Erschienen in: Advances in Database Technology EDBT 2017 : 20th International Conference on Extending Database Technology ; Proceedings / Markl, Volker et al. (Hrsg.). - Konstanz : OpenProceedings.org, 2017. - S. 414-425. - eISSN 2367-2005. - ISBN 978-3-89318-073-8
http://dx.doi.org/10.5441/002/edbt.2017.37

Exact and Approximate Algorithms for Finding k-Shortest Paths with Limited Overlap

Theodoros Chondrogiannis
Free University of Bozen-Bolzano
tchond@inf.unibz.it

Panagiotis Bouros
Aarhus University
pbour@cs.au.dk

Johann Gamper
Free University of Bozen-Bolzano
gamper@inf.unibz.it

Ulf Leser
Humboldt-Universität zu Berlin
leser@informatik.hu-berlin.de

ABSTRACT

Shortest path computation is a fundamental problem in road networks with various applications in research and industry. However, returning only the shortest path is often not satisfying. Users might also be interested in alternative paths that are slightly longer but have other desired properties, e.g., less frequent traffic congestion.

In this paper, we study alternative routing and, in particular, the k-Shortest Paths with Limited Overlap (k-SPwLO) query, which aims at computing paths that are (a) sufficiently dissimilar to each other, and (b) as short as possible. First, we propose MultiPass, an exact algorithm which traverses the network $k-1$ times and employs two pruning criteria to reduce the number of paths that have to be examined. To achieve better performance and scalability, we also propose two approximate algorithms that trade accuracy for efficiency. OnePass$^+$ employs the same pruning criteria as MultiPass, but traverses the network only once. Therefore, some paths might be lost that otherwise would be part of the solution. ESX computes alternative paths by incrementally removing edges from the road network and running shortest path queries on the updated network. An extensive experimental analysis on real road networks shows that: (a) MultiPass outperforms state-of-the-art exact algorithms for computing k-SPwLO queries, (b) OnePass$^+$ runs significantly faster than MultiPass and its result is close to the exact solution, and (c) ESX is faster than OnePass$^+$ (though slightly less accurate) and scales for large road networks and large values of k.

CCS Concepts

•**Information systems** → **Geographic information systems;** Database query processing;

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 the industry. Traditionally, the shortest path problem is addressed by Dijkstra's algorithm [9]. Additionally, a plethora of pre-processing based methods have been proposed that answer shortest path queries in almost constant time, even for continental sized networks [7, 11, 21, 25].

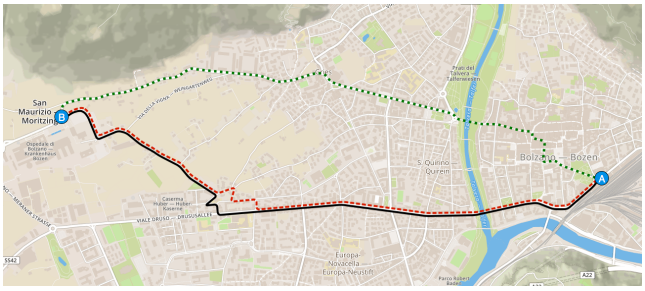


Figure 1: Motivational example

However, in many real-world scenarios, determining solely the shortest path is not enough. Most commercial route planning applications and navigation systems offer alternatives that might be longer than the shortest path but have other desirable properties (e.g., lower fuel consumption), leaving the final decision to the user. Alternative routes are also very useful for the transportation of goods using a fleet of vehicles, i.e., transportation of humanitarian aid through unsafe regions. By distributing the load into vehicles that follow different routes, the probability that at least some of the goods will arrive at the destination safely can be increased. Another interesting scenario 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, apart from the shortest, alternative paths that are sufficiently dissimilar to each other.

A first take on providing alternative routes with no prior information is to solve the K-shortest paths problem [10, 15, 24]. In most cases, though, the returned paths share large stretches, and therefore, they are of little practical value to the user. Consider the scenario illustrated in Figure 1, which shows three different paths from location A to B representing the central train station and the hospital in the city of Bolzano, respectively. The solid/black line indicates the shortest path from A to B while the dashed/red line indicates the next path by length; notice how similar these two paths are. On the other hand, the green/dotted line indicates a third path, which is clearly longer than the other two but significantly different from the shortest path. In practice, the paths cover very distant

parts of the city's road network. In many application scenarios, the green/dotted path would be considered as a better alternative to the shortest path, compared to the dashed/red path.

Existing literature has approached alternative routing from different perspectives. Notable works include methods which aim at computing alternative routes either by incrementally building a set of dissimilar paths [12] or by employing edge penalties [2]. The proposed methods, though, typically give no guarantees regarding the length of the alternative paths. Other approaches [1, 3, 5] first generate a large number of candidates and then, in a post-processing step, consider a number of constraints and criteria in order to determine the final alternative paths. However, in these works alternatives are defined based solely on their individual similarity to the shortest path, which results in alternative paths that are very similar to each other and, hence, of limited interest to the user.

Contributions. In this paper, we focus on the problem of finding *k-Shortest Paths with Limited Overlap* (k-SPwLO), previously introduced in [6]. A k-SPwLO query aims at computing paths that are (a) *sufficiently dissimilar to each other (based on a user-specified similarity threshold)*, and (b) *as short as possible*. In [6], we presented the OnePass algorithm for processing k-SPwLO queries. The algorithm outperforms a baseline solution which enumerates paths in increasing length order, but, in reality, OnePass is not practical even for mid-sized road networks. To this end, we propose MultiPass, an exact algorithm which extends and improves OnePass by employing an additional pruning criterion. In contrast to OnePass, which traverses the road network once and expands only those paths that qualify the similarity constraint, MultiPass traverses the network $k-1$ times, but examines and expands only the most *promising* paths. Any path that cannot lead to a solution is pruned. Our experimental analysis shows that MultiPass always outperforms OnePass, and, in most cases, by a large margin.

Despite its significant performance advantage over OnePass, also MultiPass cannot scale in practice for large road networks, a fact that is backed by our extensive experimental evaluation. In this spirit, we propose two approximate methods that trade result quality for efficiency. Our first approximate algorithm, OnePass$^+$, employs the pruning power of MultiPass, but traverses the road network only once, similar to OnePass. Thereby, OnePass$^+$ may prune some partial paths that, in a subsequent iteration, could become part of an alternative path. Our second approximate algorithm, ESX, computes alternative paths by incrementally removing edges from the road network that belong to previously recommended paths, and running shortest path queries on the updated network. Essentially, ESX reduces the search for alternative paths to a set of shortest path queries which require much less time to be processed. In our extensive experimental evaluation, we show that the approximate algorithm OnePass$^+$ is significantly faster than the exact algorithm MultiPass, while recommending alternative paths that are almost as short as the alternatives in the exact k-SPwLO set. We also show that ESX is the fastest algorithm and is scalable even for large road networks (i.e., one million nodes) and large values of k.

Outline. The rest of the paper is organized as follows. Section 2 briefly discusses the related work on providing alternative paths. In Section 3, we formally define the k-SPwLO problem and revisit our evaluation methodology from [6]. In Section 4, we present MultiPass, a novel exact algorithm for processing k-SPwLO queries, and conduct a preliminary experimental analysis comparing MultiPass to OnePass. Next in Section 5, we investigate the approximate evaluation of k-SPwLO. We first discuss a baseline method SVP$^+$, based on existing literature and then propose our approximate algorithms OnePass$^+$ and ESX. The results of our detailed experimental evaluation are reported in Section 6. Finally, Section 7 concludes the paper and points to future work.

2. RELATED WORK

A common approach for alternative routing is to first compute a large set of candidate paths, then examine the candidate paths with respect to a number of constraints (e.g., their length or the nodes they cross) and determine the final result set. In [5], the authors build two shortest path trees, one from the source and one from the target, and then look for paths that appear in both trees simultaneously, termed *plateaus*. This approach was revisited and formally defined in [3] (where the concept of alternative graphs has been introduced with the same functionality as the plateaus) and further improved in [17]. Abraham et al. [1] introduced the notion of *single-via paths*. The method runs Dijkstra's algorithm two times, once from the source s and once from the target t while reversing the edges of the road network. Then, for each node n apart from s and t, the algorithm constructs a single-via path by concatenating the shortest path from s to n and the shortest path from n to t. The algorithm evaluates each (simple) single-via path by employing a set of user-defined constraints, i.e., length, local optimality and stretch, and rejects all single-via paths that violate these constraints. Compared to our k-SPwLO problem, none of the aforementioned methods tackles the problem of computing multiple alternative paths that are dissimilar to each other; in contrast, the similarity only to the shortest path is considered.

Penalty-based methods generate a set of paths that are dissimilar to the shortest path by adding a penalty on the weights of the edges of the shortest path. For example, Akgun et. al. [2] propose a method which doubles the weight of each edge that lies on the shortest path. The alternative paths are computed by repeatedly running Dijkstra's algorithm on the input road network, each time with the updated weights. A similar approach is adopted in [14], where the penalty is computed in terms of both the path overlap and the total turning cost, i.e., how many times the user would have to switch between roads when following a path. The main shortcoming of penalty-based methods is that there is no intuition behind the value of the penalty applied before each subsequent iteration. In general, using a large penalty value would result in diverse but possibly very long alternative paths. On the other hand, using a small penalty value would require the algorithm to perform more iterations in order to find the desired result. Even so, penalty-based methods cannot provide a formal result set. Our last approximate algorithm ESX can also be viewed as a penalty based method where the penalty added to the weight of selected edges is $+\infty$.

To the best of our knowledge, the problem tackled in [12] is the most similar to our k-SPwLO problem. The authors devise a solution which extends Yen's algorithm [24] to produce paths that qualify a similarity constraint. In particular, given a source s and a target t, starting from the shortest path, the algorithm produces a set of candidate paths by modifying the previously found path. Among the candidate paths, the algorithm chooses the one that is most dissimilar to the previously found path and continues until a sufficiently dissimilar path is found. Apparently, the algorithm does not examine paths in length order but only based on their similarity. Thus, it does not compute alternative paths that are as short as possible, but only dissimilar. Naturally, a user finds more value in paths that are also as short as possible.

Xie et. al. [23] define alternative shortest paths using edge avoidance. Given the shortest path $p(s \rightarrow t)$ and an edge e on p, the alternative path is the shortest path from s to t which avoids edge e. To compute alternative paths, they build upon the concept of distance

oracles [20] and distance sensitivity oracles [4] and propose *iSPQF*, a quadtree-based spatial data structure inspired by [19]. Compared to our work, iSQPF computes only one alternative path instead of a set. Moreover, the use-case is different as Xie et al. find alternative paths by explicitly avoiding forbidden edges, while k-SPwLO considers the similarity between paths to propose alternative paths.

Finally, the task of alternative routing can be based on the pareto-optimal paths for multi-criteria networks [8, 13, 16]. A path p is part of the pareto-optimal set (or the route skyline) P if p is not dominated by another path $p' \in P$. Hence, path p dominates p' iff p is not worse than p' in all criteria/dimensions of the network (e.g., distance, travel time, gas consumption) and strictly better than p' in at least one of those criteria. Our definition of alternative routing, i.e., the k-SPwLO query, is not a multi-criteria optimization problem; the paths recommended by k-SPwLO cannot be obtained by first computing the pareto-optimal path set.

3. BACKGROUND

Let $G{=}(N, E)$ be a *directed weighted graph* representing a *road network* with set of nodes N and set of edges $E \subseteq N{\times}N$. The nodes of G represent road intersections and the edges represent road segments. Each edge $(n_x, n_y) \in E$ has an assigned positive weight w_{xy}, which captures the cost of moving from node n_x to n_y, e.g., travel time or distance. A (simple) *path* $p(s \rightarrow t)$ from a source node s to a target node t (or just p, if s and t are clear from the context) is a connected and cycle-free sequence of edges $\langle (s, n_x), \ldots, (n_y, t)\rangle$. The length $\ell(p)$ of a path p equals the sum of the weights of all contained edges. The *shortest path* between two nodes, $p_0(s{\rightarrow}t)$, is the path that has the shortest length among all paths connecting s to t. The length of the shortest path is also termed the *(network) distance* between s and t, i.e., $d(s,t){=}\ell(p_0(s{\rightarrow}t))$.

Problem Definition. In [6], we introduced the problem of k-*Shortest Paths with Limited Overlap* (k-SPwLO) in order to recommend alternative paths a user may take to reach her destination. In particular, let P be a set of paths from a node s to another node t on a road network G. A path $p'(s{\rightarrow}t)$ is called *alternative* to set P if p' is sufficiently dissimilar to every path $p \in P$. More formally, the similarity of p' to p is determined by their overlap ratio:

$$Sim(p', p){=}\frac{\sum_{(n_x,n_y)\in p'\cap p} w_{xy}}{\ell(p)}, \qquad (1)$$

where $p' \cap p$ denotes the set of edges shared by p' and p. Then, given a similarity threshold θ, path p' is alternative to set P *iff* $Sim(p', p) \leq \theta, \forall p \in P$ holds.

Now, given a source node s and a target node t, a k-SPwLO (s, t, θ, k) query returns a set of k paths from s to t, sorted in increasing length order, such that:

(a) the shortest path $p_0(s{\rightarrow}t)$ is always included,

(b) all k paths are pairwise dissimilar with respect to the similarity threshold θ, and

(c) all k paths are as short as possible.

Consider the road network in Figure 2. The shortest path from s to t is $p_0{=}\langle (s, n_3), (n_3, n_5), (n_5, t)\rangle$ with length $\ell(p_0){=}8$. Assume that P contains only the shortest path, i.e., $P{=}\{p_0\}$ and consider paths $p_1{=}\langle (s, n_3), (n_3, n_5), (n_5, n_4), (n_4, t)\rangle$ and $p_2{=}\langle (s, n_3), (n_3, n_4), (n_4, t)\rangle$ with $\ell(p_1){=}9$ and $\ell(p_2){=}10$, respectively, as alternatives to P. Path p_1 shares edges (s, n_3) and (n_3, n_5) with p_0, which gives

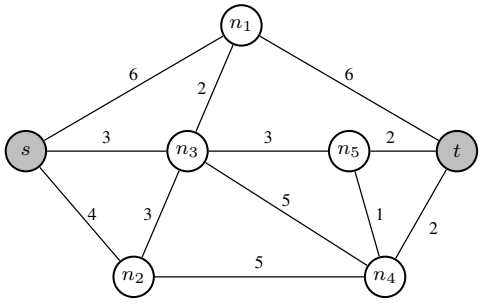


Figure 2: Running example.

$Sim(p_1, p_0){=}(w_{s,3} + w_{3,5})/\ell(p_0) = 6/8 = 0.75$, whereas $Sim(p_2, p_0){=}w_{s,3}/\ell(p_0) = 3/8 = 0.38$. Assuming a similarity threshold $\theta{=}0.5$, only p_2 is alternative to P.

Note that the asymmetric similarity metric of Equation 1 allows us to exclude needlessly long paths. Following up on our previous example, consider the shortest path p_0 and the paths $p_3{=}\langle (s, n_3), (n_3, n_4), (n_4, t)\rangle$ and $p_4{=}\langle (s, n_3), (n_3, n_2), (n_2, n_4), (n_4, t)\rangle$ with $\ell(p_3){=}10$ and $\ell(p_4){=}13$, respectively. The use of a symmetric similarity metric such as the Jaccard distance would indicate that p_4 is less similar to p_0 than p_3, although the shared length of both p_3 and p_4 with p_0 is the same. With the asymmetric definition of Equation 1 we avoid such cases. Furthermore, the pairwise dissimilarity is guaranteed as long as $\ell(p') \geq \ell(p)$ (proof excluded due to lack of space).

Evaluating k-SPwLO. A naïve approach for evaluating k-SPwLO queries is to iterate over all paths connecting s to t and compute their pairwise similarity. Naturally, such a solution is not practical. A potential improvement is to examine paths in increasing order of their length, which allows us not to examine all possible paths $p(s \rightarrow t)$. This idea was captured by the baseline method in [6], but the computation cost is still prohibitively high.

To further reduce the search space, we first introduced a pruning criterion in [6] based on the following simple observation. Let $p(s{\rightarrow}n)$ be a path connecting source s to a node n, and $p_i(s{\rightarrow}t) \in P_{LO}$ be an already recommended path. Assume that p is extended to reach target t, resulting in path $p'(s{\rightarrow}t)$. As p' contains all edges shared by p and p_i, its similarity to p_i is at least equal to the similarity of path p, i.e., $Sim(p', p_i) \geq Sim(p, p_i)$. Hence, given a threshold θ, if there exists $p_i \in P_{LO}$ such that $Sim(p, p_i) \geq \theta$, path p can be safely discarded. This observation is formally captured by the following lemma:

LEMMA 1. *Let P_{LO} be the set of already recommended paths. If p is an alternative path to P_{LO} with respect to a threshold θ,then $Sim(p', p_i) \leq \theta$ holds for every subpath p' of p and all $p_i \in P_{LO}$.*

We used this Lemma 1 as pruning criterion in the OnePass algorithm. The algorithm traverses the road network, expanding every path from the source node s that satisfies Lemma 1. OnePass employs a min priority queue in order to examine paths in increasing order of their length. Each time a new path is recommended, i.e., added to the result set P_{LO}, an update procedure takes place for all remaining incomplete paths $p(s \rightarrow n)$ in the priority queue. The algorithm terminates when either k paths are added to the result set or all paths from s to t qualifying Lemma 1 are examined.

OnePass can be viewed as an extension of *Fox's algorithm* [10] for computing the K-shortest paths. Fox's algorithm traverses the road network expanding every path from source node s. At each iteration, the algorithm expands up to K nodes, allowing each node

to be expanded up to K times. It terminates when the target node has been expanded K times. The time complexity of Fox's algorithm is $\mathcal{O}(|E| + K \cdot |N| \cdot \log |N|)$. In contrast to Fox's algorithm, OnePass allows each node to be visited an unlimited number of times. Each node can be visited by OnePass as many times as the number of paths from s to t. Note that enumerating all paths from s to t is a $\#P$-complete problem [22]. OnePass terminates when either k paths are recommended or all paths from s to t qualifying Lemma 1 are examined. Hence, the complexity of OnePass is $\mathcal{O}(|E| + K \cdot |N| \cdot \log |N|)$, where K is the number of shortest paths that have to be computed in order to cover the k results of the k-SPwLO query.

4. AN EFFICIENT EXACT ALGORITHM

Despite employing the pruning criterion of Lemma 1, OnePass still has to expand and examine a large portion of all possible $p(s \rightarrow t)$ paths. In this section, we propose a novel label-setting algorithm termed MultiPass to enhance the computation of k-SPwLO. The algorithm employs an additional powerful pruning criterion which significantly reduces the search space by avoiding expanding *non-promising* paths. Our experimental analysis in Section 4.3 demonstrates the advantage of MultiPass over OnePass in practice using real-world road networks.

4.1 Pruning Non-Promising Paths

Let $p_0(s \rightarrow t)$ be the shortest path from a source node s to a target node t as illustrated in Figure 3. In addition, let $p_i(s \rightarrow n)$ and $p_j(s \rightarrow n)$ be two distinct paths from source s to a node n of the shortest path p_0 such that $\ell(p_i) < \ell(p_j)$. Assuming that both p_i and p_j are extended to reach target t following the same path $p(n \rightarrow t)$, then any extension of p_i will be shorter than the respective extension of p_j. Furthermore, let $Sim(p_i, p_0) \leq Sim(p_j, p_0)$, i.e., the overlap ratio of p_i with p_0 is equal or lower than the ratio of p_j with p_0. Due to the monotonicity of the similarity function (Equation (1)), any extension of p_i to n will have the same or less overlap ratio with p_0 compared to the respective extension of p_j. In other words, for any extension of p_j there will always be a shorter extension of p_i with less or equal overlap ratio with p_0, and therefore, p_j can be pruned.

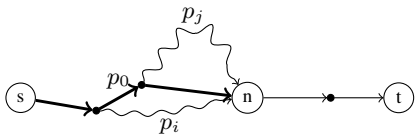


Figure 3: Pruning paths with Lemma 2.

The same idea can be utilized to prune the search space when computing the shortest alternative path to a set of paths P. Consider again p_i,p_j with $\ell(p_i) < \ell(p_j)$ and $Sim(p_i, p_0) \leq Sim(p_j, p_0)$. Path p_j is pruned if for every path $p \in P$ the overlap ratio $Sim(p_i, p)$ is lower or equal to $Sim(p_j, p)$. This pruning criterion is formally captured by the following lemma:

LEMMA 2. *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 source s to some node n. If $\ell(p_i) < \ell(p_j)$ and $\forall p \in P : Sim(p_i, p) \leq Sim(p_j, p)$ 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. Assume that an extension $p'_j = \langle (s, *), \ldots, (*, n), \ldots, (*, t) \rangle$ of $p_j(s \rightarrow n)$ 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 $p_i(s \rightarrow n)$ to target t is also an alternative path and it will be examined and recommended before p'_j.

According to the definition of alternative path, $Sim(p'_j, p) \leq \theta$ holds $\forall p \in P$ and following Lemma 1 $Sim(p_j, p) \leq \theta$ also holds $\forall p \in P$. Furthermore, due to the $\forall p \in P_{LO} : Sim(p_i, p) \leq Sim(p_j, p)$ assumption of Lemma 2, we get that $Sim(p_i, p) \leq \theta$ 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) $Sim(p'_i, p) \leq \theta$ 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. □

The pruning criterion of Lemma 2 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 be the set of paths from s to a node n created during the expansion of all paths from s. If set P_n contains a path $p'(s \rightarrow n)$ 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 overlap ratio $Sim(p', p)$ is higher than the ratio $Sim(p_n, p)$ 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) not applicable for another path already contained in P_n. To ensure that the set P_n contains only paths for which both (a) and (b) hold, every time a new path is added to P_n, we have to check whether condition (b) still holds for all paths in the set.

Unfortunately, Lemma 2 cannot be employed directly for the computation of k-SPwLO queries. Consider again the example in Figure 3. Let p_0 be the only path in the set of currently recommended alternative paths P. If during the search for the next alternative p_1 to P, p_j is pruned because $p_i \prec_P p_j$ holds, p_j cannot be part of the shortest alternative to P. However, there is no guarantee that p_j will not be part of the shortest alternative to both p_0 and p_1. In particular, if p_i is part of p_1, then, during the search for the next alternative to $P = \{p_0, p_1\}$, p_i might be pruned much earlier by Lemma 1. Hence, we have to compute k-SPwLO queries in an iterative way. Each time a new alternative is added to the k-SPwLO result set, we have to re-start the search for the next alternative from the beginning.

4.2 The MultiPass Algorithm

Next, we present MultiPass, which employs both pruning criteria of Lemma 1 and Lemma 2 to enhance the computation of k-SPwLO queries. The algorithm has the following key features. 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 $\langle n, p(s \rightarrow n) \rangle$[1]. MultiPass traverses the road network k-1 times. At each iteration, the algorithm examines paths from s in increasing order of their length and expands every path $p(s \rightarrow n)$ from s to a node n for which the following holds: (a) its similarity with any already computed result does not exceed the input threshold θ (Lemma 1) and (b) its extension can possibly lead to the shortest alternative path during the current iteration (Lemma 2). Every time a new path $p_n(s \rightarrow n)$ that qualifies conditions (a) and (b) is found, a label $\langle n, p_n \rangle$ is added to $\Lambda(n)$, and MultiPass removes all paths from $\Lambda(n)$ which do not qualify condition (b). As soon as a path to target t is found, MultiPass terminates current round, discards all stored labels, and re-traverses the network from source s. The algorithm terminates after k paths are added to result set P_{LO} or

[1]In practice, MultiPass 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.

ALGORITHM 1: MultiPass

Input: Road network $G(N, E)$, source node s, target node t, # of results k, similarity threshold θ
Output: Set P_{LO} of k paths

```
1  P_LO ← {shortest path p_0(s→t)};
2  while |P_LO|<k and last round updated P_LO do
3      initialize min-priority queue Q with ⟨s, ∅⟩;
4      ∀n ∈ N : Λ(n) ← ∅;
5      while Q not empty do
6          ⟨n, p_n⟩ ← Q.pop();                    ▷ Current path
7          if n=t then
8              P_LO ← P_LO ∪ {p_n};                ▷ Update result set
9              break;
10         else
11             foreach outgoing edge (n, n_c) ∈ E do
12                 p_c ← p_n ∘ (n, n_c);           ▷ Expand path p_c
13                 if ∃p_i ∈ P_LO : Sim(p_c, p_i)≥θ then
14                     continue;                   ▷ Pruned by Lemma 1
15                 else if ∃⟨n_c, p'_c⟩ ∈ Λ(n_c) : p'_c ≺_{P_LO} p_c then
16                     continue;                   ▷ Pruned by Lemma 2
17                 else
18                     remove from Q and Λ(n_c) all
                       ⟨n_c, p'_c⟩ : p_c ≺_{P_LO} p'_c;   ▷ Lemma 2
19                     Q.push(⟨n_c, p_c⟩);
20                     Λ(n_c) ← Λ(n_c) ∪ {⟨n_c, p_c⟩};
21 return P_LO;
```

the last iteration failed to find an alternative path. In the latter case, a complete set of k-SPwLO with respect to given θ and k values cannot be computed.

Algorithm 1 illustrates the pseudocode of MultiPass. The algorithm initializes P_{LO} with the shortest path $p_0(s{\rightarrow}t)$ (Line 1) and employs a min priority queue $\mathcal{Q}$ to traverse the road network. Before each traversal round, $\mathcal{Q}$ is initialized to $\langle s, \emptyset\rangle$ (Line 3) and the algorithm associates each node n with a (initially empty) set of labels $\Lambda(n)$ (Line 4). At each round in between Lines 5 and 20, MultiPass first pops label $\langle n, p_n\rangle$ for current path p_n in Line 6. If n is the target t, then p_n is added to P_{LO} and the round terminates (Lines 7–9). Otherwise, the algorithm expands the current path p_n considering all outgoing edges (n, n_c) (Lines 10-16). For each new path $p_c \leftarrow p_n \circ (n, n_c)$ (Line 12), the algorithm checks whether p_c qualifies the pruning criteria of Lemma 1 (Lines 13-14) and Lemma 2 (Lines 15-16). If the new path 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). Finally, MultiPass adds the new label to $\mathcal{Q}$ (Line 19) and $\Lambda(n_c)$ (Line 20) and proceeds with popping the next label from $\mathcal{Q}$.

To achieve an efficient implementation, for each label $\langle n, p_n\rangle$ MultiPass also stores a vector V_{Sim} containing the overlap ratio of p_n with all paths that were in P_{LO} at the time when the label was created. Due to the monotonicity of Equation 1, the overlap ratios stored in V_{Sim} can be updated incrementally. When a new label is created and added to $\mathcal{Q}$, our implementation of MultiPass performs lazy updates for $\mathcal{Q}$ and retains a black list to ignore labels representing pruned paths after they are removed from $\mathcal{Q}$. In order to consider results in P_{LO} that are added after the creation of the label, each time a label is popped MultiPass compares the size of V_{Sim} stored in the popped label to $|P_{LO}|$ and, if necessary, computes the missing overlaps and updates V_{Sim}.

EXAMPLE 1. *We demonstrate MultiPass using the road network of Figure 4 and the k-SPwLO*$(s, t, 0.5, 3)$ *query. During initialization, the shortest path* $p_0(s{\rightarrow}n){=}\langle(s, n_3), (n_3, n_5), (n_5, t)\rangle$ *is computed and added to* P_{LO}.

Starting from s, the first path examined by MultiPass is $p(s{\rightarrow}n_3){=}\langle(s, n_3)\rangle$. *The overlap ratio* $Sim(p(s, n_3), p_0){=}3/8{=}0.375$ *is below the similarity threshold* $\theta{=}0.5$*; hence* $p(s, n_3)$ *is not pruned. The same holds for the next two paths examined, which are* $p(s{\rightarrow}n_2){=}\langle(s, n_2)\rangle$ *and* $p(s{\rightarrow}n_1){=}\langle(s, n_1)\rangle$.

Next, MultiPass examines paths $p(s{\rightarrow}n_5){=}\langle(s, n_3), (n_3, n_5)\rangle$, $p'(s{\rightarrow}n_1){=}\langle(s, n_3), (n_3, n_1)\rangle$, $p'(s{\rightarrow}n_2){=}\langle(s, n_3), (n_3, n_2)\rangle$ *and* $p(s{\rightarrow}n_4){=}\langle(s, n_3), (n_3, n_4)\rangle$. *For path* $p(s{\rightarrow}n_5)$ *the overlap ratio* $Sim(p(s, n_5), p_0){=}6/8{=}0.75$ *exceeds the similarity threshold of* 0.5 *and so, path* $p(s{\rightarrow}n_5)$ *is pruned (Lemma 1). For path* $p'(s{\rightarrow}n_1)$ *the overlap ratio* $Sim(p'(s, n_1), p_0){=}0.375$ *does not exceed the similarity threshold. Since node* n_1 *has already been visited by path* $p(s{\rightarrow}n_1)$*, we check Lemma 2. We have* $Sim(p'(s{\rightarrow}n_1), p_0){>}Sim(p(s{\rightarrow}n_1), p_0)$ *and for the length* $\ell(p'(s{\rightarrow}n_1)){<}\ell(p(s, n_1))$*. Therefore, Lemma 2 cannot be applied and path* $p'(s{\rightarrow}n_1)$ *is not pruned. On the contrary, for path* $p'(s{\rightarrow}n_2)$ *we have* $Sim(p'(s{\rightarrow}n_2), p_0){>}Sim(p(s{\rightarrow}n_2), p_0)$ *and* $\ell(p'(s{\rightarrow}n_2)){>}\ell(p(s{\rightarrow}n_2))$*. In this case, the criterion of Lemma 2 is applied and path* $p'(s{\rightarrow}n_2)$ *is pruned. Finally, for path* $p(s{\rightarrow}n_4)$ *the overlap ratio* $Sim(p(s{\rightarrow}n_4), p_0){=}0.375$ *does not exceed the similarity threshold, hence, the path is not pruned.*

MultiPass continues the execution of the current round until the alternative path $p_1(s{\rightarrow}t){=}\langle(s, n_3), (n_3, n_4), (n_4, t)\rangle$ *with* $\ell(p_1){=}10$ *is found and subsequently added to* P_{LO}*. Next, MultiPass performs the second round in the same fashion, computes the alternative path* $p_2(s{\rightarrow}t){=}\langle(s, n_3), (n_3, n_1), (n_1, t)\rangle$ *and completes the result set* P_{LO}.

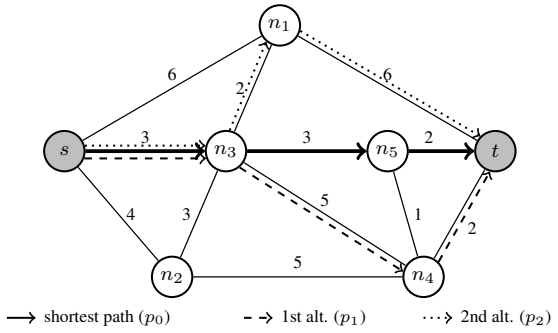


Figure 4: Result of k-SPwLO (s,t,0.5,3) query.

Compared to OnePass, MultiPass traverses the road network $k{-}1$ times instead of once (hence, the name of the algorithm). Each round works independently, i.e., builds a new path tree by expanding all paths that qualify both pruning criteria. As a result, at each round, MultiPass may potentially re-expand and re-examine paths already processed in previous rounds. On the other hand, by employing Lemma 2, the number of paths that MultiPass has to examine (including the paths examined multiple times) is lower than the number of paths processed by OnePass. Finally, OnePass has to check the simplicity of every new path, i.e., whether any cycles are contained, while MultiPass does not need to perform such a check, as Lemma 2 ensures that all non-simple paths are pruned.

Complexity analysis. Given a k-SPwLO query, MultiPass first computes the shortest path $p_0(s{\rightarrow}t)$ from source node s to target t. Naturally, the cost of this step is independent of the number of requested paths k and the similarity threshold θ. For the computa-

tion of $p_0(s \rightarrow t)$, any shortest path algorithm can be employed, e.g., Dijkstra's algorithm [9] which requires $\mathcal{O}(|E| + |N| \cdot \log |N|)$.

To find each subsequent alternative path, MultiPass expands all paths from source s that qualify the pruning criterion of Lemma 1. However, as the value of the similarity threshold θ approaches 1, the number of paths pruned by Lemma 1 significantly drops, which means that MultiPass returns the K-Shortest paths. Furthermore in practice, there exists no formula for estimating the number of paths pruned by the pruning criterion of Lemma 2. Hence, each round of MultiPass becomes equivalent to Fox's algorithm [10] with a complexity $\mathcal{O}(|E| + K \cdot |N| \cdot \log |N|)$, where K is the number of the shortest paths that have to be computed to cover the k-SPwLO result. Since MultiPass has to perform $k-1$ rounds to compute a k-SPwLO query, its total runtime complexity is $\mathcal{O}(k(|E|+K \cdot |N| \cdot \log |N|))$ where $K \gg k$. We have shown in [6] that the number of K-shortest paths that need to be computed in order to cover the k-SPwLO set is very high; in extreme cases, MultiPass may have to construct all paths from s to t.

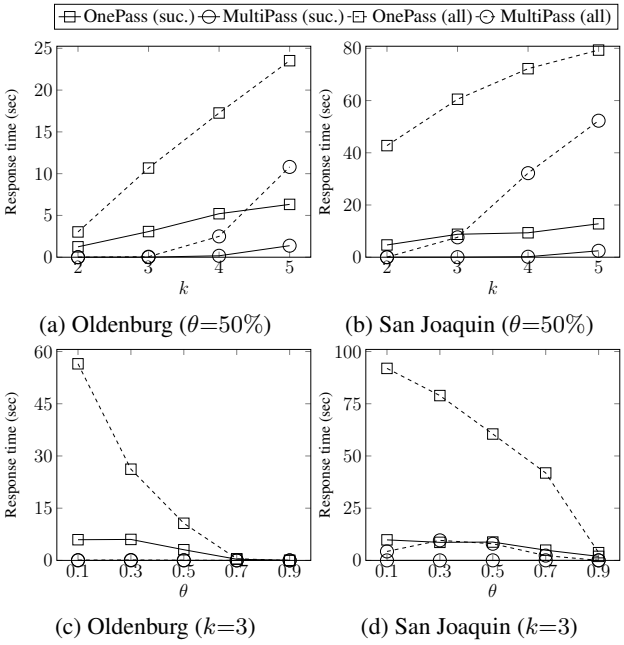
Finally, note that the time complexity of MultiPass is worse than the time complexity of OnePass. However, we show in our experimental evaluation that MultiPass is much faster than OnePass. The reason for this inconsistency is that, although by employing the pruning criterion of Lemma 2 MultiPass examines much less paths than OnePass, there can be no formal guarantees for the number of paths that are pruned. Although MultiPass has worse theoretical time complexity, in practice it is much more efficient than OnePass.

Optimization. As discussed in [6] for OnePass, the performance of MultiPass can be further enhanced by employing a lower bound, $\underline{d}(n,t)$, for the network distance $d(n,t)$ of every node n to the target t. By employing such a lower bound, MultiPass traverses at each round the network in an A^*-like fashion and directs the search towards the target, which avoids visiting nodes that are far away. In order to derive tight $\underline{d}(n,t)$ lower bounds, we first reverse the edges of the road network and then run Dijkstra's algorithm from target t to every node n of the network [18]. In practice, at the beginning of the MultiPass execution, instead of simply computing the shortest path from s to t, we compute the shortest path from target t to each node n in the road network.

4.3 Experimental Evaluation

To demonstrate its efficiency, we compare MultiPass against OnePass presented in [6] using real road networks. For each algorithm, we measure the average response time and the number of examined labels (i.e., paths) over 1,000 random queries varying parameters k and θ. Due to the high execution time of OnePass, our experiments involve only the road networks for the city of Oldenburg (6,105 nodes and 14,058 edges) and the city of San Joaquin (18,263 nodes and 47,594 edges). We also consider a timeout of 120 seconds for the evaluation of each query.

Figure 5 reports the response times of MultiPass and OnePass. The continuous lines show the time for the queries for which both algorithms finished their execution in less than 120 seconds, whereas the dashed lines show the time for all 1,000 queries including those which did not finish within 120 seconds. In Figures 5a and 5b, we observe that the performance of both OnePass and MultiPass deteriorates as the number k of requested paths increases. For all values of k though, MultiPass is clearly faster than OnePass, and in most cases MultiPass is at least two times faster. Another interesting observation is that the performance curve of OnePass is almost linear, i.e., each iteration requires approximately the same time. For instance, for Oldenburg (Figure 5a) the algorithm needs similar time to find the third and the fourth alternative path. On the other hand, MultiPass needs more time for each subsequent result. This behavior can be explained by the fact that MultiPass restarts and re-expands paths. With regard to parameter θ, we observe in Figures 5c and 5d that in all cases, MultiPass is faster than OnePass. Especially for the lowest values of θ, i.e., 0,1 and 0.3, MultiPass outperforms OnePass by at least an order of magnitude. The performance of OnePass is close to MultiPass only for θ=0.9, where the computed paths can be very similar.



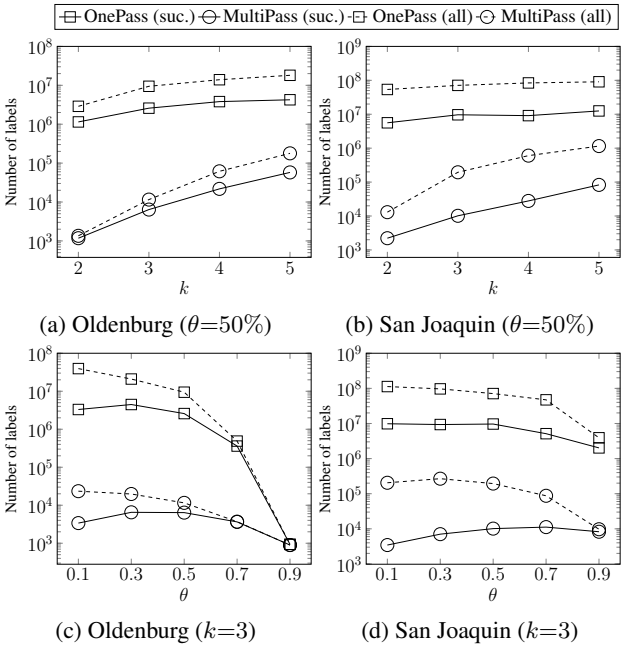
(a) Oldenburg (θ=50%) (b) San Joaquin (θ=50%)

(c) Oldenburg (k=3) (d) San Joaquin (k=3)

Figure 5: Performance comparison of MultiPass and OnePass varying requested paths k and similarity threshold θ.

To provide a better insight on the performance of MultiPass and OnePass, we report in Figure 6 the number of labels/paths each algorithm needs to examine before returning the k-SPwLO result. We observe that in all scenarios MultiPass examines significantly fewer paths than OnePass (even though we count the total number of paths from all rounds of MultiPass, hence some paths may be counted more than once). With respect to the similarity threshold θ, we observe the following important trade-off. As θ increases, the pruning power of Lemma 1 deteriorates, and both OnePass and MultiPass construct more paths (supporting measurements are not included due to lack of space). However at the same time, the next result can be determined earlier and, hence, the total runtime drops. In addition, as θ decreases, the pruning power of Lemma 2 increases, and more partial paths can be pruned. This explains the behavior of MultiPass, where the number of examined paths initially increases, but after θ=0.5 it goes down.

Finally, in Table 1 we report the percentage of timed-out/failed queries for timeout values of 30, 60 and 120 seconds. First, we observe that the failure rate of OnePass is, in most cases, much higher than the failure rate of MultiPass. More specifically, for the road network of Oldenburg, the failure rate of OnePass is more than 10% when k>3 or θ<50%. For the road network of San Joaquin, apart from the case where k=3 and θ=90%, the failure rate of OnePass is more than 30%, even when the timeout is set to 120 seconds. On the contrary, the timeout rate of MultiPass for the road network of Oldenburg is in all cases below 10%. For the road network of San Joaquin, the failure rate of MultiPass is below 10%,



(a) Oldenburg (θ=50%) (b) San Joaquin (θ=50%)

(c) Oldenburg (k=3) (d) San Joaquin (k=3)

Figure 6: Comparison of examined paths by OnePass and MultiPass varying requested paths k and similarity threshold θ.

except for the cases where k>3. However, even in cases where the failure rate of MultiPass is the highest (θ=50% and k>3), it is still significantly lower than the failure rate of OnePass.

road network	θ	k	OnePass			MultiPass		
			30	60	120	30	60	120
Oldenburg	0.1	3	46.9	45.4	44.5	0	0	0
	0.3	3	22.8	20.5	17.8	0	0	0
	0.5	3	9.1	7.7	6.6	0	0	0
	0.7	3	0.4	0.1	0.1	0	0	0
	0.9	3	0	0	0	0	0	0
	0.5	2	2.7	0.2	1.5	0	0	0
	0.5	4	15.2	12.8	10.7	2.6	1.4	0.7
	0.5	5	20.4	17.5	15	9.6	7.4	6.2
San Joaquin	0.1	3	77.5	76	74.6	3.2	1.8	1.3
	0.3	3	66.8	65.2	63.2	8.1	5.6	4.4
	0.5	3	52.3	49.4	46.6	6.8	5.1	3.5
	0.7	3	35.8	33.6	32.1	2.1	0.9	0.3
	0.9	3	3.1	2.3	1.6	0	0	0
	0.5	2	36.5	34.1	33.2	0	0	0
	0.5	4	61	59.3	56.8	28.9	25.5	22.5
	0.5	5	67.5	64.8	62.3	45.2	42	39.1

Table 1: Failure rate (%) for timeout set to 30, 60 and 120 sec.

5. APPROXIMATE ALGORITHMS

Our experimental analysis in Section 4.3 showed that MultiPass clearly outperforms OnePass presented in [6]. However, despite employing Lemma 2, MultiPass still has to examine a large number of paths, which essentially renders the algorithm not applicable to large-scale road networks. In view of this, we next investigate the approximate evaluation of k-SPwLO queries. In particular, we first discuss a baseline method, termed SVP$^+$, which builds on top of existing literature, and then we propose two novel approximate algorithms, termed OnePass$^+$ and ESX.

5.1 A Baseline Solution

Our baseline algorithm, denoted by SVP$^+$, builds upon the notion of *single-via paths*, which was introduced as an alternative

ALGORITHM 2: SVP$^+$

Input: Road network $G(N, E)$, source node s, target node t, # of results k, similarity threshold θ
Output: Set P_{LO} of k paths

1 initialize min-priority queue $\mathcal{Q}$ with $\emptyset$;
2 $T_{s\to N} \leftarrow$ shortest path tree from s to all $n \in N$;
3 $T_{N\to t} \leftarrow$ shortest path tree from all $n \in N$ to t;
4 **foreach** $n \in N$ **do**
5 $\quad \mathcal{Q}.push(\langle n, d(s,n)+d(n,t)\rangle)$;
6 $P_{LO} \leftarrow \{$shortest path $p_0(s \to t)\}$;
7 **while** P_{LO} *contains less than* k *paths* ***and*** $\mathcal{Q}$ *not empty* **do**
8 $\quad \langle n, \ell(p_n) \leftarrow \mathcal{Q}.pop()$;
9 $\quad p_n \leftarrow$ `RetrieveSingleViaPath`$(T_{s\to N}, T_{N\to t}, n)$;
10 $\quad$ **if** $Sim(p_n, p) \le \theta$ *for all* $p \in P_{LO}$ **then**
11 $\quad\quad$ add p_n to P_{LO}; ▷ `Update result set`
12 **return** P_{LO};

routing technique in [1]. As we discussed in Section 2, the original method considers the similarity of single-via paths only with regard to the shortest path and disregards the pairwise dissimilarity of all results. Instead of employing the objective criteria as in [1], SVP$^+$ iterates over the set of single-via paths aiming to find a subset of k paths which are (a) sufficiently dissimilar to each other and (b) as short as possible. Intuitively, the main idea behind SVP$^+$ is similar to the baseline method for computing k-SPwLO queries discussed in Section 3. However, instead of iterating over all possible paths connecting a source node s to a target node t and computing their pairwise overlap ratio, SVP$^+$ iterates over the much smaller set of single-via paths. As the results of k-SPwLO are not necessarily singe-via paths, SVP$^+$ can only provide approximate answers to the queries.

Algorithm 2 illustrates the pseudocode of SVP$^+$. First, the algorithm computes the two shortest path trees, one from s to every node of G (Line 2) and one from every node of G to t (Line 3). During this step all distances $d(s,n)$ and $d(n,t)$ are also computed. The algorithm orders the nodes based on the sum of $d(s,n)$+$d(n,t)$, which is also the length of the single-via path of n, using a min priority queue $\mathcal{Q}$ (Lines 4-5). In Line 6, the result set P_{LO} is initialized with p_0, i.e., the shortest path from s to t. Note that the shortest path p_0 is actually the shortest single-via path and, hence, no additional computation is required. 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). Then, SVP$^+$ checks in Line 10 whether p_n is sufficiently dissimilar to all paths currently in P_{LO}; if so, p_n is added to P_{LO} (Line 11). The algorithm terminates and returns the P_{LO} set 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.

5.2 Approximate OnePass

We propose next a novel approximate algorithm, denoted by OnePass$^+$, which combines the feature of OnePass to scan the graph only once with the pruning power of Lemma 2. OnePass$^+$ has the following key features. Given a source node s and a target node t, OnePass$^+$ traverses the road network expanding every path $p(s\to n)$ from source s to a node n that qualifies both Lemma 1 and Lemma 2. This procedure is the same with each distinct round of MultiPass. In contrast to MultiPass though, each time a new result is added to the result set P_{LO}, an update procedure takes place for all remaining incomplete paths $p(s\to n)$. In particular, for every incomplete path $p(s\to n)$, OnePass$^+$ computes the over-

lap of p with the newly found result and, then, p is checked against Lemma 1 with respect to the updated P_{LO}. The same update procedure is also employed by OnePass. The algorithm terminates when either k paths are recommended or all paths from s to t qualifying Lemma 1 and Lemma 2 have been examined.

By not restarting after the computation of each new result, OnePass$^+$ avoids expanding the network multiple times. However, the fact that OnePass$^+$ does not restart the expansion after each round implies that the next best path might get pruned and, hence, OnePass$^+$ cannot guarantee that the exact solution will be found. We already explained in Sec. 4 that for the MultiPass algorithm to find the exact solution, the restart is required as a path that is pruned as non-promising during the current round, may be promising during the next round. All such paths are excluded permanently from the search space of OnePass$^+$. Nevertheless, this case applies to only a small subset of the paths from a source s to a target t and, hence, the average length of paths in the P_{LO} set is expected to be close to the optimal one.

Algorithm 3 illustrates the pseudocode of OnePass$^+$. The algorithm employs a min priority queue $\mathcal{Q}$ (initialized with source s) to traverse the road network. Result set P_{LO} is initialized with p_0, i.e., the shortest path from s to t (Line 1). In between Lines 4 and 21, OnePass$^+$ examines the contents of $\mathcal{Q}$ until either k paths are found or $\mathcal{Q}$ is depleted. At each iteration, a label $\langle n, p_n \rangle$ is popped from $\mathcal{Q}$ (Line 5). If node n is the target t, then p_n is added to P_{LO} (Line 7) and the same update procedure as in OnePass takes place (Lines 8-10), i.e., all paths p_h with $Sim(p_h, p_c) > \theta$ are discarded. Otherwise, the algorithm expands the current path p_n considering all outgoing edges (n, n_c) (Lines 12-21). OnePass checks whether the new path $p_c \leftarrow p_n \circ (n, n_c)$ qualifies the pruning criteria of both Lemma 1 (Lines 14-15) and Lemma 2 (Lines 16-17) and updates $\mathcal{Q}$ and $\Lambda(n_c)$ accordingly. Finally, OnePass adds a new label for p_c to $\mathcal{Q}$ (Line 20) and $\Lambda(n_c)$ (Line 21) and proceeds with popping the next label from $\mathcal{Q}$.

Similar to MultiPass, for each label our implementation of OnePass$^+$ maintains and updates incrementally a vector V_{Sim} containing the overlaps of p_n with all paths that where in P_{LO} at the time when the label was created. Furthermore, OnePass$^+$ also performs lazy updates for $\mathcal{Q}$. That is, for labels that have already been created and added to $\mathcal{Q}$, OnePass$^+$ updates V_{Sim} only at the time when a label is popped from $\mathcal{Q}$. OnePass$^+$ also retains a black list to ignore labels representing pruned paths.

Complexity Analysis. Similar to MultiPass, given a k-SPwLO query from a node s to a node t, OnePass$^+$ first computes $p_0(s \rightarrow t)$ using any shortest path algorithm, e.g., Dijkstra, and adds it to the result set. To compute alternatives, OnePass$^+$ traverses the road network expanding every path $p(s \rightarrow n)$ from source s to a node n that qualifies both Lemma 1 and Lemma 2. As we discussed in the cost analysis of MultiPass, there can be no formal guarantees regarding the number of paths that are pruned using either pruning criterion. In the worst case when no paths are pruned, OnePass$^+$ is equivalent to OnePass and Fox's algorithm. Therefore, the time complexity of OnePass$^+$ is also $\mathcal{O}(|E| + K \cdot |N| \cdot \log |N|)$, where K is the number of shortest paths that have to be computed in order to cover the k-SPwLO result.

5.3 Edge Subset Exclusion

Finally, we present our second approximate algorithm, denoted by ESX, which computes k-SPwLO by iteratively excluding edges from the road network. The idea behind ESX is the following. Given a road network G, a source node s and a target node t, the algorithm first adds the shortest path p_0 to the result set P_{LO}, similar to all previously described methods. Next, ESX removes an edge of p_0 from the road network and computes the shortest path p_c from s to t on the updated road network[2]. If the overlap of path p_c with p_0 does not violate the similarity threshold θ, then p_c is added to the result set P_{LO}. Otherwise, the algorithm proceeds with removing more edges from the road network. If P_{LO} contains more than one paths, ESX removes an edge from path $p \in P_{LO}$ for which the similarity $Sim(p_c, p)$ is the highest. At each iteration, ESX removes only one edge from some path in P_{LO}. The process is repeated until a path that does not violate the similarity threshold θ is found. To compute more alternatives, the algorithm continues by removing more edges until another alternative is found, or until there are no more edges to remove.

Removing an edge from the road network may cause the network to become disconnected and prevent any subsequent iteration from finding a valid path. To avoid such a case, the algorithm has to make sure that any edge affecting the connectivity of the road network is never removed. To this end, after removing an edge from the road network, if the shortest path search fails to find a path connecting s and t, then ESX re-inserts the edge in the road network and marks it as *non-removable*. Edges marked as non-removable cannot be removed at any iteration.

The order in which we remove the edges from the road network affects both the quality of the result and the performance of ESX. However, determining the optimal order is prohibitively expensive. Therefore, to determine which edge to remove at each iteration, we employ a heuristic based on the following observation: the more shortest paths cross an edge, the greater the probability that the removal of this edge will cause a detour and lead the next result faster. As it is also prohibitively expensive to compute the all-pairs

ALGORITHM 3: OnePass$^+$

Input: Road network $G(N, E)$, source node s, target node t, # of results k, similarity threshold θ
Output: Set P_{LO} of k paths

1 $P_{LO} \leftarrow \{$shortest path $p_0(s \rightarrow t)\}$;
2 initialize min-priority queue $\mathcal{Q}$ with $\langle s, \emptyset \rangle$;
3 $\forall n \in N : \Lambda(n) \leftarrow \emptyset$;
4 **while** P_{LO} *contains less than k paths* ***and*** $\mathcal{Q}$ *not empty* **do**
5 $\quad \langle n, p_n \rangle \leftarrow \mathcal{Q}.pop()$; ▷ Current path
6 $\quad$ **if** $n = t$ **then**
7 $\quad\quad P_{LO} \leftarrow P_{LO} \cup \{p_n\}$; ▷ Update result set
8 $\quad\quad$ **foreach** *label* $\langle n', \ell(p_{n'}) \rangle$ *in* $\mathcal{Q}$ **do**
9 $\quad\quad\quad$ **if** $Sim(p_{n'}, p_i) > \theta, \forall p_i \in P_{LO}$ **then**
10 $\quad\quad\quad\quad$ remove $\langle n', \ell(p_{n'}) \rangle$ from $\mathcal{Q}$; ▷ Lemma 1
11 $\quad$ **else**
12 $\quad\quad$ **foreach** *outgoing edge* $(n, n_c) \in E$ **do**
13 $\quad\quad\quad p_c \leftarrow p_n \circ (n, n_c)$; ▷ Expand path p_c
14 $\quad\quad\quad$ **if** $\exists p_i \in P_{LO} : Sim(p_c, p_i) \geq \theta$ **then**
15 $\quad\quad\quad\quad$ **continue**; ▷ Pruned by Lemma 1
16 $\quad\quad\quad$ **else if** $\exists \langle n_c, p'_c \rangle \in \Lambda(n_c) : p'_c \prec_{P_{LO}} p_c$ **then**
17 $\quad\quad\quad\quad$ **continue**; ▷ Pruned by Lemma 2
18 $\quad\quad\quad$ **else**
19 $\quad\quad\quad\quad$ remove from $\mathcal{Q}$ and $\Lambda(n_c)$ all $\langle n_c, p'_c \rangle : p_c \prec_{P_{LO}} p'_c$; ▷ Lemma 2
20 $\quad\quad\quad\quad \mathcal{Q}.push(\langle n_c, p_c \rangle)$;
21 $\quad\quad\quad\quad \Lambda(n_c) \leftarrow \Lambda(n_c) \cup \{\langle n_c, p_c \rangle\}$;
22 **return** P_{LO};

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

shortest paths, ESX performs a local check. Given an edge $e(a, b)$ on some path $p \in P_{LO}$, let $E_{inc}(a)$ be the set of all incoming edges $e(n_i, a)$ to a from some node $n_i \in N \backslash \{b\}$ and $E_{out}(b)$ be the set of all outgoing edges $e(b, n_j)$ from b to some node $n_j \in N \backslash \{a\}$. First, ESX computes the set P_s which contains the shortest paths from every node $n_i \in E_{inc}(a)$ to every node $n_j \in E_{out}(b)$. Then, ESX defines the set P'_s which contains all paths $p \in P'_s$ that cross edge e. Finally, ESX assigns a priority to edge e, denoted by $prio(e)$, which is set to $|P'_s|$.

EXAMPLE 2. *Consider our running example in Figure 7, where $p_0(s \rightarrow t) = \langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$ is the shortest path from s to t and the only path currently in P_{LO}. For edge (n_3, n_5) we compute the shortest path from every node in $\{s, n_1, n_2, n_4\}$ to every node in $\{n_4, t\}$. Three shortest paths, $p(n_1 \rightarrow n_4)$, $p(s \rightarrow n_4)$ and $p(s \rightarrow t)$, cross edge (n_3, n_5) (bold lines). On the other hand, the rest of the shortest paths, e.g., shortest path $p(n_2 \rightarrow t)$ (dashed line), do not cross edge (n_3, n_5). Therefore, the priority of edge (n_3, n_5) is $prio(n_3, n_5) = 3$. In the same fashion, we compute the priorities for edges (s, n_3) and (n_3, t), and we have $prio(s, n_3) = 0$ and $prio(n_5, t) = 0$.*

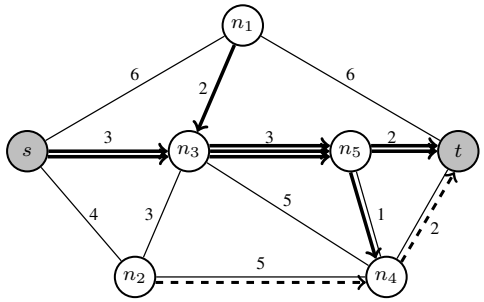


Figure 7: Computing the priority of edge (n_3, n_5).

Algorithm 4 illustrates the pseudocode of ESX. First, the algorithm initializes P_{LO} with the shortest path p_0 in Line 1 and creates a max-heap H_0, associated with p_0, in which all the edges of p_0 are enheaped and sorted based on their priority (Line 2). The algorithm also initializes the set E_{DNR} of non-removable edges (Line 3). ESX enters the outer loop in Line 4 and continues until either k results are found or there are no more edges to be removed from the graph. Next, the algorithm sets p_c to the last result found and enters the inner loop (Line 6). At each iteration the algorithm chooses p_{max} as the path in P_{LO} which has the maximum overlap $Sim(p_c, p_{max})$ and which contains edges in H_{max} (the max-heap associated with p_{max}) that can be removed from the graph. Then, the algorithm deheaps edge e_{tmp} from H_{max} (Line 8) and checks whether e_{tmp} is in E_{DNR}, i.e., it is marked as non-removable (Line 9). If it is not, edge e_{tmp} is removed (Line 10) and the algorithm computes the shortest path p_{tmp} on the updated graph (Line 11). In Lines 12-15 the algorithm checks whether p_{tmp} is a valid path and, if not, re-inserts e_{tmp} to the graph and marks it as non-removable. Otherwise, the algorithm sets p_c to p_{tmp} and proceeds to the next iteration. Finally, when the inner loop is finished, the algorithm checks if p_c is a valid alternative (there is also the possibility that all the heaps are empty and no more edge can be removed). If p_c is valid, it is added to P_{LO} and a new max-heap H_c associated with p_c is initialized with the edges of p_c. Finally, after the outer loop is finished, the algorithm returns P_{LO} in Line 20.

EXAMPLE 3. *We demonstrate the functionality of ESX using again the road network of Figure 4 and the*

ALGORITHM 4: ESX

Input: Road network $G(N, E)$, source node s, target node t, # of results k, similarity threshold θ
Output: Set P_{LO} of k paths

```
1  P_LO ← {shortest path p_0(s → t)};
2  initialize max-heap H_0 ← ⟨e_i, prio(G, e_i)⟩, ∀e_i ∈ p_0;
   ▷ Every H_i is associated with p_i
3  initialize E_DNR ← ∅;
4  while P_LO contains less than k paths and ∃H_i not empty do
5      set p_c ← last path added to P_LO;
6      while max{Sim(p_c, p_i) : p_i ∈ P_LO and H_i not empty
       } > θ do
7          Edge e_tmp ← H_i.pop();
8          if e_tmp ∈ E_DNR then
9              continue;
10         G.remove(e_tmp);
11         Path p_tmp ← ShortestPath(G, s, t);
12         if p_tmp is null then
13             re-insert e_tmp to G;
14             insert e_tmp to E_DNR;
15             continue;
16         p_c ← p_tmp;
17     if max{Sim(p_c, p_i) : p_i ∈ P_LO} then
18         add p_c to P_LO;
19         initialize max-heap H_c ← ⟨e_j, prio(G, e_j)⟩,
           ∀e_j ∈ p_c;
20 return P_LO;
```

$k{-}SPwLO(s, t, 0.5, 2)$ query. During initialization, the shortest path $p_0(s \rightarrow t) = \langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$ is computed and added to the result set P_{LO}. First, we compute the priority of each edge of the shortest path. Having computed the priorities, we first remove edge (n_3, n_5), which is the edge with the highest priority. Then, we compute the shortest path $p'(s \rightarrow t)$ on the updated graph. The shortest path is $p'(s \rightarrow t) = \langle (s, n_3), (n_3, n_4), (n_4, t) \rangle$ with $\ell(p'(s \rightarrow t)) = 10$. We check the overlap of the new path with the original shortest path and find that $Sim(p'(s \rightarrow t), p_0) = 0.375$, which does not exceed the similarity threshold. Therefore, $p'(s \rightarrow t)$ is added to the P_{LO} set.

Complexity Analysis. ESX reduces the search for an alternative path to a set of ordinary shortest path queries. In particular, given a road network $G = (N, E)$ let P_{LO} be the result set of a k-SPwLO(s, t, θ, k) query containing k paths. ESX requires $|P| \times$ total number of edges in P executions of shortest path queries, i.e., the number of shortest path queries that have to be processed is linear to the number k of paths and the size of the result paths. Furthermore, in our implementation of ESX we employed the optimization using lower bounds described in Section 4.2, which reduces the cost for retrieving each shortest path to a minimum; thus, the performance of ESX is significantly optimized.

6. EXPERIMENTAL EVALUATION

In this section, we report the results of an experimental evaluation of the algorithms for processing k-SPwLO queries. We use seven different road networks shown in Table 2. To assess the runtime performance, we measure the average response time over 1,000 random queries (i.e., pairs of nodes), varying the number k of requested paths and the similarity threshold θ. In each experiment, we vary one of the two parameters and fix the other to its default value: 3 for k and 0.5 for θ. We also report experiments on

Table 2: Road networks.

road network	# nodes	# edges
Oldenburg	6,105	14,058
San Joaquin	18,263	47,594
Vienna	19,826	54,918
Denver	73,166	196,630
San Francisco	174,956	443,604
New York City	264,346	730,100
Colorado	435,666	1,057,066

the quality of the results computed by the approximate solutions. Given the number k of requested paths, we measure (a) the number of paths returned by each approximate algorithm and (b) the average length of the computed paths in comparison to the length of the shortest path. All algorithms were implemented in C++ and the tests run on a machine with 4 Intel Xeon X5550 (2.67GHz) processors and 48GB main memory running Ubuntu Linux.

6.1 Performance

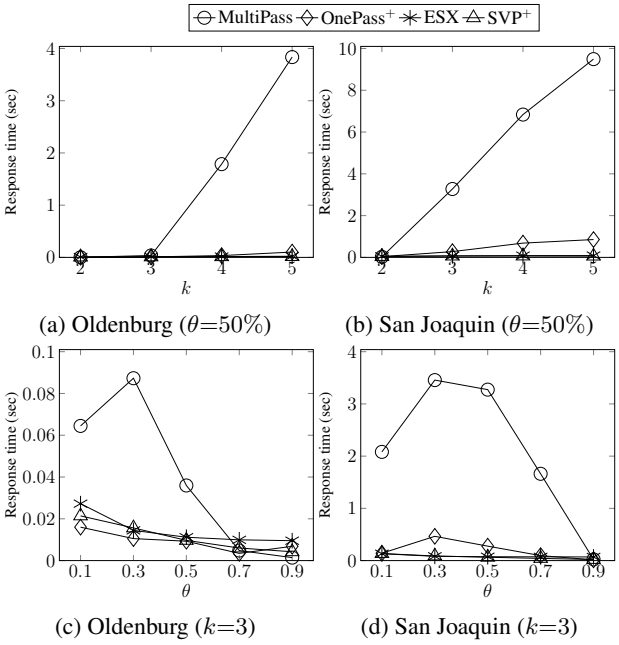
Similar to Section 4.3, we consider a timeout of 120 seconds for each query. The ratio of timed-out/failed queries for OnePass$^+$ was below 10% in all experiments. For SVP$^+$ and ESX, all queries were executed within 120 seconds. Due to space limitations, the performance results shown in this section consider only those queries which were successfully completed by all algorithms.

The first experiment in Figure 8 compares the exact algorithm MultiPass with the approximate solutions SVP$^+$, OnePass$^+$ and ESX. In Figures 8a–b we observe that the runtime of all algorithms increases with the number k of requested paths. As expected, the runtime of the approximate solutions increases only slightly, whereas the exact solution MultiPass deteriorates for large values of k. For k>3, MultiPass becomes one order of magnitude slower than OnePass$^+$ and more than two orders of magnitude slower than ESX and SVP$^+$. With regard to parameter θ, Figures 8c–d show that for θ<70%, MultiPass is one order of magnitude slower than OnePass$^+$ and two orders of magnitude slower than SVP$^+$ and ESX (for θ=30%). For large values of θ, the performance of MultiPass gets closer to the performance of the approximate algorithms (for θ=90% MultiPass is even faster than ESX and SVP$^+$).

The next experiment in Figure 9 compares the approximate algorithms using larger datasets. In Figures 9a–b, we vary the parameter k. For small values of k, the difference is not much, while for increasing values of k both SVP$^+$ and ESX clearly outperform OnePass$^+$ up two one order of magnitude. In Figures 9c–d, where the value of θ varies, we observe that OnePass$^+$ is very fast for extreme values of θ (θ=10% and θ=90%), but it is rather slow for values in between. It is clear that OnePass$^+$ is not practical for large road networks and/or values of k>3.

Another interesting observation in Figures 8c–d and 9c–d is that the performance of MultiPass and OnePass$^+$ show a local maximum for θ=0.3, which indicates the following important trade-off. As θ increases, the pruning power of Lemma 1 deteriorates, and both MultiPass and OnePass$^+$ construct more (partial) paths (supporting measurements are excluded due to lack of space). At the same time, the next result will be determined earlier, and hence the total runtime drops. In addition, as θ decreases, the pruning power of Lemma 2 also increases and more partial paths are pruned. This explains the behavior of MultiPass and OnePass$^+$, where the response time initially increases, but after θ=0.3 the runtime of both algorithms goes down.

To summarize the observations in Figures 8 and 9, the approximate solutions clearly outperform the exact algorithm MultiPass.



(a) Oldenburg (θ=50%) (b) San Joaquin (θ=50%)

(c) Oldenburg (k=3) (d) San Joaquin (k=3)

Figure 8: Performance comparison of exact and approximate algorithms varying requested paths k and similarity threshold θ.

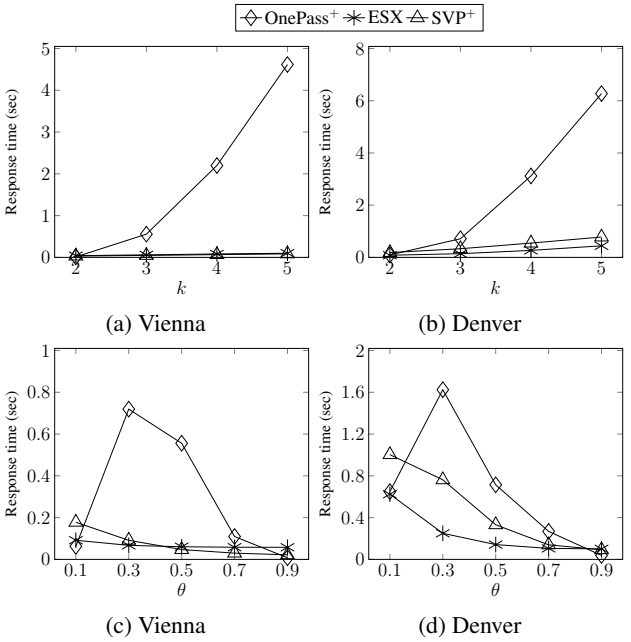
Comparing the approximate solutions, we observe that SVP$^+$ and ESX have similar performance and are the clear winners for the datasets, which are of small and medium size. OnePass$^+$ is the slowest approximate solution.

6.2 Scalability

From the previous experiments it is clear that both MultiPass and OnePass$^+$ are not scalable. For values of k>2 both algorithms are prohibitively expensive, even for a mid-sized road network such as Denver. However, the same does not apply for ESX and SVP$^+$. To demonstrate the scalability of ESX and SVP$^+$, we present in Figure 10 the results of an experiment using larger values of $k \in \{4, 8, 12, 16\}$ and we also include larger datasets. We observe that for San Francisco and Colorado, ESX is significantly faster that SVP$^+$ for all values of k. For the road networks of Denver and New York, ESX is faster than SVP$^+$ only for small values of k, whereas SVP$^+$ appears to be faster than ESX for k=12 and k=16. The reason for this behavior is that SVP$^+$ computes considerably less alternative paths than ESX (cf. Table 3 and the discussion in Sec. 6.3). Notice that the smaller result set is not due to a timeout, rather the algorithm is not able to find more alternatives. Overall, whenever ESX and SVP$^+$ find approximately the same number of alternative paths, ESX clearly outperforms SVP$^+$.

6.3 Result Quality & Completeness

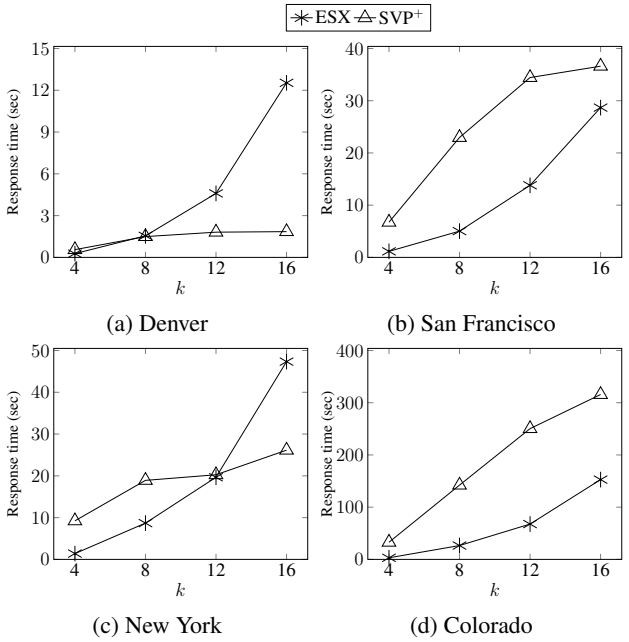
In Figure 11, we present our experiments that analyze the quality of the computed results. We consider all queries for which each algorithm returned k paths (i.e., no timeout) and compute the average length of the returned paths. Then we compare the average length of each result set to the length of the shortest path. That is, we show how much longer, on average, are the alternative paths with respect to the shortest path. Obviously, the exact results, named k-SPwLO, contain the shortest alternatives. They can be computed by any exact algorithm, such as MultiPass. Looking at the approximate solutions, OnePass$^+$ produces clearly the best alternatives, which are very close to the paths in the exact solution. Both ESX



(a) Vienna (b) Denver

(c) Vienna (d) Denver

Figure 9: Performance comparison of approximate algorithms varying requested paths k and similarity threshold θ.



(a) Denver (b) San Francisco

(c) New York (d) Colorado

Figure 10: Performance comparison of SVP$^+$ and ESX for $k \in \{2, 4, 8, 16\}$ and $\theta = 50\%$.

and SVP$^+$ recommend alternatives that, on average, are up to 15% longer than the alternatives in k-SPwLO. The alternatives recommended by ESX, though, are most of the time shorter than the alternatives recommended by SVP$^+$.

The next experiment analyzes the completeness of the result sets. As we have already seen in previous experiments, the algorithms are not always able to compute k alternative paths. Table 3 reports for each algorithm the percentage of queries for which exactly k alternative paths were found. Naturally, the exact solution k-SPwLO has the highest completion ratio. OnePass$^+$ is very close to the exact solution. In particular, for San Joaquin the completion ratio of OnePass$^+$ is always more than 99%. The completion ratio of ESX is lower than OnePass$^+$, but constantly over 95%. Finally, SVP$^+$ has generally the lowest completion ratio (except for k=3, where ESX is slightly worse).

road network	k	k-SPwLO	OnePass$^+$	ESX	SVP$^+$
Oldenburg	2	100	100	100	100
	3	99.9	99.1	98.7	99.5
	4	99.9	98.6	97.1	95
	5	99.9	98.2	95.8	85.6
San Joaquin	2	100	100	100	99.9
	3	100	99.8	98.5	99.5
	4	100	99.7	97.7	97
	5	100	99.3	95.6	94.3

Table 3: Average completeness ratio (%) per query varying requested paths k ($\theta = 50\%$) for all algorithms.

The final experiment in Table 4 compares the quality of ESX and SVP$^+$ by measuring the average number of returned paths for four road networks and values of $k \in \{4, 8, 12, 16\}$. It is evident that ESX returns more alternative paths than SVP$^+$ for all values of k. The number of alternatives returned by ESX is, in all cases, very close to k. In contrast, the number of paths returned by SVP$^+$ is significantly lower than k for k>8. For instance, for New York SVP$^+$ cannot find more than six alternatives per query on average; similar figures can be observed for Denver and San Francisco. Apparently, the set of single-via paths does not contain enough sufficiently dissimilar paths, and hence SVP$^+$ returns more and more incomplete results for an increasing k.

road network	k	ESX	SVP$^+$
Denver	4	3.96	3.95
	8	7.72	6.52
	12	11.39	7.14
	16	14.94	7.25
San Francisco	4	3.97	3.95
	8	7.92	7.03
	12	11.81	8.42
	16	15.55	8.80
New York	4	3.97	3.75
	8	7.77	5.49
	12	11.45	5.85
	16	15.02	5.91
Colorado	4	3.97	3.81
	8	7.92	7.87
	12	11.83	10.78
	16	15.71	12.55

Table 4: Average returned results per query varying requested paths k ($\theta = 50\%$) for SVP$^+$ and ESX.

7. CONCLUSIONS

We studied the problem of alternative routing on road networks and, in particular, the efficient computation and approximation of k-SPwLO queries. First, we proposed MultiPass, an exact algorithm which builds upon and optimizes the existing OnePass algorithm by employing one additional pruning criterion. Our experiments showed that MultiPass is the most efficient exact method for evaluating k-SPwLO queries as it outperforms OnePass for nearly every combination of the θ and k parameters and, in most cases, by a large margin. To achieve scalability though, we also introduced two approximate algorithms. OnePass$^+$ employs ideas from both OnePass and MultiPass and achieves to compute a set of dissimilar

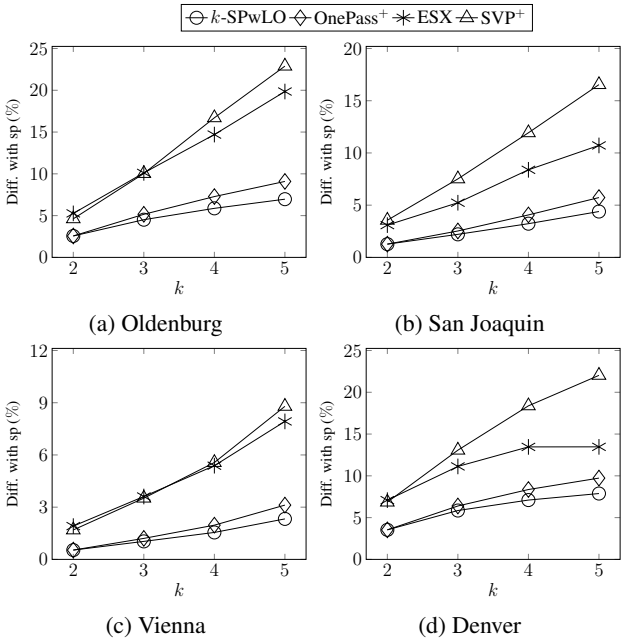


Figure 11: Result quality varying requested paths k ($\theta = 50\%$).

paths which, in terms of average length, is very close to the exact solution. ESX, our second approximate algorithm, computes alternatives by incrementally removing edges from the road network and running shortest path queries. Through an analytical experimental evaluation we showed that (a) MultiPass is the fastest exact algorithm, outperforming the existing OnePass, (b) OnePass$^+$ is significantly faster than MultiPass while its result set is close to the exact solution, and (c) in contrast to the other algorithms, ESX is scalable, i.e., it can compute approximate k-SPwLO queries for large road networks and large values of k.

In the future, we plan to extend the definition of alternative routing by considering additional constraints and criteria besides the overlap between paths and their length. We also plan to perform a qualitative study to identify which criteria users value more when deciding upon which route to follow. Finally, we plan to investigate the computation of multiple dissimilar paths on different types of networks such as social networks and web graphs.

8. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Alternative Routes in Road Networks. *Journal of Experimental Algorithmics*, 18:1–17, 2013.

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

[3] R. Bader, J. Dees, R. Geisberger, and P. Sanders. Alternative Route Graphs in Road Networks. In *Proc. of the 1st Int. ICST Conference on Practice and Theory of Algorithms in Computer Systems (TAPAS)*, pages 21–32, 2011.

[4] A. Bernstein and D. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proc. of the 41st ACM Symposium on Theory of Computing*, pages 101–110, 2009.

[5] Cambridge Vehicle Information Technology Ltd. Choice Routing, 2005.

[6] T. Chondrogiannis, P. Bouros, J. Gamper, and U. Leser. Alternative routing: k-shortest paths with limited overlap. In *Proc. of the 23rd ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, pages 68:1–68:4, 2015.

[7] T. Chondrogiannis and J. Gamper. ParDiSP: A partition-based framework for distance and shortest path queries on road networks. In *Proc. of the 17th IEEE Int. Conf. on Mobile Data Management*, pages 242–251, 2016.

[8] D. Delling and W. Dorothea. Pareto Paths with SHARC. In *Proc. of the 8th Symposium on Exterimental Algorithm (SEA)*, pages 125–136, 2009.

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

[10] B. Fox. K-th shortest paths and applications to the probabilistic networks. *ORSA/TIMS Joint National Mtg.*, 23:B263, 1975.

[11] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies : Faster and simpler hierarchical routing in road networks. In *Proc. of the 7th Int. Workshop on Experimental Algorithms (WEA)*, pages 319–333, 2008.

[12] Y.-J. Jeong, T. J. Kim, C.-H. Park, and D.-K. Kim. A Dissimilar Alternative Paths-search Algorithm for Navigation Services: A Heuristic Approach. *KSCE Journal of Civil Engineering*, 14(1):41–49, 2009.

[13] H.-P. Kriegel, M. Renz, and M. Schubert. Route skyline queries: A multi-preference path planning approach. In *Proc. of the 26th IEEE ICDE*, pages 261–272, 2010.

[14] Y. Lim and H. Kim. A Shortest Path Algorith for Real Road Network based on Path Overlap. *Journal of the Eastern Asia Society for Transportation Studies*, 6:1426 – 1438, 2005.

[15] E. Q. Martins and M. M. Pascoal. A new implementation of yen's ranking loopless paths algorithm. *4OR: A Quarterly Journal of Operations Research*, 1(2):121–133, 2003.

[16] K. Mouratidis, Y. Lin, and M. l. Yiu. Preference queries in large multi-cost transportation networks. In *2010 IEEE 26th ICDE*, pages 533–544, 2010.

[17] A. Paraskevopoulos and C. Zaroliagis. Improved Alternative Route Planning. In *Proc. of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS)*, pages 108–122, 2013.

[18] D. Sacharidis and P. Bouros. Routing directions: keeping it fast and simple. In *Proc. of the 21st ACM SIGSPATIAL Int. Conf. on Advances in Geographic Information Systems*, pages 164–173, 2013.

[19] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proc. of the 2008 ACM SIGMOD Conf.*, pages 43–54, 2008.

[20] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *Proc. of the 25th IEEE ICDE*, pages 652–663, 2009.

[21] C. Sommer. Shortest-path queries in static networks. *ACM Computing Surveys*, 46(4):1–31, 2014.

[22] L. Valiant. The Complexity of Enumeration and Reliability Problems. *Siam Journal of Computing*, 8(3):410–421, 1979.

[23] K. Xie, K. Deng, S. Shang, X. Zhou, and K. Zheng. Finding alternative shortest paths in spatial networks. *ACM Transactions on Database Systems*, 37(4):29:1–29:31, 2012.

[24] J. Y. Yen. Finding the K Shortest Loopless Paths in a Network. *Management Science*, 17(11):712–716, 1971.

[25] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest Path and Distance Queries on Road Networks: Towards Bridging Theory and Practice. In *Proc. of the 2013 ACM SIGMOD Conf.*, pages 857–868, 2013.