Finding Top-k Shortest Paths with Diversity

Huiping Liu, Cheqing Jin ✉, Bin Yang, and Aoying Zhou

Abstract—The classical K Shortest Paths (KSP) problem, which identifies the k shortest paths in a directed graph, plays an important role in many application domains, such as providing alternative paths for vehicle routing services. However, the returned k shortest paths may be highly *similar*, i.e., sharing significant amounts of edges, thus adversely affecting service qualities. In this paper, we formalize the K Shortest Paths with Diversity (KSPD) problem that identifies top-k shortest paths such that the paths are *dissimilar* with each other and the total length of the paths is minimized. We first prove that the KSPD problem is NP-hard and then propose a *generic* greedy framework to solve the KSPD problem in the sense that (1) it supports a wide variety of path similarity metrics which are widely adopted in the literature and (2) it is also able to efficiently solve the traditional KSP problem if no path similarity metric is specified. The core of the framework includes the use of two judiciously designed lower bounds, where one is *dependent* on and the other one is *independent* on the chosen path similarity metric, which effectively reduces the search space and significantly improves efficiency. Empirical studies on 5 real-world and synthetic graphs and 5 different path similarity metrics offer insight into the design properties of the proposed general framework and offer evidence that the proposed lower bounds are effective.

Index Terms—Top-k shortest paths, diversified top-k query, path finding, path similarity, path diversity.

1 Introduction

PATH-FINDING [6], [26] is one of the most prominent and ubiquitous requirements of our daily lives. Although shortest path finding has enabled highly useful services in many application domains, we are still witnessing increasing interests in other types of path-finding [14], [35], [41], [42], [47]. For instance, if the shortest path involves many traffic lights, a driver may prefer to use a slightly longer path but with less traffic lights. Such applications call for efficiently finding the top-k shortest paths, but not merely the shortest one, which provides flexibilities for users to make personalized decisions.

The classical k shortest path problem (KSP) [3], [19], [30], [45] returns the top-k shortest paths between a source and destination pair, which has been widely adopted in many applications, including path recommendation, robot motion planning, and candidate paths selection in scheduling. However, the top-k shortest paths may be quite similar [2], due to a large number of shared edges among the top-k shortest paths. This is undesired as it adversely reduces the provided flexibilities. For instance, if an accident happens on an edge and thus blocks the traffic, and the edge is shared by all top-k shortest paths, then all the returned k paths become unavailable and need to be recomputed. This situation is undesired and should be avoided especially in the applications of hazardous material shipments [9], evacuation routing [34], robotic motion planning [40] and wireless sensor networks [29]. In addition, highly overlapped paths reduce the possibility that such paths suffice to satisfy various users' diverse driving preferences. To contend with the aforementioned limitations, efficiently finding *spatially dissimilar* top-k shortest paths from a source to a destination is called for.

- H. Liu, is with the School of Computer Science and Software Engineering, East China Normal University, Shanghai, China. E-mail: hpliu@stu.ecnu.edu.cn
- C. Jin and A. Zhou are with the School of Data Science and Engineering, East China Normal University, Shanghai, China. E-mail: {cqjin, ayzhou}@sei.ecnu.edu.cn
- B. Yang is with the Department of Computer Science, Aalborg University, Aalborg, Denmark. E-mail:byang@cs.aau.dk

Figure 1 shows a weighted, directed graph. The top-4 shortest simple paths (i.e., paths without cycles) from source A to destination D are listed in Table 1. When k is set to 3, the classical KSP problem returns P_1, P_2, and P_3. However, P_2 and P_3 are highly similar as they share two long sub-paths $A \to B$ and $H \to E \to D$. Thus, returning both P_2 and P_3 to users does not significantly increase the flexibility for the users. In contrast, the fourth shortest path P_4, though slightly longer, is more different from the first two. In this sense, P_1, P_2, and P_4 form the top-3 shortest paths that are mutually dissimilar, which may satisfy more users with diverse preferences.

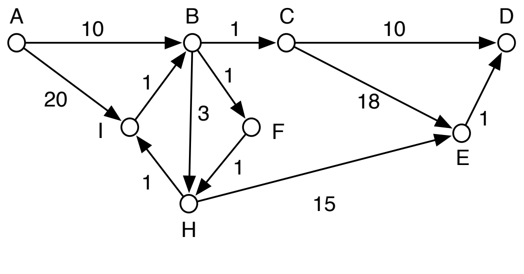


Fig. 1. A weighted, directed graph G

TABLE 1
Top-4 shortest simple paths from A to D in Fig. 1

	Path	Length
P_1	$A \to B \to C \to D$	21
P_2	$A \to B \to F \to H \to E \to D$	28
P_3	$A \to B \to H \to E \to D$	29
P_4	$A \to B \to C \to E \to D$	30

To this end, we consider the problem of finding top-k shortest simple paths with diversity (KSPD), where the similarity between any pair of the returned k paths must be below a threshold that is specified by users and the total length of the returned k paths should be minimized. However, solving the KSPD problem is non-trivial as there may exist a huge number of paths between a source and a destination and thus it may take prohibitively long time to enumerate all such paths. Moreover, we prove that the KSPD problem is NP-hard (see Section 4) and we propose an approximation algorithm for solving the KSPD problem with provable approximation ratios.

In particular, we propose a general framework to efficiently solve the KSPD problem. The framework enables us to choose the next most promising path, instead of the next shortest path, to explore. The "promisingness" of a path is defined based on a path length lower bound that is derived by the chosen path similarity metric and user specified threshold. The framework is designed in a way which is loosely coupled with the choices of path similarity metrics, and thus is applicable to a wide variety of path similarity metrics. In addition, the framework utilizes a reverse shortest path tree [8] from the destination to estimate path length lower bounds for partially-explored paths. Moreover, we also introduce *path classification* that groups partially-explored paths into different classes. These together enable effective pruning of considerable partially-explored paths and thus enable efficient enumeration of next shortest paths. Thus, the framework is also able to improve the efficiency of the classical KSP problem when no path similarity metric is specified.

To the best of our knowledge, this paper is the first to consider the KSPD problem and propose a general framework that efficiently solves both the KSP and KSPD problems and supports a wide variety path similarity metrics when solving the KSPD problem. In particular, the paper makes the following contributions. First, we formally define the top-k shortest simple paths with diversity (KSPD) problem and prove that the KSPD problem is NP-hard. Second, we propose a general greedy framework for efficiently solving the KSPD problem, which supports various path similarity metrics. Moreover, when no path similarity metric is specified, the proposed framework is also able to efficiently solve the traditional KSP problem. Third, we carefully design two path length lower bounds and a set of heuristics, which effectively prune a large number of unnecessary partially-explored paths and in turn improves the efficiency. Fourth, we conduct a comprehensive empirical study on five real-world and synthetic graphs and five different path similarity metrics to demonstrate that our proposal is efficient and effective.

The remainder of this paper is organized as follows. Section 2 summarizes related works. Section 3 gives the preliminary of our work and defines the KSPD problem. Section 4 proves the NP-hardness of KSPD problem and proposes a heuristics idea for KSPD problem. Section 5 introduces our proposed framework for KSPD problem. Section 6 and Section 7 elaborate the two lower bounds respectively, used in our framework. Section 8 gives our final algorithm in detail. Section 9 reports the evaluation and a brief conclusion is given in Section 10.

2 Related work

In general, our problem on finding top-k shortest simple paths with diversity is related to three problems in the literature, including the top-k shortest paths, diversified top-k query, and diversified paths finding.

The top-k shortest paths The KSP problem aims at finding the top-k shortest paths in weighted graphs. Yen proposes a general method to get the k shortest simple paths in weighted directed networks [45]. Initially, the shortest path is computed. Subsequently, all candidate paths that deviate from the shortest path are generated by performing Dijkstra's algorithm for many times, among which, the shortest one is selected as the next shortest simple path. The above steps repeat until k shortest paths have been found. Several prior works [10], [30] try to improve Yen's algorithm. [30] divides the previous shortest paths into different equivalence classes, then the candidate paths in each equivalence class can be found by a replacement-path-based algorithm. Finally, the next shortest path is selected among the candidate paths. [10] divides all simple paths from the source to destination into smaller subspaces, and each candidate shortest path comes from a subspace. For each subspace, they iteratively compute and tighten its corresponding lower bound, then candidate shortest paths in the subspaces are computed in best-first paradigm based on their lower bounds, so that lots of subspaces can be pruned without the time-consuming shortest path computations. Although the above improvements are efficient in practice, they share the same worst-case complexity as Yen's algorithm.

Finding top-k general shortest paths is studied in [3], [19], [32], where undirected graphs and cycles in paths are allowed. In undirected graphs, [32] proposes a much faster algorithm to find the top-k shortest simple paths by using a path branching structure. For the case that cycles are allowed in a path, the best result is an algorithm proposed by [19]. By using an indexing method, [3] efficiently answers the general top-k distance queries on large networks.

However, none of the above studies considers path diversity.

Diversified top-k query Diversified top-k query, which aims at computing the top-k results that are most relevant to a user query by taking the diversity into consideration, has been extensively studied in a wide variety of spectrum, such as, diversified keyword search in documents [5], structured databases [15] and graphs [25], diversified top-k pattern matchings [22], cliques [46] and structures [31] in a graph, and so on. However, due to different problem natures, none of the above approaches can be used to efficiently find the top-k diversified paths. A survey for different query result diversification approaches is provided by Drosou and Pitoura [18]. The complexity of query result diversification is analyzed by Deng and Fan [16]. Some other works [7], [36], [37], [39] focus on a general framework for top-k answer diversification. [7], [36], [39] focus on maximizing the diversity of k results in a given set, while in our problem, we only require that the similarity of any two paths in the result set should be less than a given threshold and then their length should be as short as possible. [37] finds the top-k diversified results, where the similarity of each pair of results should be less than a given threshold and the total score of all results should be maximized. Although the problem in [37] is similar to our problem in this paper, they are still different due to different object function. In our problem, we aim at finding the top-k diversified paths with minimal total length. Accordingly, all above frameworks cannot be directly applied to finding top-k shortest simple paths with diversity.

Diversified paths finding [2], [11], [38] studies how to find the dissimilar paths with different similarity functions, while our paper also requires to explore the graph to find the *shortest* simple paths. [29] and [40] focus on the diverse short paths in wireless sensor networks and robotic motion planning, respectively. In [29], edges in previous shortest paths are removed from the graph to get the next diverse shortest path to achieve trustful communication levels. In [40], edges near previous shortest paths are removed to avoid obstacles. Both works cannot be applied to our problem due to

TABLE 2
Notation

Notation	Meaning
$G(V, E)$	graph with vertex set V and edge set E
m, n	cardinality of edge set E and vertex set V
$P(s, t)$	path from s to t
$L(P)$	length of path P
S_P	edge set of path P
$Sim(P_i, P_j)$	similarity between path P_i and P_j
τ	similarity threshold
k	top k results are needed
$LB_1(P)$	shortest path lower bound of path P
$LB_2(P)$	diversified path lower bound of path P

different problem and application domains. Skyline path queries identify non-dominated paths when considering multiple travel costs [4], [27], [43], while our paper considers a single travel cost.

3 Preliminaries

We introduce important concepts and formalize the problem. Frequently used notation is summarized in Table 2.

Definition 1 (Graph)**.** *A directed weighted graph $G = (V, E)$ includes a vertex set V and an edge set $E \subseteq V \times V$, where $|V| = n$, and $|E| = m$. Edge $e = (u, v) \in E$, which connects vertex $u \in V$ to vertex $v \in V$, is associated with a nonnegative weight, denoted as $w(e)$.*

Definition 2 (Path)**.** *A path from vertex s to vertex t in graph G is a sequence of vertices, where each two adjacent vertices are connected by an edge, denoted by $P(s, t) : s = v_1 \to v_2 \to \cdots \to v_q = t$, where $s = head(P)$ and $t = tail(P)$ are the head and tail vertices of path P. Path P is a simple path if P consists of distinct vertices. The set of edges on path P is defined as $S_P = \{(v_i, v_{i+1})|1 \le i < q\}$. $\forall i, j$ $(1 \le i < j \le q)$, $P(v_i, v_j)$ is the subpath of $P(s, t)$ from v_i to v_j. We use '+' to concatenate two subpaths if the tail vertex of a subpath is the head vertex of another subpath.*

Definition 3 (Length of Edge Set)**.** *The length of an edge set S is the sum of the edge weights in S, i.e., $L(S) = \sum_{e \in S} w(e)$.*

Then, the length of path P, denoted as $L(P)$ equals to $L(S_P)$. To find dissimilar paths, we next formally define the similarity of paths.

Definition 4 (Path similarity)**.** *The similarity between two paths P_i and P_j, denoted as $Sim(P_i, P_j)$, can be defined using different similarity functions. Such a similarity function returns a value that ranges from 0 to 1 such that the more edges shared by the two paths, the higher value it returns.*

We consider a wide variety of widely used path similarity functions in the literature [2], [12], [13], [20], [21], [44], as summarized in Table 3. All the path similarity functions return a value between 0 and 1. If two paths are identical, their similarity is 1; while if two paths share no edges, their similarity is 0. Note that all similarity functions use $L(S_{P_i} \cap S_{P_j})$, i.e., the length of the common edges between two paths. We call $L(S_{P_i} \cap S_{P_j})$ the *intersection length*.

In particular, function $Sim_1(\cdot, \cdot)$ returns a ratio between the intersection length and the length of the union of the two paths, which can be regarded as a weighted Jaccard similarity [33]; function $Sim_2(\cdot, \cdot)$ is the arithmetic average of two ratios—the ratio between the intersection length and the length of path P_i and the ratio between the intersection length and the length of

TABLE 3
Similarity functions

Notation	Definition	References	$Sim(P_2, P_3)$ in Table 1
$Sim_1(P_i, P_j)$	$\frac{L(S_{P_i} \cap S_{P_j})}{L(S_{P_i} \cup S_{P_j})}$	[13], [20], [21], [44]	$\frac{26}{31} = 0.84$
$Sim_2(P_i, P_j)$	$\frac{L(S_{P_i} \cap S_{P_j})}{2L(P_i)} + \frac{L(S_{P_i} \cap S_{P_j})}{2L(P_j)}$	[2], [20], [21]	$\frac{26}{56} + \frac{26}{58} = 0.91$
$Sim_3(P_i, P_j)$	$\sqrt{\frac{L(S_{P_i} \cap S_{P_j})^2}{L(P_i)L(P_j)}}$	[20], [21]	$\sqrt{\frac{26^2}{28 \times 29}} = 0.91$
$Sim_4(P_i, P_j)$	$\frac{L(S_{P_i} \cap S_{P_j})}{\max\{L(P_i), L(P_j)\}}$	[20], [21]	$\frac{26}{29} = 0.90$
$Sim_5(P_i, P_j)$	$\frac{L(S_{P_i} \cap S_{P_j})}{\min\{L(P_i), L(P_j)\}}$	[12]	$\frac{26}{28} = 0.93$

path P_j; function $Sim_3(\cdot, \cdot)$ is the geometric average of the aforementioned two ratios; function $Sim_4(\cdot, \cdot)$ (or $Sim_5(\cdot, \cdot)$) returns a ratio between the intersection length and the length of the longer (or shorter) path of the two paths.

The paper's proposal is a *general* framework that solves the problem of identifying k-shortest paths with diversity, which is loosely coupled with the choice of similarity functions. Although we use the above similarity functions to exemplify the proposed framework, the framework itself is not limited to the specific similarity functions. Without loss of generality, in the following discussions, we use $Sim_1(\cdot, \cdot)$ as the default similarity function to exemplify the paper's proposal.

Next, we formally define the KSPD problem below.

Definition 5 (Top-k Shortest Paths with Diversity $KSPD(G, s, t, Sim(\cdot, \cdot), k, \tau)$)**.** *Let s, $t \in G.V$ denote two vertices in graph G, $Sim(\cdot, \cdot)$ be a path similarity function, and $\tau \in [0, 1]$ be a similarity threshold and k be an integer.*

We introduce Ω to denote a path set that consists of all simple paths from source vertex s to target vertex t. Next, we introduce $\xi \in 2^{\Omega}$ to denote a set of dissimilar path sets. In particular, for each dissimilar path set $\Psi \in \xi$, the similarity of every pair of paths in Ψ is not greater than the threshold τ, i.e., $\forall P_i, P_j \in \Psi$, $Sim(P_i, P_j) \le \tau$.

The KSPD problem aims at finding a dissimilar path set $\Psi \in \xi$ such that $|\Psi| \le k$, and there does not exist another dissimilar path set $\Psi' \in \xi$ with $|\Psi'| \le k$ such that

- *$|\Psi'| > |\Psi|$; or*
- *$|\Psi'| = |\Psi|$ and $\sum_{P_i \in \Psi'} L(P_i) < \sum_{P_j \in \Psi} L(P_j)$.*

The intuitions of Definition 5 are two folds. First, the result dissimilar path set Ψ of KSPD should have the maximal cardinality constrained by k. Second, its total length of paths should be minimized.

4 NP-hardness of KSPD

We first prove that the KSPD problem is NP-hard and then propose an approximation algorithm to solve the KSPD problem.

Lemma 1. *The KSPD$(G, s, t, Sim(\cdot, \cdot), k, \tau)$ problem is an NP-hard problem.*

Proof. We prove Lemma 1 by reducing the KSPD problem to the *Maximum Independent Set (MIS)* problem which is a well-known NP-hard problem [23]. MIS aims to find a maximum vertex set in a graph, such that each pair of vertices in this set cannot be connected by an edge.

Following the notation used in Definition 5, we use Ω to denote a path set that consists of all simple paths from s to t on G. In order to reduce the KSPD problem to the MIS problem, we need to construct a graph G'. In particular, each vertex in graph G' represents a path in Ω, then we connect two paths P_i and P_j by an edge (P_i, P_j) if $Sim(P_i, P_j) > \tau$. Next, we consider a special case of the KSPD problem, where all paths in Ω have the same length, e.g., $\forall P, L(P) = 1$, and $k = |\Omega|$. Then, finding KSPD in G is equivalent to finding the maximum independent set on G', which is an NP-hard problem. Thus, the KSPD problem is also an NP-hard problem. A concrete example of reducing KSPD to MIS is included in the supplementary document [1]. $\square$

In this paper, we propose a greedy algorithm to incrementally find an approximate result to solve the KSPD problem. Algorithm 1 shows the pseudo code of the greedy algorithm. First, we add the shortest path into the result set (lines 1–4). Next, in a greedy fashion, we always pick the next shortest path and insert it into the result set if the path is qualified (lines 5–9). Here, a path is qualified if it is dissimilar with paths that are already in the result set. Finally, the process stops when the result set has already k diversified paths or all paths in Ω have been examined (line 5).

Algorithm 1: $ApproximateKSPD(G, s, t, Sim(\cdot,\cdot), k, \tau)$

```
Input: Graph $G$, source $s$, destination $t$ similarity function
       $Sim(\cdot,\cdot)$, $k$ and $\tau$.
Output: The top-$k$ diversified shortest simple paths from $s$
        to $t$.
1  $\Omega \leftarrow$ set of all simple paths from $s$ to $t$;
2  $P \leftarrow$ the shortest path in $\Omega$;
3  $\Omega \leftarrow \Omega \setminus \{P\}$;
4  $\Psi \leftarrow \{P\}$;
5  while $|\Psi| < k$ and $|\Omega| > 0$ do
6      $P' \leftarrow$ the next shortest path in $\Omega$;
7      $\Omega \leftarrow \Omega \setminus \{P'\}$;
8      if $\forall P'' \in \Psi$, $Sim(P'', P') \le \tau$ then
9          $\Psi \leftarrow \Psi \cup \{P'\}$;
10 return $\Psi$;
```

Note that Algorithm 1 cannot always find the optimal results , i.e., Top-k Shortest Paths with Diversity according to Definition 5. An example is included in the supplementary document [1]. Lemma 2 shows the the approximate ratios of the proposed greedy algorithm.

Lemma 2. *Let Ψ denote the set of paths return by Algorithm 1 and let Ψ^* denote the set of paths returned by an exact algorithm. The approximate ratio of Algorithm 1 is defined in two-fold as the KSPD problem is defined as two sub-problems. The first sub-problem is a maximization problem, which identifies dissimilar path sets with the largest cardinality while being not greater than k. The second sub-problem is a minimization problem, which identifies, among all dissimilar path sets that are with the largest cardinality, the set with the smallest total length. For the first sub-problem, the approximation ratio of Algorithm 1 is k; for the second sub-problem, the approximation ratio of Algorithm 1 is $\frac{\max_{P\in\Psi}\{L(P)\}}{L(P_1)}$, where $L(P_1)$ is the length of the shortest path. Note that both approximation ratios are not constant.*

Proof. For the first sub-problem, Algorithm 1 always include the shortest path in Ψ. In the worst case, $|\Psi^*| = k$, i.e., there are k dissimilar shortest path, while Algorithm 1 returns Ψ that only includes the shortest path. Thus, the approximation ratio for the first sub-problem is $\alpha = \frac{|\Psi^*|}{|\Psi|} \le \frac{k}{|\Psi|} \le k$.

Next, we consider the second sub-problem when $|\Psi| = |\Psi^*|$. The approximate ratio α between approximate result Ψ and exact result Ψ^* is $\alpha = \frac{\Sigma_{P\in\Psi} L(P)}{\Sigma_{P^*\in\Psi^*} L(P^*)} \le \frac{|\Psi|\cdot\max_{P\in\Psi}\{L(P)\}}{|\Psi^*|\cdot\min_{P^*\in\Psi^*}\{L(P^*)\}} \le \frac{|\Psi|\cdot\max_{P\in\Psi}\{L(P)\}}{|\Psi^*|\cdot L(P_1)} = \frac{|\Psi|}{|\Psi^*|}\frac{\max_{P\in\Psi}\{L(P)\}}{L(P_1)} = \frac{\max_{P\in\Psi}\{L(P)\}}{L(P_1)}$. $\square$

The worst case of the approximate ratio for the first sub-problem can be k. This occurs when Algorithm 1 only returns the shortest path and the optimal result consists of k diversified paths. However, in practice, k is not large. Further, this worst case seldom happens in real world graphs (see experimental results reported in Section 9).

5 A General solution framework

We first propose a baseline algorithm that applies the greedy strategy. Next, we propose a general framework that utilizes two lower bounds to further improve the performance.

5.1 Baseline algorithm

The baseline method based on the greedy strategy enumerates shortest paths using existing algorithms that solve the KSP problem, e.g., Yen's algorithm [45]. In particular, in line 6 of Algorithm 1, we call Yen's algorithm to obtain the next shortest path.

In the Yen's algorithm, we call Dijkstra's algorithm [17] to identify the shortest path from source s to destination t, denoted as $P_1(s,t)$, and insert it into an empty priority queue where the priority is based on path lengths. If the priority queue is not empty, we keep calling the following procedure until we get k simple paths. First, we get the path with the shortest length in the priority queue as the next shortest path $P_i(s,t)$. Second, for each edge (u, u') in $P_i(s,t)$, we repeatedly call Dijkstra's algorithm to identify the shortest path $P'(u,t)$ from u to t via a different outgoing edge (u, u'') where $u' \ne u''$ and without using any vertex in $P_i(s,u)$ to avoid cycles. A candidate path $P''(s,t)$ is generated by concatenating $P_i(s,u)$ and $P'(u,t)$. We call P_i, u, and (u,u'') the *deviation path*, *deviation vertex* and *deviation edge* of $P''(s,t)$, respectively, because $P''(s,t)$ deviates from $P_i(s,t)$ at u via (u,u''). Third, all obtained candidate paths are added to the priority queue.

Suppose κ is the number of shortest simple paths that are evaluated by the baseline method to find the k diversified shortest paths. The complexity of the baseline method by using Yen's algorithm is $O(\kappa n(m + n\log n))$, because $O(n)$ times of Dijkstra's algorithms are invoked to find the next shortest simple path. Note that $\kappa \gg k$ often occurs.

The baseline algorithm is inefficient, especially when τ is small or k is large. Because some shortest paths can never be qualified before they reach the destination due to high overlaps with the diversified paths that are already in Ψ, meaning that there is no need to explore such paths. This motivates us to early prune such unpromising paths that can never be qualified.

5.2 Our Solution Framework

To efficiently find the top-k shortest simple paths with diversity, we try to improve the performance in two aspects. The first one is on finding the shortest paths and the other is on finding the diversified paths. To achieve this goal, we propose a framework (Figure 2) by using two lower bounds. Instead of directly generating the shortest *complete paths* from the source to the destination by

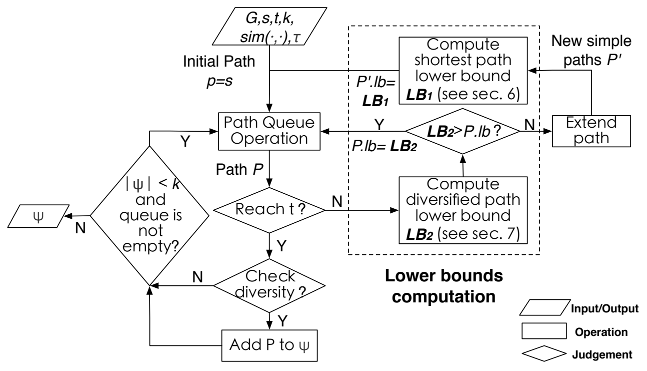


Fig. 2. Our general framework for KSPD problem

baseline method, our framework maintains all *partially explored paths* that are from source but have not reached destination during our process, and each path is attached with a lower bound lb that initially equals to its length. All paths are kept in a priority queue sorted by their lower bounds in ascending order.

The proposed general procedure adopts a best-first search strategy. Each time we get the path P with the minimal lower bound in the queue. If it has not reached the destination yet, we extend path P by creating new simple paths that are derived from P via $tail(P)$'s neighbors (where loop paths are discarded), and add these paths to the queue. Otherwise, we check if it is feasible for P to become a diversified path. If so, we add it into the result set. The above process does not terminate until k diversified paths have been found or the path queue is empty.

To accelerate the above process, we introduce t-wo lower bounds, namely *shortest path lower bound* and *diversified path lower bound*. For each newly created partially explored path, we compute its shortest path lower bound that is the lower bound of the length of its shortest simple path that reaches the destination, so that we can efficiently find the shortest complete paths by exploring partially explored paths according to their shortest path lower bounds like A* algorithm [28]. On the other hand, we also compute the diversified path lower bound for each partially explored path according to the similarity function and threshold before extending, such that the longer the length of a path's overlap (with the paths in the result set) is, the greater its diversified path lower bound is. So the paths that highly overlap with paths in the result set can be postponed to be explored, and the "best" path with the minimal lower bound which is the most promising shortest path to become a member of k diversified paths, is chosen to be explored. We note that the lower bounds are like some plugin boxes which can be put or removed without changing the correctness of our framework, but just efficiency. Moreover, when diversity is not required or $\tau = 1$, our framework can also support KSP problem.

In the next few sections, we first introduce how to compute shortest path lower bound and then diversified path lower bound, finally we elaborate our detail algorithm that uses the above lower bounds to efficiently find the top-k diversified shortest paths.

6 Shortest Path Lower Bound

We present how to calculate the shortest path lower bound, and then introduce path classification to filter unnecessary paths.

6.1 Reverse Shortest Path Trees

We compute the shortest path lower bound of a partially explored path by using reverse shortest path trees [8], [10]. Given a graph

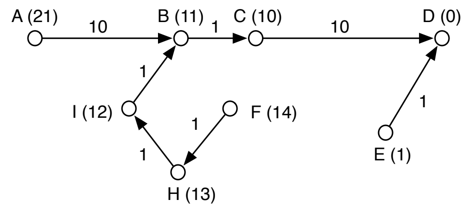


Fig. 3. Reverse shortest path tree $G(D)$

$G(V, E)$ and a vertex $t \in V$, the reverse shortest path tree of vertex t is denoted as $G(t) = G(V', E')$, where V' contains t and the vertices that are reachable by a path from t, and E' contains the edges in the shortest paths from vertices in V' to t. For each vertex $u \in V' \setminus \{t\}$, only one shortest path from u to t is considered, so that there exists exactly one outgoing edge (u, u') in $G(t)$ and t has no outgoing edge. Thus, $G(t)$ is rooted at t. We call u' the parent of u, denoted by $u.\mathsf{parent}$, and refer $u.\mathsf{sp}$ as the shortest path from u to t in $G(t)$ and $u.\mathsf{dis}$ as its length, i.e., $u.\mathsf{dis} = L(u.\mathsf{sp})$. It is convenient to construct $G(t)$ by invoking Dijkstra's algorithm from t in the reverse graph of G [10], [43].

Example 1. *Figure 2 illustrates the reverse shortest path tree $G(D)$ of the graph in Figure 1. For example, $B.\mathsf{parent} = C$, $B.\mathsf{sp} = B \to C \to D$, and $B.\mathsf{dis} = 11$.*

Lemma 3. *Given an arbitrary path $P(s,t)$ in G and the reverse shortest path tree $G(t)$, for any vertex v in $P(s,t)$, $L(P(s,v)) + v.\mathsf{dis} \leq L(P(s,t))$.*

Proof. The proof is straightforward. □

According to Lemma 3, the shortest path lower bound of $P(s,v)$, denoted as $\mathbf{LB}_1(P(s,v))$, is computed as $L(P(s,v)) + v.\mathsf{dis}$. For example, in Figure 1, the shortest path lower bound of $A \to I$ to D is 32 ($= 20 + I.\mathsf{dis}$), which happens to be the length of the shortest simple path. And the shortest path lower bound of $A \to B \to F$ is 25 ($= 11 + F.\mathsf{dis}$), which is shorter than the length of the shortest simple path P_2, which is 28 (see Table 1).

Constructing a full reverse shortest path tree that from a vertex that covers all reachable vertices in the graph may not be necessary, because not all vertices's dis are needed, especially for the vertices that are far away from the source, as they may never be visited during processing. Instead, inspired by [10], we build a partial reverse shortest path tree and incrementally update it. Algorithm 2 shows how to incrementally construct the reverse shortest path tree. Once we compute $\mathbf{LB}_1(P(s,v))$, Algorithm 2 is invoked to get $v.\mathsf{dis}$. If v has been settled, then we directly return $v.\mathsf{dis}$ which has been already computed. Otherwise, we continue to build the partial reverse shortest path tree based on a priority queue Q for vertices ordered by their dis until v has been settled. Finally, if priority queue is empty, it means that v cannot be reached from the destination, we return ∞. Initially, Q only contains the destination t with $t.\mathsf{dis} = 0$, and all other vertices with $\mathsf{dis} = \infty$. Moreover, if $landmarks$ [24] are available, the ordering key of vertex u in the priority queue can be computed as $u.\mathsf{dis} + \max_{w \in L}\{d(w,s) - d(w,u)\}$ instead of $u.\mathsf{dis}$, where s is the source vertex, L is the landmark vertex set and $d(w,*)$ is the shortest distance from landmark w to $*$ that can be precomputed offline. That is, algorithm 2 can traverse the network in an A* fashion and the vertices near source can be visited early. In this incremental way, the vertices that are never be visited won't be computed, which is more efficient than building the full reverse shortest path tree.

Algorithm 2: $ConstructPartialSPT(G, Q, v)$

```
Input: Reverse graph $G$, priority queue $Q$, vertex $v \in G$.
Output: $v$.dis.
1  if $v.isSettled$ = true then
2      return $v$.dis;
3  while $Q$ is not empty do
4      $u \leftarrow Q$.extractMin();
5      $u.isSettled \leftarrow$ true;
6      if $u = v$ then
7          return $u$.dis;
8      foreach outgoing edge $(u, u')$ of $u$ and $u'.isSettled$ = false do
9          if $u$.dis $+ w(u, u') < u'$.dis then
10             $u'$.dis $\leftarrow u$.dis $+ w(u, u')$;
11             $u'$.parent $\leftarrow u$;
12             $Q$.insert($u'$) if $u'$ is not in $Q$;
13 return $\infty$;
```

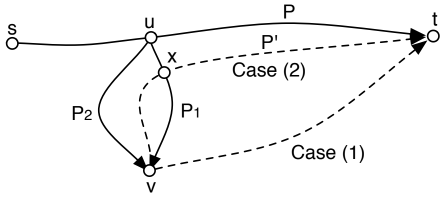


Fig. 4. An illustration of Lemma 4

6.2 Path Classification

A surprising observation is that a path may not need to be extended even if its lower bound is minimal. For example, in Figure 1, after the shortest simple path from A to D has been found, we attempt to look for the next shortest simple path. First, we extend $A \to B \to F$ (with minimal shortest path lower bound 25) via $F \to H \to I \to B \to C \to D$. Then we detect the repeated vertex B and choose $A \to B \to H$ (with minimal shortest path lower bound 26) to extend. However, there is no need to extend $A \to B \to H$ though it has the minimal lower bound, because $A \to B \to H$ is either cyclic via $H \to I \to B \to C \to D$ or longer than $A \to B \to F \to H$ via $H \to E \to D$. Hence, such path can be avoided being extended since there always exists at least one better path. This phenomenon is summarized by Lemma 4.

Lemma 4. *For any two simple partially explored paths $P_1(s, v)$ and $P_2(s, v)$ that deviate from a common shortest path at the same vertex. Let $P_1(s, t), P_2(s, t)$ be the shortest simple paths from s to t beginning with $P_1(s, v)$ and $P_2(s, v)$ respectively. If $L(P_1(s, v)) < L(P_2(s, v))$, then $L(P_1(s, t)) < L(P_2(s, t))$.*

Proof. Suppose $P_1(s, v)$ and $P_2(s, v)$ deviate from the common shortest path P at vertex u, as illustrated in Figure 4. To ease the following discussions, we distinguish two cases.

Case 1: If $P_1(s, v) + P_2(v, t)$ is simple, since $P_2(s, t) = P_2(s, v) + P_2(v, t)$ and $L(P_1(s, v)) < L(P_2(s, v))$, then $L(P_1(s, t)) \leq L(P_1(s, v)) + L(P_2(v, t)) < L(P_2(s, v)) + L(P_2(v, t)) = L(P_2(s, t))$.

Case 2: If $P_1(s, v) + P_2(v, t)$ is not simple, then there exists at least one repeated vertex between $P_1(u, v)$ and $P_2(v, t)$, say x for simplicity, forming one or more loops. Then, we can generate a new path $P'(s, t)$ based on $P_1(s, v) + P_2(v, t)$, but removing all vertices in each loop. Thus, $P'(s, t) = P(s, u) + P_1(u, x) + P_2(x, t)$ is simple and $L(P'(s, t)) < L(P_1(s, v)) + L(P_2(v, t)) < L(P_2(s, t))$. If $L(P_1(s, t)) \leq L(P'(s, t))$, then we have $L(P_1(s, t)) < L(P_2(s, t))$. Otherwise, P' is shorter than both $P_1(s, t)$ and $P_2(s, t)$, then P' should be found before $P_1(s, t)$ and $P_2(s, t)$. As a result, $P_1(s, v)$ deviates from P' at vertex x, then $P_1(s, v)$ and $P_2(s, v)$ actually deviate from different shortest paths at different vertices, which is a contradiction. □

According to Lemma 2, there is no need to extend $P_2(s, v)$ at v before $P_1(s, t)$ is found. We introduce the concept of *path classification* to avoid unnecessary path extending using Lemma 4. We partition all partially explored paths that deviate from different shortest simple paths at different vertices into different classes. Each class is denoted by $p \cdot v$, where p is the deviation path, and v is the deviation vertex. In other words, the paths which deviate from the same shortest path at the same vertex belong to the same class. Once a partially explored path, say P, has been extended at a vertex, say u, we call u has been $covered$ by the class that P belongs to. Hence, according to Lemma 2, u has no necessity to be covered again by P's class before the complete shortest simple path which begins with P has been found. That is, each vertex is covered by each class at most once before the next shortest simple path is found. For example, the shortest simple path from A to D in Figure 1 is $P_1 : A \to B \to C \to D$. Then paths $P' : A \to B \to H$ and $P'' : A \to B \to F \to H$ belong to the same class "$1 \cdot B$", indicating that they deviate from the 1st shortest path P_1 at vertex B. Since $L(P'') = 12$ is smaller than $L(P') = 13$, after P'' is extended at H, there is no need to extend P' at H before the next shortest simple path $P_2 : A \to B \to F \to H \to E \to D$ is found. We say P' is *dominated* by P'' and call P' an *inactive* path in this case, which can be temporarily removed from the path queue. After P_2 is found we *activate* it by re-adding it to the queue.

7 Diversified Path Lower Bound

To filter the shortest paths that highly overlap with paths in the result set, we compute another lower bound for each path before extension for each similarity function mentioned in Table 3.

We begin with $Sim_1(P'', P')$, where P'' is a complete path and P' is a path in the result set Ψ. If P'' is feasible, i.e., $Sim_1(P'', P') \leq \tau$, we have:

$$Sim_1(P'', P') = \frac{L(S_{P''} \cap S_{P'})}{L(P'') + L(P') - L(S_{P''} \cap S_{P'})} \leq \tau. \quad (1)$$

That is,

$$L(P'') \geq L(S_{P''} \cap S_{P'}) \times (1 + \frac{1}{\tau}) - L(P'). \quad (2)$$

In this way, the diversified path lower bound of partially explored path P, denoted as $\mathbf{LB}_2(P)$, is computed as Equation 3. We note that the diversified path lower bound of a partially explored path is monotonically increasing as it is extended, because the intersection length never decrease when the path grows. In other words, for any complete path which begins with P to be inserted into Ψ, its length is not smaller than $\mathbf{LB}_2(P)$.

$$\mathbf{LB}_2(P) = \max_{P' \in \Psi} \{L(S_P \cap S_{P'}) \times (1 + \frac{1}{\tau}) - L(P')\}. \quad (3)$$

Algorithm 3: $FindKSPD(G, s, t, Sim(\cdot,\cdot), k, \tau)$

```
Input: Graph $G$, source $s$, destination $t$ similarity function $Sim(\cdot,\cdot)$, $k$ and $\tau$.
Output: The top-$k$ shortest simple paths with diversity from $s$ to $t$.
1 Create priority queue $Q$ and local priority queues $LQ[\cdot]$;
2 $P \leftarrow$ the shortest path from $s$ to $t$;
3 $\Psi \leftarrow \{P\}$;
4 For each path $P' : v_1 \to \cdots \to v_i \to v_{i+1}$ that deviates from $P$, setting $P'$.rt $\leftarrow P'$, $P'$.len $\leftarrow L(P')$, $P'$.lb $\leftarrow \mathbf{LB}_1(P')$, $P'$.cls $\leftarrow 1 \cdot v_i$, and $LQ[v_{i+1}]$.insert$(P')$, $Q$.insert$(LQ[v_{i+1}])$;
5 while $|\Psi| < k$ and $Q$ is not empty do
6     $P \leftarrow$ FindNextPath$(G, Q, t, Sim(\cdot,\cdot), \tau)$;
7     if $\forall p' \in \Psi, Sim(p', P) \leq \tau$ then
8         Insert $P$ into $\Psi$;
9 return $\Psi$;
```

The diversified path lower bounds for other similarity functions can also be computed in a similar manner. Due to the limitation of the space, we list the results as follows:

- For $Sim_2(\cdot,\cdot)$, $\mathbf{LB}_2(P) =$
$$\max_{P' \in \Psi} \begin{cases} \frac{L(S_P \cap S_{P'})L(P')}{2\tau L(P') - L(S_P \cap S_{P'})} & 2\tau L(P') > L(S_P \cap S_{P'}) \\ \infty & 2\tau L(P') \leq L(S_P \cap S_{P'}). \end{cases}$$
- For $Sim_3(\cdot,\cdot)$, $\mathbf{LB}_2(P) = \max_{P' \in \Psi}\{\frac{L(S_P \cap S_{P'})^2}{\tau^2 L(P')}\}$.
- For $Sim_4(\cdot,\cdot)$,
$$\mathbf{LB}_2(P) = \max_{P' \in \Psi} \begin{cases} \frac{L(S_P \cap S_{P'})}{\tau} & L(P) \geq L(P') \\ L(P) & L(P) < L(P'). \end{cases}$$
- For $Sim_5(\cdot,\cdot)$,
$$\mathbf{LB}_2(P) = \max_{P' \in \Psi} \begin{cases} \infty & L(S_P \cap S_{P'}) \geq \tau L(P') \\ L(P) & L(S_P \cap S_{P'}) < \tau L(P'). \end{cases}$$

It is obvious that $\mathbf{LB}_2(P)$ should be 0 if $\Psi = \emptyset$ for all similarity functions.

So for each path, its lower bound **lb** is decided by the greater one of the shortest path lower bound $\mathbf{LB}_1$ and the diversified path lower bound $\mathbf{LB}_2$. As a result, the shortest paths that share lots of common edges with paths in the result set have greater lower bound, so that, they can be postponed to be extended, which improves the efficiency of finding the shortest diversified paths.

Algorithm 4: $FindNextPath(G, Q, t, Sim(\cdot,\cdot), \tau)$

```
Input: Graph $G$, priority queue $Q$, destination $t$, and threshold $\tau$.
Output: The next shortest feasible path.
1 while $Q$ is not empty do
2     $LQ[v] \leftarrow Q$.extractMin(), $P \leftarrow LQ[v]$.extractMin();
3     if $LQ[v]$ is not empty then $Q$.insert$(LQ[v])$;
4     while $tail(P) \neq t$ do
5         Compute $\mathbf{LB}_2(P)$ based on $Sim(\cdot,\cdot)$ and $\tau$;
6         if $\mathbf{LB}_2(P) > P.lb$ then
7             $P$.lb $\leftarrow \mathbf{LB}_2(P)$;
8             AdjustPath$(P)$;
9             $LQ[tail(P)]$.insert$(P)$;
10            $Q$.insert$(LQ[tail(P)])$ if $LQ[tail(P)]$ is not in $Q$;
11            break;
12        else if not ExtendPath$(P)$ then
13            break;
14    if $tail(P) = t$ then
15        AdjustPath$(P)$;
16        return $P$;
```

Algorithm 5: $ExtendPath(P)$

```
Input: Path $P$ to be extended.
Output: true/false.
1 $y \leftarrow$ tail$(P)$;
2 foreach $P' \in LQ[y]$, $P'$.cls $= P$.cls $\wedge$ $P'$.len $\geq P$.len do
3     Mark $P'$ inactive;
4 foreach edge $(y, u)$, $u \notin P$.rt $\wedge$ $u \neq y$.parent do
5     Create a new path $P''$ : $P''$.rt $\leftarrow P$.rt $+ u$, $P''$.len $\leftarrow P$.len $+ w(y, u)$, $P''$.lb $\leftarrow \mathbf{LB}_1(P'')$, $P''$.cls $\leftarrow P$.cls;
6     if $u$ has been covered by $P$.cls then
7         Mark $P''$ inactive;
8     else
9         $LQ[u]$.insert$(P'')$;
10        if $LQ[u]$ is not in $Q$ then $Q$.insert$(LQ[u])$;
11 if $y$.parent $\in P$.rt then
12     return false;
13 else
14     $P$.rt $\leftarrow P$.rt $+ \{y$.parent$\}$;
15     $P$.len $\leftarrow P$.len $+ w(y, y$.parent$)$;
16     return true;
```

8 The Final Algorithm

Finally, based on path classification and two lower bounds, we present the final algorithm of our framework for KSPD problem, as listed in Algorithm 3. Initially, we create an empty result set Ψ for k shortest diversified simple paths. In order to get the path with the minimal lower bound and maintain the path classifications, we use two data structures to handle the partially explored paths during the process. At first, a local priority queue, denoted as $LQ[v]$, is maintained for each vertex v to record the paths from s to v. All paths kept in $LQ[v]$ are sorted by their lower bounds lb in an ascending order. The second one is a global priority queue Q to maintain the local queues for such vertices, sorted by their minimal lower bounds in an ascending order as well. In this way, the minimal path of the minimal local queue in Q has the global minimal lower bound. For each path P in $LQ[v]$, we keep pieces of information besides lb, including the detail route rt of P which is from source to v, the path length len=L(rt), and cls referring to its path classification. Initially, lb equals to $\mathbf{LB}_1(P)$ computed as len $+ v$.dis. In the beginning, the shortest path P is added to the result set and all the paths that deviate from P are inserted to the corresponding local queues and the global queue (lines 2-4). Subsequently, we repeat invoking Algorithm

Algorithm 6: $AdjustPath(P)$

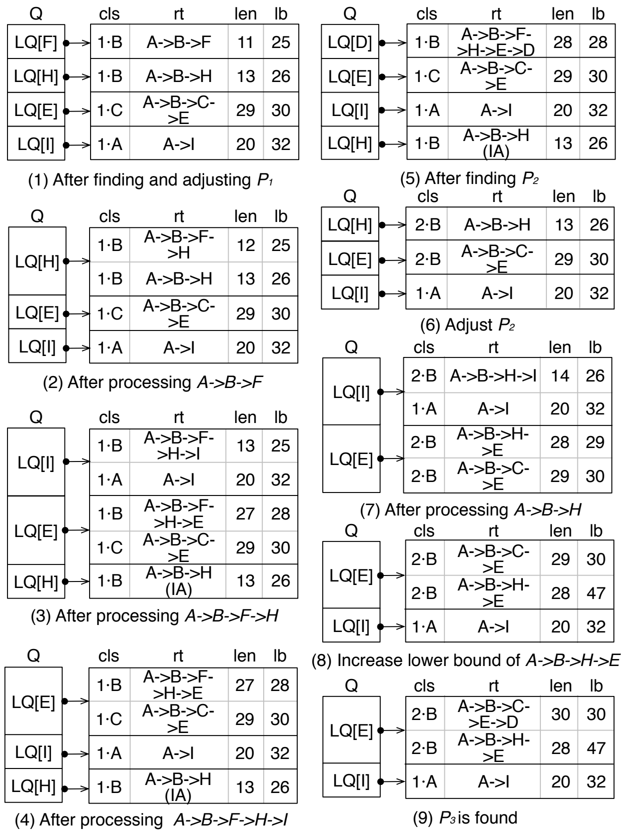
```
    Input: Path $P$ to be adjusted.
    Output: none.
1   $u \leftarrow$ the deviation vertex of $P$, $y \leftarrow$ tail($P$);
2   foreach vertex $v$, $v \in P(u, y)$ do
3       Activate the inactive paths in $LQ[v]$ that are dominated by $P$, and if $P$ reaches the destination, find all the paths with prefix $P(s, v)$ in all local queues, change their classifications to $P \cdot v$;
```

FindNextPath to find the next shortest simple path (lines 6) and check its feasibility (line 7-8), until the top-k diversified shortest paths have been found or the global priority queue is empty.

Algorithm FindNextPath (Algorithm 4) repeats extending partially explored path with the minimal lower bound until it reaches the destination, then returns it as the candidate diversified shortest path. Before we extend path P, we first check its feasibility by computing its diversified path lower bound $\mathbf{LB}_2(P)$ according to the similarity function and the threshold (line 5). If $\mathbf{LB}_2(P)$ is less than P.lb, we can safely extend P at each vertex along $tail(P)$.sp by frequently invoking Algorithm ExtendPath until P reaches the destination or P is not simple (lines 4-13), because P.lb won't change and it is always the minimal. Finally, if P reaches the destination t, we have successfully found the next shortest promising path. Meanwhile, we need to activate the inactive paths dominated by P and update the path classification of the paths derived from P according to Lemma 4 by invoking Algorithm AdjustPath (lines 14-16). On the other hand, if $\mathbf{LB}_2(P)$ is greater than P.lb, we reset P.lb to $\mathbf{LB}_2(P)$ and re-add P to the queue to postpone it to be extended, then choose another path with the minimal lower bound by breaking the loop (lines 6-11). In addition, we need to adjust P to activate the inactive paths that dominated by P, because these paths now may have smaller lower bound than P (line 8). In this way, we always select the most promising shortest path to extend, which accelerates finding the next shortest diversified path.

Algorithm ExtendPath (Algorithm 5) extends path P at $tail(P)$, say vertex y, via y.parent and creates new simple paths via the neighbors of y, then returns **false** if P is not simple, otherwise, returns **true**. According to Lemma 4, the path P' that belongs to the same class with P can avoid being extended when P'.len $\geq P$.len, we mark it inactive (lines 2-3). Then, for each outgoing edge (y, u), if u is neither in P.rt (not a loop) nor equal to y.parent, we extend P by creating new path P'' derived from P, and if u has been covered by P.cls, we set P'' inactive. Otherwise, we add P'' to the priority queue (lines 4-10). Subsequently, we extend P at y.parent. If y.parent is in P.rt, then P is not simple, we return **false**, so that Algorithm FindNextPath can stop extending P (lines 11-12). Otherwise, we update P's route and its length with y.parent (lines 13-16). Since the shortest path lower bound of P won't increase by following $tail(P)$.sp, we leave P.lb unchanged and return **true**, so that Algorithm FindNextPath can keep extending P along y.sp.

Algorithm AdjustPath (Algorithm 6) adjusts path P by activating the inactive paths that are dominated by P, and if P reaches the destination, the classifications of paths that derived from P need to be updated. In other words, for each vertex v in path from P's deviation vertex to P's tail, we activate the inactive paths dominated by P in $LQ[v]$. In addition, if P is a complete



Q	cls	rt	len	lb
LQ[F]	1·B	A->B->F	11	25
LQ[H]	1·B	A->B->H	13	26
LQ[E]	1·C	A->B->C->E	29	30
LQ[I]	1·A	A->I	20	32

(1) After finding and adjusting *P₁*

Q	cls	rt	len	lb
LQ[H]	1·B	A->B->F->H	12	25
	1·B	A->B->H	13	26
LQ[E]	1·C	A->B->C->E	29	30
LQ[I]	1·A	A->I	20	32

(2) After processing *A->B->F*

Q	cls	rt	len	lb
LQ[I]	1·B	A->B->F->H->I	13	25
	1·A	A->I	20	32
LQ[E]	1·B	A->B->F->H->E	27	28
	1·C	A->B->C->E	29	30
LQ[H]	1·B	A->B->H (IA)	13	26

(3) After processing *A->B->F->H*

Q	cls	rt	len	lb
LQ[E]	1·B	A->B->F->H->E	27	28
	1·C	A->B->C->E	29	30
LQ[I]	1·A	A->I	20	32
LQ[H]	1·B	A->B->H (IA)	13	26

(4) After processing *A->B->F->H->I*

Q	cls	rt	len	lb
LQ[D]	1·B	A->B->F->H->E->D	28	28
LQ[E]	1·C	A->B->C->E	29	30
LQ[I]	1·A	A->I	20	32
LQ[H]	1·B	A->B->H (IA)	13	26

(5) After finding *P₂*

Q	cls	rt	len	lb
LQ[H]	2·B	A->B->H	13	26
LQ[E]	2·B	A->B->C->E	29	30
LQ[I]	1·A	A->I	20	32

(6) Adjust *P₂*

Q	cls	rt	len	lb
LQ[I]	2·B	A->B->H->I	14	26
	1·A	A->I	20	32
LQ[E]	2·B	A->B->H->E	28	29
	2·B	A->B->C->E	29	30

(7) After processing *A->B->H*

Q	cls	rt	len	lb
LQ[E]	2·B	A->B->C->E	29	30
	2·B	A->B->H->E	28	47
LQ[I]	1·A	A->I	20	32

(8) Increase lower bound of *A->B->H->E*

Q	cls	rt	len	lb
LQ[E]	2·B	A->B->C->E->D	30	30
	2·B	A->B->H->E	28	47
LQ[I]	1·A	A->I	20	32

(9) *P₃* is found

Fig. 5. A running example of our final algorithm (The term "inactive" is shorten as "IA")

path, for all partially explored paths that begin with $P(s, v)$ in all local queues, we update their classes with $P \cdot v$.

Example 2. *Let's find the top-3 diversified shortest paths from A to D in Figure 1 with $Sim(\cdot, \cdot) = Sim_1(\cdot, \cdot)$ and $\tau = 0.5$.*

1) *First, we find the shortest path $P_1 : A \rightarrow B \rightarrow C \rightarrow D$ via A.sp and add it to the result set Ψ. Then we create four partially explored paths that deviate from P_1 and set their classes to "$1 \cdot *$" respectively. Figure 5 (1) illustrates the states of Q and $LQ[\cdot]$ after P_1 is found.*
2) *Subsequently, we find the next shortest diversified path. Path $A \rightarrow B \rightarrow F$ (with minimal lower bound) is chosen to be extended via F.parent, since its diversified path lower bound 10 × (1+1/0.5)-21=9 is less than its lower bound 25, which results in path $A \rightarrow B \rightarrow F \rightarrow H$, as shown in Figure 5 (2). Next, extending $A \rightarrow B \rightarrow F \rightarrow H$ ($\mathbf{LB}_2$<lb) at H generates two paths: $A \rightarrow B \rightarrow F \rightarrow H \rightarrow I$ and $A \rightarrow B \rightarrow F \rightarrow H \rightarrow E$. Meanwhile, since $A \rightarrow B \rightarrow H$ and $A \rightarrow B \rightarrow F \rightarrow H$ belong to the same class, we mark $A \rightarrow B \rightarrow H$ inactive, so that it won't be extended again in this round, which is illustrated in Figure 5 (3). Subsequently, $A \rightarrow B \rightarrow F \rightarrow H \rightarrow I$ ($\mathbf{LB}_2$<lb) is chosen to be extended. After visiting B, the path is discarded directly due to the existence of a loop, as shown in Figure 5 (4). Finally, we process $A \rightarrow B \rightarrow F \rightarrow H \rightarrow E$ ($\mathbf{LB}_2$<lb) and get the next shortest path $P_2 : A \rightarrow B \rightarrow F \rightarrow H \rightarrow E \rightarrow D$, as shown in Figure 5 (5). In addition, we adjust P_2 as illustrated in Figure 5 (6). Since $A \rightarrow B \rightarrow H$ and $A \rightarrow B \rightarrow C \rightarrow E$ share prefix $A \rightarrow B$ with P_2, their classes are changed to "$2 \cdot B$", and the inactive path*

$A \rightarrow B \rightarrow H$ which is dominated by P_2 in $LQ[H]$, is activated. Because $Sim_1(P_1, P_2) = 10/39 = 0.26 < \tau = 0.5$, we add P_2 to the result set.

3) *We then find the third shortest diversified path by first extending $A \rightarrow B \rightarrow H$ (**LB**$_2$<lb) (Figure 5 (7)). Then, we follow H.sp, but detect a loop $A \rightarrow B \rightarrow H \rightarrow I \rightarrow B$, so we give up this path and extend another one: $A \rightarrow B \rightarrow H \rightarrow E$. Its diversified path lower bound is computed as: $25 \times (1 + \frac{1}{0.5}) - 28 = 47$, which is greater than its lower bound 29, so we reset the value of entry lb to 47 (Figure 5(8)). Since no path is dominated by it, nothing needs to be changed. Finally, we process path $A \rightarrow B \rightarrow C \rightarrow E$ (**LB**$_2$<lb) through E.sp, and find shortest path $P_3 : A \rightarrow B \rightarrow C \rightarrow E \rightarrow D$. Since P_3 is qualified, we add it to the result set and get the top-3 shortest simple paths with diversity.*

As we can see, one shortest path is avoided being checked in the above example. We emphasize that we can distinguish two lower bounds. One is based on similarity functions, which is useful only when considering diversity. The other is independent on similarity functions, which can also benefit the cases when not considering diversity. In this way, our algorithm can not only solve KSPD problem but also deal with traditional KSP problem by simply setting $\tau = 1$ or **LB**$_2$=0.

8.1 Correctness

Theorem 1. *Algorithm FindKSPD correctly returns the top-k shortest paths with diversity.*

Proof. Suppose a shortest diversified path found by FindKSPD is $P = P(s, v) + v$.sp, then P.len $= P(s, v)$.len $+ v$.dis and P.lb $= P(s, v)$.lb (if P.lb increases, then we stop following v.sp and won't get P from FindNextPath). We first prove that P.len $= P$.lb. If $P(s, v)$.lb $= P(s, v)$.len $+ v$.dis, then we have P.len $= P$.lb when we get P. Otherwise, if $P(s, v)$.lb $>$ $P(s, v)$.len $+ v$.dis, which implies that $P(s, v)$.lb has been increased to the diversified path lower bound of $P(s.v)$. Hence, we have P.lb $\geq P(s, v)$.lb $> P(s, v)$.len $+ v$.dis $= P$.len. Since P is feasible, P.len $\geq P$.lb, which is a contradiction. Thus we have P.len $= P$.lb. Suppose there exists another feasible path $P' = P'(s, u) + u$.sp which is shorter than P, then P'.lb $< P$.lb, and $P'(s, u)$.lb $< P(s, v)$.lb. Since the lower bound of a path is monotonically increasing, it means that $P'(s, u)$ must have been extended before $P(s, v)$ (otherwise there must exist a better path than $P'(s, u)$ according to Lemma 4). As a result, P' must have been found before P. Accordingly, Algorithm FindKSPD returns the top-k shortest paths with diversity. □

8.2 Complexity

We then analyze the performance of our approach which is based on path classification and lower bounds. Since the lower bounds are heuristics to improve the efficiency of our framework, they do not change the theoretical complexity of our method. Theorem 2 shows the worst-case time complexity of our algorithm.

Theorem 2. *The worst-case time complexity of Algorithm FindKSPD is $O(\kappa n(m + n \log n))$, where κ is the number of shortest simple paths returned by Algorithm FindNextPath.*

Proof. According to Lemma 4, each class covers each vertex at most once before the class's corresponding shortest simple path has been found. In other words, Dijkstra's algorithm is executed at most once by each class to find its corresponding shortest simple path. Every time we get a shortest simple path, $O(n)$ new classes are generated from the new shortest path. Hence, there totally exist $O(\kappa n)$ classes after we get the top-κ shortest simple paths. As a result, the worst-case time complexity of our method is $O(\kappa n(m + n \log n))$, where $O(m + n \log n)$ is the time complexity of Dijkstra's algorithm. □

Discussion: Although our method cannot break through the complexity of the baseline method, it is worth noting that the worst case seldom happens. In most cases, we find the next shortest simple path without covering all the vertices by each class. In an ideal situation, we can directly find the next shortest simple path without trying other paths by following the reverse shortest path tree, which only costs $O(\kappa(n + \log n))$. When diversity is not considered, compared to Yen's algorithm [45] and its variants [10], [30], the advantages of our framework for KSP problem are two folds.

First, our framework applies a best-first paradigm with a (partially explored) path granularity; while Yen's algorithm and its variants apply a best-first paradigm with a subspace granularity, where each subspace corresponds to a complete candidate shortest path which deviates from previous shortest paths. As a result, in our framework, the path with minimum lower bound is the path with exact global minimum lower bound and if it is simple then it is the next shortest simple path. In contrast, in Yen's algorithm and its variants, the subspace with minimum lower bound is not an exact global minimum lower bound since the shortest simple path in the subspace with the minimum lower bound may not be the next shortest simple path, which means the lower bound in our framework is tighter than those of Yen's algorithm and its variants. Furthermore, by using path classification, the worst case complexity of our framework can be guaranteed to be equal to Yen's algorithm and its variants. In summary, fewer paths need to be examined by our framework to find the top-k shortest simple paths due to a tighter shortest path lower bound (see experiments in Section 9.2). Second, we do not generate complete candidate shortest paths unless they are members of top-k shortest simple paths. Once a partially explored path is not simple, we choose another one until the next shortest path is found, which also shows the flexibility of our framework with path granularity.

On the other hand, utilizing the diversified path lower bound, we always choose the "best" path to explore by avoiding the shortest paths that highly overlap with paths in the result set, which helps to find the k diversified shortest paths with a smaller κ.

8.3 Implementing Optimizations

We propose some optimizations for Algorithm FindKSPD to further improve its execution efficiency.

Extend path. Recall that we extend path by creating new paths via all its neighbors, and add the new paths to the priority queues. For large graph with great degree, many new paths are generated while extending path, which dramatically increases the size of priority queues (the size can exponentially increase in worst case) and deteriorates the efficiency of our algorithm. In fact, not all new paths need to be created via neighbors and added to the priority queues, because the paths with great lower bound may never be visited while processing. Hence, without changing its correctness, we extend path in a lazy extending way by only inserting the new path with minimal lower bound into the priority queue. To this end, we add a new attribute to the structure of path P, namely extendingPathList, to store P's extending paths sorted by their lower bounds (lb) in an ascending order. That is, when we extend a path P (line 4 in Algorithm 5), only the new path P' with smallest lower bound need to be added to the priority queues, and all the other new paths are inserted into P'.extendingPathList. Then once P' has been visited (line 2 in Algorithm 4), the next path P'' in P'.extendingPathList is removed and added to the priority queues and P'.extendingPathList is passed to P''.extendingPathList by setting P''.extendingPathList $=$ P'.extendingPathList and P'.extendingPathList $= \emptyset$. In this way, we extend paths on demand, which reduces the size of priority queues and improve the efficiency of our method.

Adjust path. Recall that we need to find all the paths that deviate from the new shortest path when we adjust it (line 3 in Algorithm 6), to efficiently find all such paths, we maintain a prefix tree rooted at the source vertex for all partially explored paths during processing. Once a new path is created and added to priority queues, we add it to the prefix tree. And if it has already been visited, we remove it from the prefix tree. Then, we can search all such paths which share the same prefix with the new shortest path in $O(n)$. We note that once the classification of a path is updated, all the paths in its extendingPathList need to be updated, too.

Compute $\mathbf{LB}_2$. The key to compute $\mathbf{LB}_2$ is to compute the overlap between two paths. To quickly identify the sharing edges between two paths, we use a hash table to find their common edges in $O(n)$. Moreover, since the newly created candidate path shares the same prefix path with the path they are derived from, the overlaps between the prefix path and the paths in the result set can be inherited and need not to be recomputed from scratch again. Concrete examples on the optimization of $\mathbf{LB}_2$ computation can be found in the supplementary document [1].

9 Evaluation

We report on empirical studies of the performance of the proposed algorithms.

9.1 Experimental Setup

Graphs: We use 5 real-world directed graphs with different types and properties (see Table 4). In particular, $Google$, $Google+$, and $WikiTalk$ are un-weighted graphs, where we set all edge weights to 1. $RoadCOL$ and $RoadFLA$ are weighted graphs where the weights represent the lengths of the roads in Colorado and Florida, USA, respectively.

TABLE 4
Real-World Graphs

Dataset	Type	$\|V\|$	$\|E\|$	Avg. Degree
$Google+$[1]	Social	107,614	13,673,453	254.12
$Google$[1]	Web	875,715	5,105,040	11.66
$WikiTalk$[1]	Communication	2,394,387	5,021,411	4.19
$RoadCOL$[2]	Road	435,666	1,057,066	4.85
$RoadFLA$[2]	Road	1,070,376	2,687,902	5.02

We also generate synthetic graphs using the SSCA2 generator[3]. The generated graphs are directed, weighted graphs, and made up of random-sized cliques, with a hierarchical inter-clique distribution of edges based on a distance metric. We use all the default parameters except for the parameter $SCALE$ that describes graph size. Table 5 shows the synthetic graphs we generate by setting $SCALE$ to 16, 17, 18, 19, and 20, respectively.

TABLE 5
Synthetic Graphs

Dataset	$\|V\|$	$\|E\|$	Avg. Degree
$S16$	65,536	1,564,579	47.75
$S17$	131,072	3,907,909	59.63
$S18$	262,144	10,008,022	76.36
$S19$	524,288	24,974,544	95.27
$S20$	1,048,576	63,148,387	120.45

Queries: For each graph, we randomly choose 20 different source-destination pairs as queries and report the average running time and the number of visited partially explored paths. If a query cannot stop within a pre-defined time limit (2 hours in the following experiments) or fails due to out of memory exceptions, we denote the corresponding running time as *N/A*. We vary the values of parameters in the experiments according to Table 6, where default values are shown in bold.

TABLE 6
Parameter Settings

Parameters	Values
k (for KSP)	10, 20, **30**, 40, 50
k (for KSPD)	5, **10**, 15, 20
τ	0.8, **0.6**, 0.4, 0.2
$Sim(\cdot,\cdot)$	$\mathbf{Sim_1}$, Sim_2, Sim_3, Sim_4, Sim_5

Methods: We include six different algorithms to process both KSP and KSPD problems.

IterBound is the state-of-the-art method for solving KSP problem [10]. As a variant of Yen's algorithm, IterBound computes the candidate shortest paths in a best-first manner based on their lower bounds by using an iteratively bounding approach.

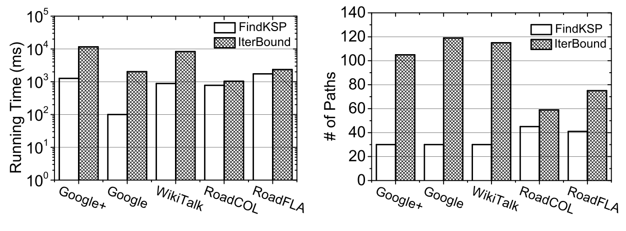
div-cut is an exact algorithm for finding the top-k diversified results [37], which is also NP-hard. This problem considers a setting where each result r has a score $S(r)$ and aims at finding a diversified result set D with cardinality less than or equal to k, in which each pair of results are dissimilar and the total score of all results is maximized. Although this problem is different from the KSPD problem, we can transfer the KSPD problem to this problem by using a score mapping function $S(P) = k\chi - L(P)$ for each simple path P from the source to destination on graph $G = (V, E)$, where $\chi = \Sigma_{e \in E} w(e)$. We omit the prove due to space limitation. To solve this problem, a diversity graph is constructed, where each node represents a result and an edge connects two results if they are similar. Next, the diversity graph is decomposed into a set of connected components, and each connected component which is loosely connected through a set of cut points can be further decomposed. In this way, the diversity graph is divided into a set of small components, and each component can be processed separately. Finally, the optimal results can be found by combining all the results in a particular way.

KSPD-Yen is the baseline method by using Yen's algorithm [45] for KSPD problem.

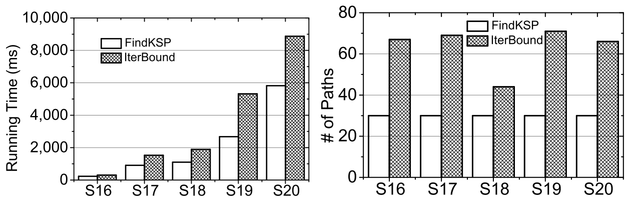
FindKSPD is the final algorithm (Algorithm 3) we propose to find the approximate KSPD with proposed optimizations.

FindKSP is the KSP version of our final algorithm FindKSPD by setting $\tau = 1$. That is the case that the diversified path lower bounds are disabled.

1. http://snap.stanford.edu/data/index.html
2. http://www.dis.uniroma1.it/challenge9/download.shtml
3. http://www.cse.psu.edu/~kxm85/software/GTgraph/



(a) Real-world graphs



(b) Synthetic graphs

Fig. 6. Efficiency on different graphs for KSP (k=30)

TABLE 7
Average hops of the top 30 shortest simple paths

Dataset	*Google+*	*Google*	*WikiTalk*	*RoadCOL*	*RoadFLA*
Avg. Hops	4	13	5	710	1,013
Dataset	$S16$	$S17$	$S18$	$S19$	$S20$
Avg. Hops	152	326	246	377	461

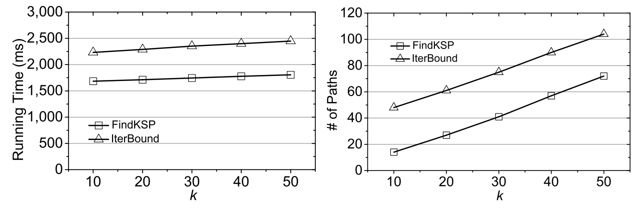
FindKSPD$^-$ is the simplified version of FindKSPD that only uses the shortest path lower bound by removing lines 5-11 from Algorithm FindNextPath.

Implementation details: All algorithms are implemented in Java 1.6 under CentOS Linux. All experiments are conducted on a 2.00GHz Intel(R) Xeon(R) CPU computer with 64 GB memory.

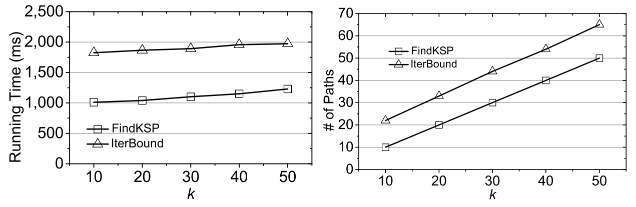
9.2 Efficiency of shortest path lower bound

Since the shortest path lower bound is used to find the shortest simple paths, we test its efficiency by finding the top-k shortest simple paths, i.e., the KSP problem. Hence, we test the performance of FindKSP and IterBound. In addition to running time, we also compare the number of paths computed by the two methods to find the top-k shortest simple paths. For FindKSP, the computed paths are the partially explored paths that have been visited by line 2 in Algorithm 4. For IterBound, the computed paths are the complete candidate shortest paths that have been generated.

Figure 6 shows the performance of the two methods on different graphs with $k = 30$. Clearly, FindKSP outperforms IterBound, the state-of-the-art method, on both real-world and synthetic graphs. Moreover, for *Google*, *Google+* and *WikiTalk*, FindKSP performs nearly one order of magnitude faster than IterBound. According to figures on the right side, the number of paths tried by FindKSP is generally slightly larger than k, while IterBound needs to generate more shortest paths. This is because the shortest path lower bound is tighter than the heuristic bound used in IterBound, and FindKSP directly finds the next shortest simple path without generating candidate shortest paths. Moreover, FindKSP tries partially explored paths instead of generating complete candidate shortest paths in IterBound. As a result, FindKSP is more efficient than IterBound. For example, to find the top 30 shortest simple paths on *RoadFLA*, FindKSP checks 41 partially explored paths and takes 1,745 ms, while the IterBound generates 75 complete candidate paths and spends 2,351 ms.



(a) *RoadFLA*



(b) $S18$

Fig. 7. Vary k on different graphs for KSP

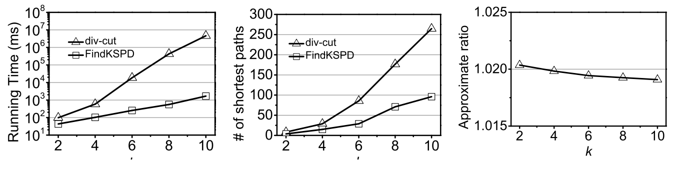
Table 7 shows the average hops of the top 30 shortest simple paths on each graph. For *Google*, *Google+* and *WikiTalk*, they have relatively small average hops. As a result, the top 30 shortest simple paths have similar lengths, which makes the lower bounds of IterBound relatively less sensitive and it needs to compute more shortest paths. On the contrary, FindKSP keeps finding the shortest simple path via the shortest path tree until it reaches the destination or it is not simple. This way, other shortest simple paths with the same length (i.e., the lower bound) are not considered. Hence, the number of paths that are checked by FindKSP is much smaller than that of IterBound. For synthetic graphs, the running time of both methods increases as the graph becomes larger, and the number of computed paths by FindKSP is more stable on graphs with different sizes.

Effect of k. We vary k from 10 to 50 to test the efficiency of the two methods. Figure 7 shows the results on two selected graphs. In general, the running time of two methods increases as k rises, and FindKSP always outperforms IterBound. Because FindKSP tries fewer paths to find the top-k shortest simple paths compared to IterBound, the running time of FindKSP is much faster. Furthermore, FindKSP does not generate complete paths unless they are members of top-k shortest simple paths, while IterBound needs to generate complete candidate shortest paths to determine the next shortest simple paths. As a result, the searching space of FindKSP is smaller than that of IterBound. For example, on $S18$, FindKSP only spends 1,102 ms visiting 30 partially explored paths to find the top 30 shortest simple paths, while IterBound spends 1,891 ms generating 44 candidate shortest paths. Although the number of paths tried by IterBound is slightly larger than that of FindKSP, IterBound takes much more time than FindKSP does due to larger searching space. Another observation is that the performance gap between FindKSP and IterBound on *S18* is much greater than that on *RoadFLA*, the reason is that $S18$ has much greater average degree than *RoadFLA*, therefore, IterBound explores much more edges to find the shortest path, while FindKSP does not need to explore all outgoing edges of vertices due to lazy extending (see Section 8.3).

9.3 Efficiency of diversified path lower bound

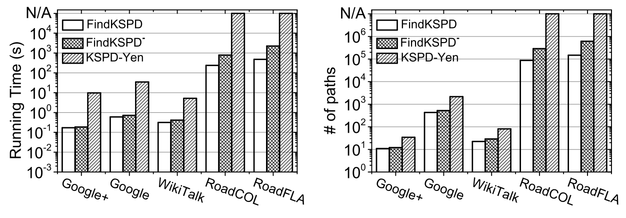
We test the efficiency of diversified path lower bound by finding the top-k diversified shortest simple paths, i.e., the KSPD problem.

Since the exact algorithm div-cut is really slow in KSPD problem, we first use a small graph to show the efficiency of

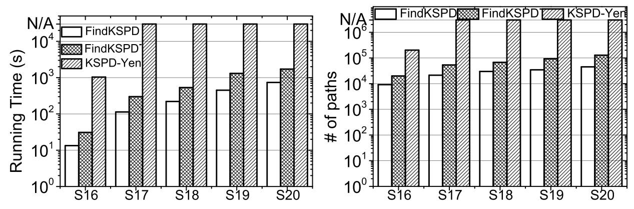


(a) Running time w.r.t k (b) Shortest paths w.r.t k (c) Approximate ratio

Fig. 8. The performance of div-cut and FindKSPD



(a) Real-world graphs



(b) Synthetic graphs

Fig. 9. Efficiency on different graphs for KSPD ($k = 10, \tau = 0.6$)

div-cut and FindKSPD. The graph[1] is a weighted undirected road network of California, USA, it contains 21,048 vertices and 21,693 edges. By setting $Sim(\cdot, \cdot) = Sim_1(\cdot, \cdot)$ and $\tau = 0.8$, Figure 8 shows the average performance of the two methods.

As we can see from Figure 8(a), the running time of div-cut dramatically increases as k gets larger. For example, it takes div-cut more than 1 hour to find the top 10 diversified shortest paths, which is very inefficient and unacceptable. While FindKSPD only takes less than 2 seconds to get the result, it is orders of magnitude faster than div-cut. Figure 8(b) shows the number of complete shortest paths checked by both methods to find the top-k diversified shortest paths. Obviously, div-cut checks much more shortest paths to find the optimal result. For example, it checks top 265 shortest paths to find top 10 diversified shortest paths, which means the shortest paths are actually quite similar. As a result, most of the diversity graphs we construct for div-cut are strongly connected with no cut points. Because FindKSPD aims at the "best" path, it checks fewer shortest paths to find the diversified shortest paths. div-cut cannot be well scaled to the KSPD problem mainly because the (adjacent) shortest paths from a source to a destination are generally quite similar. As a result, the diversity graph consists of paths is usually strongly connected and cannot be decomposed, moreover, the cut points in the diversity graph hardly exist and cannot be further decomposed, neither. That is, the optimization techniques in div-cut cannot be used in KSPD problem, which makes it very inefficient.

Since the approximate results returned by FindKSPD have the same cardinality as the optimal results, the approximate ratio of the first sub-problem is exactly 1. Then we compute the approximate ratio for the second sub-problem by total length. Figure 8(c) shows the approximate ratio of FindKSPD for the second sub-problem. As we can see, the approximate results returned by FindKSPD are very close to the optimal results, only 1.5%-2.5% larger than the optimal results.

1. http://www.cs.utah.edu/~lifeifei/SpatialDataset.htm

Although div-cut is able to identify the optimal result, it takes prohibitively long running time. Since it almost always takes more than 2 hours and thus returns *N/A* in our experimental setup with larger graph, k, and smaller τ, we omit it in our following experiments. Instead, we test the performance of FindKSPD, FindKSPD$^-$ and KSPD-Yen.

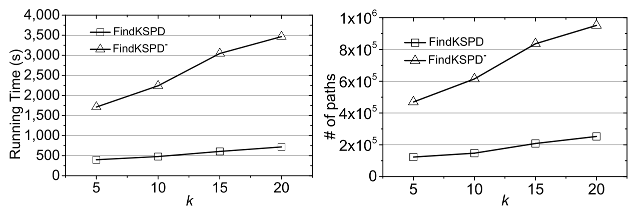
Figure 9 shows the efficiency of the methods on different graphs with $k = 10$ and $\tau = 0.6$. Clearly, FindKSPD with diversified path lower bound tries fewer paths to find the top-k diversified shortest paths, as it filters many unnecessary shortest paths. As a result, the running time of FindKSPD is much faster than that of FindKSPD$^-$ on both real-world and synthetic graphs. Meanwhile, by using the shortest path lower bound and path classification, FindKSPD$^-$ performs more than one order of magnitude faster than KSPD-Yen due to fewer shortest paths computation, and KSPD-Yen cannot get the results within 7,200 seconds in several graphs, such as $RoadCOL$, $RoadFLA$. We note that the runtimes of FindKSPD and FindKSPD$^-$ are close on social network graphs such as $Google$, $Google+$, and $WikiTalk$. Due to the small-world phenomenon, these graphs have small diameters. In particular, the diameters of $Google$, $Google+$, and $WikiTalk$ are only 21, 6 and 9, respectively. The short diameters make top-k shortest paths more diverse since there are not many opportunities to follow the shortest path. Thus, on social network graphs, the numbers of paths explored by the two methods are small. For example, FindKSPD and FindKSPD$^-$ try 11 and 12 paths, respectively, to find the top 10 diversified shortest simple paths on $Google+$. However, since Dijkstra's search by Yen's algorithm on these unweighted graphs explores more searching space to find the shortest paths, KSPD-Yen takes much more time than both FindKSPD and FindKSPD$^-$. Since KSPD-Yen is significantly slower under all experimental settings, we remove it from the comparison for subsequent experiments.

TABLE 8
Distributions of the runtime (s) on $RoadFLA$

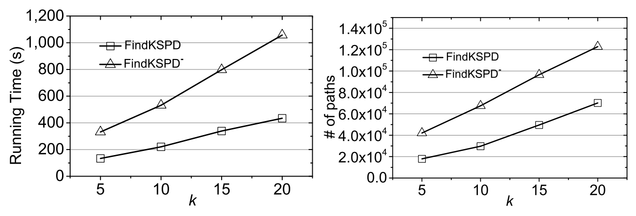
	FindKSPD	**FindKSPD$^-$**
Overall runtime	478.649	2,241.150
LB$_1$ computation time	0.015	0.014
LB$_2$ computation time	3.320	0
Priority queue maintenance time	4.601	12.440
Others time	470.712	2,228.696

Table 8 shows the distributions of the runtime of FindKSPD and FindKSPD$^-$ on $RoadFLA$. Obviously, the computations of **LB**$_1$ and **LB**$_2$ are only small portions (less than 1%) of the overall runtime due to the incremental partial shortest path tree and the optimization for computing **LB**$_2$ in Section 8.3. We note that FindKSPD's computation time for **LB**$_1$ is slightly greater than FindKSPD$^-$, because FindKSPD explores a wider range of vertices to efficiently find the diversified shortest paths by using **LB**$_2$. Since FindKSPD$^-$ examines more paths, its maintenance time for priority queues is much greater than that of FindKSPD. Apparently, the others time consist of the searching process on the graph dominate the overall runtime of KSPD and FindKSPD takes much less searching time than FindKSPD$^-$.

Effect of k. We vary k from 5 to 20 to test the efficiency of two methods. Figure 10 shows the results on $RoadFLA$ and $S18$ with $\tau = 0.6$. In general, the runtime of both methods increases as k



(a) $RoadFLA$



(b) $S18$

Fig. 10. Vary k on different graphs for KSPD ($\tau = 0.6$)

gets larger, since they both try much more paths. But, FindKSPD outperforms FindKSPD$^-$ in all cases, moreover, the greater the k is, the more advantages FindKSPD has.

Effect of τ. We vary τ from 0.2 to 0.8. Figure 11 shows the results on $RoadFLA$ and $S18$ with $k = 10$. Clearly, the runtime and the number of paths tried by two methods sharply increase as τ gets smaller, which means it costs a lot to get a little more diversified paths. On $RoadFLA$, FindKSPD$^-$ cannot get the results when $\tau = 0.2$ due to huge shortest paths to process. In all cases, FindKSPD runs faster than FindKSPD$^-$, since lots of infeasible shortest paths are filtered by FindKSPD.

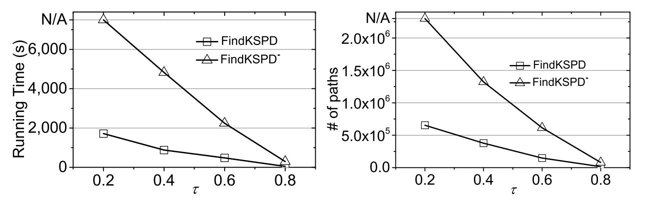
Effect of $Sim(\cdot, \cdot)$. Figure 12 shows the results by using different similarity functions on $RoadFLA$ and $S18$ with $k = 10$ and $\tau = 0.6$. It is straightforward to prove that $Sim_1(\cdot, \cdot) \leq Sim_4(\cdot, \cdot) \leq Sim_3(\cdot, \cdot) \leq Sim_2(\cdot, \cdot) \leq Sim_5(\cdot, \cdot)$. That is, paths are the most similar measured by $Sim_5(\cdot, \cdot)$. As a result, we need to check more paths to find the diversified shortest paths when paths are more similar, and the running time increases as the similarity rises. Clearly, FindKSPD performs faster than FindKSPD$^-$ in all cases due to fewer visited paths, and FindKSPD is more insensitive to the choice of similarity functions.

Additional experimental results, including (1) memory usage of different algorithms for finding KSP and KSPD, (2) a case study of KSPD, (3) using landmarks to update $\mathbf{LB}_1$, (4) distributions of path diversities, and (5) the potential of dynamic updating strategies, are included in the supplementary document [1].

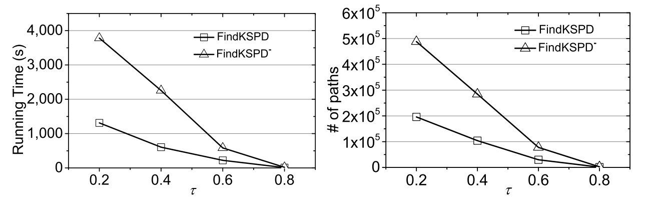
10 Conclusion

In this paper, we study the top-k shortest paths with diversity (KSPD). After defining the problem formally, we prove that the KSPD problem is NP-hard. Then we propose a general greedy framework for KSPD problem, which supports different similarity functions and also KSP problem when diversity is not required. Moreover, we utilize two lower bounds to improve the efficiency by filtering unnecessary paths, and we introduce path classification to guarantee the time complexity of our method. Experiments show the efficiency and effectiveness of our proposal.

As for future work, we plan to further improve the efficiency of KSPD in serval aspects, such as bidirectional search and dynamic updating strategy, to make it more scalable.

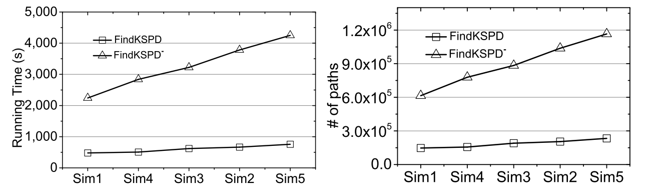


(a) $RoadFLA$

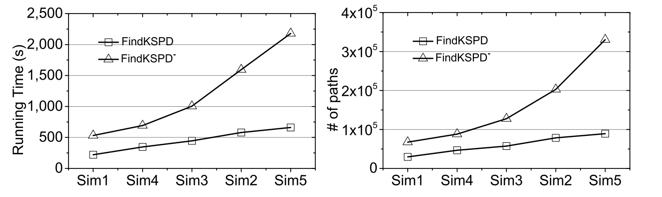


(b) $S18$

Fig. 11. Vary τ on different graphs for KSPD ($k = 10$)



(a) $RoadFLA$



(b) $S18$

Fig. 12. Vary $Sim(\cdot, \cdot)$ on different graphs for KSPD ($k = 10, \tau = 0.6$)

Acknowledgement

Our research is supported by the National Key Research and Development Program of China (2017YFC0803700 and 2016YFB1000905), NSFC (61370101, 61532021, U1501252, 61702423, and 61772327).

References

[1] Supplementary document. http://files.cnblogs.com/files/maxliu/KSPD-Supplementary.pdf.

[2] V. Akgün, E. Erkut, and R. Batta. On finding dissimilar paths. *European Journal of Operational Research*, 121(2):232–246, 2000.

[3] T. Akiba, T. Hayashi, N. Nori, Y. Iwata, and Y. Yoshida. Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling. In *Proc. AAAI*, pages 2–8, 2015.

[4] O. Andersen, C. S. Jensen, K. Torp, and B. Yang. Ecotour: Reducing the environmental footprint of vehicles using eco-routes. In *Proc. MDM*, pages 338–340, 2013.

[5] A. Angel and N. Koudas. Efficient diversity-aware search. In *Proc. SIGMOD*, pages 781–792, 2011.

[6] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. In *Algorithm Engineering*, pages 19–80. 2016.

[7] Borodin, Allan, Lee, H. Chul, Ye, and Yuli. Max-sum diversification, monotone submodular functions and dynamic updates. *Computer Science*, pages 155–166, 2012.

[8] R. S. Cahn. *Wide area network design: concepts and tool for optimization*. Morgn Kaufmann Publishers, 1998.

[9] P. Carotenuto, S. Giordani, and S. Ricciardelli. Finding minimum and equitable risk routes for hazmat shipments. *Computers and OR*, 34(5):1304–1327, 2007.

[10] L. Chang, X. Lin, L. Qin, J. X. Yu, and J. Pei. Efficiently computing top-k shortest path join. In *Proc. EDBT*, pages 133–144, 2015.

[11] Y. Cheng, J. Li, and J. P. G. Sterbenz. Path geo-diversification: Design and analysis. In *Proc. ICUMT*, pages 46–53, 2013.

[12] T. Chondrogiannis, P. Bouros, J. Gamper, and U. Leser. Alternative routing: k-shortest paths with limited overlap. In *Proc. SIGSPATIAL*, pages 68:1–68:4, 2015.

[13] J. Dai, B. Yang, C. Guo, and Z. Ding. Personalized route recommendation using big trajectory data. In *Proc. ICDE*, pages 543–554, 2015.

[14] J. Dai, B. Yang, C. Guo, C. S. Jensen, and J. Hu. Path cost distribution estimation using trajectory data. *PVLDB*, 10(3):85–96, 2016.

[15] E. Demidova, P. Fankhauser, X. Zhou, and W. Nejdl. *DivQ*: diversification for keyword search over structured databases. In *Proc. SIGIR*, pages 331–338, 2010.

[16] T. Deng and W. Fan. On the complexity of query result diversification. *PVLDB*, 6(8):577–588, 2013.

[17] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[18] M. Drosou and E. Pitoura. Search result diversification. *SIGMOD Record*, 39(1):41–47, 2010.

[19] D. Eppstein. Finding the k shortest paths. In *Proc. FOCS*, pages 154–165, 1994.

[20] E. Erkut, S. A. Tjandra, and V. Verter. Hazardous materials transportation. *Handbook in OR and MS*, 14:539 – 621, 2007.

[21] E. Erkut and V. Verter. Modeling of transport risk for hazardous materials. *Operations Research*, 46(5):625–642, 1998.

[22] W. Fan, X. Wang, and Y. Wu. Diversified top-k graph pattern matching. *PVLDB*, 6(13):1510–1521, 2013.

[23] C. Godsil and G. F. Royle. *Algebraic graph theory*, volume 207. Springer Science & Business Media, 2013.

[24] Goldberg, V. Andrew, Harrelson, and Chris. Computing the shortest path: A search meets graph theory. In *Proc. SODA*, pages 156–165, 2003.

[25] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *Proc. SIGMOD*, pages 927–940, 2008.

[26] C. Guo, C. S. Jensen, and B. Yang. Towards total traffic awareness. *SIGMOD Record*, 43(3):18–23, 2014.

[27] C. Guo, B. Yang, O. Andersen, C. S. Jensen, and K. Torp. Ecosky: Reducing vehicular environmental impact through eco-routing. In *Proc. ICDE*, pages 1412–1415, 2015.

[28] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[29] W. Henao-Mazo and A. ngel Bravo-Santos. Finding diverse shortest paths for the routing task in wireless sensor networks. In *Proc. ICSNC*, pages 53–58, 2012.

[30] 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):45, 2007.

[31] X. Huang, H. Cheng, R. Li, L. Qin, and J. X. Yu. Top-k structural diversity search in large networks. *PVLDB*, 6(13):1618–1629, 2013.

[32] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for k shortest simple paths. *Networks*, 12(4):411–427, 1982.

[33] M. Levandowsky and D. Winter. Distance between sets. *Nature*, 234(5323):34–35, 1971.

[34] Y. Lim and S. Rhee. An efficient dissimilar path searching method for evacuation routing. *Ksce Journal of Civil Engineering*, 14(1):61–67, 2010.

[35] H. Liu, C. Jin, and A. Zhou. Popular route planning with travel cost estimation. In *Proc. DASFAA*, pages 403–418, 2016.

[36] E. Minack, W. Siberski, and W. Nejdl. Incremental diversification for very large sets: a streaming-based approach. In *Proc. SIGIR*, pages 585–594, 2011.

[37] L. Qin, J. X. Yu, and L. Chang. Diversifying top-k results. *PVLDB*, 5(11):1124–1135, 2012.

[38] L. Talarico, K. Sörensen, and J. Springael. The k-dissimilar vehicle routing problem. *European Journal of Operational Research*, 244(1):129–140, 2015.

[39] M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. T. Jr., and V. J. Tsotras. On query result diversification. In *Proc. ICDE*, pages 1163–1174, 2011.

[40] C. Voss, M. Moll, and L. E. Kavraki. A heuristic approach to finding diverse short paths. In *Proc. ICRA*, pages 4173–4179, 2015.

[41] B. Yang, J. Dai, C. Guo, C. S. Jensen, and J. Hu. Pace: A path-centric paradigm for stochastic path finding. *The VLDB Journal*, 2017.

[42] B. Yang, C. Guo, and C. S. Jensen. Travel cost inference from sparse, spatio-temporally correlated time series using markov models. *PVLDB*, 6(9):769–780, 2013.

[43] B. Yang, C. Guo, C. S. Jensen, M. Kaul, and S. Shang. Stochastic skyline route planning under time-varying uncertainty. In *Proc. ICDE*, pages 136–147, 2014.

[44] B. Yang, C. Guo, Y. Ma, and C. S. Jensen. Toward personalized context-aware routing. *The VLDB Journal*, 24(2):297–318, 2015.

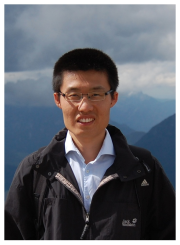
[45] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.

[46] L. Yuan, L. Qin, X. Lin, L. Chang, and W. Zhang. Diversified top-k clique search. In *Proc. ICDE*, pages 387–398, 2015.

[47] B. Zheng, H. Su, K. Zheng, and X. Zhou. Landmark-based route recommendation with crowd intelligence. *Data Science & Engineering*, 1(2):1–15, 2016.

Huiping Liu received the BS degree in Software Engineering from East China Normal University, Shanghai, China, in 2013. Currently, he is a PhD student supervised by Prof. Cheqing Jin. His research mainly focuses on location-based services, massive data mining and processing.

Cheqing Jin Professor at East China Normal University. He received his bachelor and master degrees from Zhejiang University in 1999 and 2002 respectively, and Ph.D. degree in Computer Science from Fudan University in 2005. Before joining ECNU on Oct. 2008, he worked as an Assistant Professor at East China University of Science and Technology. He is the winner of Fok Ying Tung Education Foundation Fourteenth Young Teacher Award. He is a member of Database Technology Committee of China Computer Federation, and serves as a young associate editor of Frontiers of Computer Science, an SCI journal. He has co-authored more than 80 papers, some of which received excellent paper awards, such as the best paper award of Chinese Journal of Computers, best paper award of pervasive computing and embedding from Shanghai Computer Society. His research interests include streaming data management, location-based services, uncertain data management, and sharing database management systems.



Bin Yang is an Associate Professor at Aalborg University, Denmark. He was at Aarhus University, Denmark, during 2011–2014 and at Max-Planck-Institut für Informatik, Germany, during 2010–2011. He received his B.E. and M.E. degrees from Northwestern Polytechnical University, China, in 2004 and 2007, respectively, and the Ph.D. degree in computer science from Fudan University, China in 2010. His research interests include data management and analytics. He has served on the program committees of prestigious conferences and has been an invited reviewer for several top journals, including ICDE, TKDE, The VLDB Journal, ACM Computing Surveys, ACM TSAS, and ACM TIST.

Aoying Zhou Vice President of East China Normal University (ECNU), Dean of School of Data Science and Engineering (DaSE), Professor. He got his master and bachelor degree in Computer Science from Sichuan University, in 1988 and 1985 respectively, and he won his Ph.D. degree from Fudan University in 1993. He is the winner of the National Science Fund for Distinguished Young Scholars supported by National Natural Science Foundation of China (NSFC) and the professorship appointment under Changjiang Scholars Program of Ministry of Education (MoE). He is CCF (China Computer Federation) Fellow, the Vice Director of Database Technology Committee of CCF, and Associate Editor-in-Chief of Chinese Journal of Computer. He served General Chair of ER2004, Vice PC Chair of ICDE2009 and ICDE2012, PC Co-chair of VLDB2014. His research interests include Web data management, data management for data-intensive computing, in-memory cluster computing, distributed transaction processing, benchmarking for big data and performance.