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,txue}@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, ya@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 con-
ceptual 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 \( P \) 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 \( P \) with the shortest lengths.

We adopt the best-first paradigm to recursively divide \( P \) 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 \( \tau \) of a subspace. Initially, we assign the value of \( \tau \) as the length of the (1st) shortest path. Then, we always choose the subspace with the smallest \( \tau \) 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 \( \tau \) into \( \alpha \cdot \tau \) for the subspace; otherwise the shortest path in the subspace is computed. Moreover, we propose two online-built index structures, SPT\(_P \) and SPT\(_T \), 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 \( n \) 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. Epstein’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 \( \omega \) 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 \( (v_i, v_{i+1}) \in E, 1 \leq i < l \), and we say that \( P \) consists of edges \( (v_i, v_{i+1}), 1 \leq i < l \). 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 \( \omega(P) = \sum_{(v_i, v_{i+1}) \in E} \omega(v_i, v_{i+1}) \). 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) \( \omega(P_i) \leq \omega(P_{i+1}), 1 \leq i < k \); 3) \( \omega(P_k) \leq \omega(P) \) for any
other path $P$ from $s$ to category $T$.

![Graph example](image)

**Figure 1:** An example graph

**Example 2.1:** Fig. 1 illustrates a graph $G$, where $V = \{v_1, \ldots, v_{15}\}$ and nodes $v_4, v_9, v_7$ 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$”}, 1)$, 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 a 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 direct 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 $PT_i$ 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.

![Pseudo-trees](image)

**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 $(v_3, v_5)$ nor $(v_3, v_7)$; thus $c(v_3) = (v_1, v_3, v_5, v_6, t)$, and $C_3 = \{c(v_1), c(v_3), c(v_7), c(v_3), c(v_6), c(v_5)\}$.

**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 $k$
   1. $P_i \leftarrow$ the path in $C_{i-1}$ with the shortest length;
   2. Construct $PT_i$ by inserting $P_i$ into $PT_{i-1}$, and let $d$ be the deviation vertex;
   3. 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 $C_{i-1}$;
4. 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$”}, 3\}$ 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 $P_1 = (v_1, v_3, v_7, t)$ with length 5. After inserting $P_1$ into $PT_0$, the resulting $PT_1$ is shown in Fig. 2(a), where $C_1 = \{c(v_1), c(v_3), c(v_7)\}$. 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 $v_3, v_5, v_6$ are updated or computed, and $C_2 = \{c(v_3), c(v_5), c(v_6)\}$. 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 $PT_i$, if the path formed by concatenating, 1) the path from $s$ to $u$ in $PT_2$, 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,

135
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 $P_{s,T}(G)$ 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, $P_{s,T}(G)$ is abbreviated to $P$. Then, the query $Q$ is to find the paths in $P$ with shortest lengths. Note that the size of $P$ can be exponential to $n$.

**Search Space and Subspace.** The general idea is that we regard $P$ as the entire search space $S_0$. Then, the $k$ paths in $P$ with shortest lengths can be found by recursively dividing a subspace (initially $S_0$) into smaller subspaces and computing the shortest path in each newly obtained subspace.

![Figure 3: Overview of search space division](image)

Before diving into the details, we first explain the main idea which is illustrated in Fig. 3. We conceptualize each path in $P$ 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 $P_1 = (v_1, \ldots, v_k)$ in the entire search space $S_0 = P$. Second, we divide $S_0$ into $l + 1$ subspaces, $S_1, S_2, \ldots, S_l, S_{l+1}$. Here, $S_1$ consists of only $P_1$ and is excluded from further considerations. Each of the remaining subspaces, $S_2, \ldots, S_{l+1}$, represents the set of paths in $P$ that share exactly the prefix of $P_1$ from $v_1$ to $v_{l-1}$; consequently, $S_i \neq S_j, \forall i \neq j$, and $\bigcup_{i=1}^{l+1} S_i = S_0$. Third, we compute the closest point in each of the $l$ subspaces, $S_2, S_3, \ldots, S_l, S_{l+1}$, and the one that is closest to the center among the $l$ closest points represents the 2nd shortest path. Let it be $P_2 = (v_1, \ldots, v_{i+1})$, and assume it is in $S_i$. Fourth, we further divide $S_i$ into $r + 1$ subspaces, $S_{i1}, S_{i2}, \ldots, S_{ir}, S_{i,r+1}$, and compute the closest point in each of them, where $S_{i1}$ 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 $S_{i1}, S_{i2}, \ldots, S_{i,r+1}$ 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 $S$ is represented by a tuple $(P_{s,u}, X_u)$, 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 $P$ that take $P_{s,u}$ as prefix and exclude all edges of $X_u$.

The entire search space $S_0 = P$ is represented by $(P_{s,u} = (s), X_u = \emptyset)$. Assume the shortest path in subspace $(P_{s,u}, X_u)$ is $P$, then after choosing $P$ as one of the $k$ shortest paths, the subspace is divided into $l + 1$ subspaces, where 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 $(P_{s,v}, X_u \cup \{(u, w')\})$), and one subspace corresponding to each node $v$ in the subpath of $P$ from $u$ (exclusive) to the destination node (i.e., subspace $(P_{s,v}, \{(v, w')\})$), 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 $(P_{s,u}, X_u)$ from which they are divided.

**Example 4.1:** Consider a KPJ query $Q = \{v_1, \ldots, v_8\}$ on the graph in Fig. 1. Initially, $S_0 = \{(v_1)\}$ in which the shortest path is $P_1 = (v_1, v_2, v_7)$. Then, $S_0$ is divided into four subspaces, $S_1 = \{P\}, S_2 = \{(v_1)\}, S_3 = \{(v_1, v_2)\}, S_4 = \{(v_1, v_2, v_7)\}$, and $S_5 = \{(v_1, v_2, v_7)\}$, where $S_1$ is the subspace containing only $P_1$. The 2nd shortest path is the one with shortest lengths among shortest paths in $S_2, S_3, S_5$.

**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 remedied 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 $S = (P_{s,u}, X_u)$, we define the lower bound of a subspace, denoted $lb(S)$ (or $lb(P_{s,u}, X_u)$), as the lower bound of lengths of all paths in $S$, and denote the shortest path in a subspace by $sp(S)$ (or $sp(P_{s,u}, X_u)$).

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 $Q$. Each entry of $Q$ is a tuple $(S, lb(S), P)$, where $S$ and $lb(S)$ are a subspace and its lower bound, respectively, and $P$ is either $\emptyset$ or the shortest path in $S$. Subspaces in $Q$ are ranked by their lower bounds. Initially, $Q$ contains a single entry representing the entire search space $S_0$ (Line 1). Then, we iteratively remove the subspace with smallest lower bound from $Q$, denoted $(S, lb(S), P)$ (Line 4): if $P \neq \emptyset$, then $P$ is the
Algorithm 2: BestFirst\((G, Q = \{s, T, k\})\)

1. Initialize a minimum priority queue \(Q\) to contain a single entry \((h_S, 0)\), where \(h_S = h(s)\).
2. \(i = 1\);
3. while \(i \leq k\) do
4. \(S = (P_{s, u}, X_u, h(S), P)\) \(\leftarrow\) the top entry from \(Q\);
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. \(lb(S') \leftarrow \max(\text{CompLB}(P_{s, u}, X_v), \omega(P'));\)
9. \(\text{Put } (S', lb(S'), 0) \text{ into } Q;\)
10. else
11. \(\text{sp}(S) \leftarrow \text{CompSP}(P_{s, u}, X_u);\)
12. if \(\text{sp}(S) \neq \emptyset\) then
13. \(\text{Put } (S, \omega(\text{sp}(S)), \text{sp}(S)) \text{ into } Q;\)
14. return the \(k\) paths \(P_1, \ldots, P_k;\)

next shortest path to be output (Line 6), and we divide \(S\) by \(P\) and put those newly obtained subspaces into \(Q\) (Lines 7-10); otherwise, we compute the shortest path in \(S\) and put \(S\) into \(Q\) 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 \(Q\) 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. 1, respectively. Now, we assume that this holds for general \(k \geq 1\), then we prove that it also holds for \((k + 1)\). Since the \((k+1)\)th shortest path \(P_k\) corresponds to subspace \(S\) 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 \(S\) 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 \(Q\) 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 \(Q\), 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 \(Q\) 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 \(Q\) 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 \(Q\). Conceptually, Alg. 2: BestFirst computes shortest paths in all subspaces in \(Q\) and Alg. 2: BestFirst computes shortest paths in all subspaces in \(Q\).

Figure 4: Instances of shortest path computations

\[(a) \text{Best First} \quad (b) \text{Iteratively Bounding}\)

ally, 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.

### 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 \(lb(v, V_T);\)
5. \(lb \leftarrow \min\{lb, \omega(P_{s, u}) + \omega(u, v) + lb(v, V_T)\};\)
6. return \(lb;\)

**Computing Lower Bound of a Subspace.** Given a subspace \(S = (P_{s, u}, X_u)\), 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 \(S\) is \(\omega(P_{s, u}) + lb(u, V_T)\), where \(lb(u, V_T)\) 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 \(S\) (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 \(S_0\) into \(S_1, \ldots, S_4\), lower bounds of \(S_2, S_3, S_4\) are computed. Here, we illustrate how to compute \(lb(S_2) = lb(v_1, \{v_1, v_8\})\). \(v_1\) 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, \(lb(S_2)\) is computed as \(\min\{\omega(v_1, v_2) + lb(v_2, V_T), \omega(v_1, v_3) + lb(v_3, V_T),\omega(v_1, v_11) + lb(v_11, V_T)\}\).
bound \( lb(u, v) \) of shortest distance from \( u \) to \( v \) is estimated as,
\[
lb(u, v) = \max_{w \in L} \{ \delta(w, v) - \delta(w, u) \},
\]  
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, \( lb(u, V_T) \) can be estimated as,
\[
lb(u, V_T) = \min_{v \in V_T} \{ lb(u, v) \} = \min_{v \in V_T} \max_{w \in L} \{ \delta(w, v) - \delta(w, u) \}.
\]

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,
\[
\begin{align*}
lb(u, V_T) &= \max_{w \in L} \min_{v \in V_T} \{ \delta(w, v) - \delta(w, u) \} \\
&= \max_{w \in L} \min_{v \in V_T} \{ \delta(w, v) - \delta(w, u) \} \\
&= \max_{v \in V_T} \{ \min_{w \in L} \{ \delta(w, v) - \delta(w, u) \} \}
\end{align*}
\]

The intuition is that, \( \min_{v \in V_T} \{ \delta(w, v) - \delta(w, u) \} \) is the shortest distance from \( w \) to \( t \) in \( G_Q \) (i.e., \( \delta(w, t) \)); thus, Eq. (2) estimates \( \delta(w, t) \) for all \( w \) in \( L \), prior to any lower bound estimations. In what follows, we use Eq. (2) to compute \( lb(u, V_T) \).

Remarks & Time Complexity. Note that, the landmark index \( L \) is constructed offline in \( O(|L| (m + n \log n)) \) time where \( |m| = n + \log n \) 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 \( O(|L|) \); note that this is only computed once for each query.

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

Computing Shortest Path in a Subspace. We use A* search [16] to compute \( \text{sp}(P_{s,u}, X_u) \), 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 \( S_0 \), \( Q \) contains three subspaces, \( S_2 \), \( S_3 \), \( S_4 \), together with their lower bounds. Assume the lower bounds are \( lb(S_2) = 6 \), \( lb(S_3) = 11 \), and \( lb(S_4) = 12 \). \( S_2 \) has the smallest lower bound, and is removed from \( Q \). Then, \( \text{sp}(S_2) \) is computed, which is the 2nd shortest path \( P_2 \), since its length is smaller than lower bounds of subspaces in \( Q \). Here, we compute the 2nd shortest path without computing shortest paths in subspaces \( S_3 \) and \( S_4 \).

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 \( \tau \), which is achieved by a procedure TestLB. In a nutshell, given a subspace \( S \) and a threshold \( \tau \), TestLB tests whether the shortest path in \( S \) has a length larger than \( \tau \); if it is, then the lower bound is set as \( \tau \); otherwise, the shortest path \( \text{sp}(S) \) is obtained and returned. Details of TestLB will be discussed shortly. Ideally, we can set \( \tau \) 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 \( \tau \), we iteratively enlarge \( \tau \), and the \( k \) shortest paths of a KPJ query will be found once \( \tau \) becomes no smaller than \( \omega(P_k) \).

Algorithm 4: IterBound

1. Compute the shortest path \( P' \) from \( s \) to any node in \( V_T \) in \( G \);
2. Initialize a minimum priority queue \( Q \) to contain a single entry \( (S_0, ((s), \emptyset), \omega(P'), P') \);
3. \( i \leftarrow 1 \); \( \tau \leftarrow \omega(P') \);
4. while \( i \leq k \) do
5. \( (S = (P_{s,u}, X_u), lb(S), P, \tau) \leftarrow \text{remove the top entry from } Q; \)
6. if \( P \neq \emptyset \) then
7. Same as Lines 6-10 of Alg. 3: \( P \leftarrow P - P' \); \( i \leftarrow i + 1 \), and divide \( S \) into subspaces +/−;
8. else
9. \( \tau \leftarrow \alpha \cdot \text{max}(lb(S), \text{top}(\cdot).\text{key}); +/\text{Enlarge } \tau +/−; \)
10. \( P \leftarrow \text{TestLB}(P_{s,u}, X_u, \tau) \);
11. if \( P \neq \emptyset \) then Put \( (S, \omega(P), P) \) into \( Q; \)
12. else Put \( (S, \tau, \emptyset) \) into \( Q; \)
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 \( S_0 = ((s), \emptyset) \) with path \( P' \) into \( Q \) (Line 2), and initialize \( \tau \) as \( \omega(P') \) (Line 3). Then, we iteratively remove the subspace \( S \) with smallest lower bound, together with path \( P \), from \( Q \) (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 \( \tau \) (Line 9), test whether the shortest path of \( S \) has a distance larger than \( \tau \) (Line 10), and put \( S \) back into \( Q \) together with either the computed shortest path \( P \) or a larger lower bound \( \tau \) depending on what TestLB returns (Lines 11-12).

Note that \( \tau \) controls the computation of a tighter lower bound which we want to make larger but a larger \( \tau \) will make TestLB slow. In our approach, we use a parameter \( \alpha \) to control the speed of increasing \( \tau \) iteratively. Here, \( \alpha \) can be any real number larger than 1, and we use \( \alpha = 1.1 \) as default. At Line 9, we enlarge \( \tau \) as \( \alpha \cdot \text{max}(lb(S), \text{top}(\cdot).\text{key}); +/\text{Enlarge } \tau +/−; \)

where \( \text{top}(\cdot).\text{key} \) is the key value (i.e., lower bound) in the top entry of \( Q \) and is defined to be \( +\infty \) if \( Q = \emptyset \). The intuition is that, \( \omega(P_k) \) should be not much larger than \( \omega(P_i) \), the initial \( \tau \) (Line 3). Note that \( lb(S) \) is the previous \( \tau \) we have tested for \( S \); thus, the lower bound \( \tau \) we tested for a subspace increases by a factor of at least \( \alpha \). Therefore, we iteratively enlarge \( \tau \) from \( \omega(P_i) \) to approach \( \omega(P_k) \), and obtain the \( k \) shortest paths for a KPJ query once \( \tau \) becomes no smaller than \( \omega(P_k) \).
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 $\tau$ 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 $\tau$ will be no less than $\omega(P_i)$ when the algorithm terminates, and once $\tau$ becomes no less than $\omega(P_i)$, the path $P_i$ will be computed and inserted into $Q$, thus will be output.

Lemma 5.1: $\omega(\frac{\tau}{\tau}) = 1$, and returns 0 otherwise.

Algorithm 5: TestLB($P_{s,u}, X_u, \tau$)
1. Initialize a minimum-priority queue $Q_V$ to contain $(u, 0)$;
2. $d_s(u) \leftarrow \omega(P_{s,u})$, and $d_s(v) \leftarrow +\infty$ for all other nodes;
3. while $Q_V \neq \emptyset$ do
4. if $v \in V_T$ then return the path formed by concatenating $P_{s,u}$ with the computed shortest path from $u$ to $v$;
5. else for each outgoing edge $(v, w) \in V_T$ do
6. $d_s(w) \leftarrow d_s(v) + \omega(v, w)$;
7. Compute $lb(v, V_T)$;
8. if $d_s(v) + lb(v, V_T) \leq \tau$ then
9. Put $(w, d_s(w) + lb(w, V_T))$ into $Q_V$;
10. end
11. end
12. end
13. 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 $d_s(v) + lb(v, V_T)$) in a minimum-priority queue $Q_V$, which is initialized to contain $u$; nodes in $Q_V$ are ranked by their estimated distances. Then, nodes are iteratively removed from $Q_V$ (Line 4), and their neighbors are inserted into $Q_V$ (Lines 6-11), until $Q_V = \emptyset$ or we get a node from $V_T$; the latter case implies that $\mathcal{S}$ has been computed (Line 5). Here, $V_T$ is the set of nodes removed from $Q_V$. Note that, following from [16], when a node $v$ is removed from $Q_V$, $d_s(v)$ stores the shortest distance from $s$ to $v$ constrained in $\mathcal{S}$ (i.e., $d_s(s,v)$), and each node is removed from $Q_V$ at most once.

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

Lemma 5.1: Given a subspace $\mathcal{S}$ and $\tau$, TestLB returns $\hat{\mathcal{S}}(\mathcal{S})$ if $\omega(\hat{\mathcal{S}}) \leq \tau$, and returns 0 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 $\omega(\hat{\mathcal{S}}) \leq \tau$, then TestLB returns $\hat{\mathcal{S}}$, since every node that is pruned at Line 10 will not be in $\hat{\mathcal{S}}$ due to the nature of lower bound.

Secondly, we prove that if $\omega(\hat{\mathcal{S}}) > \tau$, then TestLB returns $\emptyset$. The reason is that, for every node $v$ obtained at Line 4 of Alg. 5, we have $d_s(v) < \tau$ due to the pruning at Line 10. Thus, Alg. 5 cannot find the path $\hat{\mathcal{S}}$, 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($v_1, v_3, \{v_1, v_2, v_3\}$, 6). $v_3$ has three valid out-neighbors, $v_4, v_5, v_6$, and $v_7$ are pruned because $d_{v_3}(v_4) + \omega(v_4, v_5) > 6$ and $d_{v_3}(v_5) + \omega(v_5, v_6) > 6$. Assume $lb(v_6, V_T) = 2$, then $v_6$ is also pruned since $d_{v_3}(v_6) + (\omega(v_3, v_6) + lb(v_6, V_T) = 7$. Thus, TestLB($v_1, v_3, \{v_1, v_2, v_3\}$, 6) returns $\emptyset$. Now, let’s consider $\tau = 7$. Among the three valid out-neighbors, $v_4$ is pruned because $d_{v_3}(v_4) + \omega(v_3, v_4) = 8$; $v_5$ and $v_6$ are put into $Q_V$ with lower bounds 7. Then, $v_5$ is removed from $Q_V$, and its out-neighbor, $v_6$, is put into $Q_V$ with lower bound 7. After that, either $v_4$ or $v_5$ is removed from $Q_V$, and the shortest path in $\{v_1, v_3\}$, $\{\{v_1, v_3\}\}$ 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 $\mathcal{P}_T$, which provides a more accurate approximation of $lb(v, V_T)$. Recall that $lb(v, V_T)$ 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 $\mathcal{P}_T$ 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 $Q_T$;
2. $\mathcal{P}_T \leftarrow$ a virtual root node $t$; /* Build $\mathcal{P}_T$ */;
3. for each $w \in V_T$ do
4. Put $(w, lb(s, w))$ into $Q_T$, $d_s(w) \leftarrow 0, p(w) \leftarrow t$;
5. while $Q_T \neq \emptyset$ do
6. $v \leftarrow$ remove the top node from $Q_T$;
7. Add $v$ as a child of $p(v)$ to $\mathcal{P}_T$; /* Build $\mathcal{P}_T$ */;
8. if $v = s$ then return the path from $s$ to $t$;
9. for each incoming edge $(w, v)$ do $v$ do
10. if $d_s(v) + \omega(w, v) > d_s(w)$ then
11. $d_s(w) \leftarrow d_s(v) + \omega(w, v), p(w) \leftarrow v$;
12. Put $(w, d_s(w) + lb(w, V_T))$ into $Q_T$;

The algorithm to construct $\mathcal{P}_T$ 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 $V_T$. $\mathcal{P}_T$ is similar to $Q_T$ in Alg. 5 and initially contains all nodes of $V_T$ (Lines 3-4). Then, nodes are iteratively removed from $Q_T$ (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 $Q_T$. 

139
For all nodes \( v \) removed from \( QT \), we add \( v \) as a child of \( p(v) \) to \( SPTP \). Intuitively, \( SPTP \) contains all nodes removed from \( QT \) prior to \( s \) when computing the shortest path from \( s \) to any node in \( V_T \).

**Proposition 5.1:** For nodes \( v \in SPTP \), the path from \( v \) to \( t \) in \( SPTP \) is the shortest path from \( v \) to any node in \( V_T \). \( \Box \)

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

![Figure 5: Partial and Incremental SPT](image)

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

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

### 5.3 Incremental Shortest Path Tree

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

**Constructing SPTI.** To prune from \( SPTI \) those nodes in \( V_T \) that are far-away from \( s \), in \( SPTI \) we compute and store shortest paths from \( s \) to each node of a subset of \( V \). Recall that, in \( SPTP \), we store shortest paths from each node of a subset of \( V \) to \( V_T \). Thus, the construction of \( SPTI \) is run on \( G \) by starting from \( s \), and consists of two phases. In the first phase, we construct an initial \( SPTI \), which is a by-product of computing the shortest path from \( s \) to any node in \( V_T \) in a similar fashion to \text{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 \( SPTI \) 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 \( SPTI \) all nodes of \( V \) that are on paths from \( s \) to any node in \( V_T \) whose lengths are not larger than \( \tau \). Therefore, IncrementalSPT iteratively removes the top node \( v \) from \( QT \) (Line 2), and adds \( v \) into \( SPTI \) (Line 3). Meanwhile, the subset of \( V_T \) that are in \( SPTI \) 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 \( SPTI \) constructed, and the entire tree is the resulting \( SPTI \) for \( \tau = 7 \). Here, \( D = \{v_6, v_7\} \). \( SPTI \) has the following property.

**Proposition 5.2:** The \( SPTI \) constructed by Alg. 7 contains all nodes on paths from \( s \) to any node in \( V_T \) whose lengths are not larger than \( \tau \). \( \Box \)

**Algorithm 8: CompLB-SPTI \((P_{i,u}, X_u)\)**

1. if \( u \neq t \) then \( N(u) \leftarrow \{\text{in-neighbors of } u\} \); else \( N(u) \leftarrow D; \)
2. \( lb \leftarrow +\infty; \)
3. for each \( v \in N(u) \) do
   4. if \( v \notin P_{i,u} \) and \( (v, u) \notin X_u \) then
      5. if \( v \notin SPTI \) then Compute \( lb(v, s) \) using Eq. (2);
      6. else \( lb(v, s) \) is the distance from \( v \) to \( s \) in \( SPTI \);
      7. \( lb \leftarrow \min\{lb(v, w) + \omega(v, u) + lb(s, v)\}; \)
6. if \( u = t \) and \( lb = +\infty \) then \( D \neq V_T \) then \( lb \leftarrow 0; \)
7. return \( lb; \)

**Computing Initial Lower Bound for a Subspace using SPTI.** We compute the lower bound of a subspace using \( SPTI \) in a similar way to CompLB (Alg. 3), by considering the immediate neighbors of \( u \). The algorithm is shown in Alg. 8, denoted CompLB-SPTI. Note that, here \( P_{i,u} \) is a path from \( u \) to \( t \) in \( G \) and \( X_u \) is a subset of the incoming edges to \( u \). CompLB-SPTI differs from CompLB in the following two aspects. Firstly, \( lb(v, s) \) is estimated by utilizing \( SPTI \) 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 \( SPTI \) (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 \( lb(P_{i,u}, X_u) \), which saves a lot of computations.

The correctness of CompLB-SPTI 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 \( P_{i,u} \) or in \( X_u \), Lines 3-4), then \( lb = +\infty \). 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 \( \langle P_{i,u}, X_u \rangle \).

**Testing Lower Bound using SPTI.** By utilizing \( SPTI \), we propose a more efficient algorithm for testing lower bound, denoted TestLB-SPTI. We modify TestLB to develop TestLB-SPTI in the same fashion as the development of CompLB-SPTI. Moreover, we prune all nodes that are not in \( SPTI \) from consideration (i.e. putting into \( QT \)). Consequently, every \( lb(s, w) \) is computed as the distance from \( s \) to \( w \) in \( SPTI \); that is, Eq. (2) is not evaluated. Therefore, TestLB-SPTI is more efficient than TestLB. We prove
the correctness of TestLB-SPT\textsubscript{1} in the following lemma.

Lemma 5.2: Given a subspace \( S \) and \( \tau \), TestLB-SPT\textsubscript{1} returns \( \emptyset \) if \( \omega(sp(S)) \geq \tau \), and returns \( sp(S) \) otherwise. \( \square \)

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

Example 5.3: We show running TestLB-SPT\textsubscript{1} for subspace \( \{(v_7),\{(v_7), v_8\}\} \) with \( \tau = 6 \), where SPT\textsubscript{1} is shown in Fig. 5(b). Among \( v_7\)'s in-neighbors, only \( v_3 \) is considered, since \( v_1, v_3 \) and \( v_4 \) are not in SPT\textsubscript{1}. For \( v_3, lb(v_1, v_3) = 3 \) which is the distance in SPT\textsubscript{1}. Then, \( v_3 \) is also pruned since \( \omega(v_1, v_3) + lb(v_1, v_3) = 7 \), and TestLB-SPT\textsubscript{1} returns \( \emptyset \). For \( \tau = 7 \), SPT\textsubscript{1} remains the same. Then, \( v_3 \) is not pruned and the shortest path in \( \{(v_7),\{(v_7), v_8\}\} \) is obtained, which is \( \{(v_1, v_3, v_7)\} \) with length 7. \( \square \)

IterBound-SPT\textsubscript{1} Approach. Based on SPT\textsubscript{1} and the discussions above, we propose an approach following Alg. 4 for processing KPJ queries, denoted IterBound-SPT\textsubscript{1}. It runs on the reverse graph of \( G \), and a subspace is represented by \( \{P_i, u, X_u\} \) where \( X_u \) is a subset of the incoming edges to \( u \).

IterBound-SPT\textsubscript{1} improves the efficiency by computing SPT\textsubscript{1} and pruning all nodes not in SPT\textsubscript{1} from consideration when conducting the iteratively bounding search. That is, we take as input only the small subgraph of \( G \) induced by nodes in SPT\textsubscript{1}. Note that, SPT\textsubscript{1} enlarges for a larger \( \tau \), and the subgraph induced by nodes in SPT\textsubscript{1} also enlarges; this guarantees that we can correctly process any KPJ query.

Time Complexity. The time complexity of the IterBound-SPT\textsubscript{1} 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 \( d'(w) + lb(w, V) \leq \tau \) (see Line 10 of Alg. 5) for the largest \( \tau \) obtained in IterBound-SPT\textsubscript{1}. Note that, \( n' \) and \( m' \) are usually small in real applications; thus, IterBound-SPT\textsubscript{1} 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 \( lb(u, V_T) \). Nevertheless, when landmarks are not available, all our techniques can still be directly applied by setting all \( lb(u, V_T) \) (i.e., computed by Eq. (2)) to 0. Specifically, for IterBound-SPT\textsubscript{1}, the landmark is only used for constructing the SPT\textsubscript{1} using A* search [18], as discussed in Section 5.3; thus, without landmark, we construct the SPT\textsubscript{1} by setting \( lb(u, V_T) \) to 0 (the A* search then becomes the Dijkstra’s algorithm [11]), while other parts of IterBound-SPT\textsubscript{1} remain the same.

Moreover, without landmark, our techniques still perform well; the reasons are as follows. The IterBound-SPT\textsubscript{1} approach mainly consists of two parts: 1) incrementally constructing the partial shortest path tree SPT\textsubscript{1}, 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 IterBound-SPT\textsubscript{1} 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\textsubscript{1} [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, IterBound\textsuperscript{a}, and IterBound\textsuperscript{b}. They are the iteratively bounding approaches with or without SPT\textsubscript{1} as discussed in Section 5. IterBound\textsuperscript{a} and IterBound\textsuperscript{b} are abbreviations of IterBound-SPT\textsubscript{a} and IterBound-SPT\textsubscript{b}, respectively.
- IterBound\textsuperscript{-NL}. It is the IterBound\textsuperscript{a} 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.

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

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 ⇢ T_2 ⇢ T_3 ⇢ 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 ω(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.\footnote{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.}

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, for i < j. We use Q_2 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 IterBound\(_d\).

Choosing |L|. In our approaches, we use landmarks for estimating IB(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].\footnote{We first 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.} Fig. 6(a) shows the processing time of IterBound\(_d\) 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 IB(v, V_T). However, when |L| increases from 16 to 32, the processing time increases a little due to longer computation time of IB(v, V_T). Therefore, we choose |L| = 16.

Choosing α. The running time of IterBound\(_d\) 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 SPT. 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 α = 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.

![Figure 6: Parameter testing on CAL (Q_2, k = 20)](image)

Figure 7(a) and 7(b) show the processing time of our approaches with respect to the number of shortest path computations. In Figures 7(a)-(d), DA-SPT outperforms DA because DA-SPT online builds a full SPT to facilitate the shortest path computation. However, in Fig. 7(e) and 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, IterBound\(_d\)-NL outperforms all other approaches except IterBound\(_d\) 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 Queries. We test the approaches for processing KSP queries by setting the destination category as “Glacier” which has only one
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, IterBoundI outperforms IterBound-NL, which demonstrates the effectiveness of using landmarks for estimating lower bounds. Therefore, in the following testings, we omit DA, DA-SPT, and IterBound-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, IterBoundp, and IterBoundI, on SJ and COL.

**Vary Q.** The processing time of the approaches on SJ and COL by varying Q is shown in Fig. 9. Similar to that in Fig. 7, the running time of these four approaches increases when either Q varies from Q1 to Q2, or k increases. IterBoundI slightly outperforms BestFirst due to less number of shortest path computations, however with more expensive lower bound computations. IterBoundp performs better than IterBound because of the faster lower bound testing. IterBoundp runs faster than IterBoundI because IterBoundI can further reduce the exploration area of a graph by SPT.

**Vary k.** The processing time of the approaches on SJ and COL by varying 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 Q1 to Q2, or k increases. IterBoundI slightly outperforms BestFirst due to less number of shortest path computations, however with more expensive lower bound computations. IterBoundp performs better than IterBound because of the faster lower bound testing. IterBoundp runs faster than IterBoundI because IterBoundI can further reduce the exploration area of a graph by SPT.

**Summary.** IterBoundI outperforms the other approaches, BestFirst, IterBoundp, and IterBoundI, across all different datasets, different number of destination nodes, different Q, and different k. Thus, in the following we only evaluate our IterBoundI approach.

**Eval-IV: Scalability Testing.** The scalability testing results of IterBoundI by varying graph size and k are shown in Fig. 12. Although the running time increases when either the graph size or k increases, IterBoundI 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 IterBoundI only increases slightly (e.g., by no more than 3 times).

**Eval-V: KPJ Testing.** In this evaluation, we test the efficiency of IterBoundI over DA-SPT, the state-of-the-art approach, for KPJ.
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$I$ are similar to the previous evaluations. The improvement of IterBound$I$ 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 KPJ 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