Diversified Top-k Route Planning in Road Network

Zihan Luo[1], Lei Li[2,1,*], Mengxuan Zhang[3], Wen Hua[4], Yehong Xu[2], Xiaofang Zhou[1,2]
[1]Department of CSE, The Hong Kong University of Science and Technology, Hong Kong SAR, China
[2]DSA Thrust, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China
[3]Department of Electrical and Computer Engineering, Iowa State University, USA
[4]School of ITEE, The University of Queens
zluoat@connect.ust.hk,thorli@ust.hk,mxzhang@iastate.edu,w.hu

ABSTRACT

Route planning is ubiquitous and has a profound impact on our daily life. However, the existing path algorithms tend to produce similar paths between similar OD (Origin-Destination) pairs because they optimize query results without considering their influence on the whole network, which further introduces congestions. Therefore, we investigate the problem of diversifying the top-k paths between an OD pair such that their similarities are under a threshold while their total length is minimal. However, the current solutions all depend on the expensive graph traversal which is too slow to apply in practice. Therefore, we first propose an edge deviation and concatenation-based method to avoid the expensive graph search in path enumeration. After that, we dive into the path relations and propose a path similarity computation method with constant complexity, and propose a pruning technique to improve efficiency. Finally, we provide the completeness and efficiency-oriented solutions to further accelerate the query answering. Evaluations on the real-life road networks demonstrate the effectiveness and efficiency of our algorithm over the state-of-the-art.

PVLDB Reference Format:
Zihan Luo, Lei Li, Mengxuan Zhang, Wen Hua, Yehong Xu, Xiaofang Zhou. Diversified Top-k Route Planning in Road Network. PVLDB, 15(11): 3199 - 3212, 2022.
doi:10.14778/3551793.3551863

1 INTRODUCTION

Diversified top-k Shortest Path (DkSP) computation is an important route planning task in road networks. Given an OD (Origin s and Destination t) pair, a similarity threshold τ, and a path number k, it aims to find a path set $P_{s,t}(|P_{s,t}| = k)$ such that the similarity between any pair of the paths in $P_{s,t}$ is no larger than τ and the total length of the k paths is minimal. Figure 1 shows an example weighted graph and Table 1 lists all the shortest paths with the same length of 10 from v_0 to v_8. Suppose that we set τ to 0.4, k to 4 and use the *Jaccard Similarity*, we can have a diversified result $P_{v_0,v_8} = \{p_1, p_2, p_3, p_6\}$.

Even though various path problems have been identified and solved in the past decades, most of their outputs only consider the optimal path for one query OD pair rather than the other queries.

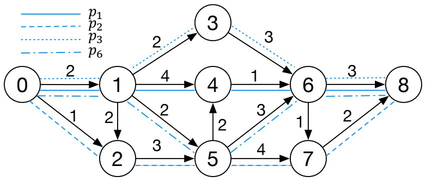


Figure 1: Example Graph

Table 1: Top-8 Shortest Paths with Length 10 from v_0 to v_8

Path ID	Path Details
p_0	$v_0 \rightarrow v_2 \rightarrow v_5 \rightarrow v_6 \rightarrow v_8$
p_1	$v_0 \rightarrow v_1 \rightarrow v_4 \rightarrow v_6 \rightarrow v_8$
p_2	$v_0 \rightarrow v_2 \rightarrow v_5 \rightarrow v_7 \rightarrow v_8$
p_3	$v_0 \rightarrow v_1 \rightarrow v_3 \rightarrow v_6 \rightarrow v_8$
p_4	$v_0 \rightarrow v_2 \rightarrow v_5 \rightarrow v_4 \rightarrow v_6 \rightarrow v_8$
p_5	$v_0 \rightarrow v_1 \rightarrow v_5 \rightarrow v_7 \rightarrow v_8$
p_6	$v_0 \rightarrow v_1 \rightarrow v_5 \rightarrow v_6 \rightarrow v_8$
p_7	$v_0 \rightarrow v_1 \rightarrow v_5 \rightarrow v_4 \rightarrow v_6 \rightarrow v_8$

Consequently, when many path queries come with localized origins and destinations, their shortest paths would have numerous overlapping, which would lead many vehicles traveling into the same roads at the same time and cause traffic congestion (travel time increases as the traffic flow increases [33]), which further deteriorates the user experience. For example, if a path algorithm chooses p_1 as the result, then all the queries from v_0 to v_8 would head to the same route while other roads are idle. If we could distribute the queries evenly to p_1, p_2, p_3 and p_6, then many roads could have fewer vehicles and lighter traffic flow ($(v_0, v1)$ and (v_6, v_8) reduce by 25% while (v_1, v_4) and (v_4, v_6) reduce by 75%) such that the traffic condition becomes better and all the queries can travel faster in practice. Hence, diversified paths provision in real-life route planning helps to ease the traffic burden by dispersing the traffic flow with minimal cost. In addition, the diversified path is also widely applied in the hazardous material shipments [6], routing in wireless sensor network [22], and evacuation planning [34]. In this paper, we study the *DkSP* problem and propose an efficient algorithm to answer the *DkSP* query.

Because the *DkSP* query requires multiple results as the output, its computation should be based on the methods that satisfy this property, and we categorize them into two streams: *disjoint edge-based* and *k-path-based*. The *disjoint edge-based* method [8, 20, 24, 25] aims to find a set of k paths connecting k OD pairs such that no edge exists on the same path, and its loose version relaxes the existence restriction to at most c paths. By setting the k OD pairs to the same source and destination, they can be used to provide diversified paths. For example, $\{p_2, p_3\}$ is a pair of edge-disjoint

Proceedings of the VLDB Endowment, Vol. 15, No. 11 ISSN 2150-8097.
doi:10.14778/3551793.3551863

[*] Lei Li is the corresponding author.

paths, and $\{p_1, p_2\}$ is another pair. However, although the results are diversified naturally, the existing algorithms only guarantee the existence of such paths without considering the distance. For example, if p_3 grows to 100, $\{p_2, p_3\}$ still has the same quality as $\{p_1, p_2\}$ from these methods' point of view. Therefore, the results could be impractically long just to avoid overlapping. Moreover, because the road intersections usually have a very small degree (at most 3 or 4), an OD pair can support only a few edge-disjoint paths, which also limits its ability to generate enough paths. Like in the example graph, we can only have at most two edge-disjoint paths at the same time. Furthermore, when there exists a natural network cut like a bridge or tunnel then the result number is capped by these cuts. Even with the loose version, it might violate the similarity requirement because returning two identical paths also satisfies the definition. Therefore, the *disjoint edge-based* methods are not suitable for real-life diversified route planning.

The second kind of methods utilizes the *k-path* enumeration, which returns the top-k loopless shortest paths between an OD pair [7, 18, 36, 41, 46, 55, 56]. However, computing the loopless *k-path* is very time-consuming, and most of the paths returned by the existing *k-path* algorithms have a very high overlapping. For example, among all the eight paths, $\{p_1, p_3, p_6\}$, $\{p_2, p_5\}$ and $\{p_4, p_7\}$ have only one vertex difference. Therefore, if we provide these sets of routes to the users, congestions become inevitable. Moreover, it requires a much larger k to find enough diversified routes, which further prolongs the query time. To prune similar paths as early as possible and only expand the dissimilar ones, *KSPD* [35] uses similarity and distance lower bound to guide the search heuristically, and *MP* [12] uses path dominance relation of the *Overlap Ratio Min* similarity function [2, 12] to prune the search space. However, they still need to search the graph, so they still take hundreds or thousands of seconds to finish a query, which is not practical for real-life use. In addition, *alternative routing* [10, 21, 26, 27, 39] is a set of heuristic *k-path* methods that are fast to run, but they suffer from drawbacks like incomplete result set, higher similarity, and longer results.

Overall, the existing solutions have several limitations which restrict them from being applicable in real-life route planning: Firstly, the graph search in these methods is the bottleneck of performance. For instance, a *Dijkstra*'s or A^* search is used to search for the next candidate path, which normally takes tens of milliseconds. Then a typical *DkSP* query might need tens of thousands of candidate paths, so the overall running time becomes hundreds of seconds. Therefore, we propose a path deviation and concatenation-based enumeration method to avoid the expensive search. Specifically, the new paths are formed incrementally by concatenating the existing sub-paths in a *Shortest Path Tree (SPT)*. Since each new path is generated through concatenation operation rather than graph traversal, the path enumeration efficiency is improved by orders of magnitude, which contributes to quick diversified path finding.

Secondly, whenever we have a path candidate, we have to compute its similarity with the existing results. However, the current best method still takes $O(|p|)$ time for each similarity computation, where $|p|$ is the number of edges in a path. Consequently, suppose we have k results and have tested N candidates, then the total time spent on similarity computation is $O(|p| \times k \times N)$. It should be noted that $|p|$ is normally on 10^2 and N is normally higher than 10^3. To reduce this high complexity, we dive into the path generation theory and reveal the ancestor-descendant relations of the generated paths, then propose an efficient similarity computation method that only takes $O(1)$ time when such relations exist.

Thirdly, N could still be huge and prolong both the overall path enumeration and similarity computation time. To shrink the candidate path set, we further propose an effective pruning method for the *Overlap Ratio Min* similarity function [2, 12], which is harder to compute than other similarity functions. By identifying the tight upper-bounds of the deviation position, we can avoid a large portion of candidate paths beforehand.

Finally, as proved in [35], finding the optimal *DkSP* is NP-H so we resort to the approximate result like all the previous works. In addition, because there are still enormous paths needed to enumerate before the results are obtained, and sometimes it is impossible to satisfy the requirement, we provide two completeness and efficiency-oriented solutions to further improve the effectiveness and efficiency. As validated in our experiments on real-life road networks, our proposed methods can achieve orders of magnitude faster than the state-of-the-art approaches.

Our major contribution can be summarized as below:

- We study the *DkSP* problem to ease the traffic congestion from the routing result's perspective and propose an edge deviation and concatenation-based path enumeration method to process the *DkSP* query efficiently.
- We analyze the path relations and propose an efficient path similarity computation method in constant time. We also propose a pruning technique for the *Overlap Ratio Min* similarity to reduce the path candidate number.
- We propose two completeness and efficiency-oriented solutions to further boost efficiency.
- We conduct extensive evaluations and verify the superiority of our approach compared with the state-of-art algorithms.

2 RELATED WORK

2.1 Route Planning

Over the years, various of path problems are identified under different scenarios, like *shortest path* [4, 14, 19, 44, 58, 61, 62] that finds a path with minimum distance in a static graph, *timetable fastest path* [49, 51] that finds paths in discrete connected environments like public transportation network, *time-dependent fastest path* [15, 28–30, 32] that considers the continuous travel time, and *constraint path* [37, 38, 50, 54] that finds paths which also satisfying other constraints like tolls and costs. As discussed previously, the results of these algorithms are only locally optimal. When it comes to answer a set of queries globally, the *batch shortest path* [31, 59, 60] algorithms utilize *path coherence* phenomenon and shared computation to reduce the overall computation costs. However, this sharing nature would lead several OD pairs traveling through a large portion of the same path, which further generates congestions in real life, so they are more suitable for route recommendation to support decision making but not the navigation.

2.2 k-Path Selection

This kind of approach tests the loopless paths in the distance increasing order [7, 18, 36, 41, 46, 55, 56] but with a very high complexity of $O(k|V|(V|log|V| + |E|))$. In addition, the resulting k paths have very high overlapping with similar distance [3], which requires a large number of paths before the results are dissimilar enough. *k-Path* is also utilized for route re-planning [40, 45, 52]. The *shortest path tree* is a structure that is widely used in the path enumeration searching towards it [18, 36, 41] to connecting to it [7, 46]. But they used it as a tool to reduce the search space but did not dig into it to reveal why and how to use SPT to enumerate correctly in theory, and what are the relations among the enumerated paths such that can be used to determine the overlapping efficiently.

2.3 Diversified and Alternative Route Planning

KSPD [35] solves the same problem by extending the *k-Path* algorithms [7, 18] with path lower bounds for pruning. However, it still has to search the graph to find the next candidate path, so its complexity is the same as the *k-Path*. *kDPwML*[11] is the only work that can output exact optimal result, but it has a very high complexity such that it can only run on toy graphs with hundreds of vertices but cannot scale to the ones with more than 1k vertices. *kSPwLO* [9, 10, 12] provide another set of search-based algorithms. But its searching strategy has no direction or bound so it has a very high complexity. Another stream of methods called *alternative routing* [27] resorts to heuristic solutions through *via-node connection* [21, 26, 39] or *penalty blocking* [10, 21] for faster query answering, but their result quality is not guaranteed (allowing longer paths, higher similarity, and incomplete result set). *Ridesharing Planning* [48, 57] also returns several alternative routes, but its goals are scheduling the workers for optimal revenue, served requests, and travel time, not the routes' diversity. Therefore, they cannot solve the diversified problem.

3 PRELIMINARY

3.1 Road Network and k-Path

A road network is denoted as a directed graph $G(V, E)$, where $V = \{v_i\}$ is a set of vertices representing the intersections and $E \subseteq V \times V = \{(v_i, v_j)\}$ is a set of edges representing the road segments. For any two vertices v_i and v_j that is connected by an edge (v_i, v_j), we say v_i is v_j's in-neighbor and (v_i, v_j) is v_j's in-edge, and v_j is v_i's out-neighbor and (v_i, v_j) is v_i's out-edge. Each edge (v_i, v_j) is also associated with a non-negative numerical weight $w(v_i, v_j)$ that represents the cost from v_i to v_j, which can be distance, travel time, fuel consumption, toll charge, and etc. A path from s to t is a sequence of vertices $p = \langle v_0, v_1, \ldots, v_k \rangle$ with $v_0 = s$, $v_k = t$, and $(v_i, v_{i+1}) \in E$. The length of this path p is $d(p) = \Sigma_{i=0}^{k-1} w(v_i, v_{i+1})$, and the shortest path is the one with the minimum $d(p)$. Obviously, there are various of different paths between any OD pair. If we rank them in a list of size k, we have the k-Path of it:

Definition 1. ***(k-Path)***. *Given a graph G, an OD pair (s, t) and a path number k, the k-Path is a set of paths $P_{s,t} = \{p_1, \ldots, p_k\}$ from s to t such that $d(p_i) \leq d(p_{i+1})$, and $d(p_j) \geq d(p_i), \forall p_i \in P_{s,t}$ and $p_j \notin P_{s,t}$.*

However, not every path in this result set is useful especially in the route planning scenario as it has requirements only on the distance but nothing else. Therefore, if we take a little detour out of the current location and travel back, this path would also appear in the list as long as its distance is no longer than the largest result. Apparently, no one would like to take this kind of obviously wasteful detour during his trip, and such a detour would also deteriorate the traffic condition rather than improve it. Therefore, restriction on the detour is essential to make the k-Path practical in real-life:

Definition 2. ***(k-Simple Path)***. *Given a graph G, an OD pair (s, t) and a path number k, the k-Path is a set of simple paths $P_{s,t} = \{p_1, \ldots, p_k\}$ from s to t such that $d(p_i) \leq d(p_{i+1})$, and $d(p_j) \geq d(p_i), \forall p_i \in P_{s,t}$ and $p_j \notin P_{s,t}$. The simple path requires $\forall p \in P_{s,t}$ and $\forall v_i \in p$, v_i only appears in p once.*

The *simple path* is also called *loopless path*. From now on, we use k-Path to denote k-Simple Path for simplicity.

3.2 Path Similarity

The paths in the k-Path results have a very high overlapping between each other because the paths are found in the distance increasing order, and also because the k-Path algorithm generates the candidates based on the existing results. Therefore, we need a metric to distinguish the difference between the paths in the k-Path result. Because the path is a special case of the trajectory data without temporal information, various trajectory similarity measurements [3, 9, 12, 13, 16, 17, 35, 47, 53] can also be applied. Given any two paths p_i and p_j, $p_i \cap p_j$ is the set of common edges and $p_i \cup p_j$ is the set of total edges. We use $d(p_i \cap p_j)$ and $d(p_i \cup p_j)$ to denote their corresponding lengths. Some of the widely used similarities are list below:

(1) *Jaccard Similarity*: $S_1 = \dfrac{d(p_i \cap p_j)}{d(p_i \cup p_j)}$ [13, 16, 17, 35, 53]

(2) *Arithmetic Average*: $S_2 = \dfrac{d(p_i \cap p_j)}{2d(p_i)} + \dfrac{d(p_i \cap p_j)}{2d(p_j)}$ [3, 16, 17, 35]

(3) *Geometric Average*: $S_3 = \sqrt{\dfrac{d(p_i \cap p_j)^2}{d(p_i)d(p_j)}}$ [16, 17, 35]

(4) *Overlap Ratio Max*: $S_4 = \dfrac{d(p_i \cap p_j)}{max(d(p_i), d(p_j))}$ [16, 17, 35]

(5) *Overlap Ratio Min*: $S_5 = \dfrac{d(p_i \cap p_j)}{min(d(p_i), d(p_j))}$ [9, 12, 35]

It should be noted our problem is not designed for any specific similarity so any measurement above can be used directly. In addition, the first four similarities have the longer paths in their denominators so their similarity values would decrease as more paths are searched. However, the fifth similarity is non-decreasing so it is stricter and harder to find the results. The similarity of a path set is defined below:

Definition 3. ***(Path Set Similarity)***. *Given a set of paths P, its max similarity $S_{max}(P) = max(S(p_i, p_j))$.*

3.3 Diversified Top-k Shortest Path

Now we are ready to define our problem below:

Definition 4. ***(Diversified Top-k Shortest Path Problem)***. *Given a graph G, an OD pair (s, t), a path number k, and a similarity threshold $\tau \in [0, 1]$, the diversified top-k shortest path problem finds a set of simple paths $P_{s,t}$ from s to t with $|P_{s,t}| \leq k$ such that $S(P_{s,t}) \leq \tau$, and $\nexists P'_{s,t}$ with $S(P'_{s,t}) \leq \tau$ such that (1) $|P_{s,t}| < |P'_{s,t}| \leq k$, or (2) $|P'_{s,t}| = |P_{s,t}|$ with $\Sigma_{p_i \in P'_{s,t}} d(p_i) < \Sigma_{p_i \in P_{s,t}} d(p_i)$.*

It should be noted that this problem actually has two objectives: one on the number of the dissimilar paths, and another one on the total path length minimization. Specifically, the first restriction requires the maximum size of the result set to prevent the case where the shortest path itself has the smallest total length, which also satisfies the similarity threshold. Moreover, there is no guarantee that a result set of size k could exist. For example, when we set τ to 0, the possible k could be as small as 3 or 4 due to the small average vertex degree in road network. The second restriction requires the minimal total length among the maximal sets.

Algorithm 1: Approximate DkSP Framework

```
Input: Graph $G$, OD Pair $(s, t)$, $k$, $\tau$
Output: Diversified Top-$k$ Shortest Paths $P_{s,t}$
$P_{s,t} \leftarrow P_{s,t} \bigcup \{p\}$; //$p$ is the shortest path
while $|P_{s,t}| < k$ and $p \leftarrow$ the next shortest path do
    if $\forall p' \in |P_{s,t}|, S(p, p') \leq \tau$ then
        $P_{s,t} \leftarrow P_{s,t} \cup \{p\}$;
return $P_{s,t}$;
```

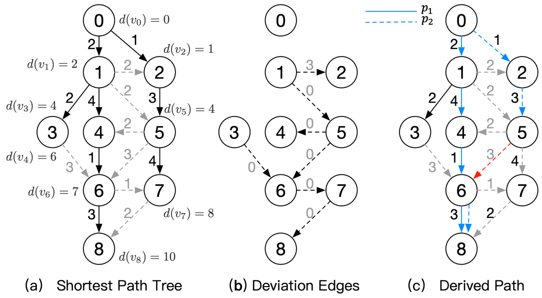
As proved in [12, 35] and validated by [11], this problem is NP-H. The intuition of the hardness is that the shortest path does not necessarily belong to the optimal path set, we do not even know which one should be the first result as we enumerate the paths. In fact, suppose the last path of optimal result is the c^{th} shortest path, then we have $\binom{c}{k}$ choices to validate their similarities and compare their total length, and c could be tens of thousands as validated in our experiments. Such a huge candidate set prohibits us from finding the optimal result. Therefore, we do not attempt to find the exact result but resort to a greedy one with approximation ratio. The overall procedure is presented in Algorithm 1. It keeps enumerating the k-Path in the distance increasing order and adds the current path if it satisfies the similarity criteria with the existing ones. The procedures terminates when k such paths are obtained or no path exists. Suppose $P^*_{s,t}$ is the optimal result. The approximation ratio for the path number is k as in the worst case, $|P^*_{s,t}| = k$ while $P_{s,t}$ only contains the shortest path. The approximation ratio for the total length is $\Sigma_{p_i \in P_{s,t}} d(p_i) / \Sigma_{p_i \in P^*_{s,t}} d(p_i)$.

4 DIVERSIFIED TOP-k SHORTEST PATH

In this section, we elaborate the shortest path tree concatenation based diversified top-k shortest path generation in detail. Section 4.1 introduces the basis of generating new longer paths from the existing one with the *SPT*. Then we classify the paths to ensure the correctness of *k-Path* generation in Section 4.2. Finally, we present how to compute the diversified k-Path in Section 4.3.

4.1 Shortest Path Tree SPT

Given a source vertex s, a shortest path tree T_s is a directed spanning tree of G rooted at s such that the path distance from s to any $v \in V$ in T_s is the same as the shortest distance from s to v in G. Because these paths all have the same source s, we can use $d(v_i)$ to denote the shortest distance for short. For example, Figure 2 shows a *SPT* of our example graph with black edges representing the tree edges and grey edges presenting the remaining ones. Accordingly, we categorize the graph edges into two types: *Tree Edge* E_T that exist in the *SPT* and *Deviation Edge* $E_D = E - E_T$ that are left over. For an edge $(v_i, v_j) \in E$, we call $c_{ij} = w(v_i, v_j) + d(v_i) - d(v_j)$ its deviation cost. Obviously, $c_{ij} = 0$ if $(v_i, v_j) \in E_T$, and $c_{ij} \geq 0$ if $(v_i, v_j) \in E_D$. The *SPT* can be constructed with *Dijkstra*'s in $O(|V| \log |V| + |E|)$ time, and the deviation costs can be obtained in $O(|E|)$ time.



(a) Shortest Path Tree (b) Deviation Edges (c) Derived Path

Figure 2: SPT and Concatenation Example

We can utilize the *SPT* and the deviation edges to generate the new longer paths from the current path. Suppose we have a shortest path $p_1 = \langle s, \ldots, t \rangle$ in T_s. Then $E^1_D = \{(v_i, v_j) | v_j \in p_1 \bigwedge v_i \notin p_1\}$ is the set of deviation edges from p_1. $\forall (v_i, v_j) \in E^1_D$, we can get a new path $p_1^{(v_i, v_j)} = p_{s \rightarrow v_i} \oplus (v_i, v_j) \oplus p_{v_j \rightarrow t}$. Apparently, the new path is the concatenation ($\oplus$) result of the following three parts:

(1) *SPT Part*: $p_{s \rightarrow v_i}$ is a shortest path from s to v_i in T_s
(2) *Deviation Edge*: (v_i, v_j)
(3) *Parent Part*: $p_{v_j \rightarrow t}$ is the sub-path of p_1 from v_j

Because the *deviation edge* $\oplus$ *parent part* won't be changed in the future path generation, we call them the *fixed part*. Therefore, the new path can also be viewed as a concatenation of the *SPT Part* and *Fixed part*. The length of this new path $d(p_1^{(v_i, v_j)})$ can be obtained directly with $d(p_1) + c_{ij}$. This is because

$$
\begin{aligned}
d(p_1^{(v_i, v_j)}) &= d(v_i) + w(v_i, v_j) + d(v_j \rightarrow t) \\
&= c_{ij} - w(v_i, v_j) + d(v_j) + w(v_i, v_j) + d(v_j \rightarrow t) \\
&= c_{ij} + d(v_j) + d(v_j \rightarrow t) \\
&= c_{ij} + d(p_1)
\end{aligned}
$$

Figure 2-(c) shows a shortest path $p_1 = \langle v_0, v_1, v_4, v_6, v_8 \rangle$ and it has three deviation edges $E^1_D = \{(v_3, v_6), (v_5, v_6), (v_5, v_4), (v_7, v_8)\}$ with deviation costs of $c_{3,6} = c_{5,6} = c_{7,8} = 0$. Suppose we choose (v_5, v_6) as the next deviation edge, we can get a new path $p_2 = p_{v_0 \rightarrow v_5} \oplus (v_5, v_6) \oplus p_{v_6 \rightarrow v_8} = \{v_0, v_2, v_5, v_6, v_8\}$ with distance $d(p_2) = d(p_1) + c_{5,6} = 10$. p_1 is called p_2's *parent path* because p_2 is directly generated from p_1. If we sort the deviation edges based on the deviation cost and generate the new paths in the increasing order, it is guaranteed the latter ones are no shorter than the earlier ones.

However, the new path generated by the above procedure may contain loops. For example, a new edge (v_8, v_4) could create a path $p_{v_0 \rightarrow v_8} \oplus (v_8, v_4) \oplus p_{v_4 \rightarrow v_8}$ that has a loop $\langle v_8, v_4, v_6, v_8 \rangle$. Although traversing the new path could detect the loop, it is time-consuming when the path is long. Therefore, we will analyze the cause of the loops theoretically and avoid it as early as possible in Section 4.3.

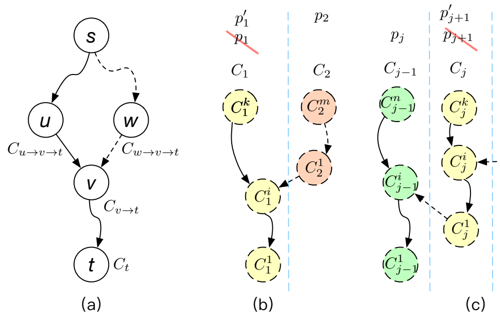


Figure 3: Path Classification Example

4.2 Path Classification

Following the previous path generation method, suppose the shortest path is $p_1 = \langle s, \ldots, u, v, \ldots, t\rangle$ and we have generated a new path $p_2 = p_{s\to w} \oplus (w, v) \oplus p_{v\to t}$ by deviating from (w, v) with the smallest deviation cost, as illustrated in Figure 3. Apparently, p_1 and p_2 share the same sub-path $p_{v\to t}$, and they are the shortest and the second shortest paths among all the paths ending with $p_{v\to t}$, which we can regard as a class of paths. Consequently, we generalize this notion and define the path class below.

Definition 5. *(Path Classification). A path class $C_{p_{v\to t}}$ contains all the paths from s to t that share the same sub-path $p_{v\to t}$.*

Because the path classes are denoted by $p_{v\to t}$, we can also say these classes are represented by their fixed part when generated, and this is the reason why we call it *fixed part*. Whenever we generated a new path by deviation and concatenation, we have created a new class of paths from the previous one. Specifically, if $p_{v\to t}$ is the sub-path of $p_{u\to t}$, then $C_{p_{v\to t}}$ is the ancestor class of $C_{p_{u\to t}}$. If $p_{u\to t} = (u, v) \oplus p_{v\to t}$, then $C_{p_{v\to t}}$ is the parent class of $C_{p_{u\to t}}$. For the shortest path p_1 with $|p_1|$ edges, it has $|p_1|$ path classes with a chain of "parent-children" relation and the class of the last edge is the ancestor of them all. In addition, $C_{\langle t\rangle}$ is the ancestor of all the possible classes. Suppose t has k in-neighbors $\{v_1, \ldots, v_k\}$, then $C_{\langle v_1,t\rangle}, \ldots, C_{\langle v_k,t\rangle}$ are the k sub-path classes that cover the entire path space, with the shortest path p_1 belonging to one of them. Therefore, enumerating the next shortest path is equivalent to finding the next shortest one among all these k sub-path classes.

4.3 Diversified k-Path Enumeration

Now we are ready to present how to enumerate the diversified k-path. Firstly, the shortest path p_1 contains a series of path classes if we keep adding the edges of p_1 reversely from t back to s. For simplicity, we change the path class's naming rule from the perspective of the fixed part to the perspective of the path ID. Specifically, we denote sub-classes covered by the shortest path p_1 as $C_1 = \{C_1^1, \ldots, C_1^k\}$ as shown in Figure 3-(b), with C_1^1 being the same as the previous $C_{\langle t\rangle}$, C_1^2 being $C_{\langle t-1,t\rangle}$, and etc. The relations of the path classes in C_1 have the following two properties:

Property 1. ***Consecutive Coverage Property:*** *$\forall 1 \le i < j \le k$, C_1^i covers the sub-spaces of C_1^j.*

Proof. C_1^i's fixed part is a sub-path of C_1^j's fixed part. □

Property 2. ***Complete Coverage Property:*** *C_1 covers the entire path space.*

Proof. Because C_1^1 is the same as $C_{\langle t\rangle}$, then $C_1^1 \in C_1$ covers the complete path space. □

The 2^{nd} Shortest Path. Because of the Property 2, if we can find the local shortest path for each of these sub-classes in C_1, then the global second shortest path is the shortest one among them all. In the *Yen*'s-based (or search-based) methods, these sub-class shortest paths are found by graph search, which is intolerably time-consuming. Fortunately, because we have already computed the deviation cost of p_1's in-neighbors, we can utilize them to avoid graph searching. Specifically, for any sub-class C_1^j, suppose it has a set of in-neighbors $\{v_{j-1}, v_i^1, v_i^2, \ldots, v_i^k\}$. Then the deviation cost of (v_{j-1}, v_j) is 0 because it is on p_1, and we do not consider it. As for the remaining in-neighbors, each of them also corresponds to a sub-class, and their shortest distance can be retrieved in constant time by adding their deviation cost with $d(p_1)$. Because they all share the same $d(p_1)$, then the one with the smallest deviation cost is the next shortest path of C_1^j. If we select the smallest one among all the in-neighbors of p_1, then we are guaranteed to find the next shortest path in C_1, which is also the global next shortest.

The 3^{rd} Shortest Path. Suppose C_1^j generates the second shortest path p_2 from v_j by connecting to $p_{s\to v_i}$ via (v_i, v_j). Since the in-neighbor (v_i, v_j) has been used in C_1, it will not be selected again (otherwise, we will keep generating the same p_2 again and again). Therefore, we have to introduce another set of path classes that have the same common fixed part $p_{(v_i,v_j)\to t}$, and call them $C_2 = \{C_2^1, \ldots, C_2^m\}$ as they are generated from p_2. An example is illustrated in Figure 3-(b). Now that C_1 and C_2 have covered the entire path space, so the next shortest path exists in $C_1 \cup C_2$, and we can find the next shortest paths from both of C_1 and C_2.

The $(j + 1)^{th}$ Shortest Path. Suppose we have obtained the $(j + 1)^{th}$ shortest path p_{j+1} as shown in Figure 3-(c). During the generation, we had the path classes $C_1 \cup \cdots \cup C_j$ and found the next shortest one for each of them. Among these next shortest paths, suppose C_j's path is chosen as the next one, and we now have a new set of path classes $C_1 \cup \cdots \cup C_j \cup C_{j+1}$, and the next shortest path is the shortest one among them. We can put these path classes into three categories: 1) C_{j+1} is the newest set of classes that have not derived any new paths yet; 2) C_j is the class that has just used C_{j+1} as its next shortest one and has not found the next of its next path; 3) All the remaining classes have their shortest paths generated but not selected in the previous round. Therefore, we only need to find the shortest paths from C_{j+1} and C_j to make sure all the path classes have their shortest path generated.

Loop Detection. Finally, each time we concatenate to the SPT via a deviate edge, the position relation between the SPT and the path's fixed part is uncontrollable, and sometimes it may create loops. The *SPT* branches can be divided into only three categories depending on the location of the *SPT* as illustrated in Figure 4-(b):

(1) *Loopless Branch:* The SPT branches that originating from p_2's SPT part (grey);
(2) *Fixed Loop Branch:* The vertices on p_2's fixed part;

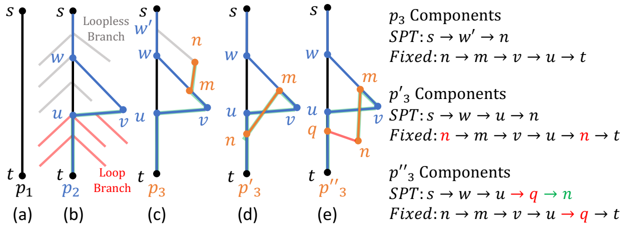

s
t
p_1
(a)
s
Loopless Branch
w
u
v
t
p_2
Loop Branch
(b)
s
w'
w
n
m
u
v
t
p_3
(c)
s
w
m
u
v
n
t
p'_3
(d)
s
w
m
u
v
q
n
t
p''_3
(e)
p_3 Components
$SPT: s \to w' \to n$
$Fixed: n \to m \to v \to u \to t$
p'_3 Components
$SPT: s \to w \to u \to n$
$Fixed: n \to m \to v \to u \to n \to t$
p''_3 Components
$SPT: s \to w \to u \to q \to n$
$Fixed: n \to m \to v \to u \to q \to t$

Figure 4: Loop Example. Fixed Parts are Shadowed in Green.

(3) *Non-Fixed Branch:* The SPT branches that originating from p_2's fixed part (red).

Then depending on which branch the deviate edge connects to, we can have the following path types:

(1) *Loopless Path*: If the deviate edge connects to the *loopless branch*, then the generated path has no loop, so it is safe to use as a candidate;

(2) *Fixed Loop Path*: If the deviate edge connects to the *fixed loop branch*, the generated path has a loop, and this loop exists in the fixed part. Because the fixed part won't be changed, the loop will exist in all the paths it generates. Thus, it cannot be a candidate and cannot be used to generate new paths;

(3) *Non-Fixed Loop Path*: If the deviate edge connects to the *non-fixed loop branch*, the generated path has a loop, but the loop is not in the fixed part. It cannot be a candidate, but it still has a chance to generate a loopless path. The branch from the fixed part to the deviate edge is the last chance to break the loop.

For example in Figure 4, (a) is the shortest path p_1 from s to t, (b) is the second shortest path p_2 (blue) deviates from u and connected to SPT on v, and (c) is the third shortest path from p_2 that has no loop. (d) and (e) are the third shortest path p_3 that generated from p_2's SPT part $s \longrightarrow w \longrightarrow v$. Suppose in (d), the deviate vertex m connects to n, which is the end of the SPT part $s \longrightarrow w \longrightarrow u \longrightarrow n$, and also on the fixed part $n \longrightarrow m \longrightarrow v \longrightarrow u \longrightarrow n \longrightarrow t$. Therefore, n appears twice in the fixed part and creates a *fixed loop* in p_3. When n does not appear in the fixed part, it can still create a loop as shown in (e): the SPT part $s \longrightarrow w \longrightarrow u \longrightarrow q \longrightarrow n$ has an overlapping segment $u \longrightarrow q$ with the fixed part $n \longrightarrow m \longrightarrow v \longrightarrow u \longrightarrow q \longrightarrow t$, thus creates a loop from q to q. However, as this loop does not entirely exist in the fixed part, there is still a chance to break it. Specifically, if we deviate anywhere from the subpath $q \longrightarrow n$ and connect to another branch of the SPT, we could break this loop. On the other hand, if deviate anywhere from the subpath $s \longrightarrow w \longrightarrow u \longrightarrow q$, this loop would become fixed and persists in the path, which generates a *fixed loop* path. In fact, we can view (d) as a special case of (e), with the subpath $q \longrightarrow n$ has no edge (q and n are the same vertex). In summary, the subpath $q \longrightarrow n$ is the last chance to get rid of this *non-fixed loop*, and we can prune all the deviate edges in subpath $s \longrightarrow w \longrightarrow u \longrightarrow q$.

The details of the diversified k-Path enumeration are shown in Algorithm 2. Firstly, we use a heap H to organize all the candidate paths in the length-increasing order and use a set of heaps D to organize the deviation edges of each path in the deviation cost increasing order. Each path p in H has a parent path $p.parent$ that generates p except for the first shortest path. During each iteration, we pop out the top path p_{top} from H as the next candidate path. If it has no non-fixed loop, then we further compute its similarity with the paths in the result set. If its similarities are smaller than τ, then we add it to the result set. After that, we generate two more candidate from p_{top} and its parent path $p_{top}.parent$. This corresponds to the previous path classes C_j and C_{j+1}. Then for each path p of these two paths, we test their next path by deviating from the smallest edge (u, v) in their deviation heap $D[p]$. The new paths is formed by concatenating $p_{s \to u}$, (u, v), and $p_{v \to t}$. After that, we test the existence of the fixed loop. Specifically, if u exists in the fixed part $p_{v \to t}$, then it contains a fixed loop so we can drop it and test the next shortest one. This procedure runs on until a path p' that has no fixed loop is found. Because p generates p', we set $p'.parent$ as p. Then if p' has no loop, we put all the deviate edges along path $p_{s,u}$ to its deviation heap $D[p']$. Otherwise, we only need to put the deviate edges along the subpath $p_{q \to u}$ that do not create a fixed loop into $D[p']$. The enumeration process terminates when k diversified results are obtained or H is empty.

Algorithm 2: Diversified k-Path Enumeration

```
Input: Graph $G$, OD Pair $(s, t)$, $k$, $\tau$, Similarity Function $S$
Output: Diversified Top-$k$ Shortest Paths $P_{s,t}$
$H.insert(p)$; //$p$ is the shortest path, $H$ is a min-heap
$D[p].insert(E_D^p)$; //$D[p]$ is a min-heap for $E_D^p$
while $|P_{s,t}| < k$ and $!H.empty()$ do
    $p_{top} \leftarrow H.pop()$;
    if $p_{top}$ has no loop and $S(P_{s,t}, p_{top}) \le \tau$ then
        $P_{s,t} \leftarrow P_{s,t} \cup \{p_{top}\}$;
    foreach $p \in \{p_{top}, p_{top}.parent\}$ do
        $(u, v) \leftarrow D[p].pop()$; $p' \leftarrow p_{s \to u} \oplus (u, v) \oplus p_{v \to t}$;
        $p'.parent \leftarrow p$; $H.insert(p')$; $p_{tmp} \leftarrow (u, v) \oplus p_{v \to t}$;
        while $v \in V$ do
            $p_{tmp} \leftarrow v \oplus p_{tmp}$;
            for $(v', v) \in E_D$ do
                if $v' \notin p_{tmp}$ then
                    $D[p'].insert((v', v))$;
            $v \leftarrow v's$ parent in SPT;
return $P_{s,t}$;
```

Theorem 1. *Algorithm 2 can find the approximate diversified top-k paths.*

Proof. Firstly, each time p and p's parent path find their next shortest path and put them into H, so H contains all the shortest paths from all the current sub-classes. Therefore, the next shortest path popped from H is the global next shortest path. Secondly, the similarity test follows the procedure in Algorithm 1. Therefore, Algorithm 2 can find the approximate diversified top-k paths. □

Complexity. Each path has at most $|E|$ deviation edges and organizing them take $O(|E| \log |E|)$ time. For the shortest path generation, suppose we have tested α paths, then we need to generate 2α paths while at most α shortest paths exist in H. Together with the *SPT* construction and the similarity computation that cost $O(Similarity)$, the total time complexity is $O(|V| \log |V| + |E| + 2\alpha \times |E| \log |E| + \alpha \log \alpha + k\alpha \times O(Similarity)) = O(|V| \log |V| + \alpha(|E| \log |E| + \log \alpha + k \times O(Similarity)))$. The space complexity is $O(\alpha |E| \log |E| + \alpha \log \alpha)$.

5 EFFICIENT PATH SIM

In this section, we present a fast
method that utilizes the structure
eration relations of the paths. In the
how to compute the similarity in
5.2 presents the framework of our
After that, Section 5.3 describes h
SPT part, and Section 5.4 and 5.5 p
ilarity of the remaining parts comp
and the other results, respectively
technique for S_5.

5.1 Baseline Similarity (

As presented in Section 3.2, give
five similarity functions are the co
$d(p_i \cap p_j)$, and $d(p_i \cup p_j)$ in consta
known, and $d(p_i \cup p_j)$ can be comp
computing $d(p_i \cap p_j)$ is at the cor
The straightforward way to compu
of the two paths in nested loop,
becomes a performance bottlenec
A better way is putting each resu
only need to loop over the curre
edges exist in the hasp map. Alth
to $O(n)$, it is still not efficient eno
of thousands of such computations during enumeration.

5.2 Similarity Computation Framework

To reduce the path intersection length computation, we propose a method that can avoid comparing the edges by utilizing the information generated during the enumeration. As explained in Section 4.1, each newly generated path is made up of two parts: *SPT part* and *fixed part*. Then we can split the path intersection computation into these two parts separately.

Specifically, each time we generate a new path p_i, we connect to a new SPT branch via a deviate edge, and this SPT branch is p_i's SPT part. Then the next time we generate a path p_j from p_i, we deviate from p_i's SPT part via another deviate edge and connect to another SPT branch, and this branch is p_j's SPT part. Therefore, the paths generated by Algorithm 2 is made up of a series of SPT segments connected by non-SPT (deviate) edges. For example, Figure 5 shows four paths, and each is generated from the previous one. Therefore, p_1 is the shortest path and it is entirely on the SPT. p_2 has one deviated edge, p_3 has two, and p_4 has three. The SPT parts are labeled in yellow, and the remaining parts are the fixed part. Because the intersection of the SPT parts is always on the SPT and irrelevant to the deviate edges, we can compute the intersection only with SPT's information in $O(1)$ time, and we will discuss it first separately. As for the fixed part intersection, the new paths inherit the fixed parts of their parents and the ancestors, so we can utilize this inheritance information to avoid repeated computation and also reduce the average computation time to constant. Because the shortest path is every path's ancestor, we discuss this special case first and then present how to compute the other result path's fixed intersections.

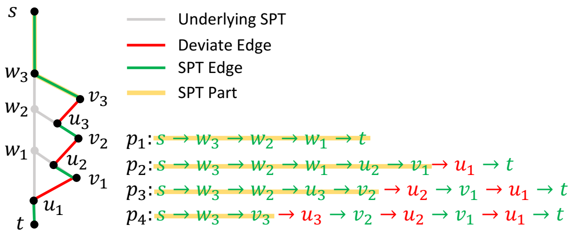

s
w_3
w_2
w_1
v_3
u_3
v_2
u_2
v_1
u_1
t
Underlying SPT
Deviate Edge
SPT Edge
SPT Part
p_1: $s \to w_3 \to w_2 \to w_1 \to t$
p_2: $s \to w_3 \to w_2 \to w_1 \to u_2 \to v_1 \to u_1 \to t$
p_3: $s \to w_3 \to w_2 \to u_3 \to v_2 \to u_2 \to v_1 \to u_1 \to t$
p_4: $s \to w_3 \to v_3 \to u_3 \to v_2 \to u_2 \to v_1 \to u_1 \to t$

Figure 5: Path Components

5.3 SPT Part Intersection

Before we dig into the SPT part intersection, we first define *Lowest Common Ancestor (LCA)* that is at the core of this part:

Definition 6. ***Lowest Common Ancestor (LCA).*** *Given an SPT, the LCA of two vertices u and v is the lowest vertex that has both u and v as descendants.*

Let's still use Figure 5 as the example. The SPT part intersection between p_1 and p_2 is $P_{s \to w_1}$, and w_1 is the *LCA* of t and v_1 on the SPT. Similarly, the SPT part intersection between p_1 and p_3 is $P_{s \to w_2}$ with w_2 being the *LCA* of t and v_2, and the SPT part intersection between p_1 and p_4 is $P_{s \to w_3}$ with w_3 being the *LCA* of t and v_3. Therefore, we have the following theorem to summarize and prove this observation:

Theorem 2. *Given any two paths p_i and p_j with v_i and v_j being their corresponding last points of SPT part, the length of their SPT part intersection equals to the length from s to the LCA w of v_i and v_j on the SPT.*

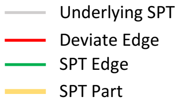
Proof. Firstly, because w is the *LCA*, then $p_{s \to w}$ shared by $p_{s \to v_i}$ and $p_{s \to v_j}$. Secondly, there is no intersection between $p_{w \to v_i}$ and $p_{w \to v_j}$, because it either means w is not the *LCA*, or some vertex w' has two parents, which violates the structure of tree. Therefore, $d(p_{s \to w})$ is the SPT part intersection. □

Now the only remaining problem is how to find the *LCA* in constant time. Because the SPT is static for each query, we build a *LCA* index [5] after the SPT is constructed. As the construction takes $O(|V|)$ time, it will not affect total enumeration complexity. Besides, given any two vertices on SPT, this *LCA* index can find the *LCA* vertex w in $O(1)$ time. Then the length of the SPT part intersection $d(s \to w)$ can be retrieved from SPT in $O(1)$ time. Finally, because the last vertex of the SPT part can also be retrieved in $O(1)$ time (previous vertex of the fixed part), the overall SPT part intersection takes $O(1)$ time.

5.4 Fixed Part Similarity with the 1^{st} Path

Because the shortest path always exists in the result set and it is the ancestor of all the other paths, we analyze and discuss it before digging into the complex cases.

Firstly, we analyze the path relations with the help of Figure 6. Because p_2 is always generated from p_1 directly, its fixed part intersection is always its parental part $p_{u_1 \to t}$. As for the third path p_3, it has three case depending on the locations of the deviation vertex v_2 and its origin:



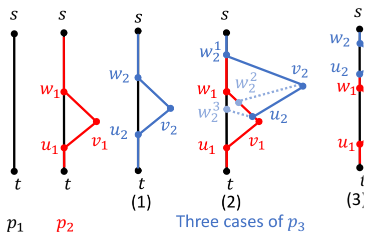


Figure 6: Path Relations with the First Shortest Path

Next, we generalize the observation above to the arbitrary generated path p_i:

Theorem 3. *Given a p_i, its fixed intersection with p_1 has and only has three situations:*

(1) *p_i is generated from p_1 directly;*
(2) *p_i is generated from some path p_j and the deviation vertex is not on p_j's LCA part with p_1;*
(3) *p_i is generated from some path p_j and the deviation vertex is on p_j's LCA part with p_1;*

Proof. Because p_i is either generated by p_1 or not by p_1, then Case 1 covers the "by p_1" case. As for the second "not by p_1" case, p_i can only be deviated from p_j's SPT part, which is made up of the *LCA* intersection part (Case 3) and the non-intersection part (Case 2). Therefore, these three cases cover all the situations. □

Now we are ready to describe how to compute the fixed part of these three cases. For Case 1, p_i's *parent part* is its fixed intersection with p_1. For Case 2, we can inherit its parent path p_j's fixed intersection length directly as no new overlapping with p_1 is introduced. For Case 3, because the deviation happens on p_j's *LCA* intersection part, the path from the deviation vertex to the *LCA* of p_1 and p_j is the new overlapping area of the fixed part. Therefore, apart from inheriting its parent's fixed part intersection, we also add its new intersection length with the help of SPT. The complexity is $O(1)$.

5.5 Fixed Part Similarity with the Other Paths

Now we discuss how to compute the fixed part intersection between any two paths. Because the paths are always generated from the previous paths, the relations of the paths form a tree structure as shown in Figure 7-(a). For any two paths, they either have an *ancestor-descendant* relation or not. Therefore, we discuss how to deal with these two cases in the following.

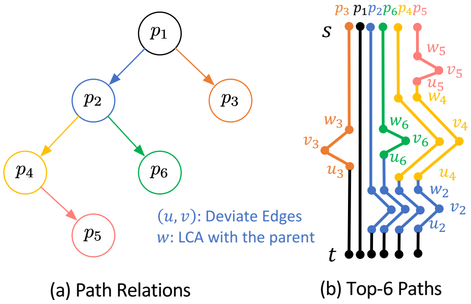
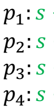


Figure 7: Path Relations with the Other Shortest Path



5.5.1 Ancestor-Descendant Relation.

Ancestor-Descendant Relation. If we look at path relation tree like the one in Figure 7, we can see that the shortest path p_1 is the root of this tree. Now we generalize this observation to any ancestral path p_j of p_i, then p_j is the root of a sub-tree which contains all its descendant paths. Then if we only focus on this sub-tree, we can view p_j as the first shortest path of all the paths in it. In this way, we can utilize fixed part Algorithm to find the fixed intersection part. For example, p_2 is p_5's ancestral path and p_4 is p_5's parent path, so $d_{fixed}(p_5 \cap p_2)$ is made up of two parts: $d_{fixed}(p_4 \cap p_2)$ that inherited from p_4, and the part $d(p_{u_5 \to w_4})$.

5.5.2 Non-Ancestor-Descendant Relation.

Non-Ancestor-Descendant Relation. If two paths do not have an ancestor-descendant relation, we cannot use fixed part Algorithm to compute it in $O(1)$ time. Nevertheless, we still have the chance to reuse part of the existing results and reduce the computational cost. For example, p_5 and p_6 are two paths and not related to each other. However, they have the same "LCA path" p_2, so they have intersect from w_2 to t from p_2. As for the remaining fixed part, we use the hash linear scan to compute the intersection.

In the example of Figure 7, all the paths are regarded as the result. However, in the DkSP problem, only a small number of the results are in $P_{s,t}$. Consequently, we only need to maintain the fixed part result of p_i to the paths in $P_{s,t}$. Furthermore, to determine the "LCA path", we only need to maintain the "ancestral result path" together with the parent path of each p_i and use it to traverse back.

Strategy 1 (**LCA Result Path Inheritance**). *If p_i and p_j has no ancestor-descendant relation, then they share their LCA result path p_k's fixed part and the fixed intersection comparison can start from p_k's deviation vertex.*

Besides, we also need to store the fixed intersection result of p_i to all the results in $P_{s,t}$ regardless of their relation to further reduce the comparison size. For example, suppose we have computed the fixed intersection between p_4 and p_3. Then we can compute p_5 and p_3's based on p_4's result.

Strategy 2 (**Parent Path Inheritance**). *If p_i and p_j has no ancestor-descendant relation, and p_i's parent p_k has computed its fixed intersection with p_j. Then the fixed intersection comparison can start from p_k's deviation vertex.*

Furthermore, as the new result path are created on the fly, the older paths (parent paths) may have not computed their intersection with new result so the Strategy 2 fails. For example, suppose p_3 is generated after p_2 so we do not have p_2's result with p_3. Now if p_2 generates p_4, we have to compute only with Strategy 1. Then if p_2 generates p_6 next, Strategy 2 still cannot be used as p_2 as a parent path still have no result with p_3. However, when we compute the intersections between p_4 and p_3, we have already covered p_2's result. Therefore, we can also update p_2's intersection with p_3 when compute p_4's, such that p_6 can inherit p_2's result now.

Strategy 3 (Parent Path Completion). *If p_i and p_j has no ancestor-descendant relation, and p_i's parent p_k has not computed its fixed intersection with p_j. Then after computing p_i and p_j result using Strategy 1, we also update p_k's result.*

With the above three strategies, we can inherit the existing results and avoid the linear comparison as much as possible.

5.6 Path Enumeration Pruning

Due to the path enumeration nature, the paths are generated in the length-increasing order. Therefore, the results are always shorter and static, while the newly generated ones are longer and uncertain. Among the five similarity functions, the first four are all dependent on the newly generated paths, while the fifth one only depends on the existing shorter paths. Therefore, the first four's values could become smaller as the new paths are generated, while the fifth one is non-decreasing. Specifically, Sim_5 has a strict length threshold $\tau \times min(d_{p_i}, d_{p_j})$. If a class of paths is guaranteed to be longer $d(p_i \cap p_j)$ than this threshold, we can discard them directly, and this provides us a chance to reduce the path enumeration number.

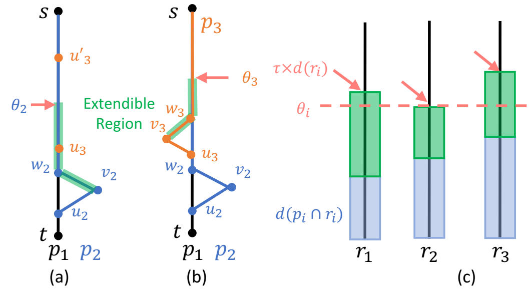


Figure 8: Pruning Example

However, although the path length keeps increasing, we cannot guarantee the length of $d(p_i \cap p_j)$ as it varies when it deviates. Fortunately, as the length variation is only introduced by the deviation and the *SPT* part, while the fixed part intersection is stable, we use the fixed intersection as the pruning condition. For example in Figure 8-(a), p_2's fixed intersection with p_1 is $p_{u_2 \to t}$ and this part cannot decrease. Then all the deviate edges from s to v_2 were considered as the next shortest path previously. However, not all the deviate vertices could generate a path satisfying the similarity threshold. This is because each time we deviate from a vertex, we have determined its fixed part implicitly. For example, when we processing the deviate edges ending with u_3, the path $p_{u_3 \to t} = \langle u_3, w_2, v_2, u_2, t \rangle$ is fixed for the new path. If $d(p_{u_3 \to t} \cap p_1)$ is longer than the threshold (like u'_3), we can prune the deviation from u_3 and all the vertices above in the SPT. Therefore, the threshold θ_2 and the SPT endpoint v_2 together determine an *extendible region* (green shadowed) where the deviation's fixed part satisfies the similarity threshold. Figure 8-(b) shows the example of p_3, and has a smaller region than p_2's because its fixed part covers p_2's fixed intersection length is no smaller than p_2's.

However, we do not know θ beforehand as it is determined by the intersection of the edges and the results. Therefore, we keep testing the edges backwardly on the SPT from the first deviate vertex v. If adding the edge length violates any of the result's threshold, then we can stop adding the deviation edges. Virtually, we have a "upper-bound" for each result path as shown in the green box of Figure 8-(c), and the smallest one (r_3) is the global "upper-bound" θ. Besides, we only add a deviate edge into the heap only if it does not violate any result path's similarity.

6 COMPLETENESS AND EFFICIENCY ORIENTED SOLUTIONS

Although the previous pruning techniques can reduce the enumerated path number, there could still be a huge number of paths to test before obtaining all the results because the paths are generated in the strictly non-decreasing order. As tested in the experiments, it is common to generate hundreds of thousands of paths, especially for the query in denser regions with longer distance, lower similarity, and higher k, which prolongs the query time to hundreds of seconds. To make things worse, some of the queries may not be able to find the k results due to the similarity requirement. Therefore, to guarantee the completeness of the query result and reduce the query time, we propose the following two heuristic techniques that either relax the similarity or block the highly repeated edges.

6.1 Dynamic Similarity Relaxation

If we keep failing to find the next result for a long time, we could temporarily pause the current search and use the best one from the enumerated path set as the next path. In this way, it is guaranteed to obtain a result set with k paths, but the similarity requirement is relaxed. Specifically, when a path p_j is added to $P_{s,t}$, we begin to accumulate the number of the newly enumerated paths. If the newly enumerated path number reaches the threshold n_p, we regard the current enumeration as incomplete and trigger the relaxation: we find the path with the smallest similarity $p_{j+1} = argmin(Sim_{max}(p_i, P_{s,t})), \forall i \in [1, n_p]$, where $Sim_{max}(p, P_{s,t}) = max(Sim(p, p')), \forall p' \in P_{s,t}$. The actual similarity of p_{j+1} is denoted as τ^*.

However, it is difficult to determine n_p and τ^* might be much larger than τ. Therefore, we propose the *dynamic similarity relaxation* to control n_p dynamically. Specifically, this method is based on the following observation: when n_p is small, τ^* should be closer to τ as we have only enumerated a few paths; when n_p is large, we can tolerate larger τ^*. Therefore, we can use a threshold function $\tau(n_p)$ to control the actual thresholds: When $n_p = 1$, $\tau(n_p) = \tau$; when n_p is very big like 10k, $\tau(n_p)$ could be close to 1. As for the values between, we can set $\tau(n_p)$ as a linear function. Now when we generate a new path p', we compare its similarity with $\tau_{n_{p'}}$. If it is larger, we discard it and generate the next one. Otherwise, we use p' as the new result path. Because it only needs an extra $\tau_{n_{p'}}$ calculation, its complexity is the same as the previous ones.

6.2 Congestion Edges Blocking

During the enumeration, some edges are occupied by more existing results than others, so avoiding them in the future enumeration could help decrease the intersection length and obtain the lower similarity results earlier. To determine which edges can be avoided, we define the edge blocking priority as follows:

Definition 7. ***Congestion Edges Order (CEO).*** *$\forall e \in p, p \in P_{s,t}$, if e appears in the current generated path, then $CEO(e)$ increases by 1. The larger the $CEO(e)$, the higher the order.*

We block the edges according to the *CEO* order. Instead of deleting them in the *SPT*, we block them by avoiding enumerating the path containing them. Because these edges may appear in different parts of the newly generated paths, we discuss their influence and behavior accordingly below:

(1) Fixed Part: it can be detected when a path popped out from H, and we do not expand it so the whole sub-class of paths are pruned;
(2) Deviate Edge: it can be detected during deviation, and we prune this path directly;
(3) SPT: it can also be detected during deviation but with the LCA computation. If the LCA of the deviation point and e's endpoints are both equal to the endpoints, then we can prune this path and deviate another one.

However, some edges are "unblockable", because blocking them may affect the connectivity of the paths such that no new paths can be generated. Therefore, before blocking an edge e, we cache the current searching status and keep counting the pruned paths. If a consecutive of β top paths fail to generate any new path, we mark e as "unblockable", roll the enumeration back to the previous status, and block the next high CEO edge. The time to trigger the blocking is a parameter similar to n_p in Section 6.1. Nevertheless, it is still non-trivial and faces the following issues:

1) Last Blocked Edge Diminish: Suppose $\{e_1, \ldots, e_k\}$ are the current blocked edges, and the next blocked edge is e_{k+1}. Then for a rough calculation, the inconnectivity caused by e_{k+1} is only $1/k+1$, while the probability of the previous k blocked edges is $k/k+1$. Therefore, a rollback of unblocking e_{k+1} has a high probability of being triggered by the previously blocked edge but not e_{k+1}, so adding back e_{k+1} does not solve the inconnectivity problem. What's worse, unblocking e_{k+1} deprives e_{k+1} and all the latter ones' blocking power: after blocking several edges, the latter edges won't be blocked and this process degenerates to the unblocking version. Therefore, the rollback e_{k+1} operation should only be triggered when the consecutive path generation failures are caused by e_{k+1}.

2) Candidate Heap Shrinking: Following the phenomenon above, the previously blocked edges have a growing influence on path pruning such that fewer and fewer candidate paths would be inserted in the heap. Then the heap would become empty earlier and the search ends with an incomplete result set. To avoid the heap shrinking too much, we need to add some blocked edges back. Specifically, we keep counting the path pruned by each blocked edge: $n_{fixed} + n_{SPT}$. When a path p is pruned by e_i in its fixed part, e_i's n_{fixed} adds p's remaining deviate edge number as they are all pruned. When p is pruned in its *SPT* or its deviate edge, e_i's n_{SPT} increases by 1. Then when the heap keeps shrinking consecutively for β paths, we add the block edge with the largest pruning number back to the graph but without rolling back.

3) CEO Bias: Because all the new paths are generated from the existing ones' fixed parts, the edges that are closer to the destination have higher *CEO* naturally. However, these edges also have a higher influence on the graph connectivity, so blocking them would cause earlier search termination with incomplete results. Therefore, instead of blocking the edges with the highest *CEO*, we choose the edge with top-20% to top-50% *CEO* randomly to block. In this way, the blocking power and graph connectivity could be balanced.

7 EXPERIMENT

7.1 Experiment Setup

Dataset. We test on four real-world road networks: 1) *Manhattan (MH)* [43]: a grid-based city center with 4,590 vertices and 25,395 edges; 2) *Tianjin (TJ)* [23, 42]: a ring network with 31,002 vertices and 86,584 edges; 3) *New York (NY)* [1]: urban city with 264,246 vertices and 733,846 edges; 4) *Colorado (COL)* [1]: state network with 435,666 vertices and 1,057,066 edges.
Query Sets. For each dataset, we randomly generate four sets of OD pairs Q_1 to Q_4 with 1000 queries. Specifically, we first estimate the diameter d_{max} of each network by finding the longest shortest path between the outer vertices like the landmarks in [19]. Then each Q_i represents a category of OD pairs falling into the distance range $[d_{max}/2^{5-i}, d_{max}/2^{4-i}]$. For example, Q_1 stores the OD pairs with distances in range $[d_{max}/16, d_{max}/8]$.
Method. We implement and compare the following algorithms: 1) *DkSP*: Our method with the efficient similarity comparison; 2) *DkSP-DS* and *DkSP-EB*: Our method with two heuristics; 3) *KSPD*: the iterative bounding pruning search-based method in [35] that dominates *Yen's* [55], *cKSP* [18], *IterBound* [7]; 4) *OP*: The *One-Pass* algorithm of *kSPwLO*; 5) *SVP$^+$-C* and *ESX-C*: The completeness version of *SVP$^+$* and *ESX* from [12]. [12];

All the algorithms are implemented in C++, compiled with full optimizations, and tested on a Dell R730 PowerEdge Rack Mount Server which has two Xeon E5-2630 2.2GHz (each has 10 cores and 20 threads) and 378G memory.

7.2 Similarity Comparison

Similarity Function Influence. In this section, we show the influence of different similarity functions on query efficiency (*Running Time*) and query result quality (*Average Length*), and their corresponding standard deviations in Figure 9. It should be noted that this set of experiment is easier to find results, so the heuristic-based methods could be slower while the non-heuristic methods are faster except on *COL*. Specifically, our *DkSP* and *OP* have similar performance with the best query quality under different similarity functions from S_1 to S_5. *KSPD* is the slowest and most unstable among all the algorithms but have similar quality because they share the same enumeration framework (slightly different due to the same length path order). The baseline heuristic methods are faster for the harder queries but have longer length and unstable quality except for *DkSP-DS* because it sacrifices the similarity. Lastly, since all the algorithms perform best on S_1 with the fastest running time and average length, we use S_1 as the default similarity function in the remaining experiments.

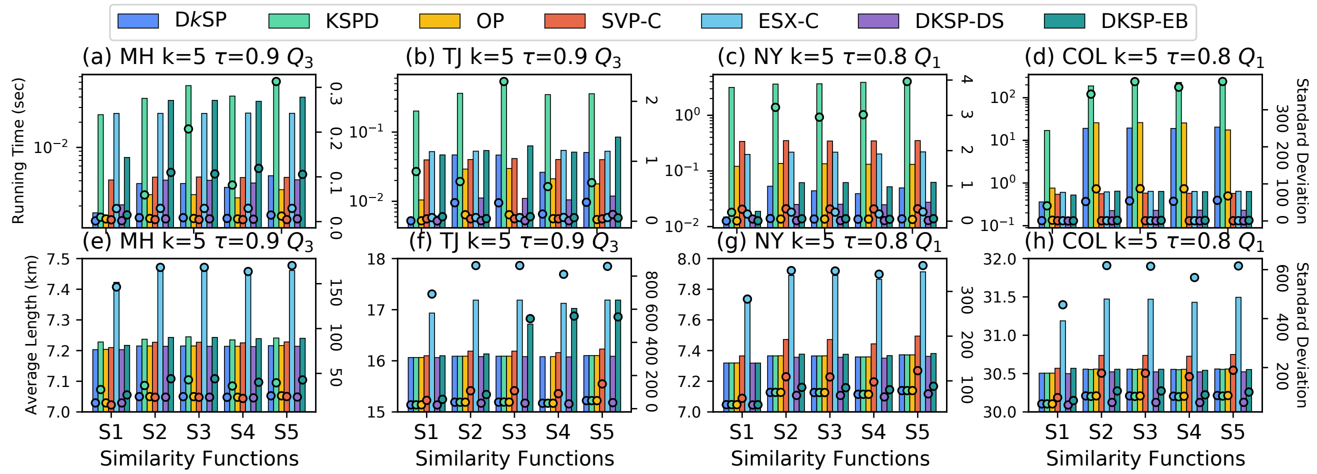


Figure 9: Similarity Function Comparison (The bars are running time and average length. The balls are standard deviations.)

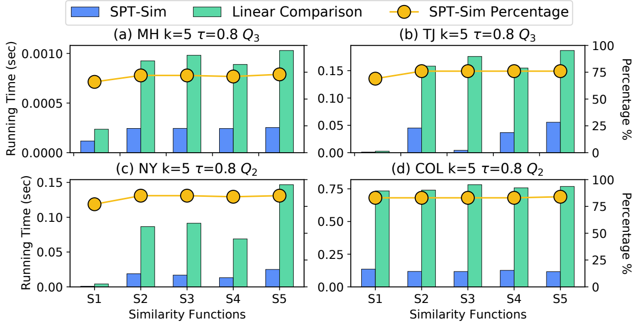


Figure 10: Similarity Computation Performance (The Bars are Running Time. The Balls are Percentage.)

Similarity Computation. In this section, we validate the effectiveness of our SPT similarity computation compared with the linear similarity computation. As shown in Figure 10, our method costs less time (around 1/3) compared with the linear comparison. This is because around 70% of the comparisons (shown as the ball) have the ancestral-descendant relation and avoid linear scan. In addition, the similarity computation with different similarity functions varies: S_4 and S_5 bring about the heaviest computation and S_1 the least.

7.3 D*K*SP Performance

Query Distance. As shown in Figure 11-(a) to (d), all algorithms take longer time in finding the diversified paths as the distance between OD pairs becomes longer. It is because the longer OD has more paths with similar distances than the shorter ones, then they have more candidate paths to enumerate and test. Moreover, our *DkSP* has the best performance in most instances among the non-heuristic methods. As for the heuristic methods, our *DkSP-DS* is generally faster than the others with shorter length. The *DkSP-EB* is not stable and effective as *DkSP-DS*. *ESX-C* always has the longest the result while the running time is never the fastest.

Similarity Threshold τ As we can see from Figure 11-(e) to (h), the running time increases as τ decreases. It is because that more and longer path candidates need to be generated to satisfy the stricter threshold, which also causes heavier computation on candidate path selection. Our *DkSP*-based methods have the best performance in almost all their corresponding exact and heuristic instances. Although *ESX-C* and *SVP-C* are fast but their results are longer especially when τ is small. *DkSP-DS* has similar efficiency but shorter length. Although lower τ implies lower similarity, the higher τ can still generate lower similarity results because this τ is the worst case bound. We will discuss it further later.

Result Path Number k**.** As shown in Figure 11-(i) to (l), all methods take longer time to find more result paths. As k increases, it is harder to find the next result because 1) longer paths are needed and the number of path candidate increases as the length increases, and 2) more paths in the current result makes the newer paths harder to satisfy the similarity requirement. In general, our proposed algorithm *DkSP* always performs better than *KSPD* under all parameter combinations because 1) in terms of path generation, our SPT-deviation-based strategy avoids the expensive graph search in *KSPD* and *OP*; 2) in terms of path similarity computation, our *SPT-Sim* also reduces the query time compared with the linear comparison. In terms of the heuristic methods, *ESX-C*'s length and running both soar up as k increases. *SVP-C* also increases, while our *DkSP-DS* remains fast with shorter length. Our *DkSP-EB* also increases but not as dramatically as the baselines.

Pruning Effectivenes. In this test, we vary the threshold τ under two parameter settings: $k = 5$ with Q_2 and $k = 3$ with $Q3$. As shown in Figure 12, the bars shows the effectiveness of our pruning technique, and the balls represent the number of the pruned path validating the pruning power. Firstly, as τ decreases, more paths are pruned because 1) the pruning power is stronger for smaller τ, and 2) more and longer candidate paths are generated. Secondly, k has larger impact on query than distance when τ is smaller, because it is harder to find more paths that are dissimilar to each other. Finally, our pruning technique can decrease the computation of S_5 by half

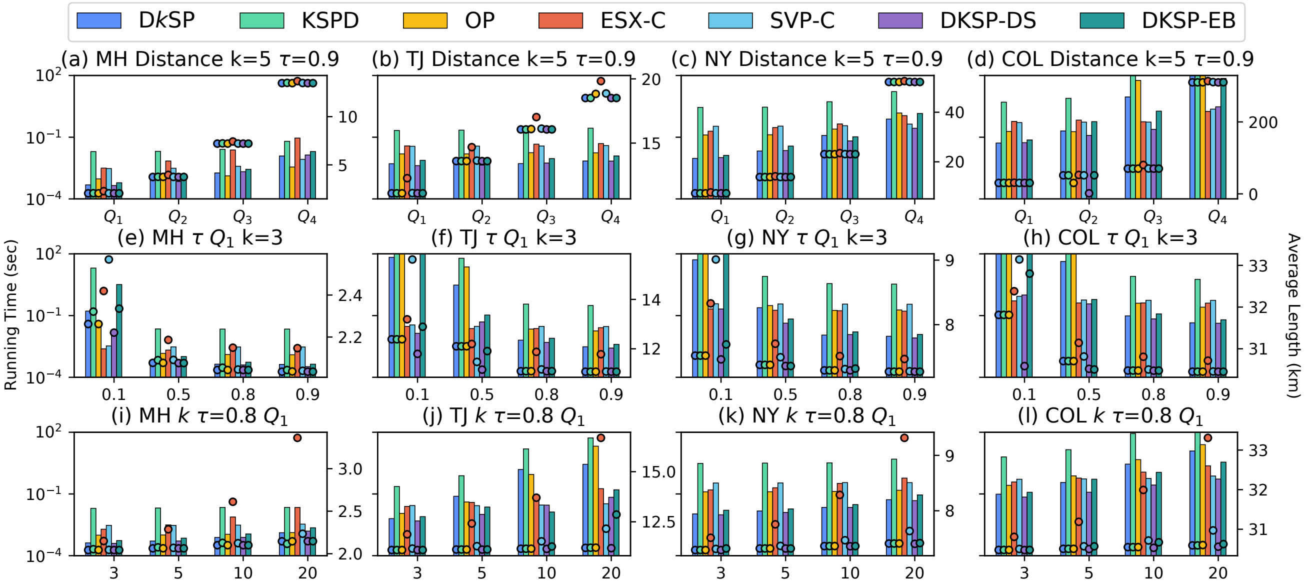


Figure 11: *DKSP* Performance under Different Distance, Similarity Threshold τ, and Path Number k (The Bars are running time. The Balls are Average Length)

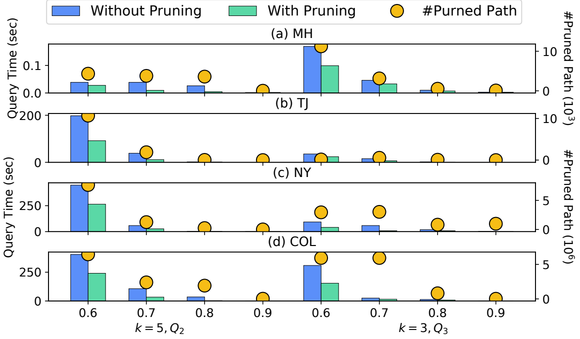


Figure 12: Pruning Power of S_5

and avoid testing millions of path candidates. Nevertheless, S_5's efficiency is still not comparable with S_1.

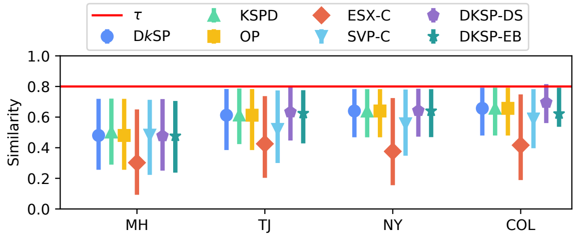


Figure 13: Similarity Quality (Max-Avg-Min) Q_1 k=5

Similarity Quality. We show the result's average max/avg/min similarities with 0.8 as threshold in Figure 13. Because 0.8 is only the upperbound, the results' average similarities are around 0.5 and 0.6, and it could be lower than 0.4. The three searching methods have similar performance as their results are nearly the same. *ESX-C* has the lowest similarity at the cost of longer length. Among the other heuristic methods, *KDSP-DS* has surpassed the threshold slightly as it sacrifices similarity for faster computation and shorter length. Therefore, even with 0.8 as threshold, our methods can still achieve lower average similarity.

Experimental Summary. Our proposed *DkSP* can generate the results faster with same quality as the baseline exact methods, while our heuristic method *DkSP-DS* can calculate the results more efficiently with shorter path length and similar similarities.

8 CONCLUSION

Route planning has an increasing impact on real-life traffic conditions, which has drawn more and more attentions recently. In this work, we attempt to address it passively by diversifying the routing results for similar queries. The existing solutions all have very high complexity due to the graph searching such that they are impractical in real-life. Therefore, we propose an efficient diversified top-k routing algorithm with SPT concatenation, constant similarity comparison, candidate path pruning, and two completeness and efficiency-oriented solutions. As validated on four real-world road networks, our solution is hundreds of times faster than the state-of-the-art solution, which indicates that diversified routing can be technically practical to use for the first time.

ACKNOWLEDGMENTS

This work was supported by the Australian Research Council (Number DP200103650 and LP180100018), Hong Kong Research Grants Council (grant# 16202722), and was partially conducted in the JC STEM Lab of Data Science Foundations funded by The Hong Kong Jockey Club Charities Trust.

REFERENCES

[1] [n.d.]. 9th DIMACS Implementation Challenge - Shortest Paths. http://users.diag.uniroma1.it/challenge9/download.shtml.

[2] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. 2013. Alternative routes in road networks. *Journal of Experimental Algorithmics (JEA)* 18 (2013), 1–1.

[3] Vedat Akgün, Erhan Erkut, and Rajan Batta. 2000. On finding dissimilar paths. *European Journal of Operational Research* 121, 2 (2000), 232–246.

[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 349–360.

[5] Michael A Bender and Martin Farach-Colton. 2000. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*. Springer, 88–94.

[6] Pasquale Carotenuto, Stefano Giordani, and Salvatore Ricciardelli. 2007. Finding minimum and equitable risk routes for hazmat shipments. *Computers & Operations Research* 34, 5 (2007), 1304–1327.

[7] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Jian Pei. 2015. Efficiently computing top-k shortest path join. In *EDBT 2015-18th International Conference on Extending Database Technology, Proceedings*.

[8] Chandra Chekuri and Sanjeev Khanna. 2003. Edge disjoint paths revisited. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. 628–637.

[9] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. 2015. Alternative routing: k-shortest paths with limited overlap. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 1–4.

[10] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. 2017. Exact and approximate algorithms for finding k-shortest paths with limited overlap. In *20th International Conference on Extending Database Technology: EDBT 2017*. 414–425.

[11] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B Blumenthal. 2018. Finding k-dissimilar paths with minimum collective length. In *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 404–407.

[12] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B Blumenthal. 2020. Finding k-shortest paths with limited overlap. *The VLDB Journal* (2020), 1–25.

[13] Jian Dai, Bin Yang, Chenjuan Guo, and Zhiming Ding. 2015. Personalized route recommendation using big trajectory data. In *2015 IEEE 31st international conference on data engineering (ICDE)*. IEEE, 543–554.

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

[15] Bolin Ding, Jeffrey Xu Yu, and Lu Qin. 2008. Finding time-dependent shortest paths over large graphs. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology (EDBT)*. 205–216.

[16] Erhan Erkut, Stevanus A Tjandra, and Vedat Verter. 2007. Hazardous materials transportation. *Handbooks in operations research and management science* 14 (2007), 539–621.

[17] Erhan Erkut and Vedat Verter. 1998. Modeling of transport risk for hazardous materials. *Operations research* 46, 5 (1998), 625–642.

[18] Jun Gao, Huida Qiu, Xiao Jiang, Tengjiao Wang, and Dongqing Yang. 2010. Fast top-k simple shortest paths discovery in graphs. In *Proceedings of the 19th ACM international conference on Information and knowledge management (CIKM)*. 509–518.

[19] Andrew V. Goldberg and Chris Harrelson. 2005. Computing the Shortest Path: A Search Meets Graph Theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Vancouver, British Columbia) *(SODA '05)*. Society for Industrial and Applied Mathematics, USA, 156–165.

[20] Longkun Guo, Yunyun Deng, Kewen Liao, Qiang He, Timos Sellis, and Zheshan Hu. 2018. A fast algorithm for optimally finding partially disjoint shortest paths. In *Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18), Stockholm, Sweden, July 13-19, 2018*. International Joint Conferences on Artificial Intelligence, 1456–1462.

[21] Christian Häcker, Panagiotis Bouros, Theodoros Chondrogiannis, and Ernst Althaus. 2021. Most Diverse Near-Shortest Paths. In *Proceedings of the 29th International Conference on Advances in Geographic Information Systems*. 229–239.

[22] Wilton Henao-Mazo and Angel Bravo-Santos. 2012. Finding diverse shortest paths for the routing task in wireless sensor networks. *Proc. ICSNC* (2012), 53–58.

[23] Alireza Karduni, Amirhassan Kermanshah, and Sybil Derrible. 2016. A protocol to convert spatial polyline data to network formats and applications to world urban road networks. *Scientific data* 3, 1 (2016), 1–7.

[24] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce Reed. 2012. The disjoint paths problem in quadratic time. *Journal of Combinatorial Theory, Series B* 102, 2 (2012), 424–435.

[25] Ken-ichi Kawarabayashi and Bruce Reed. 2009. A nearly linear time algorithm for the half integral parity disjoint paths packing problem. In *Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1183–1192.

[26] Moritz Kobitzsch. 2013. An alternative approach to alternative routes: HiDAR. In *European Symposium on Algorithms*. Springer, 613–624.

[27] Lingxiao Li, Muhammad Aamir Cheema, Hua Lu, Mohammed Eunus Ali, and Adel N Toosi. 2021. Comparing alternative route planning techniques: A comparative user study on Melbourne, Dhaka and Copenhagen road networks. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[28] Lei Li, Wen Hua, Xingzhong Du, and Xiaofang Zhou. 2017. Minimal on-road time route scheduling on time-dependent graphs. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1274–1285.

[29] Lei Li, Sibo Wang, and Xiaofang Zhou. 2019. Time-dependent hop labeling on road network. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 902–913.

[30] Lei Li, Sibo Wang, and Xiaofang Zhou. 2020. Fastest path query answering using time-dependent hop-labeling in road network. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[31] Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2020. Fast query decomposition for batch shortest path processing in road networks. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1189–1200.

[32] Lei Li, Kai Zheng, Sibo Wang, Wen Hua, and Xiaofang Zhou. 2018. Go slow to go fast: minimal on-road time route scheduling with parking facilities using historical trajectory. *The VLDB Journal* 27, 3 (2018), 321–345.

[33] Sejoon Lim and Daniela Rus. 2012. Stochastic distributed multi-agent planning and applications to traffic. In *2012 IEEE International Conference on Robotics and Automation*. IEEE, 2873–2879.

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

[35] Huiping Liu, Cheqing Jin, Bin Yang, and Aoying Zhou. 2017. Finding top-k shortest paths with diversity. *IEEE Transactions on Knowledge and Data Engineering* 30, 3 (2017), 488–502.

[36] Huiping Liu, Cheqing Jin, Bin Yang, and Aoying Zhou. 2018. Finding top-k optimal sequenced routes. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 569–580.

[37] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, Pingfu Chao, and Xiaofang Zhou. 2021. Efficient Constrained Shortest Path Query Answering with Forest Hop Labeling. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1763–1774.

[38] Ziyi Liu, Lei Li, Mengxuan Zhang, Wen Hua, and Xiaofang Zhou. 2022. FHL-Cube: Multi-Constraint Shortest Path Querying with Flexible Combination of Constraints. *Proceedings of the VLDB Endowment* 15 (2022).

[39] Dennis Luxen and Dennis Schieferdecker. 2012. Candidate sets for alternative routes in road networks. In *International Symposium on Experimental Algorithms*. Springer, 260–270.

[40] Nirmesh Malviya, Samuel Madden, and Arnab Bhattacharya. 2011. A continuous query system for dynamic route planning. In *2011 IEEE 27th International Conference on Data Engineering (ICDE)*. IEEE, 792–803.

[41] Ernesto QV Martins and Marta MB Pascoal. 2003. A new implementation of Yen's ranking loopless paths algorithm. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies* 1, 2 (2003), 121–133.

[42] Road Networks. [n.d.]. Urban (2016): Urban Road Network Data. figshare. Dataset.https://doi.org/10.6084/m9.figshare.2061897.v1. https://doi.org/10.4225/13/511C71F8612C3

[43] OpenStreetMap contributors. 2017. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org.

[44] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When hierarchy meets 2-hop-labeling: Efficient shortest distance queries on road networks. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 709–724.

[45] Juan Pan, Iulian Sandu Popa, Karine Zeitouni, and Cristian Borcea. 2013. Proactive vehicular traffic rerouting for lower travel time. *IEEE Transactions on vehicular technology* 62, 8 (2013), 3551–3568.

[46] Antonio Sedeno-Noda. 2016. Ranking One Million Simple Paths in Road Networks. *Asia-Pacific Journal of Operational Research* 33, 05 (2016), 1650042.

[47] Han Su, Shuncheng Liu, Bolong Zheng, Xiaofang Zhou, and Kai Zheng. 2020. A survey of trajectory distance measures and performance evaluation. *The VLDB Journal* 29, 1 (2020), 3–32.

[48] Yongxin Tong, Yuxiang Zeng, Zimu Zhou, Lei Chen, Jieping Ye, and Ke Xu. 2018. A unified approach to route planning for shared mobility. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1633.

[49] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. 2015. Efficient route planning on public transportation networks: A labelling approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 967–982.

[50] Sibo Wang, Xiaokui Xiao, Yin Yang, and Wenqing Lin. 2016. Effective indexing for approximate constrained shortest path queries on large road networks. *PVLDB* 10, 2 (2016), 61–72.

[51] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proceedings of the VLDB Endowment* 7, 9 (2014), 721–732.

[52] Jiajie Xu, Limin Guo, Zhiming Ding, Xiling Sun, and Chengfei Liu. 2012. Traffic aware route planning in dynamic road networks. In *International Conference on Database Systems for Advanced Applications*. Springer, 576–591.

[53] Bin Yang, Chenjuan Guo, Yu Ma, and Christian S Jensen. 2015. Toward personalized, context-aware routing. *The VLDB Journal* 24, 2 (2015), 297–318.

[54] Yajun Yang, Hong Gao, Jeffrey Xu Yu, and Jianzhong Li. 2014. Finding the cost-optimal path with time constraint over time-dependent graphs. *PVLDB* 7, 9 (2014), 673–684.

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

[56] Ziqiang Yu, Xiaohui Yu, Nick Koudas, Yang Liu, Yifan Li, Yueting Chen, and Dingyu Yang. 2020. Distributed processing of k shortest path queries over dynamic road networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 665–679.

[57] Yuxiang Zeng, Yongxin Tong, Yuguang Song, and Lei Chen. 2020. The simpler the better: An indexing approach for shared-route planning queries. *Proceedings of the VLDB Endowment* 13, 13 (2020), 3517–3530.

[58] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. 2021. Dynamic Hub Labeling for Road Networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 336–347.

[59] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2019. Efficient batch processing of shortest path queries in road networks. In *2019 20th IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 100–105.

[60] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2020. Stream processing of shortest path query in dynamic road networks. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[61] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2021. Efficient 2-hop labeling maintenance in dynamic small-world networks. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 133–144.

[62] Mengxuan Zhang, Lei Li, and Xiaofang Zhou. 2021. An Experimental Evaluation and Guideline for Path Finding in Weighted Dynamic Network. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2127–2140.