shared by dealing with them in a batch. Therefore, we process the update on the fly. As we mentioned in Section IV-B, we only consider edges that actually affect the result of k-st paths.

*1) **Efficiency on Different Datasets**:* We randomly generate 10 queries from vertices within the top 10% in the descending order of degree for each dataset, and 200 random edge updates (100 insertions and 100 deletions) are generated for each query pair (s, t). Then we report the average running time that be spent to maintain the index and enumerate the new/deleted k-st paths in Figure 7.

***CPE*$_{update}$ vs. Baselines.** The *CPE*$_{update}$ algorithm significantly outperforms *PathEnum* and *CSM** on all datasets, especially those with long query time. For example, *CPE*$_{update}$ runs 33985 times faster than *PathEnum* on TW in terms of running time. As a result of the sparsity that real-world graphs have, when we randomly generate the insertion or deletion of an edge for a random query, the most common situation is that the edge only incurs a few changes for the k-st paths. So the *CPE*$_{update}$ algorithm with $\Delta|P|$-related time cost is significantly faster than *PathEnum* with $|P|$-related time cost, especially for those query pairs costly in static graphs. And compared to *CSM**, which is also an index-based method, we still achieve better performance because our proposed pruning techniques can significantly reduce the search space and fruitless exploration is avoided. And owing to our efficient index structure, the time cost of update enumeration is negligible, and the majority of the running time of *CPE*$_{update}$ is spent to maintain the index.

Effect of Dataset. As shown in the Figure 7, the running time on different datasets with different graph topology varies greatly, which ranges from tens of microseconds to over milliseconds. The running time of *CPE*$_{update}$ depends on graph size, graph density, and the specific query $q(s, t, k)$. What stands out in Figure 7 is that the *CPE*$_{update}$ running time on BD is much more than on TS. Although they have a similar number of vertices, the d_{avg} of BD is bigger than TS. This implies that BD is a graph with considerable local density. In Figure 7, We also examine the 99.9% latency of *CPE*$_{update}$, *CSM** and *PathEnum* in terms of the response time. We can find that the latency of *CPE*$_{update}$ is much smaller than *CSM** and *PathEnum* on most graphs. But in some of graphs, they are close in latency. This is because in these graphs and some query pairs, an arrival or expiration of edge may lead a large proportion of paths to be changed that making $\Delta|P|$ be close to $|P|$, thus the cost of computing new/deleted paths is close to the cost of re-computing all paths in these queries.

*2) **Insertion vs. Deletion**:* As shown in Figure 8, the running time of *CPE*$_{update}$ for dealing with the insertion and deletion of edges are quite close. And we can find that the running time is highly relevant to the number of new/deleted paths, which is consistent with our analysis about time com-

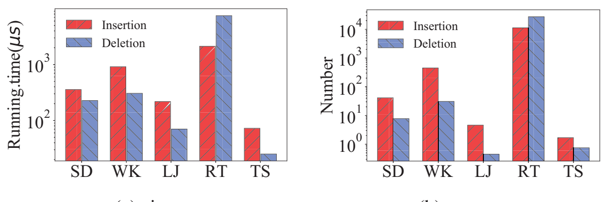

(a) time (b) count

Fig. 8. Performance for *CPE*$_{update}$ against insertion and deletion with $k = 6$.

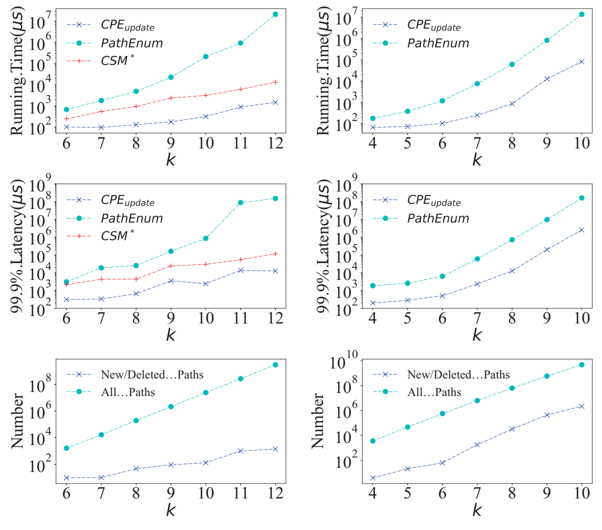

(a) AM (b) WG

Fig. 9. Update Stage on AM and WG with k varied.

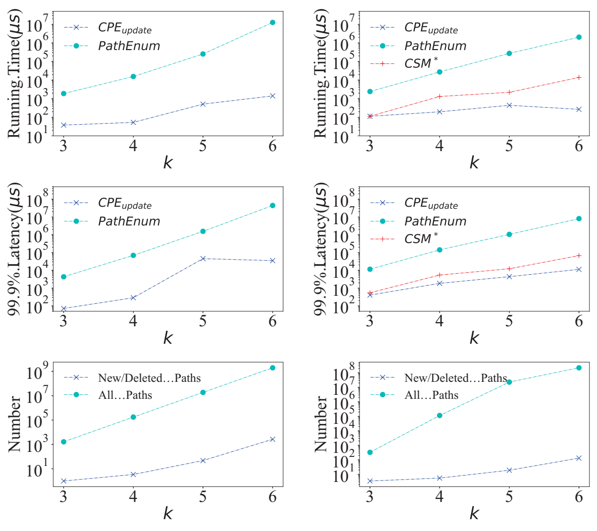

(a) EP (b) SK

Fig. 10. Update Stage on hot query pairs with k varied.

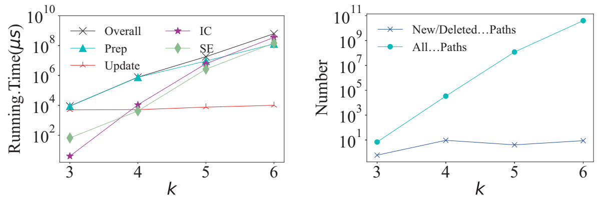

(a) Running Time (b) Number of Results

Fig. 11. Scalability evaluation on TW with k varied.

plexity in Section IV-B.

3) ***Effect of k**:* We conduct more extensive experiments on WG and AM with k varied. Figure 9 presents the average query time and 99.9% latency of *CPE*$_{update}$, *CSM** and *PathEnum* algorithm with different k. As shown in the Figure 9, *CPE*$_{update}$ significantly outperforms *CSM** and *PathEnum*, which further proves the efficiency of our *CPE*$_{update}$ algorithm. In particular, it demonstrates the great scalability of our *CPE*$_{update}$ algorithm regarding the growth of the hop constraint k. As shown in Figure 9, the number of k-st paths grows exponentially w.r.t k. But the number of new/deleted paths is not fully consistent with it, which is more relevant to the density of the induced subgraph w.r.t the specific query as we mentioned before and shows that the superiority of conducting maintenance and update instead of re-computing.

4) ***Efficiency on Hot Query Pairs**:* To testify the efficiency of *CPE*$_{update}$ algorithm on hot query pairs, we evaluate our algorithms on queries that s and t are randomly selected from vertices within the top 1% in the descending order of degree, which produce extremely dense induced graphs and a huge number of results. The query time, 99.9% latency and the number of new/deleted paths are presented in Figure 10. We can find that *CPE*$_{update}$ still can significantly outperforms *CSM** and *PathEnum* in both running time and 99.9% latency, which shows the efficiency and scalability of our *CPE*$_{update}$ algorithm. As shown in Figure 10, the running time of *CPE*$_{update}$ on hot query pairs still grows w.r.t the number of new/deleted paths.

D. *Scalability Evaluation.*

We evaluate the scalability of our algorithms on *TW* dataset that has over two billion edges. As shown in Figure 11, we report the average running time of each individual component of our algorithm in 10 random queries and 200 edge updates (100 insertions and 100 deletions). "Prep" denotes the time spent on building the shortest distance map and the induced subgraph. "IC" denotes the time spent on building our partial path index. After that, "SE" is the time spent on enumerating all k-st paths in the initial graph. "Overall" is the sum running time of "Prep", "IC", and "SE", which is the running time of a whole static path enumeration query. Then we use "Update" to represent the sum running time of index maintenance and update enumeration. Similar to the result in Figure 9, the overall number of results grows exponentially w.r.t k in Figure 11. Because the induced subgraph wouldn't be denser in TW when the k grows, the count of new/deleted paths is insensitive to k, which demonstrates great scalability again.

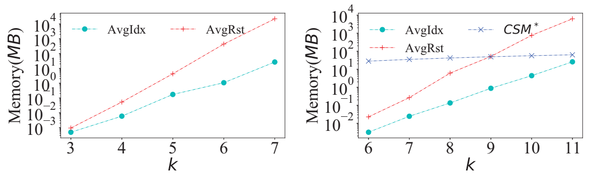

(a) BS (b) AM

Fig. 12. Average main memory usage.

E. Main Memory Usage.

Figure 12 shows the main memory usage of the partial path index on 2 datasets over 1,000 random queries with different k. To better show the tendency of our solution, we exclude memory usage of the graph storage. "AvgIdx" shows the average memory usage of our index. The memory usage grows exponentially w.r.t k due to the number of results growing exponentially w.r.t k. Note that after the construction of the index, when edges arrive or expire continuously, the space cost of our solution changes following the change of paths number $\Delta|P|$.

We use "AvgRst" to denote the average size of all k-st paths. Compared with the space cost of storing all k-st paths, our partial path index always uses much less main memory. Especially when k is not that small, compared to store all paths, our partial path index only occupies less than 1% of memory space. And the blue line shows the average index memory usage of *CSM**, which grows linearly w.r.t k because its space complexity is directly proportional to k.

VI. Related Works

k-st Path Enumeration on Static Graphs. Existing approaches [1]–[3], [5] adopt a DFS-based backtracking strategy, but introduce different pruning techniques to reduce the search space. *T-DFS* [2] and *T-DFS2* [1] prune the search space by ensuring that each search branch in the search tree leads to a result. *BC-DFS* [3] is a barrier-based method, which prunes the search space by dynamically maintaining the distance from each vertex to t to avoid repeatedly falling into the same subtree contains no result. A bidirectional search-based method *BC-JOIN* is also proposed by [3], which is the combination of *BC-DFS* and the bidirectional search paradigm. It significantly reduces the search space and avoids enumerating duplicate paths. Although *T-DFS*, *T-DFS2* and *BC-JOIN* all achieve $O(k{\times}|E|)$ polynomial delay, it was shown in [3] that *BC-JOIN* runs much faster than *T-DFS* and *T-DFS2* in practice because their pruning strategy incurs lower overhead. Additionally, *PEFP* [4] is a FPGA method based on *BC-DFS*. *HybridEnum* [6] is a scalable distributed k-st path enumeration algorithm. Sun et al. proposed *PathEnum* [5], which designs an online index to prune invalid branches, adopts a single directional join strategy based on the index, and then develops a cost-based query optimizer to further improve the performance. Different from previous algorithms, the time complexity of *PathEnum* is $O(k \times |P|)$, where $|P|$ is number of path from s to t.

In general, the existing two most competitive algorithms *BC-JOIN* and *PathEnum* are both join-based methods but adopt two different join paradigms:

BC-JOIN employs a bidirectional search-based join paradigm. It first computes all the middle vertices. Then it applies *BC-DFS* to compute paths with length at most $k/2$ starting from s and t respectively. Finally, it joins the paths starting from s and t in middle vertices to obtain the final results. *BC-JOIN* produces enough intermediate results with the potential to support subsequent updates. However, the BC-DFS method is time consuming, especially for large graphs.

PathEnum adopts a single directional search-based join paradigm. For a specific query, it first conducts a cardinality estimator to decide whether to make an full-fledged estimation and do a join operation or not. Then it computes the paths on the online index and performs the join operation with a check for path simplicity according to the estimation result. Because of the single directional paradigm and cardinality estimators, *PathEnum* produces either none or only a portion of intermediate results, which is unhelpful to avoid recomputing the result from scratch for graph updates.

Continuous Subgraph Matching. The k-st path enumeration problem on dynamic graphs can be addressed by continuous subgraph matching (CSM) methods [15]–[20], which aims to report newly generated/deleted matches on insertion or deletion of edges. [27] conducts an in-depth study about the state-of-art CSM methods, which are adopted as the baseline in the paper. However, as demonstrated in the experiments, these methods are inefficient and unscalable for k-st path enumeration.

Other Related Works. Besides, the related problems such as succinct presentation of s-t simple paths [28], [29], enumerating hop-constrained cycles against the arrival of incoming edges in dynamic graphs [8], enumerating all s-t simple paths without the hop constraint [30]–[33], finding all constrained or diversified shortest paths [34]–[38] and top-k' shortest path problem [39]–[47] are also studied in the literature. There are some works [48]–[52] on the problem of computing the shortest path when the input graph is dynamically updated. However, due to the difference of the studied problems, all these corresponding algorithms cannot be applied to the k-st path enumeration.

VII. Conclusion.

The k-st path enumeration problem is a fundamental problem in graph analysis with a wide range of applications. Though this problem has been intensively studied on static graphs, this paper is the first to investigate the problem in the context of dynamic graphs with continuous edge insertion and deletion operations. In this paper, efficient index-based algorithms have been developed with the best theoretical time complexity and practical performance compared to other solutions.

Acknowledgment

Shiyu Yang is supported by NSFC61802127, Guangzhou Basic and Applied Basic Research Foundation (202201020131) and CCF-Ant Research Fund. Dian Ouyang is supported by Guangzhou Research Foundation (202201020165). Fan Zhang is partially supported by NSFC 62002073, Guangzhou Research Foundation (202102020675). Xuemin Lin is partially supported by Guangdong Basic and Applied Basic Research Foundation (2019B1515120048). Long Yuan is supported by National Key RD Program of China 2022YFF0712100, NSFC61902184, and Science and Technology on Information Systems Engineering Laboratory WDZC20205250411.

References

[1] R. Grossi, A. Marino, and L. Versari, "Efficient algorithms for listing k disjoint st-paths in graphs," in *Latin American Symposium on Theoretical Informatics*, pp. 544–557, Springer, 2018.

[2] R. Rizzi, G. Sacomoto, and M.-F. Sagot, "Efficiently listing bounded length st-paths," in *International Workshop on Combinatorial Algorithms*, pp. 318–329, Springer, 2014.

[3] Y. Peng, Y. Zhang, X. Lin, W. Zhang, L. Qin, and J. Zhou, "Hop-constrained s-t simple path enumeration: Towards bridging theory and practice," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 463–476, 2019.

[4] Z. Lai, Y. Peng, S. Yang, X. Lin, and W. Zhang, "Pefp: Efficient k-hop constrained st simple path enumeration on fpga," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 1320–1331, IEEE, 2021.

[5] S. Sun, Y. Chen, B. He, and B. Hooi, "Pathenum: Towards real-time hop-constrained st path enumeration," in *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*, pp. 1758–1770, 2021.

[6] K. Hao, L. Yuan, and W. Zhang, "Distributed hop-constrained st simple path enumeration at billion scale," *Proceedings of the VLDB Endowment*, vol. 15, no. 2, pp. 169–182, 2021.

[7] N. R. Council *et al.*, *Dynamic social network modeling and analysis: Workshop summary and papers*. National Academies Press, 2003.

[8] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," *Proceedings of the VLDB Endowment*, vol. 11, no. 12, pp. 1876–1888, 2018.

[9] F. A. T. Force, "Fatf report: Money laundering and terrorist financing vulnerabilities of legal professionals," *Paris: FATF*, 2013.

[10] X. Li, S. Liu, Z. Li, X. Han, C. Shi, B. Hooi, H. Huang, and X. Cheng, "Flowscope: Spotting money laundering based on graphs," in *The Thirty-Fourth AAAI Conference on Artificial Intelligence*, pp. 4731–4738, AAAI Press, 2020.

[11] C. Jedrzejek, J. Bak, and M. Falkowski, "Graph mining for detection of a large class of financial crimes," in *17th International Conference on Conceptual Structures, Moscow, Russia*, vol. 46, 2009.

[12] D. Yue, X. Wu, Y. Wang, Y. Li, and C.-H. Chu, "A review of data mining-based financial fraud detection research," in *International Conference on Wireless Communications, Networking and Mobile Computing*, pp. 5519–5522, IEEE, 2007.

[13] M. Kimura and K. Saito, "Tractable models for information diffusion in social networks," in *The 10th European Conference on Principles and Practice of Knowledge Discovery in Databases*, vol. 4213, pp. 259–271, Springer, 2006.

[14] R. Misra and K. B. Misra, "Enumeration of all simple paths in a communication network," *Microelectronics Reliability*, vol. 20, no. 4, pp. 419–426, 1980.

[15] S. Choudhury, L. B. Holder, G. C. Jr., K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," in *Proceedings of the 18th International Conference on Extending Database Technology*, pp. 157–168, 2015.

[16] K. Kim, I. Seo, W.-S. Han, J.-H. Lee, S. Hong, H. Chafi, H. Shin, and G. Jeong, "Turboflux: A fast continuous subgraph matching system for streaming graph data," in *Proceedings of the 2018 International Conference on Management of Data*, pp. 411–426, 2018.

[17] M. Idris, M. Ugarte, and S. Vansummeren, "The dynamic yannakakis algorithm: Compact and efficient query processing under updates," in *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 1259–1274, 2017.

[18] M. Idris, M. Ugarte, S. Vansummeren, H. Voigt, and W. Lehner, "General dynamic yannakakis: conjunctive queries with theta joins under updates," *The VLDB Journal*, vol. 29, no. 2, pp. 619–653, 2020.

[19] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proceedings of the 2017 ACM International Conference on Management of Data*, pp. 1695–1698, 2017.

[20] S. Min, S. G. Park, K. Park, D. Giammarresi, G. F. Italiano, and W.-S. Han, "Symmetric continuous subgraph matching with bidirectional dynamic programming," *arXiv preprint arXiv:2104.00886*, 2021.

[21] J. Kunegis, "KONECT – The Koblenz Network Collection," in *Proc. Int. Conf. on World Wide Web Companion*, pp. 1343–1350, 2013.

[22] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015.

[23] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.

[24] Xtra-Computing, "The source code of pathenum." https://github.com/Xtra-Computing/PathEnum.

[25] RapidsAtHKUST, "The source code of continuous subgraph matching." https://github.com/RapidsAtHKUST/ContinuousSubgraphMatching.

[26] J. Zhang, "The source code of cpe." https://drive.google.com/drive/folders/1iwZtCkZDC-Sd0Uo42g7SQLK-CmnyURXyS?usp=sharing.

[27] X. Sun, S. Sun, Q. Luo, and B. He, "An in-depth study of continuous subgraph matching," *Proceedings of the VLDB Endowment*, vol. 15, no. 7, pp. 1403–1416, 2022.

[28] K. Böhmová, L. Häfliger, M. Mihalák, T. Pröger, G. Sacomoto, and M.-F. Sagot, "Computing and listing st-paths in public transportation networks," *Theory of Computing Systems*, vol. 62, no. 3, pp. 600–621, 2018.

[29] N. Yasuda, T. Sugaya, and S.-I. Minato, "Fast compilation of st paths on a graph for counting and enumeration," in *Advanced Methodologies for Bayesian Networks*, pp. 129–140, PMLR, 2017.

[30] E. Birmelé, R. A. Ferreira, R. Grossi, A. Marino, N. Pisanti, R. Rizzi, and G. Sacomoto, "Optimal listing of cycles and st-paths in undirected graphs," in *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1884–1896, SIAM, 2013.

[31] D. B. Johnson, "Finding all the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975.

[32] R. Kumar and T. Calders, "2scent: an efficient algorithm for enumerating all simple temporal cycles," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1441–1453, 2018.

[33] R. E. Tarjan, "Enumeration of the elementary circuits of a directed graph," *SIAM Journal on Computing*, vol. 2, no. 3, pp. 211–216, 1973.

[34] G. Y. Handler and I. Zang, "A dual algorithm for the constrained shortest path problem," *Networks*, vol. 10, no. 4, pp. 293–309, 1980.

[35] S. Wang, X. Xiao, Y. Yang, and W. Lin, "Effective indexing for approximate constrained shortest path queries on large road networks," *Proceedings of the VLDB Endowment*, vol. 10, no. 2, pp. 61–72, 2016.

[36] L. Wang, L. Yang, and Z. Gao, "The constrained shortest path problem with stochastic correlated link travel times," *European Journal of Operational Research*, vol. 255, no. 1, pp. 43–57, 2016.

[37] H. Liu, C. Jin, B. Yang, and A. Zhou, "Finding top-k shortest paths with diversity," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 3, pp. 488–502, 2017.

[38] C. Voss, M. Moll, and L. E. Kavraki, "A heuristic approach to finding diverse short paths," in *IEEE International Conference on Robotics and Automation, ICRA*, pp. 4173–4179, IEEE, 2015.

[39] L. Chang, X. Lin, L. Qin, J. X. Yu, and J. Pei, "Efficiently computing top-k shortest path join," in *Proceedings of the 18th EDBT International Conference on Extending Database Technology*, 2015.

[40] D. Eppstein, "Finding the k shortest paths," *SIAM Journal on computing*, vol. 28, no. 2, pp. 652–673, 1998.

[41] J. Gao, H. Qiu, X. Jiang, T. Wang, and D. Yang, "Fast top-k simple shortest paths discovery in graphs," in *Proceedings of the 19th ACM international conference on Information and knowledge management*, pp. 509–518, 2010.

[42] E. Q. Martins and M. M. Pascoal, "A new implementation of yen's ranking loopless paths algorithm," *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, vol. 1, no. 2, pp. 121–133, 2003.

[43] J. Y. Yen, "Finding the k shortest loopless paths in a network," *Management Science*, vol. 17, no. 11, pp. 712–716, 1971.

[44] L. Roditty and U. Zwick, "Replacement paths and k simple shortest paths in unweighted directed graphs," in *International Colloquium on Automata, Languages, and Programming*, pp. 249–260, Springer, 2005.

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

[46] J. Hershberger, M. Maxel, and S. Suri, "Finding the k shortest simple paths: A new algorithm and its implementation," *ACM Transactions on Algorithms*, vol. 3, no. 4, pp. 45–es, 2007.

[47] Z. Gotthilf and M. Lewenstein, "Improved algorithms for the k simple shortest paths and the replacement paths problems," *Information Processing Letters*, vol. 109, no. 7, pp. 352–355, 2009.

[48] D. Ouyang, L. Yuan, L. Qin, L. Chang, Y. Zhang, and X. Lin, "Efficient shortest path index maintenance on dynamic road networks with theoretical guarantees," *Proceedings of the VLDB Endowment*, vol. 13, no. 5, pp. 602–615, 2020.

[49] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies," *Transportation Science*, vol. 46, no. 3, pp. 388–404, 2012.

[50] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Customizable route planning in road networks," *Transportation Science*, vol. 51, no. 2, pp. 566–591, 2017.

[51] D. Zhang, D. Yang, Y. Wang, K.-L. Tan, J. Cao, and H. T. Shen, "Distributed shortest path query processing on dynamic road networks," *The VLDB Journal*, vol. 26, no. 3, pp. 399–419, 2017.

[52] M. S. Hassan, W. G. Aref, and A. M. Aly, "Graph indexing for shortest-path finding over dynamic sub-graphs," in *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pp. 1183–1197, ACM, 2016.