The VLDB Journal
https://doi.org/10.1007/s00778-020-00604-x

REGULAR PAPER

Finding *k*-shortest paths with limited overlap

Theodoros Chondrogiannis[1] · Panagiotis Bouros[2] · Johann Gamper[3] · Ulf Leser[4] · David B. Blumenthal[5]

Received: 20 May 2019 / Accepted: 10 February 2020

Abstract
In this paper, we investigate the computation of alternative paths between two locations in a road network. More specifically, we study the *k-shortest paths with limited overlap* (*k*SPwLO) problem that aims at finding a set of k paths such that all paths are sufficiently dissimilar to each other and as short as possible. To compute *k*SPwLO queries, we propose two exact algorithms, termed OnePass and MultiPass, and we formally prove that MultiPass is optimal in terms of complexity. We also study two classes of heuristic algorithms: (a) performance-oriented heuristic algorithms that trade shortness for performance, i.e., they reduce query processing time, but do not guarantee that the length of each subsequent result is minimum; and (b) completeness-oriented heuristic algorithms that trade dissimilarity for completeness, i.e., they relax the similarity constraint to return a result that contains exactly k paths. An extensive experimental analysis on real road networks demonstrates the efficiency of our proposed solutions in terms of runtime and quality of the result.

Keywords Alternative routing · Road networks · Query services

1 Introduction

Computing the shortest path between two locations in a road network is a fundamental problem that has attracted lots of attention by both the research community and industry. However, in many real-world scenarios, determining solely the shortest path is not enough. For instance, users of navigation systems are interested in alternative paths that might be longer than the shortest path but have other desirable properties. Another scenario where alternative routes are useful is transport of humanitarian aid goods through unsafe regions. The distribution of the load to a fleet of vehicles that follow non-overlapping routes increases the chances that at least some of the goods will be delivered. The need for alternative routes also arises in emergency situations, such as natural disasters and terrorist attacks. To avoid panic and potential catastrophic collisions while dealing with the aftermath of such events, evacuation plans should include alternative paths that overlap as little as possible.

A first take on providing alternative routes is to solve the K-shortest paths problem [16,23,37]. In most cases though, the returned paths share large stretches, and therefore, they are not helpful in scenarios such as the aforementioned ones. Consider Fig. 1, which shows three paths connecting two locations in the city of Oldenburg. The solid/black line indicates the shortest path, the dashed/red line indicates the next path by length, which however is very similar to the shortest path, and the dotted/blue line indicates a path that is clearly longer, but significantly different from the shortest path as it passes through a very distant part of the city's road network. In scenarios like the ones mentioned above, the dotted/blue path is a better alternative to the shortest path than the dashed/red one.

Existing literature has approached alternative routing from different perspectives. Notable works include methods that compute alternative routes either by incrementally building

✉ Theodoros Chondrogiannis
theodoros.chondrogiannis@uni-konstanz.de

Panagiotis Bouros
bouros@uni-mainz.de

Johann Gamper
gamper@inf.unibz.it

Ulf Leser
leser@informatik.hu-berlin.de

David B. Blumenthal
david.blumenthal@wzw.tum.de

1 University of Konstanz, Konstanz, Germany

2 Johannes Gutenberg University Mainz, Mainz, Germany

3 Free University of Bozen-Bolzano, Bolzano, Italy

4 Humboldt-Universität zu Berlin, Berlin, Germany

5 Technical University of Munich, Munich, Germany

Published online: 21 February 2020

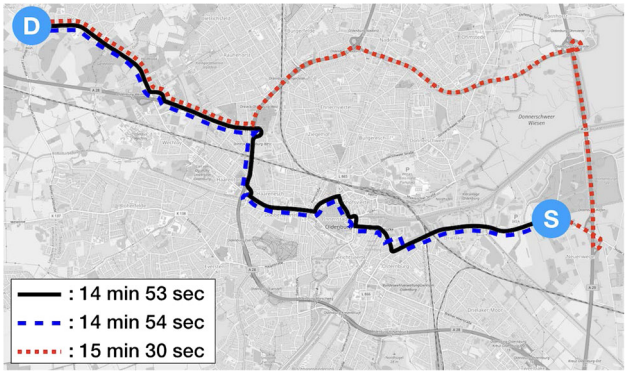


Fig. 1 Motivating example

a set of dissimilar paths [18] or by employing edge penalties [3]. These methods provide no formal definition of the alternative routing problem, and typically no guarantee regarding the length of the recommended paths. Other approaches [2,4,8] first generate a large set of candidates and, in a post-processing step, apply a number of constraints to determine the final result. In these works, alternative paths are defined solely on their individual similarity to the shortest path. This yields paths that are very similar to each other and, hence, of limited interest in many applications.

In this paper, we formalize alternative routing as the *k-Shortest Paths with Limited Overlap* (kSPwLO) problem. A kSPwLO query aims at finding a set of k paths that are (a) sufficiently dissimilar to each other with respect to a user-defined similarity threshold θ, and (b) as short as possible. We prove that the kSPwLO problem is weakly NP-hard, and we propose two exact algorithms that traverse the road network by expanding paths from the source in length order, while employing pruning criteria to reduce the number of paths that need to be examined.

To balance between performance and result quality, we present two classes of heuristic algorithms. First, to enable the processing of kSPwLO query on large road networks, we propose three performance-oriented heuristic algorithms that trade shortness (i.e., how short the recommended alternative paths are) for performance. These algorithms drastically reduce the number of examined paths and therefore scale for large road networks, but they do not guarantee that the returned paths are as short as possible.

Strictly abiding by the similarity constraint may prevent a kSPwLO algorithm from finding exactly k sufficiently dissimilar paths. In many cases though, returning a complete result set of k paths is more important than abiding by the similarity constraint. For this purpose, we introduce a procedure to gradually relax the similarity constraint, trading dissimilarity for completeness. Based on this procedure, we present two completeness-oriented heuristic algorithms, which guarantee that the result set contains exactly k paths.

This paper extends our previous work [9,10] where we presented the following contributions:

- We formally defined the k-shortest paths with limited overlap (kSPwLO) problem for computing alternative routes on road networks (Sect. 4).
- We introduced two exact algorithms for kSPwLO queries: OnePass traverses the road network once expanding only paths that qualify the similarity constraint; MultiPass improves OnePass by employing an additional pruning criterion and traversing the network $k-1$ times (Sect. 5).
- We presented three performance-oriented heuristic algorithms that limit the number of examined paths but do not minimize the length of each subsequent result: OnePass$^+$ employs the pruning power of MultiPass but traverses the network only once; SVP$^+$ selects alternative paths from the set of single-via paths [2]; ESX removes edges from the road network incrementally and computes the shortest path on the updated network (Sect. 6).

As an extension to our previous work, in this paper we present the following theoretical and technical contributions:

- We prove that the kSPwLO problem is weakly NP-hard (Sect. 4).
- We present comprehensive complexity analyses for all proposed algorithms (Sects. 5–7).
- We prove that MultiPass is optimal for the kSPwLO problem (Sect. 5).
- We examine additional edge removal criteria for ESX in order to further improve its performance and result quality (Sect. 6.4).
- We present the `Complete_kSPwLO` function that gradually relaxes the similarity constraint, and we discuss two completeness-oriented heuristic algorithms, termed ESX-C and SVP-C, that employ this function to always compute a complete result set of k paths (Sect. 7).

Through an extensive experimental analysis on real road networks, we evaluate both the algorithms presented in our previous works and the new ones in terms of runtime, quality of alternative paths and completeness of the result set (Sect. 9).

2 Related work

In this section, we overview existing works that approach alternative routing from different sides.

Pairwise dissimilar paths To the best of our knowledge, works that aim at computing a set of pairwise dissimilar paths are the closest ones to our own.

Jeong's algorithm [18] aims at computing dissimilar alternative paths by directly extending Yen's algorithm [37]. Given a length limit x and a similarity threshold y, the goal is to incrementally compute k paths not longer than x and the similarity between any two paths does not exceed y. At each step, the algorithm modifies all previously computed paths to obtain a set of candidate paths and examines the candidate path that is most dissimilar to the already recommended paths. While dissimilarity is guaranteed, in contrast to our approach, the algorithm does not minimize the length of each subsequent path in the result.

Another strategy to compute dissimilar alternative paths is to iteratively apply a penalty on the weights of edges that lie on previously computed paths. Akgun et al. [3] proposed a method that computes alternative paths by repeatedly computing the shortest path on the road network, each time with updated weights. The main shortcoming of this approach is that there is no intuition behind the penalty applied in each iteration. A large penalty would result in dissimilar but possibly long alternative paths, whereas a small penalty would require the algorithm to execute more iterations. Similar to Jeong's algorithm, penalty-based methods make no effort to minimize the length of the alternative paths.

Another approach is the k-dissimilar paths with minimum collective length (k-*DPwML*) problem introduced by Liu et al. [22]. In contrast to kSPwLO, a k-*DPwML* query computes the set of k sufficiently dissimilar paths w.r.t. a similarity threshold θ, that exhibits the lowest collective path length among all sets of k sufficiently dissimilar paths. As shown by Chondrogiannis et al. [11], the requirement to minimize the collective length of the result renders the problem strongly *NP*-hard and its exact computation prohibitively expensive. Note that Liu et al. [22] did not study the exact computation of the k-*DPwML* problem. Instead, the authors proposed a greedy approach FINDKSPD, which solves our kSPwLO problem. In fact, the kSPwLO can be seen as an approximation to the much harder but clearly less practical k-*DPwML* problem.

Candidate sets A different definition of alternative routing is to compute paths that are alternatives only to the shortest path. The *Plateaux* method [8] aims at computing paths that cross different highways of the road network. Bader et al. [4] introduced the concept of alternative graphs, which have a similar functionality as the plateaus. Abraham et al. [2] introduced the notion of *single-via paths*, which we adopt and extend for developing one of our heuristic algorithms, i.e., SVP$^+$. The proposed approach evaluates each single-via path individually by comparing it to the shortest path and checks whether it meets a set of user-defined constraints, i.e., local optimality and stretch.

Segment avoidance Another definition of alternative route is to compute paths that avoid certain segments of the road network. Xie et al. [35] study the computation of paths that avoid specific edges of the road network and then introduce iSQPF, a spatial data structure that extends the shortest path quadtree [31], to enable the efficient computation of such paths. The concept of segment avoidance has also been studied in the context of traffic management. Methods in this category utilize traffic information obtained from trajectory data [39], from sensor networks [24] or from VANETs [17]. They aim to identify congested segments of the road network and to compute paths that avoid them. Xu et al. [36] proposed a first-cut approach to compute traffic-aware routes on dynamic road networks. Li et al. [21] utilized historical traffic information and study the computation of the k traffic-tolerant paths, i.e., the paths with the minimum (historic) travel time.

Multi-objective path planning The computation of multiple routes has also been approached as a multi-objective problem. Pareto-optimal paths [13,25] and the route skyline [20] can be directly seen as alternative routes, or they can be further examined in a post-processing phase to obtain the final alternative paths. Another approach involves solving a multi-objective traffic assignment problem [27,29]. Works in this direction aim at assigning paths to different users while optimizing for a set of user preferences. Such approaches are frequently employed in urban traffic management systems to achieve flow optimization [26].

Popular route extraction Finally, there are also historical data-based methods that aim at analyzing and mining trajectory data in order to extract popular routes [6,7,34,38]. Popularity is usually measured by the number of trajectories that cross a specific edge/segment. The more popular the edges/segments of a route are, the more popular the route is. This line of work aims at exploiting the wisdom of the crowd and recommending routes that are frequently used by experienced users, e.g., taxi drivers.

3 Preliminaries

Let road network $G = (N, E)$ be represented by a *directed weighted* graph with a set of nodes N and a set of edges $E \subseteq N \times N$[1]. The nodes of the graph represent road intersections, and edges represent road segments. Every each edge $(n_i, n_j) \in E$ is assigned a *positive* weight $w(n_i, n_j)$,

[1] For ease of presentation, we draw a road network as an undirected graph in our examples. However, our proposed methods directly work on directed graphs as well.

which captures the cost of moving from node n_i to node n_j. This weight can represent any nonnegative cost, e.g., distance and travel time, or even a composite cost, e.g., a linear combination of travel time with financial cost. A (simple) *path* $p(s \rightarrow t)$ from a source node s to a target node t is a *connected* and *cycle-free* sequence of edges $\langle (s, n_i), \ldots, (n_j, t) \rangle$. The length $\ell(p)$ of a path p is the sum of the weights of all contained edges. The *shortest path* $p_{sp}(s \rightarrow t)$ is the path with the lowest length among all paths that connect nodes s and t.

Given two paths p, p' from s to t. The *similarity* Sim of p and p' is defined by their overlap ratio [2], i.e.,

$$Sim(p, p') = \frac{\sum_{\forall (n_i, n_j) \in p \cap p'} w(n_i, n_j)}{min\{\ell(p), \ell(p')\}} \quad (1)$$

where $p \cap p'$ denotes the set of edges shared by p and p'. For the similarity, we have $0 \leq Sim(p, p') \leq 1$, where $Sim(p, p') = 0$ if path p' shares no edges with p, while $Sim(p, p') = 1$ holds if $p' \equiv p$. Since only simple cycle-free paths are considered, the similarity between different paths is strictly lower than 1.

While various measures to compute the similarity between two paths have been proposed [22], we argue that the similarity measure of Eq. 1 is the most suitable one for alternative routing on road networks, as it enables us to disregard needlessly long paths when searching for alternative paths. In practice, there is no value in defining an alternative path p' to a path p, if p' is shorter than p. The shortest of two paths will always be the first option, and the longer one will be the alternative.

Given a similarity threshold θ, path p' is called an *alternative path* to p if p' is sufficiently dissimilar to p, i.e., $Sim(p, p') < \theta$. We also call a path p *alternative* to a *set of paths* P if p is sufficiently dissimilar to every path in P.

Definition 1 (*Alternative Path*) Let P be a set of paths from s to t and θ be a similarity threshold. A path p' from s to t is *alternative to set* P *iff* $\forall p \in P : Sim(p, p') \leq \theta$.

Example 1 Consider the road network in Fig. 2. Let set $P = \{p_1, p_2\}$, where $p_1 = \langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$ and $p_2 = \langle (s, n_3), (n_3, n_4), (n_4, t) \rangle$ with $\ell(p_1) = 8$ and $\ell(p_2) = 10$, respectively. Furthermore, assume a similarity threshold $\theta = 0.5$. Path $p_3 = \langle (s, n_3), (n_3, n_5), (n_5, n_4), (n_4, t) \rangle$ shares edges (s, n_3) and (n_3, n_5) with p_1, yielding $Sim(p_3, p_1) = 6/8 = 0.75 > \theta$. Therefore, p_3 is not an alternative path to P. On the contrary, the similarity of path $p_4 = \langle (s, n_2), (n_2, n_4), (n_4, t) \rangle$ to p_1 and p_2 is $Sim(p_4, p_1) = 0 < \theta$ and $Sim(p_4, p_2) = 2/10 = 0.2 < \theta$, respectively. Therefore, p_4 is an alternative path to P.

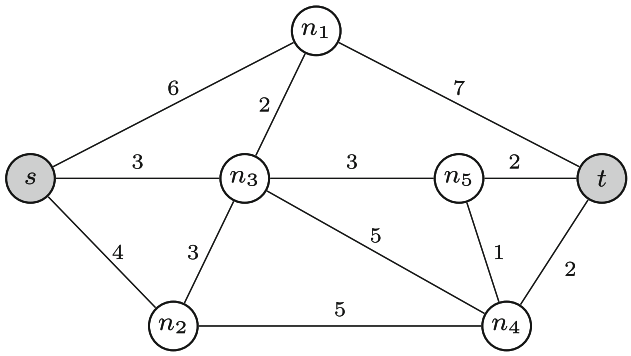


Fig. 2 Running example

4 *k*-Shortest paths with limited overlap

Intuitively, the goal of a kSPwLO(G, s, t, k, θ) query is to identify a set of k paths from a source node s to a target node t, such that (a) the shortest path $p_{sp}(s \rightarrow t)$ is always returned, (b) all returned paths are sufficiently dissimilar to each other with respect to a given similarity threshold θ, and (c) the paths are as short as possible. This is captured in the following definition.

Definition 2 (kSPwLO Problem) Given a road network $G = (N, E)$, a source node s and a target node t both in N, a requested number of paths k, and a similarity threshold $\theta \in [0, 1]$, find the set P_{LO} of k paths from s to t, such that:

(A) all paths in P_{LO} are sufficiently dissimilar to each other, i.e.,

$$\forall p_i, p_j \in P_{LO} \text{ with } i \neq j : Sim(p_i, p_j) \leq \theta$$

(B) every path $p \notin P_{LO}$ is either too long or too similar to a shorter path in P_{LO}, i.e., one of the following two conditions holds for p:

(1) $\forall p_i \in P_{LO} : \ell(p) \geq \ell(p_i)$
(2) $\exists p_i \in P_{LO} : \ell(p_i) \leq \ell(p) \wedge Sim(p, p_i) > \theta$

Condition (A) ensures the dissimilarity of the recommended paths, i.e., all paths are sufficiently dissimilar to each other w.r.t. the given similarity threshold θ. Condition (B) guarantees that each path p in P_{LO} is the shortest possible path that is sufficiently dissimilar to all paths in P_{LO} shorter than p. As a result, the shortest path $p_{sp}(s \rightarrow t)$ is always part of P_{LO}.

Example 2 Consider the road network in Fig. 2 and the query kSPwLO$(G, s, t, 3, 0.5)$. The result of the query is $P_{LO} = \{p_1, p_2, p_3\}$, where $p_1 = \langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$, $p_2 = \langle (s, n_3), (n_3, n_4), (n_4, t) \rangle$, and $p_3 = \langle (s, n_2), (n_2, n_4), (n_6, t) \rangle$ with $\ell(p_1) = 8$, $\ell(p_2) = 10$, and $\ell(p_3) = 11$, respectively. Path p_1 is the shortest path and is always included in the result

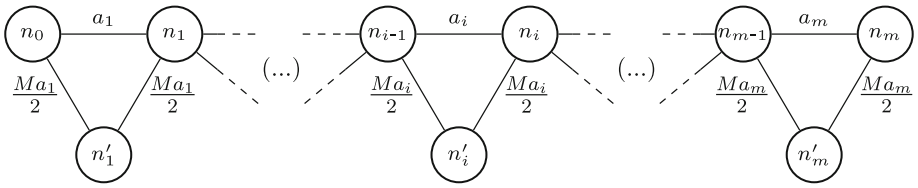


Fig. 3 Road network demonstrating the weakly NP−hardness of kSPwLO

due to Condition (B) of Definition 2. Among all other paths from s to t, path p_2 is the shortest one that is sufficiently dissimilar to p_1, i.e., $Sim(p_2, p_1) = 3/8 = 0.375 < \theta$. Subsequently, among all remaining paths from s to t, p_3 is the shortest one that is sufficiently dissimilar to both p_1 and p_2, i.e., $Sim(p_3, p_1) = 0 < \theta$ and $Sim(p_3, p_2) = 2/10 = 0.2 < \theta$.

4.1 Complexity analysis

Next, we elaborate on the complexity of the kSPwLO.

Theorem 1 *The kSPwLO problem is weakly NP-hard.*

Proof We prove the theorem by reduction from the subset sum problem, a famous weakly NP-complete problem [12]. Given natural numbers $a_1, \ldots, a_m \in \mathbb{N}$, $S \in \mathbb{N}$, the subset sum problem asks whether there is an index set $I \subseteq \{1, \ldots, m\}$ such that $\sum_{i \in I} a_i = S$. For reducing this problem to kSPwLO, we fix an instance $(\{a_i\}_{i=1}^m, S)$ of the subset sum problem and define the road network $G = (N, E)$ sketched in Fig. 3 where:

$$
\begin{aligned}
N &= \{n_i \mid i = 0, \ldots, m\} \cup \{n'_i \mid i = 1, \ldots, m\} \\
E &= \{(n_{i\text{-}1}, n_i) \mid i = 1, \ldots, m\} \\
&\quad \cup \{(n_{i\text{-}1}, n'_i) \mid i = 1, \ldots, m\} \\
&\quad \cup \{(n'_i, n_i) \mid i = 1, \ldots, m\}
\end{aligned}
$$

Furthermore, for all $i = 1, \ldots, m$, we define the edge costs ℓ as $w(n_{i\text{-}1}, n_i) = a_i$, $w(n_{i\text{-}1}, n'_i) = M{\cdot}a_i/2$, and $w(n'_i, n_i) = M{\cdot}a_i/2$, where $M > A = \sum_{i=1}^m a_i$ is a very large number. Let $\{p_{sp}, p_2\}$ be a solution for a kSPwLO$(G, n_0, n_m, 2, S/A)$ query on G. Such a solution exists because there are two edge-disjoint $(n_0{-}n_m)$ paths in G. We claim that there is an index set $I \subseteq \{1, \ldots, m\}$ such that $\sum_{i \in I} a_i = S$ if and only if $\ell(p_2) = A{\cdot}M - S{\cdot}(M-1)$. Since the size of the constructed instance of kSPwLO is polynomial in the size of $(\{a_i\}_{i=1}^m, S)$, this proves the theorem; if there was a polynomial-time algorithm for kSPwLO, then we could solve the subset sum problem in polynomial time.

For proving our claim, we first note that the shortest $(n_0{-}n_m)$ path p_{sp} equals $\{(n_{i\text{-}1}, n_i) \mid = 1, \ldots, m\}$ and has length $\ell(p_{sp}) = A$. Let p be a n_0-n_m-path with $Sim(p, p_{sp}) \leq \theta$ and let $p \cap p_{sp}$ be the intersection of p with p_{sp}. Note that $p \cap p_{sp}$ completely defines the path p: For each $(n_{i\text{-}1}, n_i) \in p_{sp} \setminus p$, we know that p contains the edges $(n_{i\text{-}1}, n'_i)$ and (n'_i, n_i). We now make two observations. First, we note that, because of the definition of Sim, the choice of θ, and $\ell(p_{sp}) = A$, it holds that $\ell(p \cap p_{sp}) \leq S$. Second, by construction of G, we have $\ell(p) = A{\cdot}M - \ell(p \cap p_{sp}){\cdot}(M-1)$, which implies that $\ell(p_2) = A{\cdot}M - \max\{\ell(p \cap p_{sp}) \mid p$ is $(n_0{-}n_m)$ path with $Sim(p, p_{sp}) \leq \theta\}{\cdot}(M-1)$. These two observations directly imply the claim and prove the theorem. □

4.2 Computing *k*SPwLO queries

A naïve approach for computing kSPwLO queries is to enumerate all paths from s to t and choose the subset that satisfies Definition 2. This is clearly impractical. A more efficient approach involves the examination of paths in increasing length order. After adding the shortest path $p_{sp}(s \rightarrow t)$ to the result set P_{LO}, every next path p in length order is constructed using some algorithm for the K-shortest paths [16,23,37]. If p is an alternative to P_{LO}, then p is added to the result. This process continues until P_{LO} contains k paths or all paths from s to t have been examined. Despite its simplicity, this approach is not practical even for small road networks. Chondrogiannis et al. [9] introduced the BSL algorithm that captures this approach and showed that the number of constructed paths is prohibitively high. In the worst case, all paths from s to t have to be constructed, which is a well-known $\#P$-complete problem [33].

To improve computation even further, Liu et al. [22] proposed the FINDKSPD algorithm that employs two lower bounds. The first bound is determined using a reverse shortest path tree, while the second is derived from the similarity function of Eq. 1. These bounds prioritize the examination of paths that are more likely to lead to the next shortest alternative path. In practice though, the number of examined paths is still very high. Our experiments in Sect. 9 show that our exact algorithms clearly outperform FINDKSPD.

4.3 Incomplete solutions

Regardless of the approach, it is important to note that computing an *exact* solution to a kSPwLO query is not always possible. For instance, consider the query kSPwLO$(G, s, t, 5, 0.3)$ on our running example in Fig. 2. By examining paths in length order aiming for constructing the P_{LO} result, we obtain the set $\{p_1, p_4, p_{11}\}$ that contains less than the

requested five paths but still satisfies both conditions of Definition 2. We call such a set of paths an *incomplete* solution. Apparently, if an exact algorithm returns an incomplete result, then an exact solution for the given combination of k and θ does not exist. Nevertheless, as an incomplete result may still be meaningful to the user, our algorithms in Sects. 5 and 6 return the incomplete result if a complete solution does not exist.

4.4 Extending *k*SPwLO

Throughout this paper, we consider a single optimization criterion (i.e., edge weight) and a single constraint (i.e., path overlap) for computing alternative routes. However, our problem definition and our solutions can be adapted to take into account more optimization criteria and/or constraints. A direct approach would be to have composite weights assigned to the edges of the road network edges using a linear combination of multiple criteria. Standard multicriteria optimization can also be supported, as the term 'shortest' can be interpreted as 'the best path according to a set of optimization criteria'. This, however, would also increase the complexity of the problem. With regard to additional constraints, despite focusing on the path overlap, our problem definition and all the algorithms we present support any monotonic similarity measure. Nevertheless, our aim is to provide a general purpose solution. Investigating optimization criteria and/or constraints that might be interesting in specific application scenarios is out of the scope of this paper.

5 Exact algorithms

In this section, we investigate the exact computation of *k*SPwLO queries, and we propose two label-setting algorithms that traverse the road network examining paths in length order.

5.1 The OnePass algorithm

OnePass, our first exact algorithm, traverses the road network expanding paths from the source node s while pruning partially expanded paths that cannot lead to a result as early as possible. We call such paths *infeasible*. For this purpose, we first introduce the notion of *one-way similarity*, which enables the comparison of partially expanded paths $p(s \rightarrow n)$ to paths $p(s \rightarrow t)$ already in the tentative result set. Formally:

$$\overrightarrow{Sim}(p, p') = \frac{\sum_{\forall (n_i, n_j) \in p \cap p'} w(n_i, n_j)}{\ell(p')} \tag{2}$$

Compared to Eq. 1, we observe that Eq. 2 is asymmetric, i.e., $\overrightarrow{Sim}(p', p) \not\equiv \overrightarrow{Sim}(p, p')$. The following lemma follows naturally from the asymmetric nature of Eq. 2.

Lemma 1 *Let p, p' be two paths and $p \cap p' = \{e_1, \ldots, e_m\}$ be the set of their shared edges. The following holds for the one-way similarity of p to p':*

$$\overrightarrow{Sim}(p, p') = \sum_{\forall e_i \in p \cap p'} \overrightarrow{Sim}(\langle e_i \rangle, p')$$

where $\langle e_i \rangle$ is the subpath of p' containing only edge e_i.

Proof From Eq. 2 we have:

$$\begin{aligned}
\overrightarrow{Sim}(p, p') &= \frac{\sum_{\forall e_i \in p \cap p'} w(e_i)}{\ell(p'} = \frac{w(e_1) + \ldots + w(e_m)}{\ell(p')} \\
&= \frac{w(e_1)}{\ell(p')} + \ldots + \frac{w(e_m)}{\ell(p')} \\
&= \overrightarrow{Sim}(\langle e_1 \rangle) + \ldots + \overrightarrow{Sim}(\langle e_m \rangle) \\
&= \sum_{e_i \in p \cap p'} \overrightarrow{Sim}(\langle e_i \rangle, p')
\end{aligned}$$

thus proving the Lemma. □

Lemma 1 unveils the monotonicity of the one-way similarity that enables the incremental computation of Eq. 1. Given two paths p and p', to compute $\overrightarrow{Sim}(p, p')$ it suffices to accumulate the individual similarities of the edges of the longer path. Formally:

$$Sim(p, p') = \sum_{e_i \in p \cap p'} \overrightarrow{Sim}(\langle e_i \rangle, p), \text{ iff } \ell(p) < \ell(p') \tag{3}$$

Apart from enabling the incremental computation of the similarity measure, Lemma 3 also enables the early pruning of partially expanded paths that cannot lead to a solution. Let p_{sub} be a subpath of p. If p_{sub} shares some edges with some $p_i \in P_{LO}$, then p contains all those edges as well. From Eq. 2, we have $\overrightarrow{Sim}(p_{sub}, p_i) \leq \overrightarrow{Sim}(p, p_i)$. Hence, given a similarity threshold θ, if there exists $p_i \in P_{LO}$ such that $\overrightarrow{Sim}(p_{sub}, p_i) \geq \theta$, then path p is infeasible and can be safely discarded. This pruning criterion is formally captured by the following lemma:

Lemma 2 *Let P_{LO} be the tentative result of a kSPwLO(G, s, t, k, θ) query and $p \notin P_{LO}$ be a path from s to t. If p is an alternative path to P_{LO}, then $\overrightarrow{Sim}(p_{sub}, p_i) \leq \theta$ holds for every subpath p_{sub} of p and for all paths $p_i \in P_{LO}$.*

Proof The proof follows directly from Eq. 2. Let $p_i \in P_{LO}$ be some already recommended path. As for both $\overrightarrow{Sim}(p, p_i)$ and $\overrightarrow{Sim}(p_{sub}, p_i)$ the denominator is the same, i.e., $\ell(p_i)$,

the numerator gets the greatest value when all edges of p_i shared by p or p_{sub} are counted for the computation. As $p_{sub} \subseteq p$, we have that $\overrightarrow{Sim}(p_{sub}, p_i) \leq \overrightarrow{Sim}(p, p_i)$, which shows that if $\overrightarrow{Sim}(p, p_i) < \theta$, then $\overrightarrow{Sim}(p_{sub}, p_i) < \theta$ as well. □

As a result of Lemma 2, we observe that, if the one-way similarity of a path $p(s \to n)$ violates the threshold θ, then all of its extensions $p(s \to n) \circ p(n \to t)$ to target node t are infeasible as they violate the similarity constraint.

Next, we present the OnePass algorithm that traverses the road network, expanding every path from source node s that qualifies the pruning criterion of Lemma 2. Similar to all label-setting algorithms, OnePass maintains a set of labels $\Lambda(n)$, where each label $\langle n, p(s \to n)\rangle$ represents a path from s to n.[2] The paths are examined in increasing length order. By doing so, OnePass ensures that the shortest alternative path to each tentative kSPwLO result is computed. Let $P_{LO} = \{p_1, \ldots, p_k\}$ be the result of a kSPwLO query. Every path p_{i+1} is the shortest alternative to each tentative result $P^i_{LO} = \{p_1, \ldots, p_i\}$. Since after the computation of each alternative path more paths are pruned but no new edges are added, every subsequent result path p_{i+1} will be longer than every $p_i \in P^i_{LO}$. Hence, $\overrightarrow{Sim}(p_{i+1}, p_i) = Sim(p_{i+1}, p_i)$ for all $p_i \in P^i_{LO}$.

Algorithm 1 illustrates the pseudocode of OnePass. First, the shortest path $p_{sp}(s \to t)$ is retrieved and the result set P_{LO} is initialized with p_{sp} in Line 1. The algorithm uses a min priority queue $\mathcal{Q}$ (initialized with label $\langle s, \emptyset\rangle$ in Line 2) to traverse the road network. Between Lines 5-16, OnePass examines the contents of $\mathcal{Q}$ until either P_{LO} contains k paths or the queue is depleted. At each round, current label $\langle n, p_n\rangle$ is popped from $\mathcal{Q}$ (Line 6). If n is the target node t, then p_n is recommended, i.e., added to P_{LO} (Line 8). Next, between Lines 8–10, for each label $\langle n_q, p_q\rangle$ in $\mathcal{Q}$, OnePass computes the similarity of p_q to the newly recommended path p_n and determines whether p_q qualifies the pruning criterion of Lemma 2; in particular, if $\overrightarrow{Sim}(p_q, p_n) > \theta$ then p_q can be safely discarded. If node n is not the target t, the algorithm expands the current path p_n considering all outgoing edges (n, n_c) (Lines 13-16), provided that the new path $p_c \leftarrow p_n \circ (n, n_c)$ qualifies the pruning criterion of Lemma 2 (Line 15). Finally, OnePass returns the result set P_{LO} in Line 17. Note that if $\mathcal{Q}$ is depleted before k paths are added to the result, then the result set P_{LO} is incomplete and an exact solution does not exist.

Example 3 Figure 4 exemplifies OnePass for the kSPwLO $(s, t, 3, 0.5)$ query. Initially, the shortest path $p_{sp} = \langle (s, n_3), (n_3, n_5), (n_5, t)\rangle$ is added to P_{LO}. Starting from s, the first alternative path examined by OnePass is p_1. Since the similarity $\overrightarrow{Sim}(p_1, p_{sp}) = 3/8 = 0.375$ is below the threshold $\theta = 0.5$, path p_1 is not pruned. The same holds for p_2, the second path examined by OnePass. Subsequently, OnePass examines paths p_3, p_4 and p_5. Paths p_3 and p_4 are not pruned as their respective similarities $\overrightarrow{Sim}(p_3, p_{sp}) = 3/8 = 0.375$ and $\overrightarrow{Sim}(p_4, p_{sp}) = 0$ do not exceed the similarity threshold. In contrast, for path p_5 the similarity $\overrightarrow{Sim}(p_5, p_{sp}) = 6/8 = 0.75$ exceeds the threshold, hence p_5 is pruned. OnePass proceeds in the same manner until alternative paths $p_{14} = \langle (s, n_3), (n_3, n_4), (n_4, t)\rangle$ and $p_{17} = \langle (s, n_2), (n_2, n_4), (n_4, t)\rangle$ are found and added to P_{LO}. At this point, $|P_{LO}| = 3 = k$ and $P_{LO} = \{p_{sp}, p_{14}, p_{17}\}$ is returned as the final result.

ALGORITHM 1: OnePass

Input: Road network $G = (N, E)$, source $s \in N$, target $t \in N$, # of results k, sim. threshold θ
Output: Result set P_{LO}

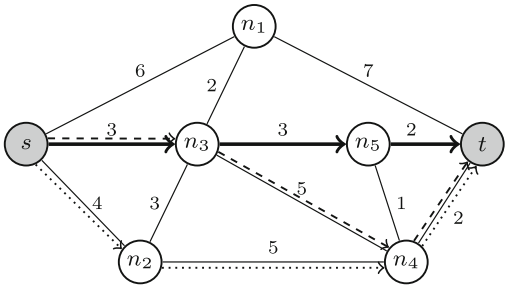
```
1  P_LO ← {p_sp(s→t)};                          ▷ Init. result with p_sp
2  initialize min-priority queue Q with ⟨s, ∅⟩;
3  foreach n ∈ N do
4      Λ(n) ← ∅;
5  while |P_LO| < k and Q not empty do
6      ⟨n, p_n⟩ ← Q.pop();                      ▷ Current path
7      if n = t then
8          P_LO ← P_LO ∪ {p_n};
9          foreach label ⟨n_q, ℓ(p_q)⟩ in Q do
10             if Sim→(p_q, p_i) > θ, ∀p_i ∈ P_LO then   ▷ Lemma 2
11                 remove ⟨n_q, ℓ(p_q)⟩ from Q;
12     else
13         foreach outgoing edge (n, n_c) ∈ E do
14             p_c ← p_n ∘ (n, n_c);            ▷ Expand path p_c
15             if ∀p_i ∈ P_LO : Sim→(p_c, p_i) ≤ θ then
16                 Q.push(⟨n_c, p_c⟩);
17 return P_LO;
```

Complexity analysis Since the pruning criterion of Lemma 2 does not give any guarantee as to how many paths are pruned, in the worst case OnePass has to enumerate all $(s \to t)$ paths. If K is the number of such paths, OnePass runs in $\mathcal{O}(\text{poly}(K))$ time. K is vastly superpolynomial, i.e., $\mathbb{E}(K) = \Omega((|N|-2)!d^{|N|})$ for random graphs with density d [28], which implies that OnePass is prohibitively expensive.

5.2 The MultiPass algorithm

Despite employing the pruning criterion of Lemma 2, OnePass still has to expand and examine a large portion of all possible $p(s \to t)$ paths. To address this shortcoming, we introduce MultiPass, our second exact algorithm. In

[2] In practice, OnePass stores only the predecessor of each label during the expansion. By tracing backwards each step of the expansion, the actual path can be retrieved at any time.



Examined paths		length
$p_1 = p_{sp}$	$\langle(s, n_3)\rangle$	3
p_2	$\langle(s, n_2)\rangle$	4
p_3	$\langle(s, n_3), (n_3, n_1)\rangle$	5
p_4	$\langle(s, n_1)\rangle$	6
(...)		
p_{13}	$\langle(s, n_2), (n_2, n_4), (n_4, n_5)\rangle$	10
p_{14}	$\langle(s, n_3), (n_3, n_4), (n_4, t)\rangle$	10
p_{15}	$\langle(s, n_3), (n_3, n_2), (n_2, n_4)\rangle$	11
p_{16}	$\langle(s, n_2), (n_2, n_3), (n_3, n_5), (n_5, n_4)\rangle$	11
p_{17}	$\langle(s, n_2), (n_2, n_4), (n_4, t)\rangle$	11

⟶ 1st alternative ($p_1 = p_{sp}$) - -> 2nd alternative (p_{14}) ·····> 3rd alternative (p_{17})

Fig. 4 OnePass computing kSPwLO(s, t, 3, 0.5)

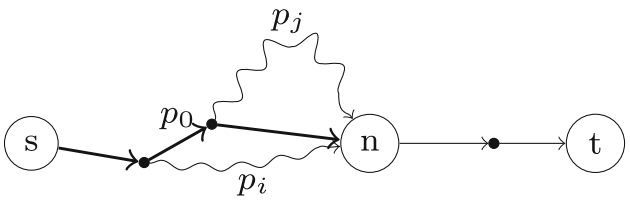


Fig. 5 Pruning paths with Lemma 3

addition to the pruning criterion of Lemma 2, MultiPass employs a second pruning criterion that aims at reducing the search space by avoiding the expansion of *non-promising* paths.

Let $p_{sp}(s \to t)$ be the shortest path from a source node s to a target t as illustrated in Fig. 5. In addition, let $p_i(s \to n)$ and $p_j(s \to n)$ be two distinct paths from source s to a node n of the shortest path p_{sp} such that $\ell(p_i) < \ell(p_j)$. Assuming that both p_i, p_j are extended to reach t following the same path $p(n \to t)$, any extension of p_i will be shorter than the respective extension of p_j. Furthermore, let $\overrightarrow{Sim}(p_i, p_{sp}) \leq \overrightarrow{Sim}(p_j, p_{sp})$, i.e., the similarity of p_i with p_{sp} is equal or lower than the similarity of p_j with p_{sp}. Due to the monotonicity of the one-way similarity, any extension of p_i to n will have the same or less similarity with p_{sp} compared to the respective extension of p_j. As a result, for any extension of p_j there will always be a shorter extension of p_i with less or equal similarity with p_{sp}; thus, p_j can be pruned. The same idea can be utilized to prune the search space when computing the shortest alternative path to a set of paths P. This pruning criterion is formally captured by the following lemma:

Lemma 3 *Let P be a set of paths from a source node s to a target node t, and p_i, p_j be two paths from s to some node n. If $\ell(p_j) > \ell(p_i)$ and $\forall p \in P : \overrightarrow{Sim}(p_i, p) \leq \overrightarrow{Sim}(p_j, p)$ hold then path p_j cannot be part of the shortest alternative path to P and we write $p_i \prec_P p_j$.*

Proof We prove the lemma by contradiction. Let $p'_j = \langle(s, *), \ldots, (*, n), \ldots, (*, t)\rangle$ be an extension of $p_j(s \to n)$ to target t is the shortest alternative path to P. Then, we show that an extension $p'_i = \langle(s, *), \ldots, (*, n), \ldots, (*, t)\rangle$ of $p_i(s \to n)$ to target t is also an alternative path and it will be examined and recommended before p'_j.

According to the definition of an alternative path, $\overrightarrow{Sim}(p'_j, p) \leq \theta$ holds $\forall p \in P$, and following Lemma 2 $\overrightarrow{Sim}(p_j, p) \leq \theta$ also holds $\forall p \in P$. Also, due to the $\forall p \in P_{LO} : \overrightarrow{Sim}(p_i, p) \leq \overrightarrow{Sim}(p_j, p)$ assumption of Lemma 3, we get that $\overrightarrow{Sim}(p_i, p) \leq \theta$ holds $\forall p \in P$.

As extension paths p'_i and p'_j share the same sequence of edges connecting n to target t, we deduce that (a) $\overrightarrow{Sim}(p'_i, p) \leq \theta$ holds $\forall p \in P$, i.e., p'_i is alternative to P and (b) $\ell(p'_i) < \ell(p'_j)$ which means that p'_i will be examined before p'_j. □

Lemma 3 can be utilized to compute the shortest alternative to a set of paths as follows. Let P be the set of paths for which we want to compute the shortest alternative path, and P_n the set of paths from s to some node n created during the expansion of all paths from s. If P_n contains a path $p'(s \to n)$ such that (a) p' is longer than any path $p_n \in P_n \setminus \{p'\}$, and (b) for every path $p \in P$ the similarity $\overrightarrow{Sim}(p', p)$ is higher than $\overrightarrow{Sim}(p_n, p)$ for all paths $p_n \in P_n \setminus \{p'\}$, then p' can be pruned. Note that the addition of a path in P_n may render Condition (B) of Definition 2 not applicable for another path already in P_n. To ensure that set P_n always contains only paths that satisfy Conditions (A) and (B), we have to check whether both conditions still hold every time a new path is added to P_n.

We now present MultiPass, an algorithm that employs the pruning criteria of both Lemma 2 and Lemma 3. For each node n of the road network, MultiPass maintains a set of labels $\Lambda(n)$. Each label represents a path from s to n and is of the form $\langle n, p(s \to n)\rangle$[3]. The algorithm examines paths from

[3] Similar to OnePass, MultiPass stores only the predecessor of each label during the expansion, thus enabling the retrieval of the actual path at any time.

s in increasing order of their length and expands every path $p(s \rightarrow n)$ from s to a node n that satisfy the conditions set by Lemma 2 and Lemma 3. Similar to OnePass, by examining paths from s in length order, MultiPass ensures that the shortest alternative path to each tentative kSPwLO result is computed.

Algorithm 2 illustrates the pseudocode of MultiPass. First, the P_{LO} result set is initialized to the shortest path in Line 1. Before each round, a min priority queue $\mathcal{Q}$ is initialized to $\langle s, \emptyset \rangle$ (Line 3) and each node n is associated with a (initially empty) set of labels $\Lambda(n)$ (Lines 4–5). At each round in Lines 5-20, MultiPass pops label $\langle n, p_n \rangle$ for current path p_n in Line 7. If n is the target t, then p_n is added to P_{LO} and the round terminates (Lines 8–10). Otherwise, MultiPass expands p_n considering all outgoing edges (n, n_c) (Lines 11–17). Each new path $p_c \leftarrow p_n \circ (n, n_c)$ (Line 13) is evaluated against the pruning criteria of Lemma 2 (Lines 14–15) and Lemma 3 (Lines 16–17). If p_c qualifies both pruning criteria, MultiPass removes from $\mathcal{Q}$ and $\Lambda(n_c)$ every label representing a path p'_n such that $p_c \prec_{P_{LO}} p'_n$ (Line 19). The new label is added to $\mathcal{Q}$ (Line 20) and $\Lambda(n_c)$ (Line 21), and the next label is popped from $\mathcal{Q}$. The loop terminates when either k paths are added to P_{LO} or $\mathcal{Q}$ is empty. Finally, MultiPass returns the result set P_{LO} in Line 22. Note that, similar to OnePass if the loop terminates before k paths are found, then the result set P_{LO} is incomplete; an exact solution does not exist.

Example 4 Figure 6 demonstrates MultiPass for the kSPwLO$(G, s, t, 3, 0.5)$ query. Initially, the shortest path $p_{sp}(s \rightarrow n) = \langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$ is computed and added to P_{LO}. The first path examined by MultiPass is p_1. The similarity $\overrightarrow{Sim}(p_1, p_{sp}) = 3/8 = 0.375$ is below the similarity threshold $\theta = 0.5$; hence, p_1 is not pruned. The same holds for p_2, which is the next path examined by MultiPass. Subsequently, MultiPass examines paths p_3, p_4 and p_5. Path p_3 is not pruned as $\overrightarrow{Sim}(p_3, p_{sp}) = 3/8 = 0.375$ does not exceed the similarity threshold. For p_4 the similarity $\overrightarrow{Sim}(p_4, p_{sp}) = 0.375$ also does not exceed the similarity threshold. Since node n_1 has already been visited by p_3 though, we also check Lemma 3, and we have $\overrightarrow{Sim}(p_3, p_{sp}) > \overrightarrow{Sim}(p_4, p_{sp})$ and $\ell(p_3) < \ell(p_4)$. Therefore, Lemma 3 cannot be applied and p_4 is not pruned. On the contrary, for p_5 the similarity $\overrightarrow{Sim}(p_5, p_{sp}) = 6/8 = 0.75$ exceeds the similarity threshold and so, p_5 is pruned by Lemma 2. MultiPass continues the execution of the current round in the same fashion until the alternative path p_{14} with $\ell(p_{14}) = 10$ is found and subsequently added to P_{LO}. Next, MultiPass performs the second round in the same fashion, computes the alternative path p'_{13} with $\ell(p'_{13}) = 11$ and completes the result set P_{LO}.

ALGORITHM 2: MultiPass

Input: Road network $G = (N, E)$, source $s \in N$, target $t \in N$, # of results k, sim. threshold θ
Output: Result set P_{LO}

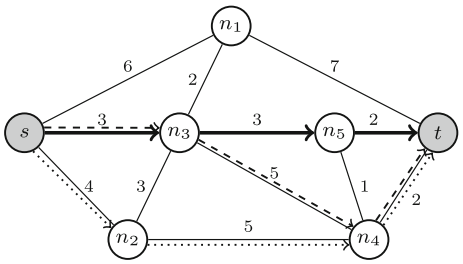
```
1  P_LO ← {p_sp(s→t)};                          ▷ Init. result with p_sp
2  while |P_LO| < k and last round updated P_LO do
3      initialize min-priority queue Q with ⟨s, ∅⟩;
4      foreach n ∈ N do
5          Λ(n) ← ∅;
6      while Q not empty do
7          ⟨n, p_n⟩ ← Q.pop();                  ▷ Current path
8          if n = t then
9              P_LO ← P_LO ∪ {p_n};
10             break;
11         else
12             foreach outgoing edge (n, n_c) ∈ E do
13                 p_c ← p_n ∘ (n, n_c);        ▷ Expand path p_c
14                 if ∃p_i ∈ P_LO : Sim(p_c, p_i) ≥ θ then
                   ▷ Lemma 2
15                     continue;
16                 else if ∃⟨n_c, p'_c⟩ ∈ Λ(n_c) : p'_c ≺_{P_LO} p_c then
                   ▷ Lemma 3
17                     continue;
18                 else
19                     remove from Q and Λ(n_c) all
                       ⟨n_c, p'_c⟩ : p_c ≺_{P_LO} p'_c;   ▷ Lemma 3
20                     Q.push(⟨n_c, p_c⟩);
21                     Λ(n_c) ← Λ(n_c) ∪ {⟨n_c, p_c⟩};
22 return P_LO;
```

Complexity analysis With regard to the complexity of MultiPass we state the following theorem:

Theorem 2 MultiPass *is optimal for the kSPwLO problem.*

Proof To determine the complexity of MultiPass, we assume without loss of generality that the edge weights $\ell(u, v)$ are natural numbers. For each node n_c and each iteration j of the main while-loop (Line 2), we define $c_j(n_c)$ as the number of non-dominated labels $\langle n_c, p_c \rangle$ such that p_c respects the similarity constraints for all previously computed paths $p_i \in P_{LO}$. Note that, because of the pruning in Lines 14, 16, and 19, we have $|\Lambda(n_c)| \leq c_j(n_c)$ throughout iteration j. Furthermore, we know that at most $\sum_{n_c \in N} c_j(n_c)$ labels are added to $\mathcal{Q}$ (Line 20). Hence, MultiPass enters the inner while-loop (Line 6) at most $\sum_{n_c \in N} c_j(n_c)$ times.

For upper-bounding $c_j(n_c)$, we observe that, for each previously computed path $p_i \in P_{LO}$, we have $\overrightarrow{Sim}(p_c, p_i) = 0$ if and only if $\ell(p_c \cap p_i) = 0$, and $\overrightarrow{Sim}(p_c, p_i) = \theta$ if and only if $\ell(p_c \cap p_i) = \theta \cdot \ell(p_i)$. Since the weights are natural numbers, we hence know that $\overrightarrow{Sim}(p_c, p_i)$ can assume at most $\lfloor \theta \ell(p_i) \rfloor + 1$ different values. Now let $\mathcal{C}(n_c)$ be a collection of $(s \rightarrow n_c)$ paths that respect the similarity constraints for all previously computed paths $p_i \in P_{LO}$. If $|\mathcal{C}(n_c)| > \prod_{i=0}^{j-1} \lfloor \theta \cdot \ell(p_i) \rfloor + 1$, then there are paths $p_c, p'_c \in \mathcal{C}(n_c)$ such



Examined paths - round 1		length
p_1	$\langle(s,n_3)\rangle$	3
p_2	$\langle(s,n_2)\rangle$	4
p_3	$\langle(s,n_3),(n_3,n_1)\rangle$	5
p_4	$\langle(s,n_1)\rangle$	6
p_5	$\langle(s,n_3),(n_3,n_2)\rangle$	6
	$(\ldots)$	
p_{13}	$\langle(s,n_2),(n_2,n_4),(n_4,n_5)\rangle$	10
p_{14}	$\langle(s,n_3),(n_3,n_4),(n_4,t)\rangle$	10

Examined paths - round 2		length
p'_1	$\langle(s,n_3)\rangle$	3
p'_2	$\langle(s,n_2)\rangle$	4
	$(\ldots)$	
p'_{12}	$\langle(s,n_2),(n_2,n_4),(n_4,n_5)\rangle$	10
p'_{13}	$\langle(s,n_2),(n_2,n_4),(n_4,t)\rangle$	11

⟶ 1st alternative ($p_1 = p_{sp}$) - -→ 2nd alternative (p_{14}) ⋯→ 3rd alternative (p'_{13})

Fig. 6 MULTIPASS computing kSPwLO(s, t, 3, 0.5)

that $\overrightarrow{Sim}(p_c, p_i) = \overrightarrow{Sim}(p'_c, p_i)$ for all $p_i \in P_{LO}$. Assume without loss of generality that $\ell(p_c) \leq \ell(p'_c)$. Then p_c dominates p'_c. This implies $c_j(n_c) \leq \prod_{i=0}^{j-1} \lfloor \theta \cdot \ell(p_i) \rfloor + 1 = \mathcal{O}((\theta L)^j)$, where $L = \sum_{e \in E} \ell(e)$.

Due to the aforementioned considerations, we know that in the j^{th} iteration of the main while-loop MULTIPASS enters the inner while-loop starting in Line 6 at most $\mathcal{O}(|N| \cdot (\theta \cdot L)^j)$ times. Moreover, if the priority queue $\mathcal{Q}$ is implemented as a Fibonacci heap, each iteration of the inner loop runs in $\mathcal{O}(|N| \cdot (\theta \cdot L)^j)$ time. By summing over iterations of the main loop, we conclude that the overall runtime complexity of MULTIPASS is $\mathcal{O}(k \cdot |N|^2 \cdot (\theta \cdot L)^{2k})$. For $k = \mathcal{O}(1)$, MULTIPASS is a pseudo-polynomial algorithm. Following on Theorem 1, as answering kSPwLO queries is weakly NP-hard even for constant k, MULTIPASS is *optimal* in terms of complexity. In other words, unless $P = NP$, there are no substantially faster algorithms for answering kSPwLO queries. □

Note that, in contrast to ONEPASS, MULTIPASS traverses the road network multiple times, i.e., $k-1$ times in total. As the pruning criterion of Lemma 3 can be utilized to compute only a single alternative path to a set of paths, there is no guarantee that a path pruned at a given round of MULTIPASS using the pruning criterion of Lemma 3, will not lead to a result during a subsequent round. Consider again the example in Fig. 5. Let p_{sp} be the only path in the tentative set P_{LO} of alternative paths. If during the search for the alternative path p_1 to P_{LO}, p_j is pruned because $p_i \prec_P p_j$ holds, p_j cannot be part of the shortest alternative to P_{LO}. However, there is no guarantee that p_j will not be part of the shortest alternative to both p_{sp} and p_1. If p_i is a subpath of p_1, then during the search for the alternative path to $P_{LO} = \{p_{sp}, p_1\}$, p_i may be pruned much earlier by Lemma 2. Consequently, MULTIPASS needs to restart the traversal to ensure the correctness of the result and may potentially re-examine paths already examined in previous rounds. However, in contrast to the runtime complexity of ONEPASS, the runtime complexity of MULTIPASS does not depend on the exponentially large number K of ($s \rightarrow t$) paths. Hence, despite traversing the network $k-1$ times instead of one, MULTIPASS is expected to be much faster than ONEPASS.

6 Performance-oriented heuristic algorithms

Despite employing the pruning criteria of Lemma 2 and Lemma 3, the exact algorithms still examine a large number of paths, which renders them impractical for large road networks. In view of this, we investigate three heuristic algorithms to accelerate the computation of kSPwLO queries. Intuitively, the algorithms treat Condition (B) in Definition 2 as a soft constraint, i.e., the alternative paths are sufficiently dissimilar to each other, but not necessarily as short as possible.

6.1 The ONEPASS$^+$ algorithm

Our first heuristic algorithm, denoted by ONEPASS$^+$, provides a first cut solution for computing kSPwLO queries. Given a source node s and a target node t, ONEPASS$^+$ traverses the road network expanding every path $p(s \rightarrow n)$ from the source to a node n that qualifies both Lemma 2 and Lemma 3. This procedure is the same as one round of MULTIPASS. In contrast to MULTIPASS though, each time a new path is added to the result set P_{LO}, ONEPASS$^+$ does not restart the traversal like MULTIPASS, but continues in a similar fashion to ONEPASS, thus traversing the network only once. Recall our discussion for MULTIPASS though, that a path which is pruned as non-promising during the current round may be promising during the next round. As such, ONEPASS$^+$ cannot guarantee that the exact solution is found. However, as this case applies to only a small subset of the examined paths, the result of ONEPASS$^+$ is expected to be close to the optimal solution in terms of length, a fact which is supported by our experiments in Sect. 9.

Algorithm 3 illustrates the pseudocode of ONEPASS$^+$. The P_{LO} result set is initialized to the shortest path $p_{sp}(s \rightarrow t)$ (Line 1). The algorithm employs a min-priority queue $\mathcal{Q}$ (initialized with s in Line 2) to traverse the road network.

ALGORITHM 3: OnePass$^+$

Input: Road network $G = (N, E)$, source $s \in N$, target $t \in N$, # of results k, sim. threshold θ
Output: Result set P_{LO}

1 $P_{LO} \leftarrow \{p_{sp}(s \rightarrow t)\}$; ▷ Init. result with p_{sp}
2 initialize min-priority queue $\mathcal{Q}$ with $\langle s, \emptyset \rangle$;
3 **foreach** $n \in N$ **do**
4 $\quad \Lambda(n) \leftarrow \emptyset$;
5 **while** $|P_{LO} < k|$ ***and*** $\mathcal{Q}$ *not empty* **do**
6 $\quad \langle n, p_n \rangle \leftarrow \mathcal{Q}.pop()$; ▷ Current path
7 $\quad$ **if** $n = t$ **then**
8 $\quad\quad P_{LO} \leftarrow P_{LO} \cup \{p_n\}$; ▷ Update result set
9 $\quad\quad$ **foreach** *label* $\langle n_q, \ell(p_q) \rangle$ *in* $\mathcal{Q}$ **do**
10 $\quad\quad\quad$ **if** $\overrightarrow{Sim}(p_q, p_i) > \theta, \forall p_i \in P_{LO}$ **then** ▷ Lemma 2
11 $\quad\quad\quad\quad$ remove $\langle n_q, \ell(p_q) \rangle$ from $\mathcal{Q}$;
12 $\quad$ **else**
13 $\quad\quad$ **foreach** *outgoing edge* $(n, n_c) \in E$ **do**
14 $\quad\quad\quad p_c \leftarrow p_n \circ (n, n_c)$; ▷ Expand path p_c
15 $\quad\quad\quad$ **if** $\exists p_i \in P_{LO} : \overrightarrow{Sim}(p_c, p_i) \geq \theta$ **then** ▷ Lemma 2
16 $\quad\quad\quad\quad$ **continue**;
17 $\quad\quad\quad$ **else if** $\exists \langle n_c, p'_c \rangle \in \Lambda(n_c) : p'_c \prec_{P_{LO}} p_c$ **then** ▷ Lemma 3
18 $\quad\quad\quad\quad$ **continue**;
19 $\quad\quad\quad$ **else**
20 $\quad\quad\quad\quad$ remove from $\mathcal{Q}$ and $\Lambda(n_c)$ all $\langle n_c, p'_c \rangle : p_c \prec_{P_{LO}} p'_c$; ▷ Lemma 3
21 $\quad\quad\quad\quad \mathcal{Q}.push(\langle n_c, p_c \rangle)$;
22 $\quad\quad\quad\quad \Lambda(n_c) \leftarrow \Lambda(n_c) \cup \{\langle n_c, p_c \rangle\}$;
23 **return** P_{LO};

Between Lines 5 and 22, OnePass$^+$ examines the contents of $\mathcal{Q}$ until either k paths are added to P_{LO} or $\mathcal{Q}$ is depleted. At each iteration, a label $\langle n, p_n \rangle$ is popped from $\mathcal{Q}$ (Line 6). If node n is target t (Line 7), then p_n is added to P_{LO} (Line 8) and the same update procedure as in OnePass takes place (Lines 9–11), i.e., all paths p_h with $\overrightarrow{Sim}(p_h, p_c) > \theta$ are discarded. Otherwise, the algorithm expands the current path p_n considering all outgoing edges (n, n_c)(Lines 13–22). OnePass$^+$ checks whether the new path $p_c \leftarrow p_n \circ (n, n_c)$ qualifies the pruning criteria of both Lemma 2 (Lines 15–16) and Lemma 3 (Lines 17–18) and updates $\mathcal{Q}$ and $\Lambda(n_c)$ accordingly. Then, OnePass$^+$ adds a new label for p_c to $\mathcal{Q}$ (Line 21) and $\Lambda(n_c)$ (Line 22) and proceeds with popping the next label from $\mathcal{Q}$. Finally, the result set P_{LO} is returned in Line 23.

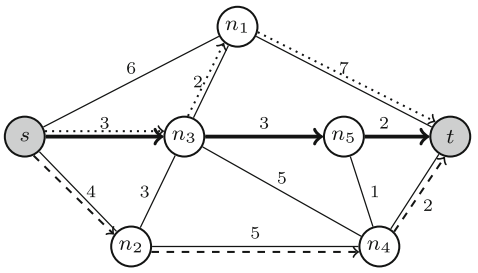
Complexity analysis OnePass$^+$ and MultiPass use the same pruning criteria. Hence, following the complexity analysis of MultiPass in Sect. 5.2, we obtain that OnePass$^+$ enters the main while-loop at most $O(|N| \cdot (\theta \cdot L)^k)$ times and that each of its iterations runs in $O(|N| \cdot (\theta \cdot L)^k)$ time. Therefore, the runtime complexity of OnePass$^+$ is $O(|N^2| \cdot (\theta \cdot L)^{2k})$ time. As OnePass$^+$ traverses the network only once, the overall runtime complexity of OnePass$^+$ corresponds to the complexity of one traversal carried out by MultiPass.

6.2 The SVP$^+$ algorithm

Our second heuristic algorithm, denoted by SVP$^+$, recommends alternative paths by employing the concept of *single-via paths* [2]. Given a road network $G = (N, E)$, a source node s and a target node t, the single-via path of every node $n \in N$ is the concatenation of the shortest paths $p_{sp}(s \rightarrow n)$ and $p_{sp}(n \rightarrow t)$. SVP$^+$ aims at finding a set of k single-via paths such that: (a) the shortest single-via path, i.e., the shortest path $p_{sp}(s \rightarrow t)$, is always recommended, (b) every single-via path is dissimilar to its predecessors with respect to a similarity threshold θ, and (c) all k single-via paths are as short as possible. The main idea behind SVP$^+$ is similar to the baseline method for computing kSPwLO queries discussed in Sect. 4.2. However, instead of iterating over all possible $(s \rightarrow t)$ paths, SVP$^+$ iterates over the much smaller set of single-via paths.

Algorithm 4 illustrates the pseudocode of SVP$^+$. In Line 1, the result set P_{LO} is initialized to the shortest path $p_{sp}(s \rightarrow t)$. Then, two shortest path trees are computed, one from s to every node n of G (Line 2) and a reverse one from every node n of G to t (Line 3). During this step, all distances $d(s, n)$ and $d(n, t)$ are computed. The algorithm organizes the nodes of the road network according to the length of their single-via path, i.e., $\ell(p_{svp}(n)) = d(s, n) + d(n, t)$, inside min-priority queue $\mathcal{Q}$ (Lines 4–6). At each iteration between Lines 7 and 11, SVP$^+$ pops from the queue the top element representing a node n (Line 8) and retrieves the single-via path p_n for node n (Line 9). The single-via paths are examined in increasing order of their length. In Line 10, SVP$^+$ checks whether p_n is simple (contains no cycles) and sufficiently dissimilar to all paths currently in P_{LO}; if so, p_n is added to P_{LO} (Line 11). The algorithm terminates when either k paths have been added to P_{LO} or there exist no more single-via paths to examine, i.e., queue $\mathcal{Q}$ is depleted, in which case the P_{LO} result set contains less than k paths. Finally, the result set P_{LO} is returned in Line 12.

Example 5 Figure 7 exemplifies SVP$^+$ for the query kSPwLO $(G, s, t, 3, 0.5)$. First, SVP$^+$ adds the shortest path $p_{sp}(s \rightarrow t) = \langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$ to the P_{LO} result set. Then, SVP$^+$ iterates over the set of single-via paths in length order. The table in Fig. 7 shows the entire set of single-via paths for the example road network. The first single-via paths examined are $p_{svp}(n_3)$ and $p_{svp}(n_5)$. Both paths are rejected as their similarity to p_{sp} exceeds the similarity threshold θ. Single-via path $p_{svp}(n_4)$ is also rejected as $Sim(p_{svp}(n_4), p) = 6/8 = 0.75$ exceeds the similarity threshold. Next, SVP$^+$ examines $p_{svp}(n_2)$ for which the similarity to the shortest path is $Sim(p_{svp}(n_2), p) =$



Single-via paths		length
$p_{svp}(n_3) = p_{sp}$	$\langle (s,n_3),(n_3,n_5),(n_5,t)\rangle$	8
$p_{svp}(n_5) = p_{sp}$	$\langle (s,n_3),(n_3,n_5),(n_5,t)\rangle$	8
$p_{svp}(n_4)$	$\langle (s,n_3),(n_3,n_5),(n_5,n_4),(n_4,t)\rangle$	9
$p_{svp}(n_2)$	$\langle (s,n_2),(n_2,n_4),(n_4,t)\rangle$	11
$p_{svp}(n_1)$	$\langle (s,n_3),(n_3,n_1),(n_1,t)\rangle$	11

⟶ 1st alternative ($p_{svp}(n_3)$) - - › 2nd alternative ($p_{svp}(n_2)$) ····› 3rd alternative ($p_{svp}(n_1)$)

Fig. 7 SVP$^+$ computing kSPwLO$(s, t, 3, 0.5)$

ALGORITHM 4: SVP$^+$

Input: Road network $G = (N, E)$, source $s \in N$, target $t \in N$, # of results k, sim. threshold θ
Output: Result set P_{LO}

1 $P_{LO} \leftarrow \{p_{sp}(s \rightarrow t)\}$; ▷ Init. result with p_{sp}
2 $T_{s \rightarrow N} \leftarrow$ shortest path tree from s to all $n \in N$;
3 $T_{N \rightarrow t} \leftarrow$ shortest path tree from all $n \in N$ to t;
4 initialize min-priority queue $\mathcal{Q}$ with $\emptyset$;
5 **foreach** $n \in N$ **do**
6 $\quad \mathcal{Q}.push(\langle n, d(s,n)+d(n,t)\rangle)$;
7 **while** $|P_{LO}| < k$ ***and*** $\mathcal{Q}$ *not empty* **do**
8 $\quad \langle n, d(s,n)+d(n,t)\rangle \leftarrow \mathcal{Q}.pop()$;
9 $\quad p_n \leftarrow$ `GetSingleViaPath`$(T_{s \rightarrow N}, T_{N \rightarrow t}, n)$;
10 $\quad$ **if** p_n *is simple* ***and*** $\forall p_i \in P_{LO} : Sim(p_n, p_i) \leq \theta$ **then**
11 $\quad\quad P_{LO} \leftarrow P_{LO} \cup \{p_n\}$; ▷ Update result
12 **return** P_{LO};

0. Hence, $p_{svp}(n_2)$ is added to the result set. Finally, single-via path $p_{svp}(n_1)$ is examined, for which we have $Sim(p_{svp}(n_1), p) = 3/8 = 0.375$ and $Sim(p_{svp}(n_1), p_{svp}(n_2)) = 0$. Thus, $p_{svp}(n_1)$ is also added to the result set. At this point, $|P_{LO}| = 3 = k$ and $P_{LO} = \{p_{sp}, p_{svp}(n_2), p_{svp}(n_1)\}$ is returned as result.

Notice that in Example 5, SVP$^+$ fails to find the exact result for the given kSPwLO query. In particular, path $p = \langle (s,n_3),(n_3,n_4),(n_4,t)\rangle$, which is in the exact kSPwLO result, is not a single-via path; hence, p is not examined by SVP$^+$. At a more general level, this example shows that SVP$^+$ is unable to compute the exact solution to a kSPwLO query if a path p is part of the exact result, but is not a single-via path.

Complexity analysis To build the set of single-via paths, SVP$^+$ needs to run Dijkstra's algorithm twice, which requires $O(|E| + |N| \cdot log|N|)$ time. Since, by definition, there is one single-via path for each node $n \in N \setminus \{s,t\}$, the number of paths that have to be examined by SVP$^+$ is in the worst case $O(|N|)$. Examining whether a given single-via path should be added to the P_{LO} set or not requires $O(k)$ time. Therefore, the overall runtime complexity of SVP$^+$ is $O(|N| \cdot k + |E| + |N| \cdot log|N|)$.

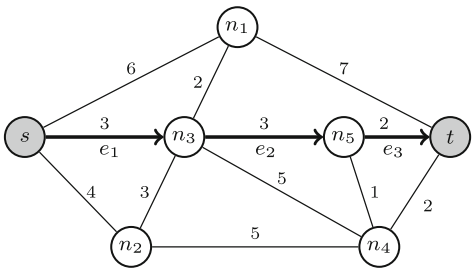
6.3 The ESX algorithm

Our third heuristic algorithm, denoted by ESX, computes kSPwLO by executing shortest path searches while progressively excluding edges from the road network[4]. We identify two important factors that affect the processing of a kSPwLO query with ESX: (1) the order in which edges are removed from the road network and (2) the maintenance of the connectivity of the network. We investigate the former in detail in Sect. 6.4. Regarding the latter, removing an edge from the road network may cause the network to become disconnected. This prevents any subsequent iteration from finding a valid path. To avoid such cases, if the shortest path search fails to find a path from s and t after the removal of an edge e from the road network, ESX re-inserts e in the network and marks it as *non-removable*. Non-removable edges are never removed from the road network regardless of their priority.

Algorithm 5 illustrates the pseudocode of ESX. The algorithm maintains a heap $\mathcal{H}_i$ for each path $p_i(s \rightarrow t)$ added to the P_{LO} result set. The heap organizes every edge e_j contained in p_i according to their priority $\texttt{prio}(e_j)$; $\mathcal{H}_i$ can be either a min-heap or a max-heap depending on the strategy in which the edges are prioritized (see Sect. 6.4). Initially, the $p_{sp}(s \rightarrow t)$ shortest path is added to the result and the associated heap $\mathcal{H}_{sp}$ is initialized with the edges of $p_{sp}(s \rightarrow t)$ (Lines 1–2). The algorithm keeps track of the non-removable edges inside set E_{DNR} (initialized to an empty set in Line 3).

In Lines 4–19, ESX iterates over the already computed paths and their heaps to determine the next alternative path, until either the P_{LO} result set contains exactly k paths or there are no more edges to be removed from the road network. Specifically, ESX first accesses the most recently recommended path, denoted by p_c in Line 5 and then executes the loop in Lines 6–16. At each iteration of this loop, the

[4] In practice, the edges are not actually deleted from the network but only marked as such to be ignored by the search.



Edge priorities			
edges	weight	stretch	paths through
e_1	3	4	0
e_2	3	3	3
e_3	2	1	0

Computed paths		length
p_{sp}	$\langle(s,n_3),(n_3,n_5),(n_5,t)\rangle$	8
p_2	$\langle(s,n_3),(n_3,n_5),(n_5,n_4),(n_4,t)\rangle$	9
p_3	$\langle(s,n_3),(n_3,n_4),(n_4,t)\rangle$	10
p_4	$\langle(s,n_2),(n_2,n_4),(n_4,t)\rangle$	11

Fig. 8 ESX computing kSPwLO$(s,t,3,0.5)$ and edge priorities

ALGORITHM 5: ESX

Input: Road network $G=(N,E)$, source $s\in N$, target $t\in N$, # of results k, sim. threshold θ
Output: Result set P_{LO}

1 $P_{LO}\leftarrow\{p_{sp}(s\to t)\}$; ▷ Init. result with p_{sp}
2 initialize heap $\mathcal{H}_{sp}\leftarrow\langle e_i,\texttt{prio}(e_i)\rangle,\forall e_i\in p_{sp}$; ▷ An $\mathcal{H}_i$ heap assigned to each $p_i\in P_{LO}$
3 $E_{DNR}\leftarrow\emptyset$;
4 **while** $|P_{LO}|<k$ ***and*** $\exists\mathcal{H}_i$ *not empty* **do**
5 $p_c\leftarrow$ last path added to P_{LO};
6 **while** $max\{Sim(p_c,p_i):p_i\in P_{LO}\}>\theta$ *and* $\mathcal{H}_i$ *not empty* **do**
7 $\langle e,\texttt{prio}(e)\rangle\leftarrow\mathcal{H}_i.pop()$;
8 **if** $e\in E_{DNR}$ **then**
9 **continue**;
10 $E\leftarrow E\setminus e$; ▷ Excluding edge from G
11 $p_c\leftarrow\texttt{ShortestPath}(G,s,t)$;
12 **if** p_c *is null* **then**
13 $E\leftarrow E\cup e$; ▷ Re-inserting edge to G
14 $E_{DNR}\leftarrow E_{DNR}\cup e$;
15 **continue**;
16 **if** $\forall p_i\in P_{LO}:Sim(p_c,p_i)\leq\theta$ **then**
17 $P_{LO}\leftarrow P_{LO}\cup\{p_c\}$;
18 initialize heap $\mathcal{H}_c\leftarrow\langle e_j,\texttt{prio}(G,e_j)\rangle,\forall e_j\in p_c$;
19 **return** P_{LO};

already computed path $p_i\in P_{LO}$ with the highest similarity to p_c is chosen as long as p_i contains edges that can be removed from the road network. ESX then considers edge e of path p_i, i.e., the top of the $\mathcal{H}_i$ max-heap (Line 7). If e is not marked as non-removable, then the algorithm removes the edge from road network in Line 10 and computes the new shortest path p_c in Line 11. If p_c is null, then the removal of e has rendered the network disconnected. Consequently, e is re-inserted to the road network (Line 13) and is inserted to E_{DNR} (Line 14), and ESX proceeds to the next round. Otherwise, p_c is checked whether it is an alternative to P_{LO} (Line 16), the result set is updated accordingly in Line 17 and a new heap $\mathcal{H}_c$ associated with p_c is initialized with the edges of p_c in Line 18. This process is repeated until either k paths have been added to P_{LO} or there are no more edges that can be removed, in which case the P_{LO} result set contains less than k paths. Finally, the result set P_{LO} is returned in Line 19.

Example 6 Figure 8 exemplifies ESX for the query kSPwLO $(G,s,t,3,0.5)$. To determine the priority of an edge, we consider its *stretch* shown on the upper table in Fig. 8. Without loss of generality, assume that ESX removes the edge with the smallest stretch first; hence, every $\mathcal{H}_i$ is a min-heap.

Initially, the shortest path from s to t, i.e., $p_{sp}(s\to t)=\langle(s,n_3),(n_3,n_5),(n_5,t)\rangle$, is computed and added to the P_{LO} result set. Based on the edge priorities, ESX first removes the (n_5,n_t) edge of p_{sp} and compute the shortest path on the updated network, $p_2=\langle(s,n_3),(n_3,n_5),(n_5,n_4),(n_4,t)\rangle$ with $\ell(p_2)=9$. Path p_1 is not an alternative to P_{LO} as $Sim(p_2,p_{sp})=0.75>\theta$. Hence, ESX proceeds by removing edge $(n_3,n_5)\in p_{sp}$ and computing the new shortest path $p_3=\langle(s,n_3),(n_3,n_4),(n_4,t)\rangle$ with $\ell(p_3=10)$. We now have $Sim(p_3,p_{sp})<\theta$ and hence, p_3 is added to the result set P_{LO}. Subsequently, ESX updates the edge priorities table by computing the priorities of (n_3,n_4) and (n_4,t) and proceed to the next round. As the current path is p_3 (the last path added to the result set) and e_1 has the highest stretch, either (n_3,n_4) and (n_4,t) is removed. For this example, let ESX remove edge (n_3,n_4) and compute the new shortest path $p_4=\langle(s,n_2)(n_2,n_4)(n_4,t)\rangle$. The new path p_4 is sufficiently dissimilar to both paths in the set, i.e., $Sim(p_4,p_{sp})=0$ and $Sim(p_4,p_2)=0.2$, and therefore, it is added to P_{LO}. At this point $|P_{LO}|=3{=}k$ and ESX terminates, returning $P_{LO}=\{p_{sp},p_3,p_4\}$ as the final result.

Complexity analysis The total amount of time ESX spends in the if-block that starts in Line 16 is $O(k\cdot|N|)$: ESX enters it $(k-1)$ times and then has to initialize the $\mathcal{H}_c$ heap. This initialization is linear in the length of p_c, which is upper bounded by $|N-1|$. In each iteration of the inner while-loop starting in Line 6, one edge is popped from the head associated to one of the first $(k-1)$ paths. Since all paths have length at most $|N|-1$, ESX enters the inner while-loop as most $O(k\cdot|N|)$ times. The most expensive operation that has to be carried out in one iteration of the inner while-loop is the shortest path computation, which requires $O(|N|\cdot\log|N|+|E|)$ time. Therefore, ESX runs in $O(k{\cdot}|N|{\cdot}(|N|{\cdot}\log|N|+|E|))$ time.

6.4 Optimizing ESX

The order in which edges are removed from the road network affects both the result quality and the performance of ESX. Since determining the optimal order is prohibitively expensive, in what follows we describe three strategies to determine which edge to remove at each iteration. Depending on the strategy and the nature of the heap (min or max) that is used to organize the edges of a path, we discuss six variants of ESX.

Smallest/largest edge weight The first strategy uses the edge weight to select which edge to remove at each iteration. We prioritize either edges with small weight (MinW variant) or edges with large weight (MaxW variant). Removing first edges with a large weight causes the next path to be less similar to the already computed paths, thereby enabling the algorithm to terminate sooner. However, there is a higher chance that ESX will miss alternative paths that have small differences in length with the paths already found. Hence, on average the result set is expected to contain longer paths. On the contrary, prioritizing edges with a small weight decreases the chances of missing such paths, but leads to more iterations of the algorithm, thereby increasing the overall runtime of ESX.

Minimum/maximum stretch Our second strategy is to remove edges based on their stretch. Given an edge $e = (n_i, n_j)$, let p be the shortest path from n_i to n_j computed by excluding e from the network. The stretch of an edge e is the difference between the length of p and the length of e, i.e., $stretch(e) = |\ell(p) - \ell(e)|$. Similar to MinW/MaxW, the removal of edges with a high stretch (MaxS variant) is more likely to cause a detour, leading to paths that are less similar to the paths already in the result set and, hence, allowing ESX to find a result sooner. In contrast, prioritizing edges with a small stretch (MinS variant) leads to the examination of more paths. This increases the overall runtime, but at the same time, it also increases the chances that on average the result set will contain shorter paths.

Least/most local shortest paths Our third strategy is inspired by the edge betweenness. Given an edge $e(a, b)$ on some path $p \in P_{LO}$, let $E_{inc}(a)$ and $E_{out}(b)$ be the set of all incoming edges $e(n_i, a)$ to a from some nodes $n_i \in N \backslash \{b\}$ and the set of all outgoing edges $e(b, n_j)$ from b to some nodes $n_j \in N \backslash \{a\}$, respectively. First, ESX computes the set P_s which contains the shortest paths $p_{sp}(n_i, n_j)$ such that $n_i \in E_{inc}(a)$ and $n_j \in E_{out}(b)$. Then, ESX defines the set P'_s of all paths $p_{sp}(n_i, n_j)$ that cross e. Finally, ESX assigns a priority to e, denoted by $prio(e)$, which is set to $|P'_s|$. Similar to the previous variants, the intuition behind MaxP is that the more (shortest) paths cross an edge, the more the chances that removing this edge causes a detour; thus, ESX will find an alternative path sooner. By employing MinP, ESX examines more paths and increases the chance of computing alternative paths that are shorter on average, at the cost of an increased overall runtime.

7 Completeness-oriented heuristic algorithms

Up to this point, we have discussed how to efficiently compute the exact result to kSPwLO queries as well as approximations, where the paths in the result set are not necessarily as short as possible. Since guaranteeing the dissimilarity threshold of the returned paths may lead to incomplete results (cf. Sect. 4.3), we next investigate the approximate computation of kSPwLO queries while treating the Condition (A) in Definition 2 as a soft constraint in order to ensure that the result set contains exactly k paths.[5]

7.1 Relaxation of θ

A naive solution to deal with an incomplete result set P_{LO} is to execute multiple kSPwLO queries, each time increasing the original similarity threshold θ manually. Such a trial-and-error approach is impractical unless there is a hint on how much to increase θ. Furthermore, executing multiple queries means that all previously computed intermediate results, e.g., rejected paths, are disregarded. To this end, we aim for a solution that determines the smallest increase of θ that yields a complete result automatically and computes the complete result without running another query from scratch.

Let kSPwLO(G, s, t, k, θ) be a query whose P_{LO} result set is incomplete, i.e, $|P_{LO}| < k$. In addition, let P_{cand} be a set of paths from s to t with $P_{LO} \subseteq P_{cand}$. We will elaborate on the nature of P_{cand} and how it is computed in Sect. 7.2. Without loss of generality, assume for now that P_{cand} contains all possible $p(s \rightarrow t)$ paths in road network G. By the definition of the kSPwLO problem, for every path $p \in P_{cand} \backslash P_{LO}$, there exists a shorter path $p' \in P_{LO}$ such that $Sim(p, p') > \theta$. Based on this, we define the maximum similarity of a path $p \in P_{cand}$ to result set P_{LO} as:

$$Sim_{max}(p, P_{LO}) = \max_{\forall p_i \in P_{LO}: \ell(p') \leq \ell(p)} Sim(p, p') \quad (4)$$

Consequently, path $p \in P_{cand}$ cannot be part of the result set P_{LO}, as long as either p' is part of the result, or the similarity threshold is greater than θ and lower than $Sim_{max}(p, P_{LO})$. In fact, even if we use a new similarity threshold equal or higher than $Sim_{max}(p, P_{LO})$, we cannot guarantee that p will be included in the new result set P_{LO}.

[5] Provided that at least k distinct paths from s to t exist on road network G.

That is because the new threshold may cause a shorter path p'' with $\ell(p'') \leq \ell(p)$ to join P_{LO} that keeps p out. In other words, with this new threshold, we can only guarantee that the result set P_{LO} will indeed change. In this context, we next define θ_{min} as the minimum value for the new similarity threshold such that query kSPwLO(G, s, t, k, θ) will return an updated result as:

$$\theta_{min} = \min_{\forall p \notin P_{LO}} Sim_{max}(p, P_{LO}). \tag{5}$$

Essentially, if the new threshold is set inside $[\theta, \theta_{min})$, the P_{LO} result remains unchanged, while a value equal or higher than θ_{min} will cause P_{LO} to change.

To ensure a complete result, we need to progressively increase θ_{min} until P_{LO} contains exactly k paths. This iterative procedure is captured by `Complete_kSPwLO` illustrated in Function 1. The function receives as input a set of candidate paths P_{cand} out of which the result will be extracted, the number k of requested paths, and an initial similarity threshold θ. The first step is to sort the paths in P_{cand} in increasing length order. Note that if P_{cand} contains less than k paths, it is impossible to return a complete result; hence, the function terminates and $P_{LO} \leftarrow P_{cand}$.

Next, between Lines 4 and 15 the function progressively relaxes the similarity threshold θ until the P_{LO} result is complete. At each round, θ_{min} and P_{LO} are re-initialized to 1 and the shortest path p_{sp}, respectively (Lines 5-6); note that p_{sp} is the first path in P_{cand}. Then, in between Lines 7 and 14, `Complete_kSPwLO` examines the paths in P_{cand} in increasing length order. Fix such a path p. If p is alternative to the current result set P_{LO}, p is added to P_{LO}. At this point, the function will terminate if P_{LO} already contains k paths as the result set is now complete. In contrast, if the current path p is not alternative to P_{LO}, the current θ_{min} is checked against Sim_{max} of p to P_{LO} in Line 13 and is updated in Line 14 accordingly using Eqs. 4 and 5. Finally, the value of θ is relaxed, i.e., increased to θ_{min} to prepare for the next round. Note that the while loop eventually terminates due to the condition in Line 10, i.e., after k paths are added to P_{LO}.

Example 7 Consider the kSPwLO$(G, s, t, 5, 0.3)$ query on the road network in Fig. 9 and its incomplete result $P_{LO} = \{p_1, p_5, p_{11}\}$. Also assume that P_{cand} contains all 24 possible paths from s to t. Figure 10 illustrates the process of relaxing similarity threshold θ and completing the result set. The paths contained inside the initial P_{LO} are marked with $(*)$.

The first round starts by setting $\theta_{min} = 1$ and $P_{LO} = \{p_1\}$. We then iterate over the paths in P_{cand}. Path p_2 is more than 30% similar to p_1 and so it is ignored but enables us to update $\theta_{min} = 0.75$. In the same manner, p_3 is ignored but allows us to set $\theta_{min} = 0.375$. The first sufficiently dissimilar path to p_1 is p_4, hence P_{LO} is updated to $\{p_1, p_4\}$. All subsequent paths are not alternative to the current P_{LO} until p_{11}, which is added to P_{LO}. `Complete_kSPwLO` continues in the same manner until all 24 paths are examined. Note that when p_{17} is examined, θ_{min} is set to 0.365, which is also the value at the end of the round. Consequently, θ gets a new value $\theta = \theta_{min} = 0.364$, indicating that in order for the P_{LO} to change, paths should be allowed to be at most 36.4% similar to each other instead of the initial 30%. Columns 4–6 in Fig. 10 report all path similarities computed during the first round.

The second round starts by setting θ_{min} and P_{LO} to 1 and $\{p_1\}$, respectively; remember that θ is now 0.364. `Complete_kSPwLO` operates exactly as in the first round until p_{17} is examined. This time, the path is sufficiently dissimilar to the current result set, and P_{LO} is updated to $\{p_1, p_4, p_{11}, p_{17}\}$. At the end of the second round, the similarity threshold is further relaxed to $\theta = 0.375$. For illustration purposes, Fig. 10 reports only the extra path similarities computed in the second round.

FUNCTION 1: `Complete_kSPwLO`

Input: Set of paths P_{cand}, # of results k, similarity threshold θ
Output: Result set P_{LO}

```
1  sort P_cand by length in ascending order;
2  if |P_cand| ≤ k then
3      return P_cand;                    ▷ Result cannot be completed
4  while true do
5      θ_min ← 1;                        ▷ Initialization for next round
6      P_LO ← {p_sp};                    ▷ First path in P_cand
7      foreach p ∈ P_cand \ {p_sp} do
8          if ∀p_i ∈ P_LO : Sim(p, p_i) ≤ θ then
9              P_LO ← P_LO ∪ {p};
10             if |P_LO| = k then
11                 return P_LO;          ▷ Result completed
12         else
13             if θ_min > Sim_max(p, P_LO) then
14                 θ_min ← Sim_max(p, P_LO);   ▷ Update θ_min, Equations 4, 5
15     θ ← θ_min;                        ▷ Relax similarity threshold
```

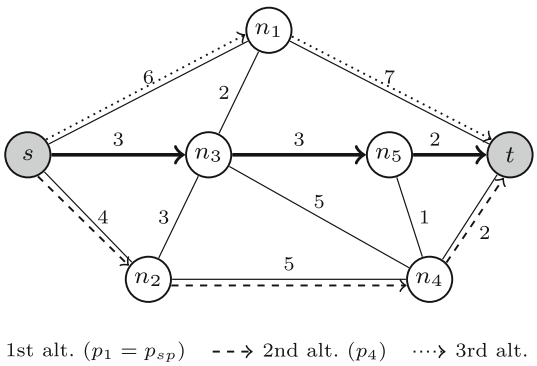


Fig. 9 Result set of query kSPwLO$(G, s, t, 5, 0.3)$

All paths P_{all} in length order		length	Round 1				Round 2
			$Sim(p_i, p_1)$	$Sim(p_i, p_4)$	$Sim(p_i, p_{11})$	$\cdots$	$Sim(p_i, p_{17})$
p_1=$p_{sp}(*)$	$\langle(s,n_3),(n_3,n_5),(n_5,t)\rangle$	8	-	-	-		-
p_2	$\langle(s,n_3),(n_3,n_5),(n_5,n_4),(n_4,t)\rangle$	9	0.75	-	-		-
p_3	$\langle(s,n_3),(n_3,n_4),(n_4,t)\rangle$	10	**0.375 (R2)**	-	-		-
$p_4(*)$	$\langle(s,n_2),(n_2,n_4),(n_4,t)\rangle$	11	-	-	-		-
p_5	$\langle(s,n_3),(n_3,n_4),(n_4,n_5),(n_5,t)\rangle$	11	0.625	0	-		-
p_6	$\langle(s,n_3),(n_3,n_1),(n_1,t)\rangle$	12	0.375	0	-		-
p_7	$\langle(s,n_2),(n_2,n_4),(n_4,n_5),(n_5,t)\rangle$	12	0.25	0.818	-		-
p_8	$\langle(s,n_2),(n_2,n_3),(n_3,n_5),(n_5,t)\rangle$	12	0.625	0.364	-		-
p_9	$\langle(s,n_2),(n_2,n_3),(n_3,n_5),(n_5,n_4),(n_4,t)\rangle$	13	0.375	0.545	-		-
p_{10}	$\langle(s,n_3),(n_3,n_2),(n_2,n_4),(n_4,t)\rangle$	13	0.375	0.636	-		-
$p_{11}(*)$	$\langle(s,n_1),(n_1,t)\rangle$	13	-	-	-		-
p_{12}	$\langle(s,n_1),(n_1,n_3),(n_3,n_5),(n_5,t)\rangle$	13	0.625	0	0.462	$\cdots$	-
p_{13}	$\langle(s,n_3),(n_3,n_2),(n_2,n_4),(n_4,n_5),(n_5,t)\rangle$	14	0.625	0.455	0		-
p_{14}	$\langle(s,n_2),(n_2,n_3),(n_3,n_4),(n_4,t)\rangle$	14	0	0.545	0		-
p_{15}	$\langle(s,n_1),(n_1,n_3),(n_3,n_5),(n_5,n_4),(n_4,t)\rangle$	14	0.375	0.182	0.462		-
p_{16}	$\langle(s,n_1),(n_1,n_3),(n_3,n_4),(n_4,t)\rangle$	15	0	0.182	0.462		-
p_{17}	$\langle(s,n_2),(n_2,n_3),(n_3,n_4),(n_4,n_5),(n_5,t)\rangle$	15	0.25	**0.364 (R1)**	0		-
p_{18}	$\langle(s,n_2),(n_2,n_3),(n_3,n_1),(n_1,t)\rangle$	16	0	0.364	0.538		0.467
p_{19}	$\langle(s,n_1),(n_1,n_3),(n_3,n_4),(n_4,n_5),(n_5,t)\rangle$	16	0.25	0	0.462		0.533
p_{20}	$\langle(s,n_1),(n_1,n_3),(n_3,n_2),(n_2,n_4),(n_4,t)\rangle$	18	0	0.636	0.462		0
p_{21}	$\langle(s,n_1),(n_1,n_3),(n_3,n_2),(n_2,n_4),(n_4,n_5),(n_5,t)\rangle$	19	0.25	0.455	0.462		0.2
p_{22}	$\langle(s,n_2),(n_2,n_4),(n_4,n_3),(n_3,n_5),(n_5,t)\rangle$	19	0.625	0.818	0		0.4
p_{23}	$\langle(s,n_2),(n_2,n_4),(n_4,n_5),(n_5,n_3),(n_3,n_1),(n_1,t)\rangle$	22	0	0.818	0.538		0.333
p_{24}	$\langle(s,n_2),(n_2,n_4),(n_4,n_3),(n_3,n_1),(n_1,t)\rangle$	23	0	0.818	0.538		0.267

Fig. 10 Sample execution of `Complete_kSPwLO` function for query kSPwLO($G, s, t, 5, 0.3$) on our running example

Finally, the third round commences by setting again $\theta_{min} = 1$ and $P_{LO} = \{p_1\}$. Due to the new threshold $\theta = 0.375$, `Complete_kSPwLO` adds to P_{LO} paths p_1, p_3, p_4, p_6, and p_{15}, in this order. Note that it is possible to add p_{15} because p_{11} is now excluded from the result as $Sim(p_{11}, p_6) = 0.58 > \theta$. The process terminates after adding p_{15}, as P_{LO} contains $k = 5$ paths.

Theorem 3 *Given a set of paths P_{cand} from s to t and a similarity threshold θ, `Complete_kSPwLO` determines the lowest value for $\theta_{min} \geq \theta$ such that there is a solution set $P_{LO} \subseteq P_{cand}$ with $|P_{LO}| = k$.*

Proof Let P_{cand} be an input set of at least k distinct paths from s to t. Also, let `Complete_kSPwLO` terminate after I iterations of its main while-loop. Let θ_i denote the value of the similarity threshold and k_{θ_i} the size of the result set P_{LO} during the i-th iteration. We prove by induction on $i \in \{1, \ldots, I\}$ that $k_{\theta'} < k$ holds for all $\theta' \in [\theta, \theta_I)$. If $I = 1$, i.e., the function `Complete_kSPwLO` terminates after a single iteration, we have $\theta_1 = \theta$ which is by definition the minimum possible value. Now, if the inductive hypothesis holds for $i < I$ but not for $i+1$, there is a $\theta' \in [\theta, \theta_{i+1})$ such that $k_{\theta'} = k$. We know from the hypothesis that $\theta' \geq \theta_i$. Furthermore, we know from the definition of θ_i and θ_{i+1} that all $\theta' \in [\theta_i, \theta_{i+1})$ yield the same result set (cf. Eqs. 4 and 5). Together, these observations imply that $k_{\theta_i} = k$, which contradicts the fact that function `Complete_kSPwLO` did not terminate at the i-th iteration. □

Complexity analysis The runtime of function `Complete_kSPwLO` depends on the size of $|P_{cand}|$. At each round, the number of paths examined by `Complete_kSPwLO` is $O(|P_{cand}|)$. To determine whether a path should be added to P_{LO} or not requires $O(k)$ time. Furthermore, as for each path we keep at most $k-1$ similarities with paths in the result set, to add a single path to P_{LO} we need, in the worst case, to run an iteration for all the similarities of all paths, i.e., $O(|P_{cand}| \cdot k)$ iterations. Since, we need to fill the result set with k paths, the total number of iterations is $O(|P_{cand}| \cdot k^2)$. Therefore, the overall runtime complexity of `Complete_kSPwLO` is $O(|P_{cand}|^2 \cdot k^3)$.

7.2 The SVP-C and ESX-C algorithms

As discussed in the previous subsection, the `Complete_kSPwLO` function can operate with any arbitrary set of ($s \rightarrow t$) paths as input. The only requirement dictated by Definition 2 for P_{cand} is to include the shortest path from s to t.

Paradigm 1 outlines the completeness-oriented computation of kSPwLO(G, s, t, k, θ) queries. Initially, the query is processed by a kSPwLO algorithm. Besides P_{LO}, the algorithm also returns the candidate set P_{cand} of paths from s to t. If P_{LO} contains k paths, the result is already complete and the computation terminates. Otherwise, the completeness process takes over in between Lines 4 and 6. To deliver a complete result set, P_{cand} must contain k or more paths. To this end, if P_{cand} contains less than k paths, we add to it

PARADIGM 1: Complete P_{SPwLO} computation

Input: Road network $G = (N, E)$, source $s \in N$, target $t \in N$, # of results k, sim. threshold θ
Output: Result set P_{LO}

```
1 ⟨P_LO, P_cand⟩ ← kSPwLOAlgorithm(G, s, t, k, θ);
2 if |P_LO| = k then
3     return P_LO;
4 if |P_cand| < k then
5     P_cand ← P_cand ∪ kShortestPaths(G, s, t, k);
6 return Complete_kSPwLO (P_cand, k, θ);
```

the k-shortest paths from s to t (Lines 4-5). Finally, P_{cand} is fed to the `Complete_kSPwLO` to produce a complete P_{LO} result in Line 6.

In practice, using all possible paths from s to t as P_{cand} is prohibitively expensive. Therefore, we rely on the kSPwLO algorithms to provide a set of candidate paths. Not all of our kSPwLO algorithms are compatible with Paradigm 1 though. Algorithms that traverse the original network, i.e., exact OnePass, MultiPass and heuristic OnePass$^+$, do not qualify for this purpose, as the only $(s \rightarrow t)$ paths these algorithms construct are the ones that constitute the result set. On the contrary, this is possible with SVP$^+$ and ESX; recall that SVP$^+$ considers concatenations of single-via paths as candidate results (cf. Algorithm 4, Line 9) while ESX constructs candidate paths by removing edges (cf. Algorithm 5, Line 11). We denote by SVP- C and ESX- C the algorithms that follow Paradigm 1 and employ SVP$^+$ and ESX respectively to compute an initial P_{LO} result and a set of candidate paths P_{cand}.

8 Optimization with lower bounds

To further improve the performance of both our exact and heuristic algorithms, we employ a lower bound $\underline{d}(n, t)$ for the network distance $d(n, t)$ of every node n to the target t. Such a lower bound enables algorithms to direct the traversal toward the target and has been employed by various existing works as well [22,30]. Also, such bounds can be computed in a preprocessing phase [32]. However, to enable our algorithms to work on road networks with changing edge weights, we compute bounds on query time.

To derive tight $\underline{d}(n, t)$ lower bounds, we run Dijkstra's algorithm [14] in reverse from target t to every node n of the road network. By executing such an all-to-one query, we obtain for every node n its exact distance $d(n, t)$ to the target t, which is the tightest possible lower bound. This computation takes place at the beginning of the execution of all algorithm that employ this optimization, i.e., OnePass, MultiPass, OnePass$^+$, and ESX. Instead of simply computing the shortest path from s to t, we compute the shortest path tree from target t to each node n in the road network.

Optimizing OnePass/MultiPass/OnePass$^+$. In principle, OnePass, MultiPass and OnePass$^+$ utilize lower bounds in the same fashion. Instead of sorting labels into the priority queue based on their distance from the source, each label associated with some node n is sorted based on the total estimated distance $d(s, n) + \underline{d}(n, t)$. Apart from reducing the search space of the traversal, the pruning power of the algorithms is enhanced as well. Paths to nodes that are far away from the target have less chances of sharing edges with already recommended paths. Instead, paths to nodes that are closer to the target are more likely to share edges with already recommended paths and therefore have more chances to be pruned.

Optimizing ESX. As we explained in Sect. 6.3, ESX computes alternative paths by executing shortest path searches repeatedly. By employing the aforementioned lower bounds, ESX uses A^*-search [15]. Since the lower bounds are the tightest possible ones, the search space of the traversal is expected to be small. While the quality of the bounds drops after each iteration as they are not updated after each edge removal, the correctness of the A^*-search is still ensured.

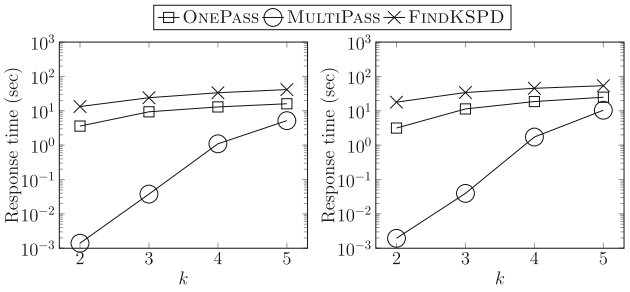
9 Experimental evaluation

In this section, we report the results of our experiments that involve ten real-world road networks obtained from three different sources [1,5,19]. We selected road networks with different structural characteristics. Table 1 shows the number of nodes and edges, and the structure of each road network.

To assess the performance of all algorithms, we measure their average runtime over 1000 kSPwLO queries with randomly selected source and target nodes, while varying the number k of requested paths and the similarity threshold θ. In each experiment, we vary one of the two parameters and set

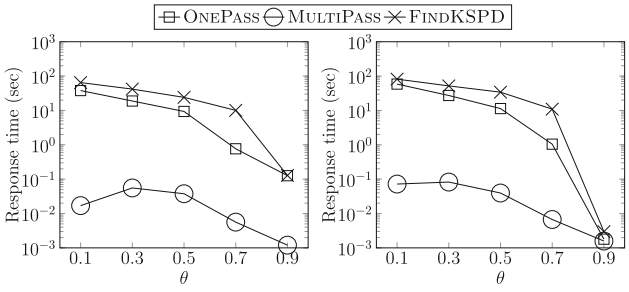
Table 1 Datasets

Road networks	Nodes	Edges	Structure
Rome	3353	8870	City-center
Oldenburg	6105	14,058	City-center
San Joaquin	18,263	47,594	Grid-based
Tianjin	31,002	86,584	Ring-based
Porto Alegre	63,751	187,364	Grid-based
Beijing	74,383	222,778	Ring-based
Milan	187,537	525,296	Ring-based
Chicago	386,533	1,121,620	Grid-based
Colorado	435,666	1,057,066	State
Florida	1,070,376	2,712,798	State



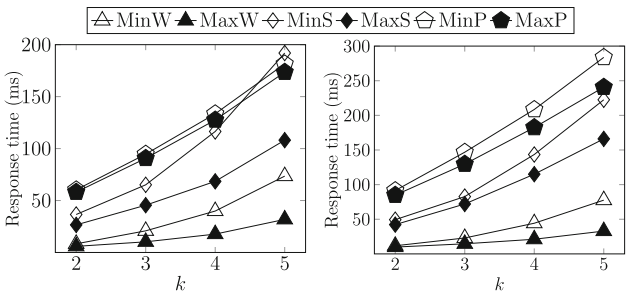
(a) Rome **(b)** Oldenburg

Fig. 11 Runtime of exact algorithms for kSPwLO queries varying requested paths k ($\theta = 0.5$)



(a) Rome **(b)** Oldenburg

Fig. 12 Runtime of exact algorithms for kSPwLO queries varying similarity threshold θ ($k = 3$)



(a) San Joaquin **(b)** Tianjin

Fig. 13 Response time of ESX variants varying requested paths k ($\theta = 0.5$)

the other to its default value, i.e., $k = 3$ and $\theta = 0.5$. For our performance-oriented heuristic algorithms, we also measure the quality, i.e., the shortness of the alternative paths, by comparing their average length to the length of the shortest path for each query, and the completeness of the result set, i.e., the percentage of queries for which an algorithm returns exactly k paths. For our completeness-oriented heuristic algorithms, we identify the maximum pairwise similarity between the results of each query such that the result set contains k paths, and we report the average similarity value for all queries.

All algorithms were implemented in C++[6], the code was compiled using GNU G++ 8, and the tests run on a machine with 12 Intel Xeon E5-2650 (2.20GHz) processors and 256GB RAM running Ubuntu Linux. Moreover, our implementations of OnePass, MultiPass and OnePass$^+$ employ the lower bounds of Sect. 8.

9.1 Exact algorithms

Figure 11 reports the runtime of our exact algorithms for processing kSPwLO queries while varying parameter k. As expected, the runtime of all algorithm goes up for increasing values of k since more paths need to be examined. MultiPass clearly outperforms both OnePass and FindKSPD [22] and, in most cases, by a large margin. By utilizing the pruning criterion of Lemma 3, MultiPass is able to significantly reduce the total number of examined paths, even though it scans the network multiple times. Furthermore, while OnePass is always faster than FindKSPD, none of these two algorithms scale. Even for the road networks of Rome and Oldenburg, the smallest networks used in our experiment, both algorithms require in most cases several seconds on average to process kSPwLO queries.

Figure 12 reports the runtime of our exact algorithms for processing kSPwLO queries while varying parameter θ. Similar to varying k, MultiPass is clearly the fastest exact algorithm. We also observe that the runtime of OnePass and FindKSPD increases for decreasing values of θ while the runtime of MultiPass peaks for $\theta = 0.3$. This result reveals an important trade-off: as θ increases, the pruning power of Lemma 2 deteriorates, and, MultiPass constructs more (partial) paths. At the same time, the next path added to the result set is shorter due to the higher similarity threshold, and hence, the algorithm terminates earlier. With a decreasing θ, the pruning power of Lemma 3 also increases and more partial paths are pruned.

9.2 Heuristic algorithms

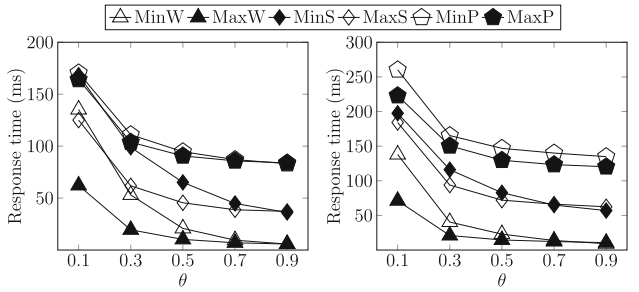
Next, we report the performance, result quality, and completeness of our heuristic algorithms.

9.2.1 Comparison of ESX variants

Before presenting the results of our experiments for all heuristic algorithms, we analyze the effectiveness of different ESX variants/edge removal strategies (cf. Sect. 6.4) to determine the most efficient one. For this purpose, we present our measurements on the road networks of San Joaquin and Tianjin.

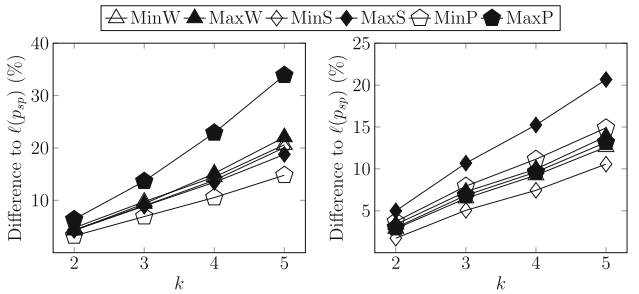
Runtime Figure 13 and 14report on the runtime of the different ESX variants, i.e., ESX varying the number of requested paths k and the similarity threshold θ. Clearly, MinW and

[6] https://github.com/tchond/kspwlo



(a) San Joaquin **(b)** Tianjin

Fig. 14 Response time of ESX variants varying similarity threshold θ ($k = 3$)



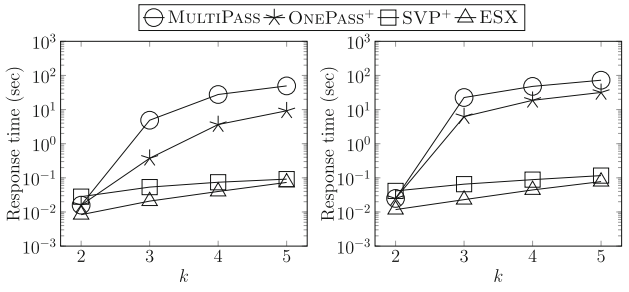
(a) San Joaquin **(b)** Tianjin

Fig. 15 Average length difference to $\ell(p_{sp})$ of ESX variants varying requested paths k ($\theta = 0.5$)

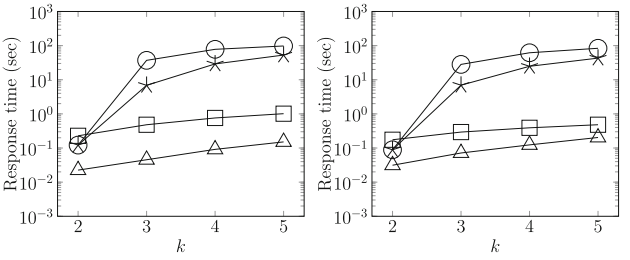
MaxW are the fastest variants. That is mainly because these two variants incur no computational overhead to determine the priority of each edge. The MinS and MaxS variants come in second place, while MinP and MaxP are the slowest ones, in almost all cases. Regarding the prioritization of edges with low priority (i.e., Min* variants) or high priority (i.e., Max* variants), as expected removing first edges with high priority is more efficient.

Result shortness Figure 15 reports the average length difference to p_{sp} of the alternative paths produced by each ESX variant. For San Joaquin, MinP is the variant that returns the shortest paths on average. MinW, MinS and MaxS return paths of similar average length, MaxW recommends even longer paths, while MaxP recommends the longest paths on average among all variants. For Tianjin, we observe that MinS returns the sets with the shortest paths on average, MinW, MaxW and MaxP recommend slightly longer paths, MinP returns even longer paths, while MaxS returns the longest paths on average.

In summary, MinW is the most consistent variant of ESX. It produces alternative paths quickly, being the second fastest variant in all cases, and with low average length, being ranked either second or third in all cases. Based on these observations, we use the MinW variant of ESX for the rest of our experiments.

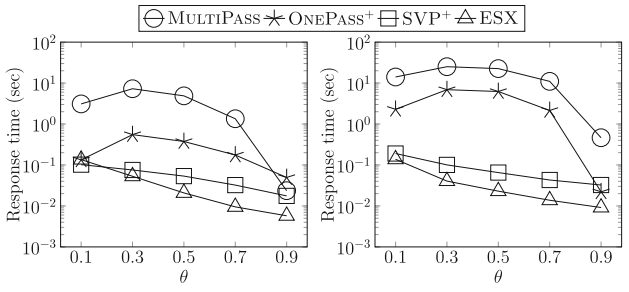


(a) San Joaquin **(b)** Tianjin

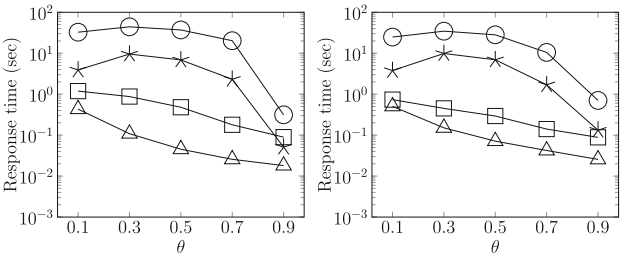


(c) Porto Alegre **(d)** Beijing

Fig. 16 Performance of kSPwLO algorithms varying requested paths k ($\theta = 0.5$)



(a) San Joaquin **(b)** Tianjin



(c) Porto Alegre **(d)** Beijing

Fig. 17 Performance of kSPwLO algorithms varying similarity threshold θ ($k = 3$)

9.2.2 Performance-oriented heuristic algorithms

Runtime Figures 16 and 17 report the response time of our performance-oriented heuristic algorithms OnePass$^+$, SVP$^+$ and ESX, on four clearly larger road networks than the ones we used for measuring the performance of the exact algorithms. We also include the fastest exact algorithm, i.e., MultiPass.

Figure 16 reports the response time of the heuristic algorithms and MULTIPASS varying the number of requested paths k. While, as expected, the runtime of all algorithms increases with k, the efficiency of MULTIPASS and ONEPASS$^+$ deteriorates much faster. For $k \geq 3$, MULTIPASS is approximately three times slower than ONEPASS$^+$ and more than three orders of magnitude slower than SVP$^+$ and ESX. ONEPASS$^+$ is also clearly outperformed by SVP$^+$ and ESX on all road networks for $k \geq 3$ by approximately three orders of magnitude. In brief, ESX is clearly the fastest algorithm in all cases, SVP$^+$ comes second in all networks for $k \geq 3$, while MULTIPASS and ONEPASS$^+$ have comparable performance to SVP$^+$ and ESX only for $k = 2$.

Figure 17 reports the response time of the heuristic algorithms and MULTIPASS varying the similarity threshold θ. The overall picture is the same as in Fig. 16, i.e., SVP$^+$ and ESX are the clear winners, MULTIPASS is approximately three times slower than ONEPASS$^+$, which in turn is up to two orders of magnitude slower than SVP$^+$ and ESX. An interesting observation though is that while the response time of SVP$^+$ and ESX decreases with increasing values of θ, MULTIPASS and ONEPASS$^+$ show a local maximum for $\theta = 0.3$. This indicates an important trade-off: as θ increases, the pruning power of Lemma 2 deteriorates, and both MULTIPASS and ONEPASS$^+$ construct more (partial) paths. At the same time, the higher similarity threshold causes each next path to be determined faster and the algorithms to terminate earlier. With θ decreasing, the pruning power of Lemma 3 also increases and more partial paths are pruned.

Result shortness Figure 18 shows the average length difference to p_{sp} of the exact solution and the computed results of the heuristic algorithms ONEPASS$^+$, SVP$^+$ and ESX. Note that only queries for which all algorithms returned a complete result set, i.e., a set of k paths, are considered. Naturally, the exact kSPwLO result provides the shortest alternatives. Looking at the heuristic algorithms, ONEPASS$^+$ produces the shortest alternative paths, which are very close to the paths in the exact solution. Both SVP$^+$ and ESX recommend alternative paths that are up to 15% longer on average than the paths in kSPwLO, with SVP$^+$ returning slightly shorter ones than ESX.

Completeness As already discussed, the algorithms for kSPwLO queries are not always able to compute all requested k alternative paths. Table 2 reports, for each algorithm, the percentage of queries for which exactly k alternative paths were found. Naturally, the exact solution kSPwLO has the highest completeness ratio. ONEPASS$^+$ is very close to the exact solution, achieving a completeness ratio of more than 90% in all scenarios. SVP$^+$ and ESX show similar completeness ratio, i.e., over 90%, in all scenarios, apart from the case where $k = 3$ and $\theta = 0.1$ (i.e., the alternative paths are very dissimilar to each other). In this case, the completeness ration

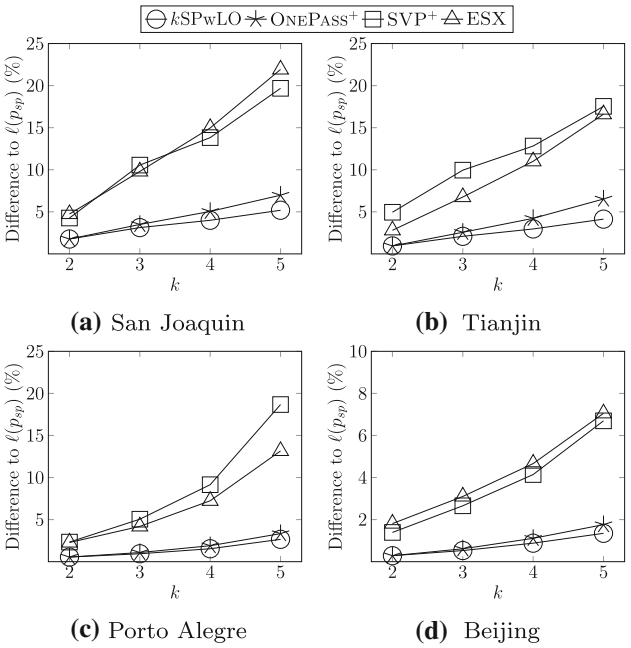


Fig. 18 Average length difference to $\ell(p_{sp})$ of heuristic algorithms varying requested paths k ($\theta = 0.5$)

of SVP$^+$ and ESX is clearly lower than that of the exact solution and ONEPASS$^+$. Nevertheless, the completeness ratio of ESX is above 80% is all cases and is clearly higher than the completeness ratio of SVP$^+$.

Scalability From previous experiments, it is clear that neither the exact algorithm MULTIPASS nor the heuristic algorithm ONEPASS$^+$ are scalable. However, the situation is different for SVP$^+$ and ESX. To this end, in what follows we evaluate the efficiency of SVP$^+$ and ESX for large values of k on large road networks.

In Fig. 19, we analyze the runtime performance of SVP$^+$ and ESX for large values of k and large road networks setting $\theta = 0.5$. For $k \leq 10$, we observe that the runtime of ESX and the runtime of SVP$^+$ are similar, with ESX being slightly faster. For $k > 10$ and the road networks of Milan and Florida, we observe that ESX is clearly faster the SVP$^+$. However, observe that for $k = 20$ and the road networks of Chicago and Colorado SVP$^+$ is faster than ESX. This result connected to the completeness of SVP$^+$ that we analyze below.

In Fig. 20, we report the average length difference to $\ell(p_{sp})$ of the result paths computed by SVP$^+$ and ESX. For the road networks of Milan and Chicago, we observe that the two algorithms recommend alternative paths of similar length. More specifically, for $k \leq 10$ the two algorithms compute alternative paths of similar length with the ones returned by SVP$^+$ being slightly shorter, while for $k > 10$, ESX clearly computes shorter alternative paths. On the sparser road networks of Colorado and Florida, for

Table 2 Completeness ratio (%) of kSPwLO algorithms for different values of k and θ

Road net.	k	θ	kSPwLO	OnePass$^+$	SVP$^+$	ESX
San Joaquin	2	0.5	100	100	100	100
	3	0.5	100	99.8	99.6	99.5
	4	0.5	99.88	99	96.5	97.8
	5	0.5	99.67	98.3	94.1	96.9
	3	0.9	100	100	99.9	100
	3	0.7	100	99.9	99.7	99.8
	3	0.3	99.1	98.5	92.2	96.5
	3	0.1	92.1	89.6	55.3	81.7
Tianjin	2	0.5	99.9	99.9	99.8	99.9
	3	0.5	99.9	99.8	99.8	99.5
	4	0.5	99.8	98.6	99.4	98.6
	5	0.5	99.7	97.7	98.6	96.4
	3	0.9	100	100	100	100
	3	0.7	100	99.7	100	100
	3	0.3	99.4	99.2	97.7	97.8
	3	0.1	95	93.5	74	87.5
Porto Alegre	2	0.5	100	100	100	100
	3	0.5	100	99.9	99.9	99.2
	4	0.5	100	99.5	99.5	98.3
	5	0.5	100	98.6	98.6	96.8
	3	0.9	100	100	100	100
	3	0.7	100	100	100	99.9
	3	0.3	100	99.7	91.2	97.6
	3	0.1	98.5	97.5	53.2	90.9
Beijing	2	0.5	100	100	100	100
	3	0.5	100	99.9	99.5	99.6
	4	0.5	100	98.7	98.5	99.4
	5	0.5	100	97.4	98.5	98.3
	3	0.9	100	100	100	100
	3	0.7	100	100	100	99.9
	3	0.3	99.7	99.6	98.2	98.6
	3	0.1	95.4	94.6	85.2	88

$k = 5$, ESX and SVP$^+$ compute alternative paths of similar length, with the ones returned by ESX being slightly shorter. For $k > 8$ though, the alternative paths computed by ESX are clearly much shorter than the ones computed by SVP$^+$.

Finally, Table 3 reports on the completeness of the SVP$^+$ and ESX results. For the road networks of Milan and Chicago, we observe that while ESX demonstrates a higher completeness ratio than SVP$^+$, the completeness ratio of both algorithms is in most cases over 90%. For $\theta < 0.5$ though, ESX is clearly better than SVP$^+$, while both algorithms struggle to compute a complete result for $\theta = 0.1$. For the road networks of Colorado and Florida, the superiority of ESX is even more apparent. While ESX demonstrates a completeness ratio of over 90% for $\theta > 0.1$, SVP$^+$ demonstrates a comparable ratio only for $k = 5$ and $\theta > 0.5$. Especially for $\theta < 0.3$ the completeness ratio of SVP$^+$ is below 10%. Similar to the results for Milan and Chicago though, both algorithms struggle to compute a complete result for $\theta = 0.1$.

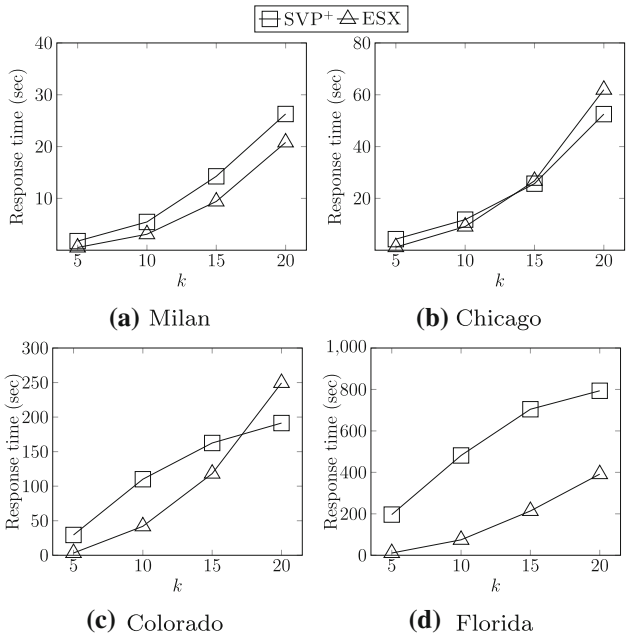


Fig. 19 Performance of SVP$^+$ and ESX varying requested paths k ($\theta = 0.5$)

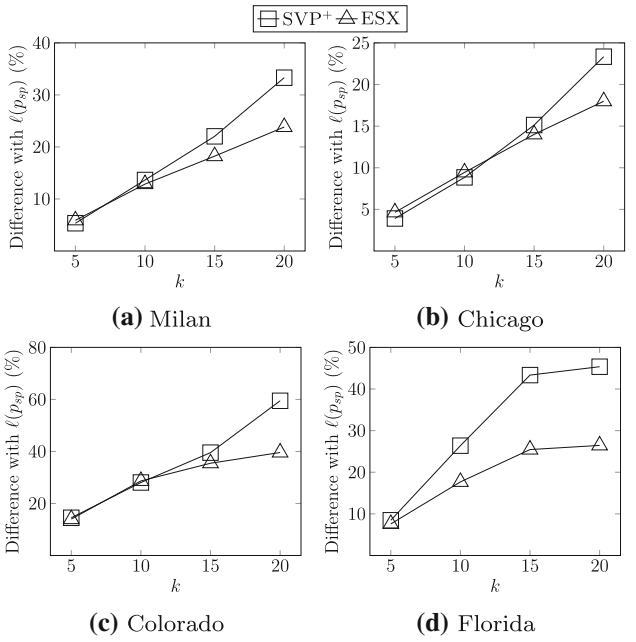


Fig. 20 Average length difference to $\ell(p_{sp})$ of SVP$^+$ and ESX varying requested paths k ($\theta = 0.5$)

Table 3 Completeness ratio (%) of SVP$^+$ and ESX for different values of k and θ

Milan				Chicago			
k	θ	SVP$^+$	ESX	k	θ	SVP$^+$	ESX
5	0.5	100	98.6	5	0.5	100	98.7
10	0.5	99	95.7	10	0.5	98.8	95.3
15	0.5	91.9	94.2	15	0.5	98.2	93.1
20	0.5	72.8	92.6	20	0.5	93.1	91.4
10	0.9	100	99.7	10	0.9	100	99.6
10	0.7	100	98.8	10	0.7	100	99.1
10	0.3	36.5	86.4	10	0.3	69.9	86.9
10	0.1	0	37	0	0.1	0	50.7
Colorado				Florida			
k	θ	SVP$^+$	ESX	k	θ	SVP$^+$	ESX
5	0.5	99.4	98.6	5	0.5	94.8	99.1
10	0.5	78.6	96.6	10	0.5	57.1	97.8
15	0.5	43.7	95.2	15	0.5	20.1	97
20	0.5	22.5	94.5	20	0.5	3.8	96.7
10	0.9	100	99.5	10	0.9	100	99.8
10	0.7	99.9	98.8	10	0.7	100	99.4
10	0.3	8.8	91.1	10	0.3	1.1	94.4
10	0.1	0	25.2	10	0.1	0	62

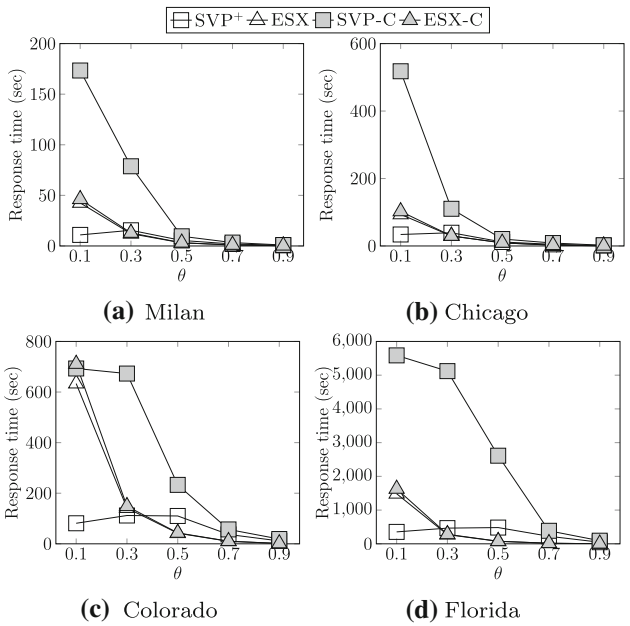


Fig. 21 Performance of SVP$^+$, ESX, SVP- C and ESX- C varying similarity threshold θ ($k = 10$)

9.2.3 Completeness-oriented heuristic algorithms

Runtime In Fig. 21, we report the runtime of SVP- C and ESX- C, varying the similarity threshold θ with $k = 10$, and we compare them to their performance-oriented counterparts, i.e., SVP$^+$ and ESX. In the road networks of Milan and Chicago for $\theta \geq 0.5$ and in the road networks of Colorado and Florida $\theta \geq 0.7$, we observe that all algorithms have similar runtime. For kSPwLO queries where SVP$^+$ and ESX return a complete result, SVP- C and ESX- C do not need to invoke function `Complete_kSPwLO` . Hence, the runtime of SVP- C and ESX- C is almost the same with SVP$^+$ and ESX, respectively. For smaller values of θ though, we observe in all networks that ESX- C clearly outperforms SVP- C. Also, the difference between the runtime of SVP$^+$ and SVP- C is much greater than the difference between the runtime of ESX and ESX- C. As the completeness ratio of SVP$^+$ is fairly low for small values of θ, function `Complete_kSPwLO` has to be invoked many times to relax θ and compute a complete result.

To get additional insights in the performance of our completeness-oriented algorithms, in Fig. 22 we report the number of paths in the candidate set of SVP- C and ESX- C for all datasets, for the case where both SVP- C and ESX- C demonstrated the lowest completeness ratio, i.e., $k = 10$ and $\theta = 10\%$. We observe that the P_{cand} used by SVP- C is around three orders of magnitude larger than the P_{cand} used by ESX- C. This difference justifies the difference in the runtime of the two algorithms, since ESX- C invokes `Complete_kSPwLO` using a much smaller P_{cand} as input.

Pairwise Similarity. Figure 23 reports the average relaxed similarity threshold θ_{min} for which algorithms SVP- C and ESX- C return a complete result, varying the initial similar-

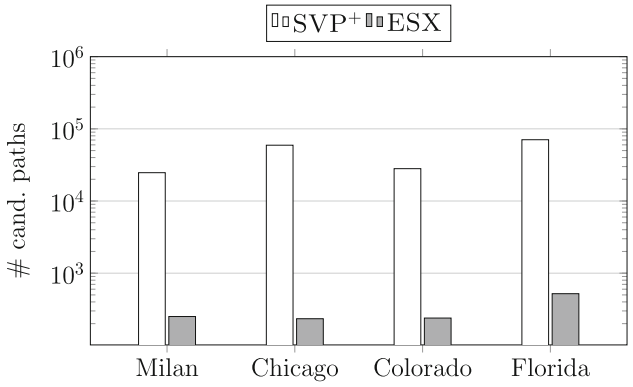


Fig. 22 Size of candidate sets constructed by ESX- C and SVP- C for $k = 10$ and $\theta = 10\%$

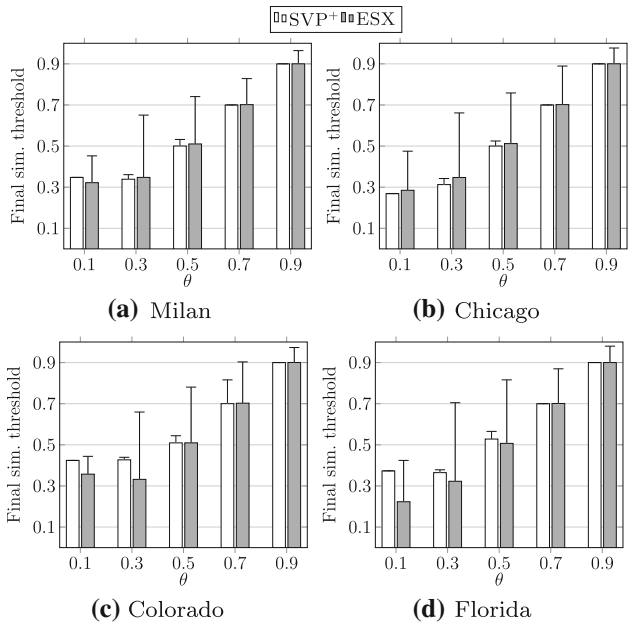


Fig. 23 Updated similarity threshold for ESX- C and SVP- C varying similarity threshold θ ($k = 10$)

ity threshold θ. The line above each bar indicates the average θ_{min} only for queries that the initial θ resulted in an incomplete result. In all road networks, for $\theta > 0.5$, the relaxed similarity threshold θ_{min} computed by SVP- C and ESX- C is almost the same. For $\theta \leq 0.5$, SVP- C performs slightly better in the road networks of Chicago and Milan, while the opposite is true in the road networks of Colorado and Florida. However, this result is influenced directly by the completeness ratios of SVP$^+$ and ESX. On the contrary, for queries that the initial θ leads to an incomplete result, SVP- C finds a complete result for a much smaller θ_{min} than ESX- C. These observations hint that the P_{cand} constructed by SVP- C is not only larger but also more diverse than the one constructed by ESX- C.

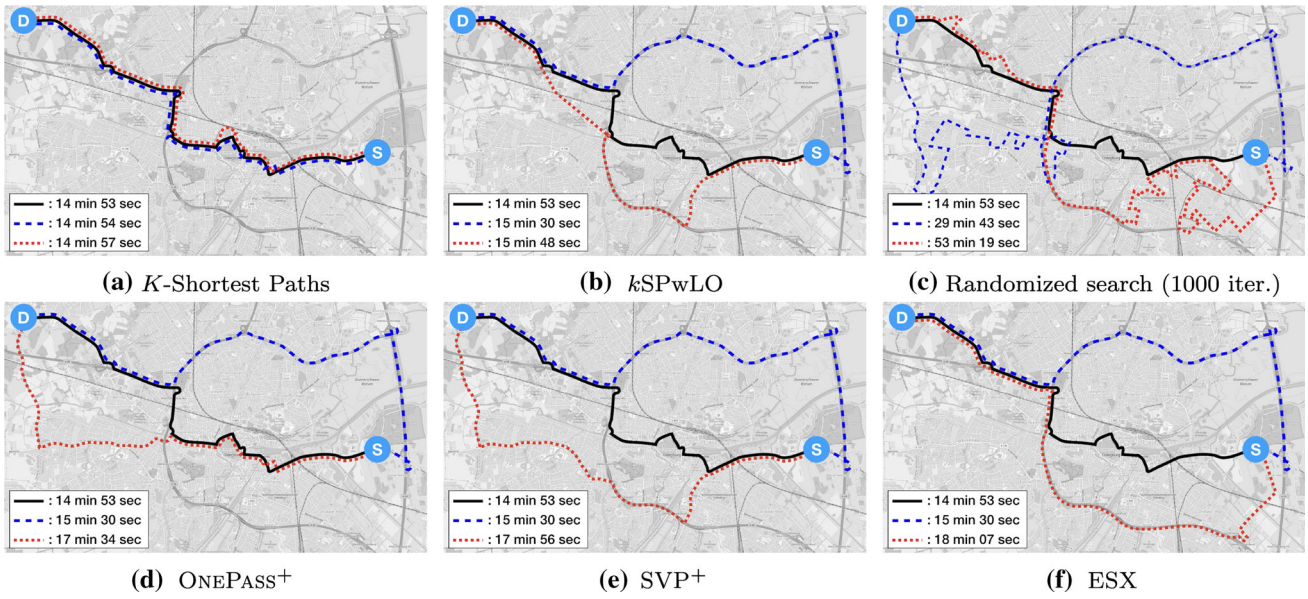
9.3 Summary of findings

To sum up, our experimental analysis concludes into three key findings. First, our tests for the exact computation of kSPwLO are in line with the theoretical analysis on the optimality of MultiPass (cf. Sect. 5.2). MultiPass is the fastest exact algorithm outperforming both OnePass and FindKSPD [22] by a significant margin. However, MultiPass is practical only for $k = 2$. For $k > 2$, MultiPass is practical only on small road networks, as its response time even for mid-sized road networks is prohibitively high.

Two out of our three performance-oriented heuristic algorithms manage to address this scalability issue. More specifically, despite being an improvement over MultiPass in terms of runtime and computing a result set that is close to the exact solution, OnePass$^+$ does not scale on large road networks. On the contrary, SVP$^+$ and ESX are both significantly faster than OnePass$^+$ and able to scale, but on average they compute slightly longer alternative paths. Overall, ESX is the best choice as it is the fastest performance-oriented heuristic algorithm while recommending more and shorter alternative paths than SVP$^+$.

Finally, in applications where a complete result is required, i.e., exactly k alternative paths must be retrieved, we distinguish between two cases. If the response time matters more than the quality of the complete result set, ESX- C is the algorithm of choice as it inherits the performance advantage of ESX. However, if result quality is more important, SVP- C is preferred as its candidate set enables the `Complete_kSPwLO` to find a result set of more dissimilar paths than ESX- C.

To provide additional insights on how our approach could be used in practice, in Fig. 24 we visualize different sets of 3 alternative paths between two locations in the city of Oldenburg, setting the similarity threshold $\theta = 50\%$. Figure 24a shows the K-shortest paths, which clearly have little practical value since they are too similar to each other. On the contrary, both the kSPwLO shown in Fig. 24b and the results of our heuristic algorithms shown in Fig. 24d–f offer much more attractive alternative paths. As the heuristic algorithms compute fairly similar results to the kSPwLO, in applications where response time is important, we expect these results to be satisfactory. We also visualize the result of a randomized search to examine how better or worse the alternative paths would be if they were to be selected at random. We first add the shortest path into the result set, and then, we execute $k-1$ random walks in order to obtain k paths in total. We repeat this process multiple times and keep the best result in terms of shortness. Figure 24c shows that the alternative paths obtained using this randomization are too long and contain too many needless detours, even after 1000 iterations to improve the result quality.



(a) K-Shortest Paths **(b)** kSPwLO **(c)** Randomized search (1000 iter.)

(d) OnePass$^+$ **(e)** SVP$^+$ **(f)** ESX

Fig. 24 Example results of exact and approximate algorithms for the kSPwLO in Oldenburg ($k = 3$, $\theta = 50\%$)

10 Conclusions

In this paper, we studied the problem of alternative routing on road networks. Our goal was to recommend k paths that are sufficiently dissimilar to each other and as short as possible. To this end, we proposed kSPwLO, which minimizes the length of each individual path in the result set and we showed that kSPwLO is weakly NP-hard. For answering kSPwLO queries, we presented two exact algorithms, three performance-oriented heuristic algorithms and two completeness-oriented heuristic algorithms. Through an extensive experimental evaluation, we demonstrated the performance of all algorithms in terms of runtime and result quality, and we identified use-cases each algorithm is useful for and trade-offs each algorithm comes with.

In the future, we plan to extend the definition of alternative routing by considering multiple criteria and constraints to match the requirements of a wider range of applications. Moreover, we plan to adapt our algorithms for time-dependent and dynamic traffic-aware road networks. Finally, we plan to investigate the computation of dissimilar paths on different types of networks such as social networks and web graphs.

Acknowledgements Open Access funding provided by Projekt DEAL. This research has been partially supported by Grant No. GR 4497/2 of the Deutsche Forschungsgemeinschaft(DFG)

References

1. 9th DIMACS Implementation Challenge. http://www.dis.uniroma1.it/challenge9/
2. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Alternative routes in road networks. J. Exp. Algorithmics **18**, 1–17 (2013)
3. Akgun, V., Erkut, E., Batta, R.: On finding dissimilar paths. Eur. J. Oper. Res. **121**(2), 232–246 (2000)
4. Bader, R., Dees, J., Geisberger, R., Sanders, P.: Alternative route graphs in road networks. In: Proceedings of the 1st International ICST TAPAS Conference, pp. 21–32 (2011)
5. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica **6**(2), 153–180 (2002)
6. Ceikute, V., Jensen, C.S.: Vehicle routing with user-generated trajectory data. In: Proceedings of the 16th IEEE MDM Conference, pp. 14–23 (2015)
7. Chen, Z., Shen, H.T., Zhou, X.: Discovering popular routes from trajectories. In: Proceedings of the 27th IEEE ICDE, pp. 900–911 (2011)
8. Choice Routing. Cambridge Vehicle Information Technology Ltd. (2005). http://www.camvit.com
9. Chondrogiannis, T., Bouros, P., Gamper, J., Leser, U.: Alternative Routing: K-shortest paths with limited overlap. In: Proceedings of the 23rd ACM SIGSPATIAL GIS Conference, pp. 68:1–68:4 (2015)
10. Chondrogiannis, T., Bouros, P., Gamper, J., Leser, U.: Exact and approximate algorithms for finding k-shortest paths with limited overlap. In: Proceedings of the 20th EDBT Conference, pp. 414–425 (2017)

11. Chondrogiannis, T., Bouros, P., Gamper, J., Leser, U., Blumenthal, D.B.: Finding *k*-dissimilar paths with minimum collective length. In: Proceedings of the 26rd ACM SIGSPATIAL GIS Conference, pp. 404–407 (2018)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press (2001). https://doi.org/10.5555/500824
13. Delling, D., Dorothea, W.: Pareto Paths with SHARC. In: Proceedings of the 8th SEA, pp. 125–136 (2009)
14. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
15. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE TSSC **4**(2), 100–107 (1968)
16. Hershberger, J., Maxel, M., Suri, S.: Finding the K shortest simple paths: a new algorithm and its implementation. ACM Trans. Algorithms **3**(4), 26–36 (2007)
17. Jeong, J., Jeong, H., Lee, E., Oh, T., Du, D.H.C.: Saint: self-adaptive interactive navigation tool for cloud-based vehicular traffic optimization. IEEE Trans. Veh. Technol. **65**(6), 4053–4067 (2016)
18. Jeong, Y.J., Kim, T.J., Park, C.H., Kim, D.K.: A dissimilar alternative paths-search algorithm for navigation services: a heuristic approach. KSCE J. Civil Eng. **14**(1), 41–49 (2009)
19. Karduni, A., Kermanshah, A., Derrible, S.: A protocol to convert spatial polyline data to network formats and applications to world urban road networks. Sci. Data **3**, 160046 (2016)
20. Kriegel, H.P., Renz, M., Schubert, M.: Route skyline queries: a multi-preference path planning approach. In: Proceedings of the 26th IEEE ICDE, pp. 261–272 (2010)
21. Li, P.H., Yiu, M.L., Mouratidis, K.: Discovering historic traffic-tolerant paths in road networks. GeoInformatica **21**(1), 1–32 (2017)
22. Liu, H., Jin, C., Yang, B., Zhou, A.: Finding top-k shortest paths with diversity. IEEE TKDE **99**, 1 (2017)
23. Martins, E.Q.V., Pascoal, M.M.B.: A new implementation of Yen's ranking loopless paths algorithm. 4OR **1**(2), 121–133 (2003)
24. Moghadam, K.R., Nguyen, Q., Krishnamachari, B., Demiryurek, U.: Traffic matrix estimation from road sensor data: A case study. In: Proceedings of the 23rd ACM SIGSPATIAL GIS Conference, pp. 65:1–65:4 (2015)
25. Mouratidis, K., Lin, Y., lu Yiu, M.: Preference queries in large multi-cost transportation networks. In: Proceedings of the 26th IEEE ICDE, pp. 533–544 (2010)
26. Nellore, K., Hancke, G.P.: A survey on urban traffic management system using wireless sensor networks. Sensors **16**(2), 157 (2016)
27. Raith, A., Wang, J.Y.T., Ehrgott, M., Mitchell, S.A.: Solving multi-objective traffic assignment. Ann. Oper. Res. **222**(1), 483–516 (2014)
28. Roberts, B., Kroese, D.P.: Estimating the number of s-t paths in a graph. J. Gr. Algorithms Appl. **11**(1), 195–214 (2007)
29. Saber, T., Ventresque, A., Murphy, J.: Rothar: Real-time on-line traffic assignment with load estimation. In: Proceedings of the 17th IEEE/ACM DS-RT Symposium, pp. 79–86 (2013)
30. Sacharidis, D., Bouros, P., Chondrogiannis, T.: Finding the most preferred path. In: Proceedings of the 25th ACM SIGSPATIAL GIS Conference, pp. 5:1–5:10 (2017)
31. Samet, H., Sankaranarayanan, J., Alborzi, H.: Scalable network distance browsing in spatial databases. In: Proceedings of the 2008 ACM SIGMOD Conference, pp. 43–54 (2008)
32. Sommer, C.: Shortest-path queries in static networks. ACM Comput. Surv. **46**(4), 1–31 (2014)
33. Valiant, L.: The complexity of enumeration and reliability problems. SIAM J. Comput. **8**(3), 410–421 (1979)
34. Wei, L.Y., Zheng, Y., Peng, W.C.: Constructing popular routes from uncertain trajectories. In: Proceedings of the 18th ACM SIGKDD Conference, pp. 195–203 (2012)
35. Xie, K., Deng, K., Shang, S., Zhou, X., Zheng, K.: Finding alternative shortest paths in spatial networks. ACM TODS **37**(4), 29:1–29:31 (2012)
36. Xu, J., Guo, L., Ding, Z., Sun, X., Liu, C.: Traffic aware route planning in dynamic road networks. In: Proceedings of the 17th International DASFAA Conference, pp. 576–591 (2012)
37. Yen, J.Y.: Finding the K shortest loopless paths in a network. Manag. Sci. **17**(11), 712–716 (1971)
38. Yuan, J., Zheng, Y., Zhang, C., Xie, W.: T-Drive : Driving directions based on taxi trajectories. In: Proceedings of the 18th ACM SIGSPATIAL GIS Conference, pp. 99–108 (2010)
39. Zhan, X., Zheng, Y., Yi, X., Ukkusuri, S.V.: Citywide traffic volume estimation using trajectory data. IEEE TKDE **29**(2), 272–285 (2017)