Efficiently Computing Top-K Shortest Path Join

Lijun Chang[†], Xuemin Lin[†#], Lu Qin[¶], Jeffrey Xu Yu[‡], Jian Pei[§]

[†] *University of New South Wales, Australia, {ljchang,lxue}@cse.unsw.edu.au*
[#] *East China Normal University, China*
[¶] *University of Technology, Sydney, Australia, lu.qin@uts.edu.au*
[‡] *The Chinese University of Hong Kong, Hong Kong, China, yu@se.cuhk.edu.hk*
[§] *Simon Fraser University, Canada, jpei@cs.sfu.ca*

ABSTRACT

Driven by many applications, in this paper we study the problem of computing the top-k shortest paths from one set of target nodes to another set of target nodes in a graph, namely the top-k shortest path join (KPJ) between two sets of target nodes. While KPJ is an extension of the problem of computing the top-k shortest paths (KSP) between two target nodes, the existing technique by converting KPJ to KSP has several deficiencies in conducting the computation. To resolve these, we propose to use the best-first paradigm to recursively divide search subspaces into smaller subspaces, and to compute the shortest path in each of the subspaces in a prioritized order based on their lower bounds. Consequently, we only compute shortest paths in subspaces whose lower bounds are larger than the length of the current k-th shortest path. To improve the efficiency, we further propose an iteratively bounding approach to tightening lower bounds of subspaces. Moreover, we propose two index structures which can be used to reduce the exploration area of a graph dramatically; these greatly speed up the computation. Extensive performance studies based on real road networks demonstrate the scalability of our approaches and that our approaches outperform the existing approach by several orders of magnitude. Furthermore, our approaches can be immediately used to compute KSP. Our experiment also demonstrates that our techniques outperform the state-of-the-art algorithm for KSP by several orders of magnitude.

1. INTRODUCTION

Data are often modeled as graphs in many real applications such as social networks, information networks, gene networks, protein-protein interaction networks, and road networks. With the proliferation of graph data, significant research efforts have been made towards analyzing large graph data. These include the problem of computing the top-k shortest paths between two target nodes in a graph, namely the k shortest path (KSP) query.

KSP is a fundamental graph problem with many applications. In general, KSP is used in the applications that besides lengths, other constraints against the paths could not be precisely defined [12, 25]. For example, computing KSP between two sensitive accounts in a large social network enables end-users to identify all accounts involved in the top-k shortest paths [14]. In gene networks, the lengths of top-k shortest paths may be used to define the importance of a target gene to a source gene [26]. Other applications of KSP include multiple object tracking in pattern recognition [3], hypothesis generation in computational linguistics, and trip planning against road networks. Thus, KSP has been extensively studied [8, 9, 14, 15, 18, 24, 28].

The problem of computing the top-k shortest paths between two "conceptual" target nodes (instead of between two physical nodes) in a graph, called *the top-k shortest path join* (KPJ), is recently investigated in [15]. A conceptual node is a set of physical nodes in the graph, which can be identified by categories, concepts, and keywords in the above applications. While a KSP query is a special case of a KPJ query where each of the two conceptual target nodes only contains one physical node, KPJ can support more general application scenarios than KSP since a target node is allowed to be a set of physical nodes. For example, in a social network, the KPJ query can be used to detect user accounts involved in the top-k shortest paths between two criminal gangs to identify other "most suspicious" user accounts; the KPJ query can also be used in route planning where the destination is any one from a group of nodes (e.g., "IKEA"). In this paper, we study KPJ.

Motivations and Challenges. KPJ query shares similarity with but is different from the well-studied KSP query. To process a KPJ query, [15] reduces it to a KSP query by introducing a virtual target node for each conceptual target node and connecting every physical node in the conceptual node to it. Then, [15] proposes to use the state-of-the-art algorithm for KSP developed in [15]. Since the technique for solving KSP in [15] is based on the deviation paradigm [9, 28], applying KSP to solve KPJ, as proposed in [15], has the following two deficiencies. 1) Firstly, the deviation based techniques for KSP need to compute $O(k \cdot n)$ "candidate paths". The candidate paths are computed by iteratively extending all "prefixes" of the obtained l-th ($l < k$) shortest path, where n is the number of nodes in a graph, and each candidate path is computed by running an expensive shortest path algorithm; this is time-consuming. 2) Secondly, edges that are added to connect to a virtual target node for processing KPJ by using the KSP techniques depend on queries. This makes the existing index structures [7, 10] for computing shortest paths inapplicable. Thus, the candidate paths are computed by traversing the graph exhaustively; this is very costly.

Our Approaches. For presentation simplicity, in this paper we present our techniques against the simplified case, where one conceptual target node consists of one physical node only - called *source* node s, and the other conceptual target node may consist of multiple physical nodes - called *destination* nodes; then we extend our techniques to the general case where source nodes may also be multiple. Let $\mathcal{P}$ denote the set of all *simple* paths (i.e., paths without loops) from s to any of the destination nodes as the entire search space. Clearly, the result of KPJ is the set of k paths in $\mathcal{P}$ with the shortest lengths.

We adopt the best-first paradigm to recursively divide $\mathcal{P}$ into smaller subspaces, and then compute shortest paths for the generated subspaces in a prioritized order based on their lower bounds, where lower bound of a subspace is the lower bound of the length of all paths in the subspace. The top-k shortest paths may be iteratively obtained over the subspaces whose lower bounds are smaller than the length of the current k-th shortest path, while other subspaces can be safely pruned without the time-consuming shortest path computation.

We further propose to iteratively "guess" and tighten the lower bound τ of a subspace. Initially, we assign the value of τ as the length of the (1st) shortest path. Then, we always choose the subspace with the smallest τ to test whether the shortest path in it has length larger than $\alpha \cdot \tau$ (for an $\alpha > 1$). If the shortest path in the subspace can be determined larger than $\alpha \cdot \tau$, then we enlarge τ into $\alpha \cdot \tau$ for the subspace; otherwise the shortest path in the subspace is computed. Moreover, we propose two online-built index structures, $\mathsf{SPT_P}$ and $\mathsf{SPT_I}$, to significantly reduce the exploration area of a graph in the lower bound testing as briefly described above.

Contributions. Our primary contributions are summarized as follows.

- We propose a framework based on the best-first paradigm for processing KPJ queries which significantly reduces the number of shortest path computations.
- We propose an iteratively bounding approach to guessing and tightening the lower bounds, as well as two online-built index structures to speed-up the lower bound testing.
- We conduct extensive performance studies and demonstrate the scalability of our approaches which outperform the baseline approach [15] by several orders of magnitude.
- Moreover, our approaches can be immediately used to process KSP queries, and our experiments also demonstrate that our techniques outperform the state-of-the-art algorithm for KSP query by several orders of magnitude.

Organization. The rest of this paper is organized as follows. A brief overview of related work is given below. We give the preliminaries and problem statement in Section 2. The existing KSP-based approach is given in Section 3, in which we also discuss its deficiencies. We present the best-first paradigm in Section 4, in which we also implement a best-first approach. Under this paradigm, in Section 5 we propose an iteratively bounding approach and two online-built indexes for efficiently processing KPJ queries. In Section 6, we extend our techniques to cover other applications including the case that the source node has multiple physical nodes. We conducted extensive experimental studies and report our findings in Section 7, and we conclude this paper in Section 8.

Related Work. Given two nodes s and t in a graph G, the problem of computing the top-k shortest paths from s to t is a long-studied problem, which can be classified into two categories, 1) top-k simple shortest paths, and 2) top-k general shortest paths.

1) Top-k Simple Shortest Path. The existing algorithms for computing top-k simple shortest paths are based on the deviation paradigm proposed by Yen [9, 28], which has a time complexity of $O(k \cdot n \cdot (m + n \log n))$, where m is the number of edges in G and $O(m + n \log n)$ is the time complexity of computing single source shortest paths. Techniques to improve its efficiency in practice have been studied in [8, 14, 15, 18, 24], which shall be discussed in Section 3. We discuss using these techniques to process KPJ queries in Section 3.

2) Top-k General Shortest Path. Finding top-k general shortest paths is studied in [2, 12, 19], where cycles in paths are allowed. Since not enforcing paths to be simple, the top-k general shortest path problem is generally easier than its counterpart. Eppstein's algorithm [12] has the best time complexity, $O(m+n \log n+k)$, which is achieved by precomputing a shortest path tree rooted at the destination node and building a sophisticated data structure. Recently, the authors in [1] propose a heuristic search algorithm that has the same time complexity as [12]. However, due to different problem natures, these techniques are inapplicable to finding top-k simple shortest paths.

Finding Top-k Objects by Keywords. Finding k objects closest to a query location and containing user-given keywords has been studied in [13, 20, 22, 23, 29]. Ranking spatial objects by the combination of distance and relevance score is also studied in [4, 5, 27]. Distance oracles for node-label queries in a labeled graph are studied in [6, 17]; that is, given a query which contains a node and a label, it returns approximately the closest node to the query node that contains the query label. Nevertheless, the above queries are inherently different from KPJ, and their techniques cannot be applied to process KPJ queries.

2. PRELIMINARY

In this paper, we focus on a *weighted* and *directed graph* $G = (V, E, \omega)$, where V and E represent the set of nodes and the set of edges of G, respectively, and ω is a function assigning a weight to each edge in E. In G, nodes belong to categories, and each category represents a conceptual node consisting of all nodes belonging to that category. The number of nodes and the number of edges of G are denoted by $n = |V|$ and $m = |E|$, respectively.

A *path* P in G is a sequence of nodes $(v_1, \ldots, v_l)$ such that $(v_i, v_{i+1}) \in E, \forall 1 \leq i < l$, and we say that P consists of edges $(v_i, v_{i+1}), \forall 1 \leq i < l$. Here, v_1 and v_l are called the *source* node and *destination* node of P, respectively. P is a *simple* path if and only if all nodes in P are distinct (i.e., $v_i \neq v_j, \forall i \neq j$). A *prefix* of P is a subpath of P starting from the source node of P. The *length* of a path is defined as the total weight of its constituent edges; that is $\omega(P) = \sum_{(v_i, v_{i+1}) \in P} \omega(v_i, v_{i+1})$. The shortest distance from v_1 to v_2 is the shortest length among all paths from v_1 to v_2, denoted $\delta(v_1, v_2)$.

Definition 2.1: Given a category T, a path P is said to be a ***path to category*** T if its destination node is in V_T, where V_T is the set of nodes belonging to category T. □

Problem Statement: Given a graph G, we study the *top-k shortest path join* (KPJ) query, which aims at finding the top-k shortest simple paths $P_1, \ldots, P_k$ from a source node s to a category T (i.e., to any node in V_T).

Formally, a KPJ query is given as $Q = \{s, T, k\}$, where s is a source node in G, T represents a destination category, and k specifies the number of paths to find. It is to find k simple paths $P_1, \ldots, P_k$ such that: 1) each P_i is a path from s to category T; 2) $\omega(P_i) \leq \omega(P_{i+1}), \forall 1 \leq i < k$; 3) $\omega(P_k) \leq \omega(P)$ for any other path P from s to category T.

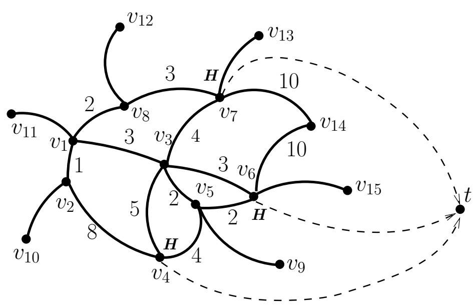


Figure 1: An example graph

Example 2.1: Fig. 1 illustrates a graph G, where $V = \{v_1, \ldots, v_{15}\}$ and nodes v_4, v_6, v_7 belong to category *"H"* (i.e., hotel). Here, edges are bidirectional, and weights are shown besides them with a default value 1. Consider a KPJ query $Q = \{v_1, \text{"}H\text{"}, 1\}$, which is to find the top-1 shortest path from v_1 to category "H". The top-1 path is $P_1 = (v_1, v_8, v_7)$ with $\omega(P_1) = 2 + 3 = 5$. □

For a KPJ query, V_T is the set of destination nodes. In the following, we assume that an inverted index [21] is offline built on the categories of nodes such that V_T can be efficiently retrieved online, and assume that a path is a simple path.

3. THE EXISTING KSP-BASED APPROACH

Reducing KPJ Query to KSP Query. The most related problem to KPJ is k shortest path (KSP) query defined below.

Definition 3.1:[28] Given a graph G, a KSP query $Q' = \{s, t, k\}$ is to find k simple paths $P_1, \ldots, P_k$ from s to t such that, 1) $\omega(P_i) \leq \omega(P_{i+1}), \forall 1 \leq i < k$, and 2) $\omega(P_k) \leq \omega(P)$ for any other path P from s to t. □

KSP query is a special case of KPJ query where V_T contains only one node. In other words, KSP query considers a single destination node while KPJ query considers multiple destination nodes. To process a KPJ query $Q = \{s, T, k\}$, [15] reduces it to a KSP query by adding a virtual destination node t to G and adding a directed edge from each node in V_T to t with a weight 0. Then, the result of Q on G is the same as the result of the KSP query $Q' = \{s, t, k\}$ on G_Q. Reconsider the KPJ query in Example 2.1, the modified graph G_Q is also shown in Fig. 1 with the virtual node t and the additional edges (dashed lines).

Deviation Algorithm (DA) for KSP Queries. The existing algorithms for KSP queries are based on the deviation paradigm [9, 28], denoted DA. It maintains a set C of candidate paths which include the next shortest path from s to t, and chooses k shortest paths from C one by one in a non-decreasing length order by incrementally updating C.

Pseudo-tree. The set of already chosen paths are encoded using a compact trie-like structure [21], called pseudo-tree. It is named because the same node may appear at several places in the tree; thus, we refer to nodes in a pseudo-tree as vertices to distinguish them from nodes in a graph. Let PT_i denote the pseudo-tree constructed for paths $P_1, \ldots, P_i$, and PT_0 consists of a single vertex s. PT_{i+1} is constructed by inserting P_{i+1} into PT_i by sharing the longest prefix; let d be the last vertex of the shared prefix, it is called the deviation vertex of P_{i+1} from PT_i. For example, Fig. 2 shows PT_1, PT_2, and PT_3, where PT_3 is constructed by inserting path (v_1, v_3, v_7, t) into PT_2, and v_3 is the deviation vertex.

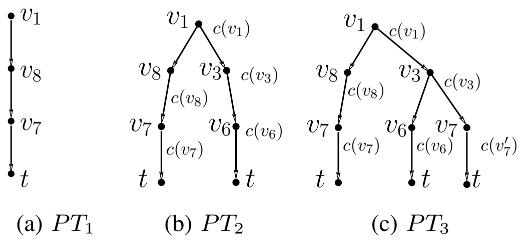


Figure 2: pseudo-trees

Candidate Path. Given a pseudo-tree PT_i, the DA algorithm maintains a set C_i of candidate paths, one corresponding to each vertex u in PT_i, denoted $c(u)$, which is *the shortest one among all paths from s to t that takes the path from s to u in PT_i as prefix and contains none of the outgoing edges of u in PT_i.* For example, in Fig. 2(c), $c(v_3)$ is the shortest one among all paths from s to t that take edge (v_1, v_3) as prefix and contain neither (v_3, v_6) nor (v_3, v_7); thus $c(v_3) = (v_1, v_3, v_5, v_6, t)$, and $C_3 = \{c(v_1), c(v_8), c(v_7), c(v_3), c(v_6), c(v_7')\}$.

Lemma 3.1: *[28]. Given a pseudo-tree PT_i and the corresponding C_i of candidate paths, the $(i+1)$-th shortest path from s to t is the path in C_i with shortest length.* □

Following from Lemma 3.1, the pseudocode of processing a KPJ query using DA is shown in Alg. 1, which is self-explanatory. The ingredient of Alg. 1 is to incrementally maintain PT_i and C_i after choosing each of the top-k paths.

Algorithm 1: DA($G_Q, Q' = \{s, t, k\}$)

```
1 Initialize $PT_0$ to contain a single vertex $s$;
2 Compute the shortest path $c(s)$ from $s$ to $t$, and $C_0 = \{c(s)\}$;
3 for each $i \leftarrow 1$ to $k$ do
4     $P_i \leftarrow$ the path in $C_{i-1}$ with the shortest length;
5     Construct $PT_i$ by inserting $P_i$ into $PT_{i-1}$, and let $d$ be the
      deviation vertex;
6     Construct $C_i$ by removing $P_i$ from $C_{i-1}$, computing the
      candidate paths corresponding to vertices in $P_i$ from $d$ to $t$, and
      inserting them into $C_{i-1}$;
7 return the $k$ paths $P_1, \ldots, P_k$;
```

Example 3.1: Fig. 2 demonstrates a running example for a KPJ query $Q = \{v_1, \text{"}H\text{"}, 3\}$ on the graph in Fig. 1. We first transform the graph G into G_Q, and reduce Q to a KSP query $Q' = \{v_1, t, 3\}$. The shortest path is $P_1 = (v_1, v_8, v_7, t)$ with length 5. After inserting P_1 into PT_0, the resulting PT_1 is shown in Fig. 2(a), where $C_1 = \{c(v_1), c(v_8), c(v_7)\}$. The 2nd shortest path is computed as $P_2 = c(v_1) = (v_1, v_3, v_6, t)$ which has the shortest length in C_1, and PT_2 is shown in Fig. 2(b). Then, candidate paths for v_1, v_3, v_6 are updated or computed, and $C_2 = \{c(v_8), c(v_7), c(v_1), c(v_3), c(v_6)\}$. The 3rd shortest path is $P_3 = c(v_3) = (v_1, v_3, v_7, t)$ with length 7. □

DA-SPT: Optimizations. The most time-consuming part of DA (Alg. 1) is Line 6, which needs to compute $O(k \cdot n)$ candidate paths in total. To efficiently compute a candidate path, several optimization techniques have been recently proposed [14, 24]. Pascoal [24] observes that, when computing $c(u)$ for u in a pseudo-tree PT, if the path formed by concatenating, 1) the path from s to u in PT, 2) an edge (u, v) in G_Q, and 3) the shortest path from v to t in G_Q, is simple, then it is $c(u)$. By preprocessing G_Q to generate a shortest path tree (SPT) storing shortest paths from all nodes to t, the path described above, if exists, can be found in constant time; otherwise, a shortest path algorithm is run to compute $c(u)$. Gao et al. [14, 15] improve Pascoal's approach by iteratively testing the above property during running Dijkstra's algorithm [11], and obtaining $c(u)$ once a simple path is found; this is known as the state-of-the-art approach, denoted DA-SPT, since a full SPT is built online.

Deficiencies of DA and DA-SPT. Both DA and DA-SPT are inefficient for processing KPJ queries due to the following three reasons. 1) Firstly, both need to compute $O(k \cdot n)$ candidate paths which are computed by iteratively extending all prefixes of the obtained l-th ($l < k$) shortest path; this is time-consuming. 2) Secondly, for processing a KPJ query using KSP techniques, the edges added to connect nodes in V_T to the virtual destination node depend on queries; this makes the existing index structures [7, 10] for efficiently computing shortest paths inapplicable. Thus, the candidate paths are computed by traversing the graph exhaustively, which is very costly. 3) Thirdly, although DA-SPT, compared to DA, can compute candidate paths more efficiently, it is time-consuming to construct the full SPT, which may be the dominating cost especially when the k shortest paths are short.

4. A BEST-FIRST APPROACH

In this section, to remedy the deficiencies of using the existing KSP techniques to process KPJ queries, we adopt a best-first paradigm which significantly reduces the number of shortest path computations thus enables fast query processing. In the following, we first discuss the paradigm, and then give an implementation of a best-first approach.

4.1 Best-First Paradigm

Given a KPJ query $Q = \{s, T, k\}$, let $\mathcal{P}_{s,T}(G)$ denote the set of all paths in G from s to category T (i.e., to any node in V_T). When the context is clear, $\mathcal{P}_{s,T}(G)$ is abbreviated to $\mathcal{P}$. Then, the query Q is to find the k paths in $\mathcal{P}$ with shortest lengths. Note that the size of $\mathcal{P}$ can be exponential to n.

Search Space and Subspace. The general idea is that we regard $\mathcal{P}$ as the entire search space $\mathcal{S}_0$. Then, the k paths in $\mathcal{P}$ with shortest lengths can be found by recursively dividing a subspace (initially $\mathcal{S}_0$) into smaller subspaces and computing the shortest path in each newly obtained subspace.

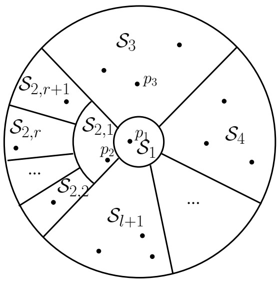


Figure 3: Overview of search space division

Before diving into the details, we first explain the main idea which is illustrated in Fig. 3. We conceptualize each path in $\mathcal{P}$ as a point, whose distance to the center (i.e., the origin) indicates the length of the path. Thus, the k paths with shortest lengths correspond to the k points closest to the center which can be computed as follows. First, we compute the closest point $P_1 = (v_1, \ldots, v_l)$ in the entire search space $\mathcal{S}_0 = \mathcal{P}$. Second, we divide $\mathcal{S}_0$ into $l+1$ subspaces, $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_l, \mathcal{S}_{l+1}$. Here, $\mathcal{S}_1$ consists of only P_1 and is excluded from further considerations. Each of the remaining subspaces, $\mathcal{S}_2, \ldots, \mathcal{S}_{l+1}$, represents the set of paths of $\mathcal{P}$ that share exactly the prefix of P_1 from v_1 to v_{i-1}; consequently, $\mathcal{S}_i \neq \mathcal{S}_j, \forall i \neq j$, and $\bigcup_{i=1}^{l+1} \mathcal{S}_i = \mathcal{S}_0$. Third, we compute the closest point in each of the l subspace, $\mathcal{S}_2, \ldots, \mathcal{S}_{l+1}$, and the one that is closest to the center among the l closest points represents the 2nd shortest path. Let it be $P_2 = (v'_1, \ldots, v'_r)$, and assume it is in $\mathcal{S}_2$. Fourth, we further divide $\mathcal{S}_2$ into $r+1$ subspaces, $\mathcal{S}_{2,1}, \mathcal{S}_{2,2}, \ldots, \mathcal{S}_{2,r}, \mathcal{S}_{2,r+1}$, and compute the closest point in each of them, where $\mathcal{S}_{2,1}$ consists of only P_2 and is excluded from further considerations. Thus, the point that is closest to the center among closest points in all subspaces $\mathcal{S}_3, \cdots, \mathcal{S}_{l+1}, \mathcal{S}_{2,2}, \ldots, \mathcal{S}_{2,r+1}$ represents the 3rd shortest path. We can repeat this process until k shortest paths are computed.

Subspace Division. We formally define a subspace below.

Definition 4.1: A ***subspace*** $\mathcal{S}$ is represented by a tuple $\langle P_{s,u}, X_u \rangle$, where $P_{s,u}$ is a path from s to u and X_u is a subset of the outgoing edges of u. It consists of all paths in $\mathcal{P}$ that *take* $P_{s,u}$ as prefix and *exclude* all edges of X_u. □

The entire search space $\mathcal{S}_0(=\mathcal{P})$ is represented by $\langle P_{s,s} = (s), X_s = \emptyset \rangle$. Assume the shortest path in subspace $\langle P_{s,u}, X_u \rangle$ is P, then after choosing P as one of the k shortest paths, the subspace is divided into $l+1$ subspaces, where l is the number of nodes in the subpath of P from u to the destination node. The $l+1$ subspaces consist of a subspace containing only P, a subspace corresponding to node u (i.e., subspace $\langle P_{s,u}, X_u \cup \{(u,w)\}\rangle$), and one subspace corresponding to each node v in the subpath of P from u (exclusive) to the destination node (i.e., subspace $\langle P_{s,v}, \{(v,w')\}\rangle$), where $P_{s,v}$ is the prefix of P to v, and (u,w) and (v,w') are edges in P. It is important to note that the $l+1$ subspaces are disjoint, and their union is the original subspace $\langle P_{s,u}, X_u \rangle$ from which they are divided.

Example 4.1: Consider a KPJ query $Q = \{v_1, \text{“}H\text{”}, 2\}$ on the graph in Fig. 1. Initially, $\mathcal{S}_0 = \langle (v_1), \emptyset \rangle$ in which the shortest path is $P_1 = (v_1, v_8, v_7)$. Then, $\mathcal{S}_0$ is divided into four subspaces, $\mathcal{S}_1 = \{P\}$, $\mathcal{S}_2 = \langle (v_1), \{(v_1, v_8)\}\rangle$, $\mathcal{S}_3 = \langle (v_1, v_8), \{(v_8, v_7)\}\rangle$, and $\mathcal{S}_4 = \langle (v_1, v_8, v_7), \emptyset \rangle$, where $\mathcal{S}_1$ is the subspace containing only P_1. The 2nd shortest path is the one with shortest lengths among shortest paths in $\mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4$. □

Paradigm. It is easy to verify that there is a one-to-one correspondence between candidate paths defined in Section 3 and subspaces defined above. In the deviation paradigm, subspaces are implicitly maintained by storing candidate paths based on the fact that *each candidate path is the shortest path in a subspace*. Considering that shortest paths are expensive to compute, we remedy the deficiency of deviation paradigm by computing lower bounds of subspaces and pruning subspaces based on their lower bounds.

Definition 4.2: For a subspace $\mathcal{S} = \langle P_{s,u}, X_u \rangle$, we define the ***lower bound of a subspace***, denoted $\mathsf{lb}(\mathcal{S})$ (or $\mathsf{lb}(P_{s,u}, X_u)$), as the lower bound of lengths of all paths in $\mathcal{S}$, and denote the ***shortest path in a subspace*** by $\mathsf{sp}(\mathcal{S})$ (or $\mathsf{sp}(P_{s,u}, X_u)$). □

Based on lower bounds of subspaces, the best-first paradigm is shown in Alg. 2. Instead of directly computing the shortest path for each newly obtained subspace, we compute its lower bound first. All obtained subspaces and their lower bounds are maintained in a minimum priority queue $\mathcal{Q}$. Each entry of $\mathcal{Q}$ is a triple $\langle \mathcal{S}, \mathsf{lb}(\mathcal{S}), P \rangle$, where $\mathcal{S}$ and $\mathsf{lb}(\mathcal{S})$ are a subspace and its lower bound, respectively, and P is either $\emptyset$ or the shortest path in $\mathcal{S}$. Subspaces in $\mathcal{Q}$ are ranked by their lower bounds. Initially, $\mathcal{Q}$ contains a single entry representing the entire search space $\mathcal{S}_0$ (Line 1). Then, we iteratively remove the subspace with smallest lower bound from $\mathcal{Q}$, denoted $\langle \mathcal{S}, \mathsf{lb}(\mathcal{S}), P \rangle$ (Line 4): if $P \neq \emptyset$, then P is the

Algorithm 2: BestFirst$(G, Q = \{s, T, k\})$

```
1  Initialize a minimum priority queue $\mathcal{Q}$ to contain a single entry
   $\langle \mathcal{S}_0 = \langle (s), \emptyset \rangle, \mathsf{lb}(\mathcal{S}_0), \emptyset \rangle$;
2  $i \leftarrow 1$;
3  while $i \le k$ do
4      $\langle \mathcal{S} = \langle P_{s,u}, X_u \rangle, \mathsf{lb}(\mathcal{S}), P \rangle \leftarrow$ remove the top entry from $\mathcal{Q}$;
5      if $P \neq \emptyset$ then
6          $P_i \leftarrow P$; $i \leftarrow i + 1$;
7          for each node $v$ in the subpath of $P$ from $u$ to the destination node do
8              Create a subspace $\mathcal{S}' = \langle P_{s,v}, X_v \rangle$;
9              $\mathsf{lb}(\mathcal{S}') \leftarrow \max\{\mathsf{CompLB}(P_{s,v}, X_v), \omega(P)\}$;
10             Put $\langle \mathcal{S}', \mathsf{lb}(\mathcal{S}'), \emptyset \rangle$ into $\mathcal{Q}$;
11     else
12         $\mathsf{sp}(\mathcal{S}) \leftarrow \mathsf{CompSP}(P_{s,u}, X_u)$;
13         if $\mathsf{sp}(\mathcal{S}) \neq \emptyset$ then Put $\langle \mathcal{S}, \omega(\mathsf{sp}(\mathcal{S})), \mathsf{sp}(\mathcal{S}) \rangle$ into $\mathcal{Q}$;
14 return the $k$ paths $P_1, \ldots, P_k$;
```

next shortest path to be output (Line 6), and we divide $\mathcal{S}$ by P and put those newly obtained subspaces into $\mathcal{Q}$ (Lines 7-10); otherwise, we compute the shortest path in $\mathcal{S}$ and put $\mathcal{S}$ into $\mathcal{Q}$ again with the computed shortest path (Lines 12-13). We will present an implementation of computing lower bound of a subspace and shortest path in a subspace in the next subsection.

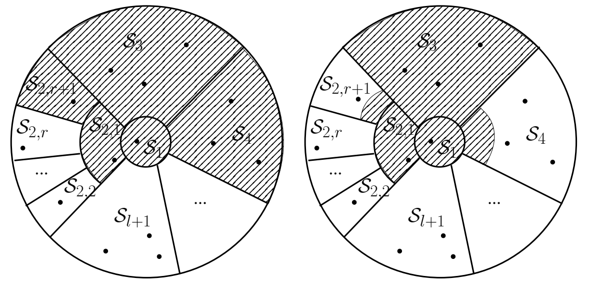
Lemma 4.1: *The set of shortest paths computed in Alg. 2 is a subset of that computed in Alg. 1; thus, the number of shortest path computations in Alg. 2 is not larger than that in Alg. 1. Assume that computing lower bound takes less time than computing shortest path for a subspace, then the time complexity of Alg. 2 is not larger than that of Alg. 1.* □

Proof Sketch: We prove the first part of Lemma 4.1 by proving that there is a one-to-one correspondence between subspaces inserted into $\mathcal{Q}$ in Alg. 2 and candidate paths computed in Alg. 1. This can be proved by induction. For $k = 1$, this is true, since there is only one subspace and one candidate path in Alg. 2 and Alg. 2, respectively. Now, we assume that this holds for general $k \ge 1$, then we prove that it also holds for $(k+1)$. Since the k-th shortest path P_k corresponds to subspace $\mathcal{S}$ in Alg. 2, to obtain the $(k+1)$-th shortest path, we generate $O(n)$ new candidate paths from P_k in Alg. 1 and $O(n)$ new subspaces from $\mathcal{S}$ in Alg. 2; moreover, there is a one-to-one correspondence between these newly generated subspaces and newly generated candidate paths. Thus, there is a one-to-one correspondence between subspaces inserted into $\mathcal{Q}$ in Alg. 2 and candidate paths computed in Alg. 1. Considering that we compute shortest paths only for a subset of the subspaces inserted into $\mathcal{Q}$, the first part of the lemma holds.

Moreover, if we set all lower bounds computed at Line 9 of Alg. 2 to be 0, then the number of shortest path computations in Alg. 2 is the same as that in Alg. 1.

Second, in Alg. 2, any subspace is inserted into $\mathcal{Q}$ at most twice: once with computed lower bound (i.e., $P = \emptyset$), and once with computed shortest path. Thus, given that computing lower bound takes less time than computing shortest path for a subspace, the time complexity of Alg. 2 is not larger than that of Alg. 1. □

Following the proof of Lemma 4.1, we can also see that the maximum size of $\mathcal{Q}$ in Alg. 2 is $O(k \cdot n)$; moreover, given any algorithm computing a lower bound of a subspace, Alg. 2 correctly processes a KPJ query. Let P_k be the k-th shortest path for a KPJ query. Obviously, Alg. 2 does not compute shortest paths in subspaces whose lower bounds are larger than $\omega(P_k)$. Note that, in contrast, Alg. 1 needs to compute shortest paths in all subspaces in $\mathcal{Q}$. Conceptually, Fig. 4(a) shows by shadow the subspaces in which the shortest paths are computed: Alg. 2 computes only 5 shortest paths instead of $l + r + 2$ which is done by Alg. 1.



(a) Best First (b) Iteratively Bounding

Figure 4: Instances of shortest path computations

4.2 An Implementation of BestFirst

In the following, we present efficient techniques for computing a lower bound of a subspace (CompLB at Line 9 of Alg. 2) and for computing the shortest path in a subspace (CompSP at Line 12 of Alg. 2), denote the approach as BestFirst.

Algorithm 3: CompLB$(P_{s,u}, X_u)$

```
1  $lb \leftarrow +\infty$;
2  for each outgoing edge $(u, v)$ of $u$ do
3      if $v \notin P_{s,u}$ and $(u, v) \notin X_u$ then
4          Compute $\mathsf{lb}(v, V_T)$;
5          $lb \leftarrow \min\{lb, \omega(P_{s,u}) + \omega(u, v) + \mathsf{lb}(v, V_T)\}$;
6  return $lb$;
```

Computing Lower Bound of a Subspace. Given a subspace $\mathcal{S} = \langle P_{s,u}, X_u \rangle$, the set of paths in it corresponds to the set of paths from s to any node in V_T in a subgraph G' of G obtained as follows. We first remove all edges of X_u from G, and then for each node $v(\neq u)$ in $P_{s,u}$, we remove from G all outgoing edges of v except the one that is in $P_{s,u}$. Thus, for any two nodes u and v, the shortest distance from u to v in G' is lower bounded by that in G. Consequently, a naive lower bound of $\mathcal{S}$ is $\omega(P_{s,u}) + \mathsf{lb}(u, V_T)$, where $\mathsf{lb}(u, V_T)$ is the lower bound of shortest distance from u to any node in V_T, whose computation shall be discussed shortly. However, this is loose considering that many outgoing edges of u (i.e., X_u) are removed from G. Therefore, to estimate the lower bound of $\mathcal{S}$ (i.e., the shortest distance from s to any node in V_T in G') more accurately, we consider all valid outgoing edges (u, v) of u (i.e., $v \notin P_{s,u}$ and $(u, v) \notin X_u$), and choose the minimum estimation. The pseudocode is shown in Alg. 3 which is self-explanatory, and its correctness immediately follows from the above discussions.

Example 4.2: Continuing Example 4.1. After dividing $\mathcal{S}_0$ into $\mathcal{S}_1, \cdots, \mathcal{S}_4$, lower bounds of $\mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4$ are computed. Here, we illustrate how to compute $\mathsf{lb}(\mathcal{S}_2) = \mathsf{lb}((v_1), \{(v_1, v_8)\})$. v_1 has three valid outgoing edges, (v_1, v_2), (v_1, v_3), and (v_1, v_{11}), that can be considered for lower bound estimation. Thus, $\mathsf{lb}(\mathcal{S}_2)$ is computed as $\min\{\omega(v_1, v_2) + \mathsf{lb}(v_2, V_T), \omega(v_1, v_3) + \mathsf{lb}(v_3, V_T), \omega(v_1, v_{11}) + \mathsf{lb}(v_{11}, V_T)\}$. □

Computing $\mathsf{lb}(u, V_T)$. We propose a landmark-based approach [16] to estimating $\mathsf{lb}(u, V_T)$ in the following. Note that, the computation of $\mathsf{lb}(u, V_T)$ has not been studied in the literature.

A *landmark* is a subset of nodes, $L \subseteq V$. With L, the lower bound $\mathsf{lb}(u,v)$ of shortest distance from u to v is estimated as, $\mathsf{lb}(u,v) \doteq \max_{w \in L}\{\delta(w,v) - \delta(w,u)\}$, [1] where $\delta(w,v)$ and $\delta(w,u)$ are the shortest distance from w to v and to u, respectively, and are precomputed. This estimation is based on the fact that $\delta(w,u) + \delta(u,v) \geq \delta(w,v)$. Then, $\mathsf{lb}(u,V_T)$ can be estimated as,

$$\begin{aligned} \mathsf{lb}(u,V_T) &\doteq \min_{v \in V_T}\{\mathsf{lb}(u,v)\} \\ &= \min_{v \in V_T}\max_{w \in L}\{\delta(w,v) - \delta(w,u)\} \end{aligned} \quad (1)$$

However, the computation time is $O(|L| \cdot |V_T|)$ which is too costly especially when V_T is large. Therefore, motivated by the transformed graph G_Q in Section 3, we propose a new lower bound as follows,

$$\begin{aligned} \mathsf{lb}(u,V_T) &\doteq \max_{w \in L}\min_{v \in V_T}\{\delta(w,v) - \delta(w,u)\} \\ &= \max_{w \in L}\{\min\{\delta(w,v) \mid v \in V_T\} - \delta(w,u)\} \end{aligned} \quad (2)$$

The intuition is that, $\min\{\delta(w,v) \mid v \in V_T\}$ is the shortest distance from w to t in G_Q (i.e., $\delta(w,t)$); thus, Eq. (2) estimates $\mathsf{lb}(u,t)$. Consequently, $\mathsf{lb}(u,V_T)$ can be computed in $O(|L|)$ time by precomputing $\delta(w,t)$ for all $w \in L$ prior to any lower bound estimations. In what follows, we use Eq. (2) to compute $\mathsf{lb}(u,V_T)$.

Remarks & Time Complexity. Note that, the landmark index L is constructed offline in $O(|L|(m + n\log n))$ time where $O(m + n\log n)$ is the time complexity of a shortest path algorithm, while its space complexity is $O(|L| \cdot n)$. At the initialization phase of query processing, we compute $\delta(w,t)$ which is query dependent. The time complexity for computing $\delta(w,t)$ for all $w \in L$ is $O(|L| \cdot |V_T|)$; note that this is only computed once for each query.

Therefore, the time complexity of lower bound computation (i.e., Alg. 3) is $O(d(u)|L|)$ where $d(u)$ is the degree of u in G, since we traverse at most $d(u)$ edges at Line 2 while computing each $\mathsf{lb}(v,V_T)$ takes $O(|L|)$ time.

Computing Shortest Path in a Subspace. We use A* search [16] to compute $\mathsf{sp}(P_{s,u}, X_u)$, denoted CompSP. In A* search, we consider only the valid edges (same as Line 3 of Alg. 3), and use Eq. (2) to estimate the shortest distance to destination. We omit the pseudocode.

Example 4.3: Continuing Example 4.2, after dividing $\mathcal{S}_0$, $\mathcal{Q}$ contains three subspaces, $\mathcal{S}_2, \mathcal{S}_3, \mathcal{S}_4$, together with their lower bounds. Assume the lower bounds are $\mathsf{lb}(\mathcal{S}_2) = 6$, $\mathsf{lb}(\mathcal{S}_3) = 11$, and $\mathsf{lb}(\mathcal{S}_4) = 12$. $\mathcal{S}_2$ has the smallest lower bound, and is removed from $\mathcal{Q}$. Then, $\mathsf{sp}(\mathcal{S}_2)$ is computed, which is the 2nd shortest path P_2, since its length is smaller than lower bounds of subspaces in $\mathcal{Q}$. Here, we compute the 2nd shortest path without computing shortest paths in subspaces $\mathcal{S}_3$ and $\mathcal{S}_4$. □

5. AN ITERATIVELY BOUNDING APPROACH

In this section, following the best-first paradigm in Section 4, we propose a new iteratively bounding approach to iteratively "guessing" and tightening lower bounds for subspaces in Section 5.1. Moreover, we propose two online-built indexes, in Section 5.2 and Section 5.3, respectively, based on which we can reduce the exploration area of a graph dramatically in tightening lower bounds.

5.1 Iteratively Bounding

In BestFirst, we prune subspaces whose lower bounds are larger than $\omega(P_k)$, where P_k is the k-th shortest path for a KPJ query. Therefore, BestFirst will run fast if we can prune more subspaces based on their lower bounds (i.e., by computing tighter/larger lower bounds for subspaces), considering that computing shortest path is time-consuming. However, in general, computing a tighter lower bound takes longer time, and in the extreme case computing the shortest path in a subspace provides the tightest lower bound. In Alg. 3, we present a light-weight lower bound estimation by considering only the immediate neighbors of u. Intuitively, we can compute a tighter lower bound by exploring multi-hop neighbors of u (e.g., neighbors of neighbors).

We propose to guess and tighten lower bounds of subspaces in a controlled manner by a threshold τ, which is achieved by a procedure TestLB. In a nutshell, given a subspace $\mathcal{S}$ and a threshold τ, TestLB tests whether the shortest path in $\mathcal{S}$ has a length larger than τ: if it is, then the lower bound is set as τ; otherwise, the shortest path $\mathsf{sp}(\mathcal{S})$ is obtained and returned. Details of TestLB will be discussed shortly. Ideally, we can set τ as $\omega(P_k)$, then all subspaces whose lower bounds are larger than $\omega(P_k)$ are pruned with the least amount of effort; however, P_k is unknown. Considering that TestLB takes longer time for a larger τ, we iteratively enlarge τ, and the k shortest paths of a KPJ query will be found once τ becomes no smaller than $\omega(P_k)$.

Algorithm 4: $\mathsf{IterBound}(G, Q = \{s, T, k\})$

```
1  Compute the shortest path $P'$ from $s$ to any node in $V_T$ in $G$;
2  Initialize a minimum priority queue $\mathcal{Q}$ to contain a single entry
   $\langle \mathcal{S}_0 = \langle (s), \emptyset \rangle, \omega(P'), P' \rangle$;
3  $i \leftarrow 1$; $\tau \leftarrow \omega(P')$;
4  while $i \leq k$ do
5      $\langle \mathcal{S} = \langle P_{s,u}, X_u \rangle, \mathsf{lb}(\mathcal{S}), P \rangle \leftarrow$ remove the top entry from $\mathcal{Q}$;
6      if $P \neq \emptyset$ then
7          Same as Lines 6-10 of Alg. 3;   /* $P_i \leftarrow P$, $i \leftarrow i+1$,
           and divide $\mathcal{S}$ into subspaces */;
8      else
9          $\tau \leftarrow \alpha \cdot \max\{\mathsf{lb}(\mathcal{S}), \mathcal{Q}.top().key\}$; /* Enlarge $\tau$ */;
10         $P \leftarrow \mathsf{TestLB}(P_{s,u}, X_u, \tau)$;
11         if $P \neq \emptyset$ then Put $\langle \mathcal{S}, \omega(P), P \rangle$ into $\mathcal{Q}$;
12         else Put $\langle \mathcal{S}, \tau, \emptyset \rangle$ into $\mathcal{Q}$;
13 return the $k$ paths $P_1, \ldots, P_k$;
```

The algorithm IterBound is given in Alg. 4, which is similar to Alg. 2. We first compute the shortest path P' in G (Line 1), put subspace $\mathcal{S}_0 = \langle (s), \emptyset \rangle$ with path P' into $\mathcal{Q}$ (Line 2), and initialize τ as $\omega(P')$ (Line 3). Then, we iteratively remove the subspace $\mathcal{S}$ with smallest lower bound, together with path P, from $\mathcal{Q}$ (Line 5). If P is not empty, then it is the next shortest path P_i, and we perform the same subspace division as Lines 6-10 of Alg. 2. Otherwise, we enlarge τ (Line 9), test whether the shortest path of $\mathcal{S}$ has a distance larger than τ (Line 10), and put $\mathcal{S}$ back into $\mathcal{Q}$ together with either the computed shortest path P or a larger lower bound τ depending on what TestLB returns (Lines 11-12).

Note that τ controls the computation of a tighter lower bound which we want to make larger but a larger τ will make TestLB slow. In our approach, we use a parameter α to control the speed of increasing τ iteratively. Here, α can be any real number larger than 1, and we use $\alpha = 1.1$ as default. At Line 9, we enlarge τ as $\alpha \cdot \max\{\mathsf{lb}(\mathcal{S}), \mathcal{Q}.top().key\}$, where $\mathcal{Q}.top().key$ is the key value (i.e., lower bound) in the top entry of $\mathcal{Q}$ and is defined to be $+\infty$ if $\mathcal{Q} = \emptyset$. The intuition is that, $\omega(P_k)$ should be not much larger than $\omega(P_1)$, the initial τ (Line 3). Note that $\mathsf{lb}(\mathcal{S})$ is the previous τ we have tested for $\mathcal{S}$; thus, the lower bound τ we tested for a subspace increases by a factor of at least α. Therefore, we iteratively enlarge τ from $\omega(P_1)$ to approach $\omega(P_k)$, and obtain the k shortest paths for a KPJ query once τ becomes no smaller than $\omega(P_k)$.

[1]Note that this triangle inequality holds for the shortest distances based on any distance metrics not only Euclidean distance. Thus, our techniques work for general graphs.

Theorem 5.1: *Given* TestLB, IterBound *correctly computes k shortest paths for a* KPJ *query.* □

Proof Sketch: IterBound follows the best-first paradigm of Alg. 2, except that we iteratively compute lower bounds of subspaces. Moreover, it is easy to prove that the lower bound τ computed for the same subspace $\mathcal{S}$ at Line 9 is strictly increasing (assuming that $\alpha > 1$). Therefore, let P_i be the correct i-th shortest path, we can prove that τ will be no less than $\omega(P_i)$ when the algorithm terminates, and once τ becomes no less than $\omega(P_i)$, the path P_i will be computed and inserted into $\mathcal{Q}$, thus will be output. □

IterBound acts the same as BestFirst if we set τ as $+\infty$. Nevertheless, by iteratively enlarging τ with an initial value $\omega(P_1)$, IterBound runs much faster than BestFirst by pruning more subspaces. Conceptually, Fig. 4(b) shows the subspaces in which the shortest paths are computed by IterBound. Compared to Fig. 4(a), $\mathcal{S}_4$ and $\mathcal{S}_{2,r+1}$ are pruned based on their tighter lower bounds that are computed by TestLB. The shadow areas of $\mathcal{S}_4$ and $\mathcal{S}_{2,r+1}$ in Fig. 4(b) indicate the exploration areas of TestLB in testing lower bounds.

Testing Lower Bound. TestLB tests whether the shortest path in a subspace has length larger than a given threshold τ. This is achieved by considering multi-hop neighbors of u, denoted V'. Given τ, V' are those nodes v with $\delta_{\mathcal{S}}(s,v)+\mathsf{lb}(v,V_T) \le \tau$, where $\delta_{\mathcal{S}}(s,v)$ is the shortest distance from s to v constrained in $\mathcal{S}$ (i.e., G' defined in Section 4.2). After obtaining V', if $\mathsf{V}' \cap V_T = \emptyset$ then $\omega(\mathsf{sp}(\mathcal{S})) > \tau$, otherwise, $\mathsf{sp}(\mathcal{S})$ is obtained by backtracking from $\mathsf{V}' \cap V_T$.

Algorithm 5: TestLB($P_{s,u}, X_u, \tau$)

```
1  Initialize a minimum-priority queue $\mathcal{Q}_V$ to contain $\langle u, 0\rangle$;
2  $d_s(u) \leftarrow \omega(P_{s,u})$, and $d_s(v) \leftarrow +\infty$ for all other nodes;
3  while $\mathcal{Q}_V \neq \emptyset$ do
4      $v \leftarrow$ remove the top node from $\mathcal{Q}_V$;
5      if $v \in V_T$ then  return the path formed by concatenating $P_{s,u}$
       with the computed shortest path from $u$ to $v$;
6      else for each outgoing edge $(v,w)$ of $v$ do
7          if $w \notin P_{s,u}, (v,w) \notin X_u$ and $d_s(v) + \omega(v,w) < d_s(w)$
           then
8              $d_s(w) \leftarrow d_s(v) + \omega(v,w)$;
9              Compute $\mathsf{lb}(w, V_T)$;
10             if $d_s(w) + \mathsf{lb}(w, V_T) \le \tau$ then
11                 Put $\langle w, d_s(w) + \mathsf{lb}(w, V_T)\rangle$ into $\mathcal{Q}_V$;
12 return $\emptyset$;
```

The pseudocode of TestLB is shown in Alg. 5, which is similar to A* search [16], where $d_s(v)$ stores the length of a path from s to v constrained in $\mathcal{S}$. We maintain the explored nodes together with their estimated distances (i.e., node v with $d_s(v) + \mathsf{lb}(v, V_T)$) in a minimum-priority queue $\mathcal{Q}_V$, which is initialized to contain u; nodes in $\mathcal{Q}_V$ are ranked by their estimated distances. Then, nodes are iteratively removed from $\mathcal{Q}_V$ (Line 4), and their neighbors are inserted into $\mathcal{Q}_V$ (Lines 6-11), until $\mathcal{Q}_V = \emptyset$ or we get a node from V_T; the latter case implies that $\mathsf{sp}(\mathcal{S})$ has been computed (Line 5). Here, V' is the set of nodes removed from $\mathcal{Q}_V$. Note that, following from [16], when a node v is removed from $\mathcal{Q}_V$, $d_s(v)$ stores the shortest distance from s to v constrained in $\mathcal{S}$ (i.e., $\delta_{\mathcal{S}}(s,v)$), and each node is removed from $\mathcal{Q}_V$ at most once.

The efficiency of TestLB is due to that, we only put into $\mathcal{Q}_V$ those nodes whose estimated distance are not larger than τ, as ensured by Line 10, which prunes a lot of nodes especially for a small τ.

Lemma 5.1: *Given a subspace $\mathcal{S}$ and τ,* TestLB *returns* $\mathsf{sp}(\mathcal{S})$ *if* $\omega(\mathsf{sp}(\mathcal{S})) \le \tau$*, and returns $\emptyset$ otherwise.* □

Proof Sketch: First of all, we remark that if we remove Lines 9-10 from Alg. 5, then it is the same as the A* search algorithm [16] that computes the shortest path in $\mathcal{S}$. Thus, if $\omega(\mathsf{sp}(\mathcal{S})) \le \tau$, then TestLB returns $\mathsf{sp}(\mathcal{S})$, since every node that is pruned at Line 10 will not be in $\mathsf{sp}(\mathcal{S})$ due to the nature of lower bound.

Secondly, we prove that if $\omega(\mathsf{sp}(\mathcal{S})) > \tau$, then TestLB returns $\emptyset$. The reason is that, for every node v obtained at Line 4 of Alg. 5, we have $d_s(v) \le \tau$ due to the pruning at Line 10. Thus, Alg. 5 cannot find the path $\mathsf{sp}(\mathcal{S})$, and returns $\emptyset$. □

Time Complexity. The time complexity of Alg. 5 is $O(m'+n' \log n')$, where n' and m' are the number of visited nodes and edges in Alg. 5 (specifically, at Line 6), respectively. In the worst case, $n' = n$ and $m' = m$; thus the time complexity is $O(m + n \log n)$. However, in practice, n' and m' are usually small and much smaller than n and m, respectively.

Example 5.1: First, let's consider TestLB($(v_1, v_3), \{(v_3, v_6)\}, 6$). v_3 has three valid out-neighbors, v_4, v_5, v_7. v_4 and v_7 are pruned because $d_{v_1}(v_3) + \omega(v_3, v_4) > 6$ and $d_{v_1}(v_3) + \omega(v_3, v_7) > 6$. Assume $\mathsf{lb}(v_5, V_T) = 2$, then v_5 is also pruned since $d_{v_1}(v_3) + \omega(v_3, v_5) + \mathsf{lb}(v_5, V_T) = 7$. Thus, TestLB($(v_1, v_3), \{(v_3, v_6)\}, 6$) returns $\emptyset$. Now, let's consider $\tau = 7$. Among the three valid out-neighbors, v_4 is pruned because $d_{v_1}(v_3) + \omega(v_3, v_4) = 8$; v_5 and v_7 are put into $\mathcal{Q}_V$ with lower bounds 7. Then, v_5 is removed from $\mathcal{Q}_V$, and its out-neighbor, v_6, is put into $\mathcal{Q}_V$ with lower bound 7. After that, either v_6 or v_7 is removed from $\mathcal{Q}_V$, and the shortest path in $\langle (v_1, v_3), \{(v_3, v_6)\}\rangle$ is obtained. □

5.2 Partial Shortest Path Tree

Motivated by DA-SPT, in this subsection we propose to compute and store a partial SPT, denoted $\mathsf{SPT_P}$, which provides a more accurate estimation of $\mathsf{lb}(v, V_T)$. Recall that $\mathsf{lb}(v, V_T)$ is used in TestLB to prune nodes, thus a more accurate estimation of TestLB will result in faster computation time. In contrast to DA-SPT which online constructs a full SPT by incurring high overheads, we obtain $\mathsf{SPT_P}$ as a by-product of computing the shortest path in G (i.e., Line 1 of Alg. 4) without any extra cost.

Algorithm 6: PartialSPT(G, s, T)

```
1  Initialize an empty minimum-priority queue $\mathcal{Q}_T$;
2  $\mathsf{SPT_P} \leftarrow$ a virtual root node $t$;        /* Build $\mathsf{SPT_P}$  */;
3  for each $w \in V_T$ do
4      Put $\langle w, \mathsf{lb}(s,w)\rangle$ into $\mathcal{Q}_T$, $d_t(w) \leftarrow 0, p(w) \leftarrow t$;
5  while $\mathcal{Q}_T \neq \emptyset$ do
6      $v \leftarrow$ remove the top node from $\mathcal{Q}_T$;
7      Add $v$ as a child of $p(v)$ to $\mathsf{SPT_P}$;  /* Build $\mathsf{SPT_P}$  */;
8      if $v = s$ then  return the path from $s$ to $t$;
9      for each incoming edge $(w,v)$ of $v$ do
10         if $d_t(v) + \omega(w,v) < d_t(w)$ then
11             $d_t(w) \leftarrow d_t(v) + \omega(w,v), p(w) \leftarrow v$;
12             Put $\langle w, d_t(w) + \mathsf{lb}(s,w)\rangle$ into $\mathcal{Q}_T$;
```

The algorithm to construct $\mathsf{SPT_P}$ is given in Alg. 6, denoted PartialSPT, which is the A* search algorithm for computing the shortest path from s to any node in V_T in G by adding Lines 2,7. The algorithm runs in the reverse graph of G, since we want to compute shortest paths from different nodes to any node in V_T. $\mathcal{Q}_T$ is similar to $\mathcal{Q}_V$ in Alg. 5 and initially contains all nodes of V_T (Lines 3-4). Then, nodes are iteratively removed from $\mathcal{Q}_T$ (Line 6) and their incoming edges are explored (Lines 9-12). The shortest path from s to any node in V_T is obtained when s is removed from

$\mathcal{Q}_T$ (Line 8). For all nodes v removed from $\mathcal{Q}_T$, we add v as a child of $p(v)$ to $\mathsf{SPT_P}$. Intuitively, $\mathsf{SPT_P}$ contains all nodes removed from $\mathcal{Q}_T$ prior to s when computing the shortest path from s to any node in V_T.

Proposition 5.1: *For nodes $v \in \mathsf{SPT_P}$, the path from v to t in $\mathsf{SPT_P}$ is the shortest path from v to any node in V_T in G.* □

Computing $\mathsf{lb}(v, V_T)$ using $\mathsf{SPT_P}$. Both CompLB and TestLB, which are invoked by IterBound, require computing $\mathsf{lb}(v, V_T)$ for any $v \in V$. Eq. (2) computes $\mathsf{lb}(v, V_T)$ using a landmark-based approach. By utilizing $\mathsf{SPT_P}$, we can compute a more accurate $\mathsf{lb}(v, V_T)$ as follows. If v is in $\mathsf{SPT_P}$, then $\mathsf{lb}(v, V_T)$ is computed as the length of the path from v to t in $\mathsf{SPT_P}$, the correctness of which directly follows from Proposition 5.1; otherwise, it is computed by Eq. (2). Here, we give $\mathsf{SPT_P}$ a higher priority, because if $v \in \mathsf{SPT_P}$, then the lower bound computed using $\mathsf{SPT_P}$ is guaranteed to be not smaller than that by Eq. (2); for lower bound, the larger the better.

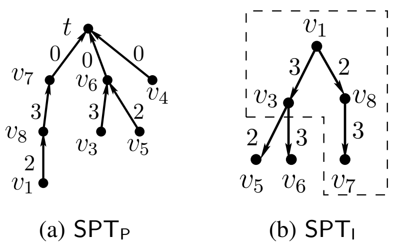


Figure 5: Partial and incremental SPT

Example 5.2: Fig. 5(a) shows the $\mathsf{SPT_P}$ constructed for $Q = \{v_1, \text{“}H\text{”}, 3\}$. For each node v in $\mathsf{SPT_P}$, its distance to t in $\mathsf{SPT_P}$ is the shortest distance from v to any node in V_T and can be used as an estimation of $\mathsf{lb}(v, V_T)$. For example, $\mathsf{lb}(v_3, V_T)$ is estimated as 3. □

IterBound-$\mathsf{SPT_P}$ Approach. We denote the approach that uses $\mathsf{SPT_P}$ to estimate $\mathsf{lb}(v, V_T)$ in Alg. 4 as IterBound-$\mathsf{SPT_P}$. The correctness of IterBound-$\mathsf{SPT_P}$ directly follows from that of IterBound and the above discussions.

5.3 Incremental Shortest Path Tree

$\mathsf{SPT_P}$ includes all nodes of V_T which can be large for a KPJ query, thus may take long time to construct. In this subsection, we propose an incremental SPT, denoted $\mathsf{SPT_I}$, by pruning nodes in V_T that are far-away from the source node s. Moreover, we incrementally enlarge $\mathsf{SPT_I}$, based on which we identify a new property for reducing the exploration area of a graph by TestLB.

Constructing $\mathsf{SPT_I}$. To prune from $\mathsf{SPT_I}$ those nodes in V_T that are far-away from s, in $\mathsf{SPT_I}$ we compute and store shortest paths from s to each node of a subset of V. Recall that, in $\mathsf{SPT_P}$, we store shortest paths from each node of a subset of V to V_T. Thus, the construction of $\mathsf{SPT_I}$ is run on G by starting from s, and consists of two phases. In the first phase, we construct an initial $\mathsf{SPT_I}$, which is a by-product of computing the shortest path from s to any node in V_T in a similar fashion to PartialSPT (i.e., Alg. 6) while running on G and starting from s; this is invoked at Line 1 of Alg. 4. In the second phase, we incrementally enlarge $\mathsf{SPT_I}$ by IncrementalSPT, which is invoked after Line 9 and before Line 10 of Alg. 4.

The pseudocode of IncrementalSPT is shown in Alg. 7, which is self-explanatory. The general idea is to include into $\mathsf{SPT_I}$ all nodes of V that are on paths from s to any node in V_T whose lengths are not larger than τ. Therefore, IncrementalSPT iteratively removes

Algorithm 7: IncrementalSPT(G, T, τ)
```
1 while $\mathcal{Q}_T.top().key \le \tau$ do
2     $v \leftarrow$ remove the top node from $\mathcal{Q}_T$;
3     Add $v$ as a child of $p(v)$ to $\mathsf{SPT_I}$;
4     if $v \in V_T$ then Add $v$ to $D$;
5     for each outgoing edge $(v, w)$ of $v$ do
6         if $d_s(v) + \omega(v, w) < d_s(w)$ then
7             $d_s(w) \leftarrow d_s(v) + \omega(v, w)$, $p(w) \leftarrow v$;
8             Put $\langle w, d_s(w) + \mathsf{lb}(w, V_T)\rangle$ into $\mathcal{Q}_T$;
```

the top node v from $\mathcal{Q}_T$ (Line 2), and adds v into $\mathsf{SPT_I}$ (Line 3). Meanwhile, the subset of V_T that are in $\mathsf{SPT_I}$ is maintained into a set D (Line 4), which will be used later to improve the performance of testing lower bound for a subspace. In Fig. 5(b), the subtree in the rectangle shows the initial $\mathsf{SPT_I}$ constructed, and the entire tree is the resulting $\mathsf{SPT_I}$ for $\tau = 7$. Here, $D = \{v_6, v_7\}$. $\mathsf{SPT_I}$ has the following property.

Proposition 5.2: *The $\mathsf{SPT_I}$ constructed by Alg. 7 contains all nodes on paths from s to any node in V_T whose lengths are no larger than τ.* □

Algorithm 8: CompLB-$\mathsf{SPT_I}$ $(P_{t,u}, X_u)$
```
1 if $u \neq t$ then $N(u) \leftarrow$ {in-neighbors of $u$} else $N(u) \leftarrow D$;
2 $lb \leftarrow +\infty$;
3 for each $v \in N(u)$ do
4     if $v \notin P_{t,u}$ and $(v, u) \notin X_u$ then
5         if $v \notin \mathsf{SPT_I}$ then Compute $\mathsf{lb}(s, v)$ using Eq. (2);
6         else $\mathsf{lb}(s, v) \leftarrow$ the distance from $s$ to $v$ in $\mathsf{SPT_I}$;
7         $lb \leftarrow \min\{lb, \omega(P_{t,u}) + \omega(v, u) + \mathsf{lb}(s, v)\}$;
8 if $u = t$ and $lb = +\infty$ and $D \neq V_T$ then $lb \leftarrow 0$;
9 return $lb$;
```

Computing Initial Lower Bound for a Subspace using $\mathsf{SPT_I}$. We compute the lower bound of a subspace using $\mathsf{SPT_I}$ in a similar way to CompLB (Alg. 3), by considering the immediate neighbors of u. The algorithm is shown in Alg. 8, denoted CompLB-$\mathsf{SPT_I}$. Note that, here $P_{t,u}$ is a path from u to t in G and X_u is a subset of the incoming edges to u. CompLB-$\mathsf{SPT_I}$ differs from CompLB in the following two aspects. Firstly, $\mathsf{lb}(s, v)$ is estimated by utilizing $\mathsf{SPT_I}$ in the same way as that in Section 5.2. Secondly, when $u = t$, instead of considering all nodes of V_T, we consider only the subset that is in $\mathsf{SPT_I}$ (i.e. D). The reason is that, the entire V_T can be very large, while the small subset D is sufficient for our purpose of computing $\mathsf{lb}(P_{t,u}, X_u)$, which saves a lot of computations.

The correctness of CompLB-$\mathsf{SPT_I}$ when $u \neq t$ directly follows from that of CompLB. However, when $u = t$, there are two cases depending on whether there are any valid incoming edges to u. 1) If there is no valid incoming edge to u (i.e., every node of $N(u)$ is either in $P_{t,u}$ or in X_u, Lines 3-4), then lb will be $+\infty$. We reassign lb to 0 if $D \neq V_T$. 2) Otherwise, $lb \neq +\infty$, which is guaranteed to be a lower bound of the subspace $\langle P_{t,u}, X_u\rangle$.

Testing Lower Bound using $\mathsf{SPT_I}$. By utilizing $\mathsf{SPT_I}$, we propose a more efficient algorithm for testing lower bound, denoted TestLB-$\mathsf{SPT_I}$. We modify TestLB to develop TestLB-$\mathsf{SPT_I}$ in the same fashion as the development of CompLB-$\mathsf{SPT_I}$. Moreover, *we prune all nodes that are not in $\mathsf{SPT_I}$ from consideration* (i.e., from putting into $\mathcal{Q}_V$). Consequently, every $\mathsf{lb}(s, w)$ is computed as the distance from s to w in $\mathsf{SPT_I}$; that is, Eq. (2) is not evaluated. Therefore, TestLB-$\mathsf{SPT_I}$ is more efficient than TestLB. We prove the correctness of $\mathsf{TestLB\text{-}SPT_I}$ in the following lemma.

Lemma 5.2: *Given a subspace $\mathcal{S}$ and τ, $\mathsf{TestLB\text{-}SPT_I}$ returns $\emptyset$ if $\omega(\mathsf{sp}(\mathcal{S})) \geq \tau$, and returns $\mathsf{sp}(\mathcal{S})$ otherwise.* □

Proof Sketch: The lemma follows from Lemma 5.1 and Proposition 5.2. □

Example 5.3: We show running $\mathsf{TestLB\text{-}SPT_I}$ for subspace $\langle(v_7), \{(v_7, v_8)\}\rangle$ with $\tau = 6$, where $\mathsf{SPT_I}$ is shown in Fig. 5(b). Among v_7's in-neighbors, only v_3 is considered, since v_{13} and v_{14} are not in $\mathsf{SPT_I}$. For v_3, $\mathsf{lb}(v_1, v_3) = 3$ which is the distance in $\mathsf{SPT_I}$. Then, v_3 is also pruned since $\omega(v_3, v_7) + \mathsf{lb}(v_1, v_3) = 7$, and $\mathsf{TestLB\text{-}SPT_I}$ returns $\emptyset$. For $\tau = 7$, $\mathsf{SPT_I}$ remains the same. Then, v_3 is not pruned and the shortest path in $\langle(v_7), \{(v_7, v_8)\}\rangle$ is obtained, which is (v_1, v_3, v_7) with length 7. □

$\mathsf{IterBound\text{-}SPT_I}$ **Approach.** Based on $\mathsf{SPT_I}$ and the discussions above, we propose an approach following Alg. 4 for processing KPJ queries, denoted $\mathsf{IterBound\text{-}SPT_I}$. It runs on the reverse graph of G, and a subspace is represented by $\langle P_{t,u}, X_u\rangle$ where X_u is a subset of the incoming edges to u.

$\mathsf{IterBound\text{-}SPT_I}$ improves the efficiency by computing $\mathsf{SPT_I}$ and pruning all nodes not in $\mathsf{SPT_I}$ from consideration when conducting the iteratively bounding search. That is, we take as input only the small subgraph of G induced by nodes in $\mathsf{SPT_I}$. Note that, $\mathsf{SPT_I}$ enlarges for a larger τ, and the subgraph induced by nodes in $\mathsf{SPT_I}$ also enlarges; this guarantees that we can correctly process any KPJ query.

Time Complexity. The time complexity of the $\mathsf{IterBound\text{-}SPT_I}$ approach is $O(kn(m' + n' \log n'))$, where n' and m' are the number of nodes and edges, respectively, in the subgraph G' of G that is induced by nodes w with $d_s(w) + lb(w, V_T) <= \tau$ (see Line 10 of Alg. 5) for the largest τ obtained in $\mathsf{IterBound\text{-}SPT_I}$. Note that, n' and m' are usually small in real applications; thus, $\mathsf{IterBound\text{-}SPT_I}$ runs much faster than DA (see Alg. 1).

6. EXTENSIONS

In the following, we extend our techniques to other applications including the case that the source node also has multiple physical nodes and the case that landmarks are not available.

General KPJ. A general KPJ (GKPJ) query is an extension of KPJ query where the source node also has multiple physical nodes, denote as $Q = \{S, T, k\}$, where both S and T are categories. It is to compute the top-k shortest paths from any node in V_S to any node in V_T, where V_S is the set of nodes of V belonging to category S. We can convert a GKPJ query to a KPJ query by introducing a virtual source node s and connecting s to all nodes in V_S with weights 0. Then, Q is reduced to a KPJ query $Q' = \{s, T, k\}$ on the new graph, and all our proposed techniques can be used to process the query.

Computing without Landmark. Our techniques are presented based on landmarks which are used to estimate $\mathsf{lb}(u, V_T)$. Nevertheless, when landmarks are not available, all our techniques can still be directly applied by setting all $\mathsf{lb}(u, V_T)$ (i.e., computed by Eq. (2)) to be 0. Specifically, for $\mathsf{IterBound\text{-}SPT_I}$, the landmark is only used for constructing the $\mathsf{SPT_I}$ using A* search [16], as discussed in Section 5.3; thus, without landmark, we construct the $\mathsf{SPT_I}$ by setting $\mathsf{lb}(u, V_T)$ to be 0 (the A* search then becomes the Dijkstra's algorithm [11]), while other parts of $\mathsf{IterBound\text{-}SPT_I}$ remain the same.

Moreover, without landmark, our techniques still perform well; the reasons are as follows. The $\mathsf{IterBound\text{-}SPT_I}$ approach mainly consists of two parts: 1) incrementally constructing the partial shortest path tree $\mathsf{SPT_I}$, 2) computing lower bound or shortest path for a subspace. The dominating cost comes from the second part, while landmark is only used in the first part. Thus, by running $\mathsf{IterBound\text{-}SPT_I}$ without landmark will only increase the cost of the first part which is not a big factor in the total cost.

7. PERFORMANCE STUDIES

We conduct extensive performance studies to evaluate the efficiency of our approaches against the baseline approaches for processing KPJ queries. The following algorithms are implemented:

- DA [28] and DA-SPT [15]. They are in the deviation paradigm as discussed in Section 3, and are the baseline approaches for processing KPJ queries.
- BestFirst. It is the best-first approach as discussed in Section 4.
- IterBound, $\mathsf{IterBound_P}$, and $\mathsf{IterBound_I}$. They are the iteratively bounding approaches with or without SPT as discussed in Section 5. $\mathsf{IterBound_P}$ and $\mathsf{IterBound_I}$ are abbreviations of $\mathsf{IterBound\text{-}SPT_P}$ and $\mathsf{IterBound\text{-}SPT_I}$, respectively.
- $\mathsf{IterBound_I}$-NL. It is the $\mathsf{IterBound_I}$ approach however without landmark, as discussed in Section 6.

Note that, 1) in our testings, a KSP query is also considered as a KPJ query where the query category uniquely identifies the destination node; 2) the baseline approaches for processing KPJ queries are the state-of-the-art techniques for processing KSP queries.

All algorithms are implemented in C++ and compiled with GNU GCC by -O3 option. All tests are conducted on a PC with an Intel(R) Xeon(R) 2.66GHz CPU and 4GB memory running Linux. We evaluate the performance of the algorithms on real datasets as follows.

Dataset	#Nodes	#Edges
CAL	106,337	213,964
SJ	18,263	47,594
SF	174,956	443,604
COL	435,666	1,042,400
FLA	1,070,376	2,687,902
USA	6,262,104	15,119,284

Table 1: Summary of dataset

Datasets. We use six real road networks with real/synthetic points of interest (POIs), and each POI belongs to a category. They are: California road network (CAL), San Joaquin road network (SJ), San Francisco road network (SF), Colorado road network (COL), Florida road network (FLA), and Western USA road network (USA). The first three are downloaded from `www.cs.utah.edu/~lifeifei/SpatialDataset.htm`, and the last three are from DIMACS (`www.dis.uniroma1.it/~challenge9/download.shtml`). A summary is given in Table 1.

POIs. The CAL dataset is provided with real POIs, which have 62 different categories. For the other five road networks, we generate synthetic POIs randomly located on nodes. For each road network, we generate four sets of POIs, denoted T_1, T_2, T_3, T_4, corresponding to different number of physical destination nodes (i.e., $n \times 10^{-4}, 5n \times 10^{-4}, 10n \times 10^{-4}, 15n \times 10^{-4}$ POIs, respectively), where n is the number of nodes in a road network. For example, for

COL, $|T_1| = 43$, $|T_2| = 217$, $|T_3| = 435$, and $|T_4| = 653$. Note that, we generate the POIs in such a way that $T_1 \subset T_2 \subset T_3 \subset T_4$.

Graphs. We model each road network with POIs as a graph G. Here, an edge (u, v) in G represents a road segment, and has a non-negative weight $\omega(u, v)$ which can be any measure of the road segment, such as distance, travel time, travel cost, and etc. We take distance as weight in our experiments. Each node belongs to the categories of POIs that are located on it.[2]

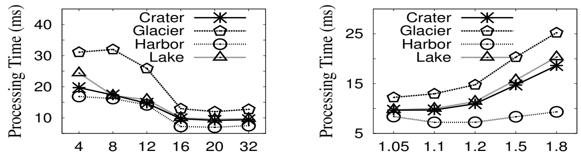
Queries. A query consists of a source node s, a destination node set V_T indicated by category T, and a value k indicating the number of paths to found. For each query, we first choose a category T, and then randomly generate source nodes. For the CAL dataset, we consider four representative categories, *"Glacier"*, *"Lake"*, *"Crater"*, and *"Harbor"*, which have 1, 8, 14, and 94 physical nodes, respectively. For the other datasets, we consider T_1, T_2, T_3, and T_4, and choose T_2 by default.

For a destination category T, the source nodes in a query are randomly generated as follows. We sort all nodes in increasing order regarding their shortest path lengths to category T, partition them into 5 groups, and generate 5 query sets, Q_1, Q_2, Q_3, Q_4, Q_5, each of which consists of 100 nodes randomly selected from the corresponding group. Thus, nodes in Q_i are closer to destination nodes than nodes in Q_j do, for $i < j$. We use Q_3 as the default query set.

k is chosen from 10, 20, 30, and 50, with 20 as the default.

7.1 Experimental Results

Eval-I: Parameters. We evaluate the influence of landmark size $|L|$ and parameter α on the performance of $\mathsf{IterBound_l}$.



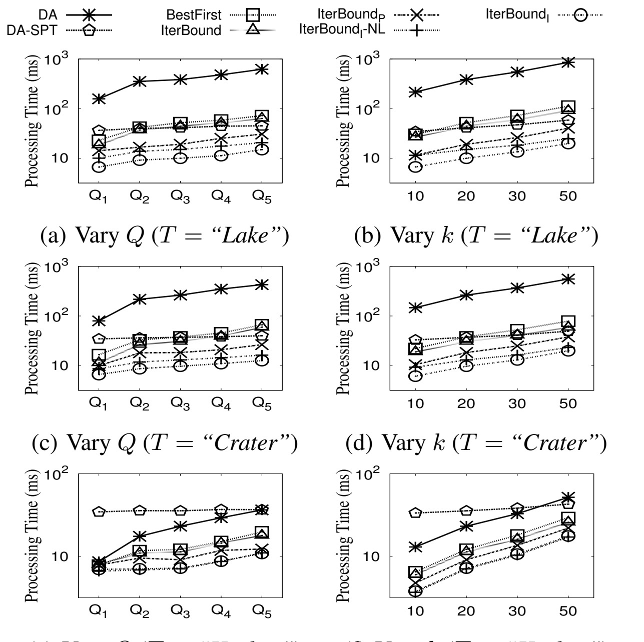
(a) Varying $|L|$ (b) Varying α

Figure 6: Parameter testing on CAL ($Q_3, k = 20$)

Choosing $|L|$. In our approaches, we use landmarks for estimating $\mathsf{lb}(v, V_T)$, the lower bound of shortest distance from v to any node in V_T. The landmarks are chosen following the most popular way in [16].[3] Fig. 6(a) shows the processing time of $\mathsf{IterBound_l}$ for different $|L|$ values. Clearly, when $|L|$ increases from 4 to 16, the processing time decreases, because more landmarks can provide more accurate estimation of $\mathsf{lb}(v, V_T)$. However, when $|L|$ increases from 16 to 32, the processing time increases a little due to longer computation time of $\mathsf{lb}(v, V_T)$. Therefore, we choose $|L| = 16$.

Choosing α. The running time of $\mathsf{IterBound_l}$ for different α values are illustrated in Fig. 6(b). Recall that α is used in our iterative bounding approaches for controlling the increasing ratio of τ (i.e., controlling the computation of a tighter lower bound, see Alg. 4). The running time increases when α increases from 1.1 to 1.8 due to building a larger $\mathsf{SPT_l}$. However, when α decreases from 1.1 to 1.05, the processing time also increases due to taking more iterations to reach the final τ. Therefore, we choose $\alpha = 1.1$.

Note that: 1) among our parameters, τ is determined by α; 2) better choices of $|L|$ and α will improve the performance of our algorithm marginally as shown in Fig. 6. It will be our future work to automatically find the best choice of $|L|$ and α.



(a) Vary Q (T = *"Lake"*) (b) Vary k (T = *"Lake"*)

(c) Vary Q (T = *"Crater"*) (d) Vary k (T = *"Crater"*)

(e) Vary Q (T = *"Harbor"*) (f) Vary k (T = *"Harbor"*)

Figure 7: Against baseline approaches on CAL (Varying Q, k)

Eval-II: Against the Baseline Approaches. Here, we evaluate the performances of our approaches against the baseline approaches on CAL dataset.

KPJ *Query.* The processing time of the seven approaches by varying query sets and k is demonstrated in Fig. 7, where the destination category is chosen from *"Lake"*, *"Crater"*, and *"Harbor"*. In general, all our approaches, BestFirst, IterBound, $\mathsf{IterBound_P}$, $\mathsf{IterBound_l}$, and $\mathsf{IterBound_l}$-NL, outperform the two baseline approaches, DA and DA-SPT. This is because our approaches use a best-first paradigm to reduce the number of shortest path computations. In Figures 7(a)-7(d), DA-SPT outperforms DA because DA-SPT online builds a full SPT to facilitate the shortest path computation. However, in Fig. 7(e)-7(f), DA-SPT performs worse due to the dominating cost of building the full SPT. When the lengths of shortest paths increase (i.e., varying Q from Q_1 to Q_5), the running time of all approaches increases except DA-SPT which is steady, due to the dominating cost of constructing the full SPT. Moreover, although without landmarks, $\mathsf{IterBound_l}$-NL outperforms all other approaches except $\mathsf{IterBound_l}$ across all testings. The trend of the processing time of these approaches by varying k is similar to that of varying query set. One exception is that, the processing time of DA-SPT also slightly increases due to computing more shortest paths for larger k.

KSP *Query.* We test the approaches for processing KSP queries by setting the destination category as *"Glacier"* which has only one

[2]For simplicity, we assume that POIs are located on the nodes of G. When a POI is on an edge (u, v), we can add a new node w to G and connect w with u and v to replace (u, v). Note that, given a query with category T, we only need to consider the set of POIs belonging to category T.

[3]We firstly pick a random start node and select the farthest node from the start node as the first landmark, and then iteratively choose the node that is farthest away from the current set of landmarks as the next landmark.

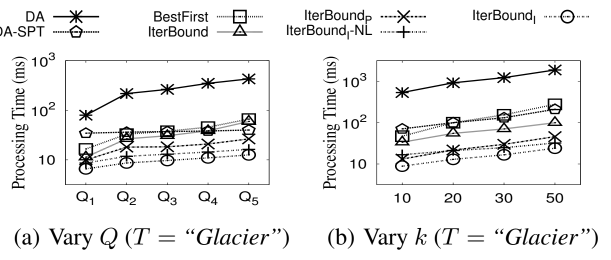
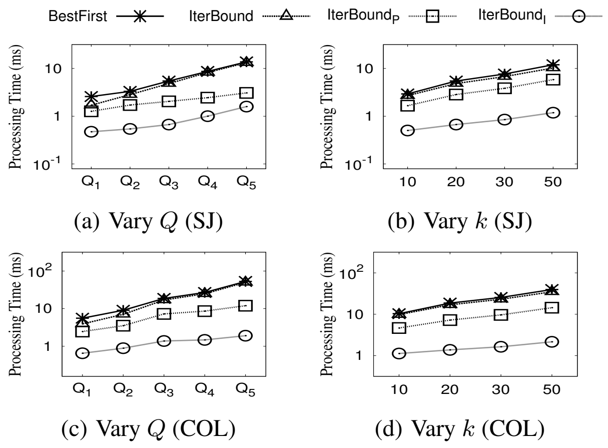

(a) Vary Q (T = *"Glacier"*) (b) Vary k (T = *"Glacier"*)

Figure 8: Testing KSP queries on CAL (Varying Q and k)

physical destination node; thus, the KPJ query is a KSP query. The results are shown in Fig. 8, which are similar to that for KPJ queries in Fig. 7.

Summary. There is no clear winner between DA and DA-SPT, and all our approaches perform better than these two baseline approaches. Despite using the same techniques, IterBound$_I$ outperforms IterBound$_I$-NL, which demonstrates the effectiveness of using landmarks for estimating lower bounds. Therefore, in the following testings, we omit DA, DA-SPT, and IterBound$_I$-NL, and evaluate the other approaches which use different techniques and all use landmark.

Eval-III: Evaluating Our Approaches. In this testing, we evaluate the efficiency of our different approaches, BestFirst, IterBound, IterBound$_P$, and IterBound$_I$, on SJ and COL.


(a) Vary Q (SJ) (b) Vary k (SJ)

(c) Vary Q (COL) (d) Vary k (COL)

Figure 9: Our approaches by varying Q and k ($T = T_2$)

Varying Q and k. The running time of the approaches on SJ and COL by varying Q and k is shown in Fig. 9. Similar to that in Fig. 7, the running time of these four approaches increases when either Q varies from Q_1 to Q_5 or k increases. IterBound slightly outperforms BestFirst due to less number of shortest path computations, however with more expensive lower bound computations. IterBound$_P$ performs better than IterBound because of the faster lower bound testing. IterBound$_I$ runs faster than IterBound$_P$ because IterBound$_I$ can further reduce the exploration area of a graph by SPT$_I$.

Varying Number of Destination Nodes ($|T|$). Fig. 10 shows the processing time of these four approaches on SJ and COL by varying the number of destination nodes (i.e., varying $|T|$). For all these four approaches, the processing time decreases, when the number of destination nodes increases (i.e., T varies from T_1 to T_4). This is because the shortest paths become shorter for more number

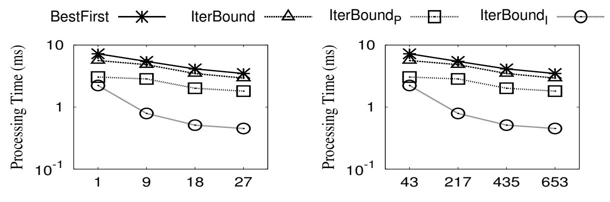

(a) SJ (b) COL

Figure 10: Vary #(destination nodes) ($Q = Q_3$, k = 20)

of destination nodes as shown in Fig. 11 which will be discussed shortly. IterBound$_I$ outperforms IterBound$_P$ which then outperforms BestFirst and IterBound. The improvement of IterBound$_I$ over IterBound$_P$ becomes more significant when there are more destination nodes, since IterBound$_I$ can prune destination nodes and reduce the exploration area of a graph by SPT$_I$.

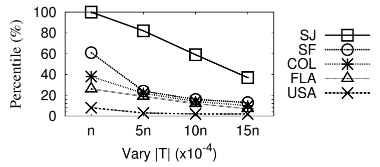


Figure 11: Shortest path length (Varying #(destination nodes))

Fig. 11 illustrates the influence of the number of destination nodes on the shortest path lengths. Specifically, for each category T_i, we compute the longest length of shortest paths from nodes to T_i, and report its percentile position in the observations of all $n \cdot n$ shortest path lengths in the graph. For all datasets, the shortest path lengths decrease with more number of destination nodes; thus all approaches run faster as shown in Fig. 10. Note that, for a specific T_i, the number of destination nodes belonging to T_i are different, thus the shortest path lengths vary for different datasets; for example, for T_1, the number of destination nodes for SJ, SF, COL, FLA, USA are 1, 17, 43, 107, and 626, respectively.

Summary. IterBound$_I$ outperforms the other approaches, BestFirst, IterBound, and IterBound$_P$, across all different datasets, different number of destination nodes, different Q, and different k. Thus, in the following we only evaluate our IterBound$_I$ approach.

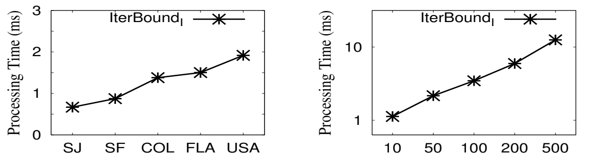
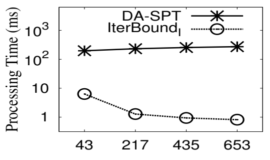

(a) Vary graph size (k = 20) (b) Vary k (COL)

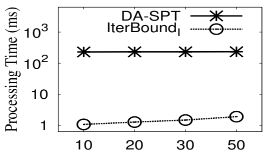
Figure 12: Scalability of IterBound$_I$ ($T = T_2$, $Q = Q_3$)

Eval-IV: Scalability Testing. The scalability testing results of IterBound$_I$ by varying graph size and k are shown in Fig. 12. Although the running time increases when either the graph size or k increases, IterBound$_I$ is scalable enough to process very large graphs. For example, when the graph size increases 40 times (i.e., from SJ to USA), the running time of IterBound$_I$ only increases slightly (e.g., by no more than 3 times).

Eval-V: GKPJ Testing. In this evaluation, we test the efficiency of IterBound$_I$ over DA-SPT, the state-of-the-art approach, for GKPJ



(a) Vary #(destination nodes) ($k = 20$)



(b) Vary k ($T = T_2$)

Figure 13: GKPJ query (COL)

queries $Q = \{S, T, k\}$. Here, the source category S has 4 physical nodes which are randomly chosen. Fig. 13 shows the running time by varying the number of destination nodes (i.e., $|T|$) or k. The trends of running time of DA-SPT and IterBound$_l$ are similar to the previous evaluations. The improvement of IterBound$_l$ over DA-SPT is more significant (e.g., by two orders of magnitude). This is because the lengths of k shortest paths become smaller with multiple source nodes.

8. CONCLUSION

In this paper, we studied the problem of top-k shortest path join (KPJ). We adopted the best-first paradigm to reduce the number of shortest path computations, compared to the existing deviation paradigm, by pruning subspaces based on their lower bounds. To improve the efficiency, we further proposed an iteratively bounding approach to tightening lower bounds of subspaces which is achieved by lower bound testing. Moreover, we proposed index structures to significantly reduce the exploration area of a graph in lower bound testing. We conducted extensive performance studies using real road networks, and demonstrated that our proposed approaches significantly outperform the baseline approaches for KPJ queries. Furthermore, our approaches can be immediately used to process KSP queries, and they also outperform the state-of-the-art algorithm for KSP queries by several orders of magnitude.

Acknowledgements. Lijun Chang is supported by ARC DE150100563. Xuemin Lin is supported by NSFC61232006, ARC DP120104168, ARC DP140103578, and ARC DP150102728. Lu Qin is supported by ARC DE140100999. Jeffrey Xu Yu is supported by Research Grants Council of the Hong Kong SAR, China, 14209314 and 418512. Jian Pei is supported in part by an NSERC Discovery grant.

9. REFERENCES

[1] H. Aljazzar and S. Leue. K*: A heuristic search algorithm for finding the k shortest paths. *Artif. Intell.*, 175(18):2129–2154, 2011.

[2] R. Bellman and R. Kalaba. On kth best policies. *Journal of the Society for Industrial and Applied Mathematics*, 1960.

[3] J. Berclaz, F. Fleuret, E. Türetken, and P. Fua. Multiple object tracking using k-shortest paths optimization. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(9), 2011.

[4] X. Cao, G. Cong, and C. S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *PVLDB*, 3(1), 2010.

[5] X. Cao, G. Cong, C. S. Jensen, and B. C. Ooi. Collective spatial keyword querying. In *Proc. of SIGMOD'11*, 2011.

[6] S. Chechik. Improved distance oracles for vertex-labeled graphs. *CoRR*, abs/1109.3114, 2011. informal publication.

[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5), 2003.

[8] E. de Queirós Vieira Martins and M. M. B. Pascoal. A new implementation of yen's ranking loopless paths algorithm. *4OR*, 1(2), 2003.

[9] E. de QueirÍős Vieira Martins, M. M. B. Pascoal, and J. L. E. dos Santos. The k shortest loopless paths problem, 1998.

[10] D. Delling, A. V. Goldberg, R. Savchenko, and R. F. Werneck. Hub labels: Theory and practice. In *Proc. of SEA'14*, 2014.

[11] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1), 1959.

[12] D. Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2), 1998.

[13] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *Proc. of ICDE'08*, pages 656–665, 2008.

[14] J. Gao, H. Qiu, X. Jiang, T. Wang, and D. Yang. Fast top-k simple shortest paths discovery in graphs. In *Proc. of CIKM'10*, 2010.

[15] J. Gao, J. X. Yu, H. Qiu, X. Jiang, T. Wang, and D. Yang. Holistic top-k simple shortest path join in graphs. *IEEE Trans. Knowl. Data Eng.*, 24(4):665–677, 2012.

[16] A. V. Goldberg and C. Harrelson. Computing the shortest path: *A** search meets graph theory. In *Proc. of SODA'05*, 2005.

[17] D. Hermelin, A. Levy, O. Weimann, and R. Yuster. Distance oracles for vertex-labeled graphs. In *Proc. of ICALP'11*, 2011.

[18] J. Hershberger, M. Maxel, and S. Suri. Finding the *k* shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms*, 3(4), 2007.

[19] W. Hoffman and R. Pavley. A method for the solution of the nth best path problem. *J. ACM*, 6, October 1959.

[20] C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *Proc. of GIS'03*, 2003.

[21] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[22] M. R. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *Proc. of VLDB'04*, 2004.

[23] S. Nutanong and H. Samet. Memory-efficient algorithms for spatial network queries. In *Proc. of ICDE'13*, 2013.

[24] M. M. B. Pascoal. Implementations and empirical comparison of *k* shortest loopless path algorithms, Nov. 2006.

[25] D. Quercia, R. Schifanella, and L. M. Aiello. The shortest path to happiness: Recommending beautiful, quiet, and happy routes in the city. In *Proc. of Hypertext'14*, 2014.

[26] Y.-K. Shih and S. Parthasarathy. A single source *k*-shortest paths algorithm to infer regulatory pathways in a gene network. *Bioinformatics*, 28(12), 2012.

[27] P. Venetis, H. Gonzalez, C. S. Jensen, and A. Y. Halevy. Hyper-local, directions-based ranking of places. *PVLDB*, 4(5), 2011.

[28] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17, 1971.

[29] C. Zhang, Y. Zhang, W. Zhang, and X. Lin. Inverted linear quadtree: Efficient top k spatial keyword search. In *Proc. of ICDE'13*, 2013.