Efficient Algorithms for Route Planning Problems on Road Networks

PH.D. THESIS IN COMPUTER SCIENCE

DOKTORARBEIT IN INFORMATIK

TESI DI DOTTORATO DI RICERCA IN INFORMATICA

Theodoros Chondrogiannis

Relatore - *Doktorvater* - Faculty Advisor: Prof. Dr. Johann Gamper

Examination committee:
Prof. Kyriakos Mouratidis (chair), Singapore Management University
Prof. Dr. Matthias Renz, George Mason University
Prof. Johann Gamper (secretary), Free University of Bozen-Bolzano
Prof. Dr. Günther Specht (chair substitute member), University of Innsbruck
Prof. Dr. Peer Kröger (substitute member), Ludwig Maximillians University of München
Prof. Prof. Alessandro Artale (internal substitute member), Free University of Bozen-Bolzano

Keywords: Shortest Path, Graph Partitioning, Spatial Network Queries, WebGIS, Multimodal Networks, Alternative Routing

ACM categories: DATA STRUCTURES, Algorithms, Spatial databases and GIS

Acknowledgments

First and foremost, I want to thank my supervisor Prof. Johann Gamper for all his contributions of time, ideas and funding to make my PhD experience exciting and productive. He has taught me how deepened research in the field of computer science is done. The joy and enthusiasm he has for his research was truly inspiring for me.

Second, I would also like to thank Dr. Panagiotis Bouros and Prof. Dr. Ulf Leser for their contributions to my work. I am grateful to them for inviting me to Humboldt-Universität zu Berlin during my study period abroad. They taught me how important it is to collaborate and exchange ideas with other researchers in order to conduct top level research. My collaboration with them was an invaluable experience.

Third, I would like to thank all of my group members and colleagues (current and former ones), for their valuable inputs during the inspirational discussions in our seminars or in the coffee breaks. Many thanks also to my colleagues from the University of Zurich and the University of Salzburg for the valuable inputs during our annual retreat seminars.

I would also like to thank Roberto Cavaliere and Patrick Ohnewein from IDM Südtirol for their collaboration and their feedback during the development of the prototype system MoTrIS presented in this thesis.

Also many thanks to Maria for her constant support and for putting up with me during the last year of my PhD studies, especially during the time I was writing my thesis.

Lastly, I would like to thank my family for all their love and encouragement. Especially my parents Petros and Eleftheria, and my grandfather Nikos, who raised me with a love of science and taught me how to pursue with passion any targets I set in my life. Thank you.

Bolzano, 15/5/2017
Theodoros Chondrogiannis

Abstract

Route planning services have become very popular over the past decade. The increased availability of road network data has triggered the development of a variety of new applications. In this spirit, the overall goal of this thesis is to propose efficient algorithms for tackling route planning problems on road networks. In particular, we study the following two problems: (a) the efficient processing of distance and shortest path queries and (b) the computation of dissimilar yet short alternative paths.

Distance and shortest path queries can be used as building blocks for more complex queries on road networks. Therefore, the efficient processing of both types of queries is of great importance. However, most state-of-the-art methods focus solely on one type of query and do not efficiently support the other. To address this shortcoming, we propose the *Partition-based framework for Distance and Shortest Path queries* (ParDiSP), which combines ideas from state-of-the-art approaches in a novel way and provides exceptional query times for both distance and shortest path queries. Our framework first partitions the road network into components. Then exploits the properties of the partitioning for precomputations to boost query processing. ParDiSP answers distance queries solely by accessing precomputed distance tables and significantly limits the part of the road network that has to be accessed to process shortest path queries. A detailed experimental evaluation shows that: ParDiSP outperforms the state-of-the-art for shortest path queries, is comparable to the state-of-the-art for distance queries, and, for mixed query loads containing both distance and shortest path queries, outperforms even a combination of the best methods for each query type.

In many real-world scenarios though, returning only the shortest path is not enough. Most commercial navigation systems recommend, apart from the shortest path, a number of alternative paths with different characteristics, leaving the final decision to the user. In this context, we formally introduce the k-*Shortest Paths with Limited Overlap* (k-SPwLO) problem seeking to compute k alternative paths which are as short as possible and sufficiently dissimilar based on a user-controlled similarity threshold. We present three algorithms that examine the paths from a source s to a target t in increasing order of their length and progressively construct the result set. First, the baseline algorithm BSL builds upon the computation of K-shortest paths. Second, OnePass traverses the network to expand every path from the source that qualifies the similarity constraint. Third, MultiPass traverses the network $k-1$ times and employs two

pruning criteria to reduce the number of paths that have to be examined. In an extensive experimental evaluation we show that MultiPass is the fastest algorithm for processing k-SPwLO queries as it outperforms both BSL and OnePass and, in most cases, by a large margin.

To achieve scalability, we also propose two heuristic 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 OnePass$^+$ runs significantly faster than MultiPass with its result being close to the exact solution, and ESX is faster than OnePass$^+$ (though slightly less accurate) and scales for large road networks and large values of k.

Finally, we present MoTrIS, a service-oriented *Multimodal Transport Information System* for routing services on road and transportation networks. We have implemented and integrated two of the algorithms presented in this thesis into MoTrIS: ParDiSP for processing distance and shortest path queries, and ESX to recommend alternative paths. In addition, we have implemented an algorithm for processing distance and shortest path queries on multimodal transportation networks. MoTrIS enables developers to create customized routing services and submit queries via a public API. We also show that MoTrIS is highly extensible. New algorithms can be easily integrated to support the processing of more types of routing queries.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Motivation and Problem Setting

The increasing popularity of online mapping services such as Google Maps, Bing Maps and OpenStreetMap, has motivated research on the efficient processing of various types of queries on (spatial) road networks. Route planning queries are frequently employed by users to plan trips by foot, car or public transportation. Apart from being widely used by mapping and navigation services, route planning queries are often used as building blocks for more complex queries and, hence, are an integral part of systems and applications in various fields, e.g., transport services and crisis management systems. For these reasons, the efficient processing of such queries has recently attracted a considerable interest from both research and industry.

1.1.1 Modelling and Querying Road Networks

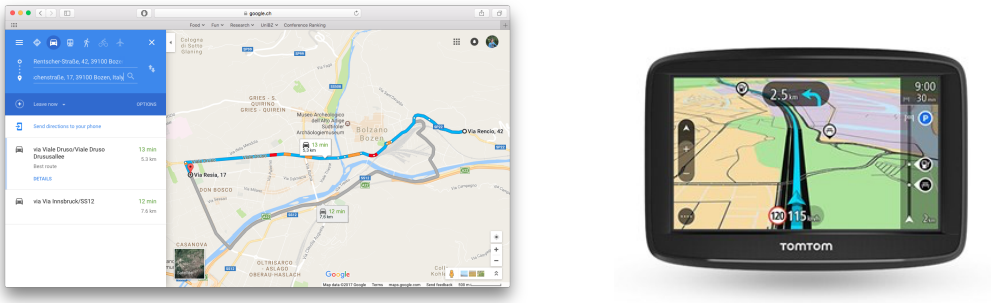
The most common way to represent a road network is as an *undirected (or directed) weighted graph*. Let $G = (N, E)$ be a weighted graph representing a *road network* with nodes N and edges $E \subseteq N \times N$, where nodes represent road intersections and edges represent road segments. Each edge $e = (n_i, n_j)$, $e \in E$, has an assigned weight $\ell(e)$, which captures the cost of moving from node n_i to node n_j, e.g., distance or travel time. 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 $\langle e_1{=}(s, n_i), \ldots, e_m{=}(n_j, t)\rangle$. The *length* $\ell(p)$ of a path p equals the sum of the weights of all contained edges, i.e.,

$$\ell(p) = \sum_{\forall e \in p} \ell(e).$$

The *shortest path* $p_s(s{\rightarrow}t)$ between two nodes s and t is the path that has the shortest length among all paths that connect s and t. The length of the shortest path is also termed the *(network) distance* between s and t, i.e., $d(s, t) = \ell(p_s(s{\rightarrow}t))$.

1.1.2 Distance and Shortest Path Queries

Given two locations s and t in a road network, a distance query returns the network distance from s to t, while a shortest path query computes the shortest actual route the user has to follow to reach t starting from s. These two queries find applications in various fields. For example, Figure 1.1a illustrates GoogleMaps[1], the most popular web mapping service, while Figure 1.1b shows a GPS navigator for cars developed by TomTom[2]. Distance and shortest path queries are an integral part of both applications.



(a) GoogleMaps

(b) TomTom navigator for cars

Figure 1.1: Two applications offering routing services

The classic solution for distance and shortest path queries is *Dijkstra's algorithm* [32]. Given a road network G, a starting node s and a target node t, Dijkstra's algorithm traverses the vertices in G in ascending order of their distance from s. Despite its simplicity, Dijkstra's algorithm is, however, inefficient for large road networks. To achieve better performance, a variety of preprocessing methods have been proposed, e.g., [11, 80, 86]. In particular, for distance queries the state-of-the-art methods are Bounded-hop Methods [2, 7, 20], which reduce the processing of distance queries to a number of lookups on precomputed distance tables. For shortest path queries, the most efficient methods are Hierarchical Methods [36, 73, 94], which precompute a hierarchy of shortcuts and employ it to process queries.

The aforementioned state-of-the-art methods come with a particular shortcoming. All approaches focus on a single type of query, either distance or shortest path queries, and do not efficiently support the other. For instance, to compute a shortest path query, bounded-hop methods execute an A^*-search using the real distance to the target as a lower bound. Such an operation requires the processing of a (usually very large) set of distance queries, making the processing of shortest path queries orders of magnitude slower than the processing of distance queries. On the other hand, despite offering superior query performance for shortest path queries, hierarchical methods are not as efficient as bounded-hop methods for distance queries as they require a sort of scan on the hierarchy. It has been shown in [11] that the state-of-the-art hierarchical methods are orders of magnitude slower than the state-of-the-art bounded-hop methods for distance queries.

However, various applications require efficient query processing for both types of queries. A representative example is itinerary planning [14]. Given a set of points of interest and a

[1] https://maps.google.com/
[2] https://www.tomtom.com/

time budget, itinerary planning computes one or more itineraries that visit as many points of interest as possible within the given time budget. During the planning phase, many distance queries need to be answered between points of interest, and once valid combinations of points are found, the distinct paths connecting those points must be determined. In such scenarios, to achieve optimal performance, the maintenance of two separate structures is necessary. This leads to the first problem addressed in this thesis:

Problem 1. *Existing solutions focus on one query type, either distance or shortest path queries, and do not efficiently support the other.*

1.1.3 Alternative Routing

In many real-world scenarios, determining solely the shortest path is not enough. Most commercial route planning applications and navigation systems recommend alternative paths 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 routing is 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 which overlap as little as possible.

Consider the scenario illustrated in Figure 1.2, which shows three distinct paths from location s to t in the city center of Bolzano. The solid/black line indicates the shortest path from s to t, whereas the dotted/red line indicates the next path in length order. Notice how similar these two paths are. On the other hand, the green/dashed line indicates a third path which is clearly the longest but is significantly different from the shortest path. In practice, the paths cover very distant parts of the city's road network. In applications like the ones discussed in the previous paragraph, only the green/dashed path can be considered as a good and useful alternative to the shortest path.

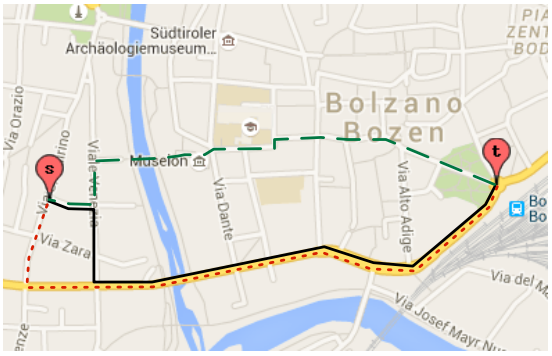


Figure 1.2: Illustration of alternative paths

Existing works for recommending alternative paths come with two important shortcomings. First, many approaches define alternative paths based on their similarity to the shortest path,

which may result in alternative paths being very similar to each other and, hence, of limited interest to the user. Second, most existing methods typically give no guarantees regarding the length of the alternative paths. Naturally, the users have more interest in paths that are as short as possible. This leads to the second problem addressed in this thesis:

Problem 2. *There exist no solutions that compute k alternative paths that are as short as possible and at the same time sufficiently dissimilar to each other.*

1.2 Objectives and Contributions

The overall goal of this thesis is to propose efficient algorithms for route planning on road networks. In particular, we study two different problems related to route planning: the *efficient processing of distance and shortest path queries*, and *the computation of dissimilar yet short alternative paths*. In what follows, we summarize the technical contributions of this thesis.

1.2.1 ParDiSP Framework

We propose a *Partition-based framework for Distance and Shortest Path queries* (ParDiSP) that efficiently computes both types of queries. ParDiSP combines ideas from state-of-the-art approaches in a novel way, thereby providing exceptional query times for both distance and shortest path queries. More specifically, ParDiSP partitions the road network into k components and precomputes auxiliary information. The precomputed information enables ParDiSP to processes distance queries as a bounded-hop method, i.e., by executing a number of table look-ups. For processing shortest path queries, ParDiSP utilizes the result of a distance query to identify the subset of the road network that needs to be accessed to process the given query. Furthermore, in contrast to most existing methods, ParDiSP provides flexibility, i.e, the number k of components can be used to adjust the trade-off between performance and space overhead.

In practice, ParDiSP exploits the properties of the road network partitioning to precompute both distance tables and graph structures. ParDiSP answers distance queries by combining distances from exactly three precomputed distance tables. Shortest path queries are decomposed into three segments, which can be computed in parallel by accessing only a small part of the original road network. To compute the longest of the three segments, ParDiSP employs a state-of-the-art hierarchical method over a precomputed part of the road network. We evaluate ParDiSP in terms of performance and preprocessing cost and show that: ParDiSP outperforms two state-of-the-art solutions for shortest path queries; it is comparable to the state-of-the-art for distance queries; and, for mixed query loads containing both distance and shortest path queries, ParDiSP outperforms a combination of the best methods for each query type, while its space requirements are significantly smaller.

1.2.2 k-SPwLO Queries

We propose a novel definition of alternative paths. More specifically, we recommend a set of k paths (including the shortest path) such that every path in the result is (a) sufficiently dissimilar to all shorter paths in the set and (b) as short as possible. We formalize this form of alternative routing as the *k-Shortest Paths with Limited Overlap* (k-SPwLO) problem. We

also present three algorithms to evaluate k-SPwLO queries. First, the baseline algorithm BSL builds upon the computation of the K-shortest paths. Second, OnePass traverses the road network once expanding every path from the source that qualifies the similarity constraint. Third, MultiPass extends and improves OnePass by employing an additional pruning criterion and processes queries by traversing the network $k-1$ times. In an extensive experimental evaluation we show that MultiPass is the fastest algorithm for processing k-SPwLO queries outperforming both BSL and OnePass.

Despite MultiPass being the fastest exact solution for processing k-SPwLO queries, the algorithm is not practical for large road networks. Therefore, we also propose two heuristic algorithms[3] that trade result quality for efficiency. Our first heuristic algorithm, OnePass$^+$, employs the pruning power of MultiPass, but, similar to OnePass, traverses the road network only once. The second heuristic algorithm, ESX, reduces the search for alternative paths to a set of shortest path queries by incrementally removing edges from the road network. In the experimental evaluation, we compare the heuristic algorithms with MultiPass, the most efficient exact solution, in terms of performance and result quality. OnePass$^+$ runs significantly faster than MultiPass and its result is close to the exact solution, while ESX is faster than OnePass$^+$ (though slightly less accurate) and it scales for large road networks and large values of k.

1.2.3 MoTrIS Framework

Finally, we present MoTrIS, a Multimodal Transport Information System, which integrates two of the algorithm presented in this thesis: ParDiSP for processing distance and shortest path queries and ESX to recommend alternative routes. Apart from route planning on road networks, MoTrIS tackles the challenge of combining different types of networks, i.e., road and transportation networks, into a single multimodal network. Developers can create customized routing services over specific regions and with specific transportation modes. MoTrIS also provides a public API which enables developers to submit queries and integrate the functionality directly into their applications. We also show that MoTrIS is highly extensible. New algorithms can be easily integrated to support the processing of more types of routing queries on road and multimodal transportation networks.

1.3 Publications

The results presented in this thesis have been published at the following conferences:

- T. Chondrogiannis and J. Gamper, Exploring Graph Partitioning for Shortest Path Queries on Road Networks, In *Proceedings of the 26th Grundlagen von Datenbanken (GvDB'14)*, pages 71-76, 2014

- T. Chondrogiannis, P. Bouros, J. Gamper and U. Leser, Alternative Routing: k-Shortest Paths with Limited Overlap, In *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'15)*, pages 68:1-68:4, 2015

[3] In [18] the term "approximate algorithms" was used instead of "heuristic algorithms". Since our algorithms come with no error guarantees, we changed the terminology as the term "heuristic" is more accurate.

- T. Chondrogiannis and J. Gamper, ParDiSP: A Partition-based Framework for Distance and Shortest Path Queries on Road Networks, In *Proceedings of the 17th IEEE International Conference on Mobile Data Management (MDM'16)*, pages 242-251, 2016

- T. Chondrogiannis, J. Gamper, R. Cavaliere and P. Ohnewein, MoTrIS: A Framework for Route Planning on Multimodal Transportation Networks, In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'16)*, pages 82:1-82:4, 2016

- T. Chondrogiannis, P. Bouros, J. Gamper and U. Leser, Exact and Approximate Algorithms for Finding k-Shortest Paths with Limited Overlap, In *Proceedings of the 20th International Conference on Extending Database Technology (EDBT'17)*, pages 414-425, 2017

1.4 Thesis Organization

Chapter 2. This chapter discusses related research work in the areas of distance and shortest path query processing as well as alternative routing. In addition, since our prototype system is part of the thesis, we review state-of-the-art implementations with similar functionalities.

Chapter 3. This chapter presents the *Partition-based Framework for Distance and Shortest Path Queries* (ParDiSP) on road networks, which combines ideas from state-of-the-art approaches for distance and shortest path processing in a novel way.

Chapter 4. In this chapter, we first introduce the k-SPwLO query for alternative routing on road networks. We also propose and evaluate three algorithms for processing k-SPwLO queries which examine the paths from the source node in increasing order of their length and progressively construct the result set.

Chapter 5. In this chapter, we study heuristics to compute k-SPwLO queries and we propose two heuristic algorithms which trade accuracy for efficiency. We also present the results of an extensive experimental evaluation, comparing the heuristic algorithms with the most efficient exact solution, both in terms of performance and result quality.

Chapter 6. This chapter presents MoTrIS, our service-oriented platform which integrates our algorithms. We describe the system architecture and we present use-cases which demonstrate the main functionalities of the platform.

Chapter 7. This chapter summarizes the achieved results and points out some interesting directions for future research work.

CHAPTER 2

Related Work

In this chapter, we discuss related research work, which we divide in four parts. The first part in Section 2.1 focuses on state-of-the-art preprocessing-based methods for distance and shortest path query processing on static road networks. The second part in Section 2.2 reviews algorithms for computing alternative routes on road networks. The third part in Section 2.3 discusses the problem of route planning on multimodal transportation networks, i.e., using different transportation modes. Finally, Section 2.4 provides an overview of popular open source and commercial routing applications and systems.

2.1 Distance and Shortest Path Queries on Road Networks

In the last twenty years, the processing of spatial network queries has attracted considerable interest and a variety of methods to process such queries have been proposed. In [65], a storage model for spatial network databases has been proposed along with algorithms for various spatial network queries, i.e., k-nearest neighbor (kNN) queries, range queries and closest pair queries. For nearest neighbor and kNN queries in particular, various approaches have been proposed [23, 30, 49, 65, 72]. Many algorithms have also been proposed for processing variants of nearest neighbor queries, i.e., aggregate nearest neighbor queries [64, 90], group nearest neighbor queries [63] and reverse nearest neighbor queries [91].

Among spatial network queries, distance and shortest path queries are the most fundamental and among the most popular queries. The classical solution for processing such queries is *Dijkstra's algorithm* [32]. Given a road network $G = (N, E)$, a starting node s and a target node t, Dijkstra's algorithm traverses the nodes in G in ascending order of their distances from s. In practice, Dijkstra's algorithm works as follows: each node is associated with a tentative distance which is initially set to $+\infty$ for all nodes, apart from s for which the tentative distance is set to 0. Starting from s, the algorithm expands all outgoing edges of s, checks whether by traversing the current edge the distance to the adjacent node is lower than the current tentative distance and, if necessary, updates the tentative distances of the adjacent nodes. Once all outgoing edges are visited, the node is called expanded. At each iteration, the node with the smallest tentative distance that has not yet been expanded is examined. The expansion terminates either when the target node t is encountered or there are no more nodes to expand, in which case there is no path connecting nodes s and t.

A simple improvement to Dijkstra's algorithm is to perform a *bidirectional search* [67], i.e., to execute Dijkstra's algorithm simultaneously from the source s and backwards from the target t. The search stops when a valid meeting point of the two shortest path trees is determined. Bidirectional search can improve the execution time of Dijkstra's algorithm by a factor of two. However the improvement is not sufficient; like Dijkstra's algorithm, bidirectional search is also impractical for large road networks.

In order to make distance and shortest path queries scalable for large road networks, a variety of preprocessing based methods have been proposed [11, 80]. Such methods aim at precomputing auxiliary information offline, inflicting a relatively high one time cost, and employ the precomputed information in order to reduce the query processing time. Depending on the precomputed information and the way each method utilizes it, we can classify existing methods into five main categories: *Speed-up methods*, *Spatial Coherence-based methods*, *Bounded-hop methods*, *Hierarchical methods*, and *Partition-based methods*. In what follows, we present the most important methods in each category.

2.1.1 Speed-up Methods

Speed-up or *goal-directed methods* employ a modified version of Dijkstra's algorithm along with heuristics to prioritize the expansion of nodes that are closer to the target. As Dijkstra's algorithm has to visit a very large part of the road network, speed-up methods aim at reducing the part of the network that the search algorithm has to expand.

A^*-search [40] is a classic goal-directed algorithm which employs lower bounds to reduce the search space and speed-up shortest path query processing. The lower bound is determined

by employing a heuristic function $h : N - \Re$ on the nodes of the input road network. The algorithm then employs a modified version of Dijkstra's algorithm setting the priority of each node n to $dist(s, n) + h(n, t)$ causing the nodes that are closer to the target to be visited first. Naturally, the tighter the lower bound, the less the nodes that the search algorithm is going to visit. For example, in the case where we have the tightest possible lower bounds, hence $h(n, t) = dist(n, t)$, then, for any shortest path query from s to t, A^*-search visits only nodes on the shortest path from s to t. Like Dijkstra's algorithm, A^*-search can also run in a bidirectional fashion too [38].

Common heuristics employed by *A^*-search* on road networks is the *Euclidean distance* and the *Manhattan distance*. However, such heuristics fail to take into consideration the structure of the road network and, therefore, in many cases, the improvement is minimal. An alternative way to obtain lower bounds is to employ landmarks. *Landmark-based A^*-search* (ALT) [38] precomputes distances from all nodes of the road network to a small subset of nodes, called landmarks. During the execution of a shortest path query from a source node s to a target node t, the lower bound of the distance from a node n visited by the algorithm to t is computed by employing the triangle inequality. More precisely, for any landmark l, we have $dist(n, t) \geq dist(n, l) - dist(t, l)$ and $dist(n, t) \geq dist(l, t) - dist(l, n)$. For each node n the algorithm always picks the tightest possible lower bound among all bounds computed using different landmarks. Apparently, the quality of the lower bounds depends heavily on the selection of landmarks during preprocessing; several techniques for selecting lanmarks have been proposed [68].

The ALT algorithm has been improved by incorporating reach labels [39] resulting in the *Reach-based ALT* (REAL) [37] algorithm. The reach of a node n is defined as $R_{st}(n) = min\{dist(s, n), dist(n, t)\}$. The shortest path search can be pruned at nodes with a reach too small to get to the source or the target. Although reach values are determined during the preprocessing phase, computing exact reaches requires the computation of the all-pairs shortest paths; such an operation is prohibitively expensive. The result of the query is correct even if the reach of a node represents an upper bound. Such upper bounds can be obtained much faster by computing partial shortest path trees.

Despite offering a significant improvement to Dijkstra's algorithm, speed-up methods are not efficient enough for large road networks. Speed-up methods are two to three orders of magnitude slower than state-of-the-art methods for distance and shortest path queries [11]. In contrast to state-of-the-art methods though, speed-up methods are usually space efficient, i.e., have much lower memory requirements.

2.1.2 Spatial Coherence-based Methods

Spatial coherence-based methods exploit the property of road networks that shortest paths are often spatially coherent, i.e., many shortest paths between different pairs of nodes share common parts. To illustrate the concept of spatial coherence, let us consider four locations s, s', t and t' on a road network. If s is close to s' and t is close to t', the shortest path from s to t is more likely to share nodes with the shortest path from s' to t'. Spatial coherence-based methods precompute the all-pair shortest paths and employ some data structure to index the paths and answer queries.

Spatially Induced Linkage Cognizance (SILC) [72, 74] precomputes and indexes the shortest paths between all pairs of nodes using a quad-tree [34]. For each node n of the input road

network, SILC computes the shortest paths from n to all the other nodes of the road network. Then SILC imposes a grid on the road network and splits the grid into areas such that: (i) every area contains exactly one neighbor of n and (ii) the shortest path from n to any node inside an area passes through the neighbor of n assigned in the same area. Hence, given a shortest path query from a source node s to a target node t, starting from s, SILC can identify which of the neighbors of the node examined at each iteration lies on the shortest path to t. It has been shown that every lookup of SILC requires $\mathcal{O}(\log n)$, while the number of lookups depends on the number of nodes on the shortest path.

Path-Coherent Pairs Decomposition (PCPD) [77] imposes a grid over the road networks and precomputes the shortest paths between all pairs of nodes, like SILC. Then, PCPD stores all paths in a concise format called *path coherent pairs*. A path coherent pair is a triple $\langle A, B, n\rangle$, where A and B are two disjoint square regions of the grid and n is a node of the road network such that all shortest paths $p(s{\rightarrow}t)$ from some node s located inside A to some node t located inside B pass through node n. Similar to SILC, in order for PCPD to retrieve a shortest path, it requires linear time to the size of the path, i.e., the same number of lookups as the number of nodes on the path. Each lookup using the aforementioned path coherent pairs scheme costs $\mathcal{O}(m)$ where m is the number of unique path coherent pairs computed during preprocessing.

Spatial coherence is also employed by *distance oracles*, an efficient approach for approximate distance query processing. In [82], the authors propose an $(1+\epsilon, 0)$-approximate distance oracle, a multi-level approach which answers approximate distance queries in almost constant time while inflicting relatively low space overhead. Another method for approximate distance query processing is the ϵ-approximate distance oracle presented in [76]. The oracle requires $\mathcal{O}(n/\epsilon^2)$ space and retrieves the approximate network distance in $\mathcal{O}(\log n)$ time using a B-tree. Although very efficient, methods based on distance oracles cannot answer exact distance queries and approximate distance query processing is out of the scope of this thesis.

The main shortcoming of spatial coherence-based methods is that they incur significant preprocessing time and space overhead. In particular, although SILC and PCDP offer exceptional query times for both distance and shortest path queries, their space overhead is exponential to the number of nodes [86]. Both methods have prohibitively high memory requirements even when applied on road networks with less than a million nodes. Hence, spatial coherence-based methods are clearly not practical for large road networks with several millions of nodes.

2.1.3 Bounded-hop Methods

Bounded-hop methods precompute and store distances between selected pairs of nodes into a set of distance tables. The distance between any pair of nodes is computed by accessing only the precomputed distance tables, and then the shortest path is retrieved by running an A^*-search from the source to the target using the exact distance as a lower bound; hence, the retrieval of the shortest path is linear to the size (number of nodes) of the path.

The *2-hop cover* [20] is an early theoretical distance labeling scheme which works as follows. During preprocessing, every node n of the road network is assigned with a set of labels $L(n)$ containing distances to selected nodes such that for any pair of nodes s and t, $L(s) \cap L(t)$ contains at least one node on the shortest path from s to t. This method ensures that the shortest path from s to t is covered by a node in $L(s) \cap L(t)$, i.e., the distance from s to t can be found by combining only distances in $L(s)$ and $L(t)$. The 2-hop cover is the the minimum set of nodes that can be used as labels to compute the shortest paths between any pair of nodes on the

road network. However, the computation of the 2-hop cover requires the computation of the all-pair shortest paths, thus is prohibitively expensive.

Hub Labeling (HL) [2, 3], is a labeling technique for distance queries on road networks based on the theory of 2-hop cover. Each node is associated with a label $L(n)$ which contains distances to a set of nodes, i.e., the hubs of n. HL guarantees that the cover property of the 2-hop cover is obeyed; hence, any distance $dist(s,t)$ can be determined in linear time by combining exactly two distance tables, i.e., $L(s)$ and $L(t)$. Apparently, the performance of distance queries depends on the size of the distance tables. Even though HL does not guarantee that the size of the distance tables is minimum, the proposed label selection strategy leads to small distance tables and, therefore, exceptional query times.

Another popular bounded-hop technique is *Transit Node Routing* [12, 8]. In contrast to HL, TNR combines three distance tables to compute the distance between two nodes. Given a road network $G = (N, E)$, during preprocessing TNR selects a small set $T \subseteq N$ of *transit nodes* and computes all pairwise distances between them. Then, the algorithm assigns to each node $n \in N$ a set of *access nodes* $A(n) \subseteq T$ and precomputes the distances from and to every access node in the assigned set. To determine the access nodes, the algorithm imposes a grid on the road networks such that each grid cell contains at most one node. TNR chooses as access nodes of a given node, the nodes that are located inside neighboring grid cells. The result for a given distance query $q(s,t)$ is $dist(s,t) = min\{d(s,a_s) + d(a_s,a_t) + d(a_t,t)\}$, where $a_s \in A(s)$ and $a_t \in A(t)$.

Pruned Highway Labeling (PHL) [7] introduces a cost efficient preprocessing method to select labels. Although a pure bounded-hop technique, PHL combines features from different aspects in the literature. In particular, PHL repeatedly separates input road networks by shortest paths, and then stores distances from nodes to the shortest paths. This is an idea also employed by distance oracles for approximate distance query procesing [82]. PHL also utilizes the concept of highways [73, 78] in order to compute the set of labels assigned to each node. During preprocessing, PHL computes labels such that any shortest path from a node s to a node t can be expressed as a sequence of three paths $\langle p(s,u), p(u,v), p(v,t)\rangle$, where path $p(u,v)$ is a highway; distance $dist(u,v)$ is precomputed while u and v are stored as labels of s and t along with their respective distances from s and t.

Bounded-hop methods and, in particular, HL [3], are the most efficient approaches for processing distance queries. Experiments presented in [11] have shown that HL requires less than a microsecond to process distance queries even on continental road networks. In comparison to other approaches, HL is eight orders of magnitude faster than Dijkstra's algorithm and five times faster than TNR. However, HL requires a lot of preprocessing time and memory. In terms of performance, the method closest to HL is PHL [7]. PHL answers distance queries approximately in one microsecond, i.e., it is slightly slower than HL. However, PHL requires significantly less memory and preprocessing time than HL.

Although very efficient for distance queries, bounded-hop methods are not as efficient for shortest path queries. As we mentioned before, to retrieve the shortest path, bounded-hop methods execute an A^*-search from the source to the target using the real distance to the target as a lower bound. Processing a shortest path query is essentially equivalent to processing a (large) set of distance queries, which depends heavily on the network structure and the length of the shortest path.

2.1.4 Hierarchical Methods

Hierarchical methods impose a hierarchical structure on the road network and process queries by running a bidirectional search over the precomputed structure. The hierarchical structure usually consists of shortcuts which aim at abstracting the main arteries of the road network. By traversing the hierarchy instead of the original road network, the search space for computing shortest path queries is significantly reduced.

Highway Hierarchies (HH) [73] are based on the following observation. Certain edges of the road network, i.e., the highway edges, tend to be on many shortest paths where the source and the target are far apart. HH employs two sub-routines to build a hierarchy of shortcuts on the road network. Node reduction introduces shortcuts to bypass nodes of low degree, i.e., one or two. Edge reduction adds shortcuts to bypass non-highway edges. To identify non-highway edges, HH performs a local check for each edge of the road network. To process queries, HH employs a modified version of bidirectional search [67], which avoids expanding most of the non-highway edges.

Contraction Hierarchies (CH) [36] is a direct successor of HH, which organizes the nodes on a road network into a hierarchy, based on their relative importance. During preprocessing, CH determines the importance of each node by employing a set of heuristics, generates a node order and contracts each node on the road network following the precomputed order. The result of the contraction process is the construction of a multi-level hierarchical structure of shortcuts. Like HH, to process shortest path queries a modified bidirectional search is executed over the hierarchical structure. At each step, the search algorithm visits nodes that are on the same or a higher level than the last expanded node.

Arterial Hierarchy (AH) [94] is a method inspired by CH, which precomputes shortcuts by imposing a grid on the road network. AH organizes the nodes of the road network into levels such that during query processing the network traversal will always visit a node from a higher level than the current one. AH is the only hierarchical method which comes with theoretical guarantees regarding the space overhead and the query processing time. AH builds the shortcut hierarchy in a way that it guarantees the levels of the hierarchy will be $\mathcal{O}(\log n)$. As a consequence, AH provides a time complexity of $\mathcal{O}(\log n)$ for processing queries.

Hierarchical methods and, in particular, AH and CH are the most efficient methods for shortest path queries. In [94] it is shown that AH outperforms CH in both distance and shortest path queries at the cost of extra space and preprocessing time. Despite offering superior query performance for shortest path queries though, hierarchical methods are not as efficient as bounded-hop methods for distance queries as they require a sort of scan on the hierarchy. It has been shown in [11] that CH is approximately three orders of magnitude slower than HL for distance queries.

2.1.5 Partition-based Methods

Partition-based methods first partition the input road network into a number of components. For this purpose, a third-party partitioning method is usually employed, which splits the nodes into balanced components while attempting to minimize the number of connecting edges, i.e., edges between border nodes of neighboring components. By employing the inherent properties of the partition, shortcuts and/or distance tables are precomputed to boost query processing. Most partition-based methods share some characteristics with methods from other categories,

i.e., speed-up, bounded-hop or hierarchical methods.

Precomputed Cluster Distances (PCD) [60] is a speed-up technique which partitions the input road network into components, precomputes the distances between all pairs of components and uses these distances to compute lower bounds. More specifically, PCD first partitions the road network into k components $\{C_1 \dots C_k\}$. During the preprocessing phase, the algorithm computes the distances between all pairs of components. The distance between two components C_i and C_j is defined as the minimum distance between two nodes $n_i \in C_i$ and $n_j \in C_j$. For computing a shortest path query from a source node $s \in C_S$ to a target node $t \in C_t$, PCD executes an A^*-search employing the distance between components to compute lower bounds. For any visited node $n \in C_n$, a valid lower bound on its distance to t is $dist(s, n) + dist(C_n, C_t) + dist(b_t, t)$, where b_t is the border node of C_t that is closest to t.

Arc Flags [48] partitions the road network into k components and assigns to each edge of the road network a vector of k bits (arc flags). The i^{th} bit of the vector is set if the edge lies on a shortest path to some node of component i. While processing a shortest path query from s to t, the search algorithm prunes edges which do not have the bit set for the component containing target t. The arc flags for a component i are computed by growing a backward shortest path tree from each border node (of component i), setting the i^{th} flag for all edges on the tree.

Hierarchical Encoded Path Views (HEPV) [45] is a hierarchical partition-based method which employs Spatial Partition Clustering (SPC) [42], a custom partitioning algorithm for road networks. First, HEPV partitions the input road network into components using SPC. During the preprocessing phase, the shortest path between every pair of border nodes is computed. HEPV stores and maintains the entire shortest path between two border nodes instead of the mere distance, in the form of a path view. Next, an auxiliary graph is created, which consists of several partial graphs. Each partial graph keeps all the path views which store shortest paths between the border nodes of its associated component. Next, HEPV partitions the resulting auxiliary graph into subgraphs and computes shortest paths in the same fashion in order to populate the next level of the hierarchy. The process continues until the top auxiliary graph is sufficiently small. To retrieve the shortest path between two nodes, HEPV retrieves and combines the partial paths from the appropriate components, usually from two different levels of the hierarchy. *HiTi Graphs* [46] also employ a similar approach to HEPV.

Customizable Route Planning (CRP) [26] partitions the road network into components and precomputes distances between border nodes in each component. To partition the input road network, CRP employs PUNCH [27], another graph partitioning algorithm tailored to road networks. To process queries, a modified bidirectional search algorithm is employed, which expands only the shortcuts and the edges in the source and the target component. In contrast to HEPV and HiTi, CRP stores only the distances between border nodes. Hence, for each shortcut on the shortest path, CRP has to retrieve the path from the original road network. Since the precomputed information is limited to one distance table per node, the memory requirements of CRP are quite low. Hence, CRP is able to handle various arbitrary metrics by precomputing more distance tables per node.

Finally, *G-tree* [93], *PTree* [84] and *G*-tree* [5] use the same hierarchical partition-based structure for processing spatial network queries. All these methods partition the road network recursively and construct a hierarchy of components. For components at the lowest-level the distances between every node and all border nodes are stored, whereas for the other components only the distances between pairs of border nodes are stored. All approaches handle distance

queries using precomputed distance tables. In addition, G-Tree uses the structure to compute nearest neighbor queries, PTree employs dynamic programming to retrieve shortest paths, and G*-tree [93] employs a different partitioning strategy to process k-closest pairs queries.

2.2 Alternative Routing on Road Networks

A first take on providing alternative routes on road networks is to solve the K-shortest (simple) paths problem. It has been shown that Yen's algorithm, initially proposed in [89] and further optimized in [41, 59], is the most efficient algorithm for computing the $K-$shortest paths. The main idea behind Yen's algorithm is that, given a source node s and a target node t, in order to compute the K^{th} shortest path from s to t we must have already computed the first $K-1$ shortest paths. Hence, the first step of Yen's algorithm is to employ any traditional algorithm to compute the shortest path from s to t, e.g., Dijkstra's algorithm. By analyzing the shortest path, a candidate path will be generated for each node of the shortest path, and the shortest among the candidate paths will be chosen as the next shortest path. The process continues until the K^{th} shortest path has been determined. In practice , the $K-$shortest paths cannot be employed for alternative routing. In most cases the $K-$shortest paths share large stretches and, therefore, they are of little practical value as alternative routes.

In [44], the authors propose an algorithm which directly extends Yen's algorithm [89] to compute k-dissimilar paths on road networks. Given a source node s and target node t, a length limit x and a similarity threshold y, the goal is to incrementally compute k paths, the length of which does not exceed x and their pairwise similarity does not exceed y. The similarity between two paths is defined based on the length of their shared edges. The shortest path from s to t is always included in the final result set. At each round, the algorithm computes a set of candidates, selects the most dissimilar path to the previously computed ones as the top candidate and, if the candidate path satisfies the x, y constraints, it is added to the result set. The algorithm terminates when k valid paths have been found. Although pairwise dissimilarity is guaranteed, the algorithm does not aim at minimizing the length of the recommended paths.

In what follows, we describe different approaches presented in the bibliography on how to generate alternative paths.

2.2.1 Penalty-based Methods

Penalty-based methods focus on the process of generating a set of paths different from the shortest path without, however, providing a formal definition of alternative routing. The main idea of penalty-based methods is to compute the shortest path and then update the road network by adding a penalty on the weights of the edges that lie on the shortest path. For example, in [6] the authors propose a method which doubles the weight of each edge that lies on the shortest path. The alternative paths are computed by repeatedly running a shortest path algorithm, such as Dijkstra's algorithm, on the input road network, each time with the updated weights. A similar approach is adopted by [52] where the penalty is computed in terms of both the path overlap and the total turning cost, i.e., how many times the user has 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 iteration. In general, using a large penalty would

result in dissimilar but possibly very long alternative paths. On the other hand, using a small penalty 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 and, hence, their efficiency depends upon the user's choice of the penalty.

2.2.2 Candidate Set-based Methods

Another approach for alternative routing is to first compute a large set of candidate paths. Then, during a post-processing step, we examine the candidates with respect to a number of constraints (e.g., their length or the nodes they cross) and determine the final result set. For example, the *Plateaux* method [1] aims at computing paths that cross different highways of the road network. Since highways rarely overlap, the produced paths are dissimilar. The problem and the proposed method in [1] were revisited and formally defined later in [9], which introduces the concept of alternative graphs having the same functionality as the plateaus, and were further improved in [66].

In [4], the authors use the set of *single-via paths* as their candidate set to compute alternative paths. The proposed method first selects a subset of the network nodes called *via-nodes*, i.e., all nodes of the road network apart from s and t. Then, it computes a single-via path for each via-node v by concatenating the shortest path from source s to v and the shortest path from v to target t. Each single-via path is compared to the shortest path based on a set of objective user-defined criteria, i.e., excess length, local optimality and stretch. The single-via paths that satisfy all of the user-defined criteria are considered alternative paths. Optimizations for this method were recently presented in [56].

The main shortcoming of methods based on candidate sets is that none of the proposed methods tackles the problem of computing multiple alternative paths dissimilar to each other. For example, the Plateaux method requires the existence of highways to recommend dissimilar alternative paths. However, in many real-world scenarios, i.e., within cities, the presence of highways is not always guaranteed. With regard to the method proposed in [4] where multiple single-via paths may be selected as alternative paths, their similarity only to the shortest path is considered. Hence, the recommended paths may be very similar to each other. In Chapter 5 we extend the method of [4] and propose the SVP$^+$ algorithm, which builds upon the concept of single-via paths and computes alternative paths dissimilar to each other.

2.2.3 Historical Data-based Methods

Historical data-based methods analyze historical information in order to extract information about the road network and provide more robust route planning services. A common approach is to analyze historical traffic information to compute traffic tolerant paths over time-dependent road networks [29, 51, 70, 88]. For alternative routing in particular, *the k traffic-tolerant paths* (TTP) problem [51] takes an $s-t$ pair and historic traffic information as input, and returns k paths that minimize the aggregate (historic) travel time. In contrast to our work though, TTP computes alternative paths without taking into consideration the similarity of the result paths. Furthermore, similar to all historical data-based methods, TTP relies on the availability of historical traffic information in order to compute alternative paths. In scenarios where such information is not available, TPP cannot be applied.

Other historical data-based methods analyze and mine trajectory data in order to extract popular routes [16, 17, 54, 81, 85, 92]. The main idea backing these methods is that experienced drivers tend to follow routes based on their own preferences and/or their knowledge about the traffic conditions in a particular area. By mining trajectory data, useful information can be obtained and used by recommender systems in order to provide more reliable results. However, trajectory-based methods compute routes based on trajectories collected over a period of time under normal circumstances. Therefore, the number of routes that can be obtained by using solely trajectory data is limited. Moreover, similar to historical traffic data for TPP, the availability of quality trajectory data is not always guaranteed.

2.2.4 Other Methods

Apart from the methods we described above, which define alternative routes using path similarity, there are also other methods that define alternative routes in a different way. In [87], alternative shortest paths using *edge avoidance* are introduced. Given the shortest path $p(s \to t)$ 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, the authors combine the concepts of distance oracles [75] and distance sensitivity oracles [13] and propose *iSPQF*. The shortest path between every pair of nodes avoiding each edge is precomputed and stored in a quadtree-based data structure inspired by [72]. Given a road network $G(N, E)$, iSPQF structure stores $|N|^2$ quadtrees in total ($|N|$ quadtrees per node). Alternative path queries are computed in almost constant time. Also, the authors show that by merging the quadtrees a worst case space complexity of $O(n^{1.5})$ is achieved. Although very efficient, iSPQF is limited to computing only one alternative route instead of a set.

Finally, the task of alternative routing can also be based on the pareto-optimal paths or the route skyline query for multi-criteria networks [25, 50, 58, 61, 79]. A path p is part of the pareto-optimal set or the route skyline P if p is not dominated by another path $p' \in P$. Path p dominates p' iff p is no 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. The pareto-optimal paths or the route skyline can be directly seen as alternative routes to move from source node s to target node t or can be further examined in a post-processing phase to provide the final alternative paths. Nevertheless, our definition of alternative routing is not a multi-criteria problem and the recommended paths by k-SPwLO cannot be obtained by first computing the pareto-optimal path set.

2.3 Multimodal Networks

The multimodal route planning problem seeks journeys combining schedule-based transportation (e.g., buses and trains) with unrestricted modes (e.g., walking and driving). This problem is significantly harder than its individual components as it involves the combination of two or more networks, usually of different types, into a single multimodal network.

A general approach to construct a multimodal network requires to build an individual network for each transportation mode. However, while a pedestrian network can be modeled using a static graph, i.e., the weights of edges do not change over time, public transportation networks are usually modeled as schedule-based networks, a special case of time-dependent networks. In

time-dependent networks, the weight of an edge varies depending on the time of the day that it is crossed. Each edge is associated with a time-dependent function which takes as an argument a timestamp t and returns the weight of the edge on time t. For schedule-based networks in particular, the function is used to query the schedule of the given transportation mode.

Pyrga et al. [69] summarize two different models for modeling timetable information. The *time-expanded model* constructs the time-expanded digraph in which every node corresponds to a specific time event (departure or arrival) at a station and edges between nodes represent either elementary connections between the two events (i.e., served by a train that does not stop in-between) or waiting within a station. The *time-dependent model* constructs the time-dependent digraph in which every node represents a station and two nodes are connected by an edge if the corresponding stations are connected by an elementary connection. The costs on the edges are assigned "on-the-fly", i.e., the cost of an edge depends on the time in which the particular edge will be expanded by the shortest-path algorithm to answer the query.

After each individual network graph is constructed, all individual networks are merged into a single multimodal network by adding link edges between nodes of different networks. For example, in order to combine a pedestrian network and a bus network, we add links so that every node of the bus network is connected to some node of the pedestrian network. Typical examples [28, 62] model walking as a static graph and public transportation networks using the realistic time-dependent model.

Various approaches for route planning [28, 31] and alternative routing [10, 24] on multimodal networks have been proposed. A direct solution to compute the shortest path between two nodes in a multimodal network though is to employ a modified version of Dijkstra's algorithm [32]. Since the multimodal network contains both edges with a fixed weight and time-dependent edges, the weight of each edge is determined "on-the-fly" based on its type. Also, when the search algorithm switches from the road (static) network to a transportation (time-dependent) network, the waiting time, i.e., the time between the arrival of the user at a stop and the departure time of the vehicle from the stop, has to be considered.

2.4 Routing Applications and Systems

Numerous systems that offer route planning services have been proposed in the last decade. *OSRM/MoNav* [55] obtains data from OpenStreetMap and allows the processing of distance and shortest path queries on road networks. To optimize query processing OSRM/MoNav employs CH [36]. *Graphhopper* [1] is a similar service-oriented open-source system which also employs CH. *TransDec* [29] is a real-world data-driven framework which obtains data from sensors and analyzes the traffic conditions on the road network. In addition, it stores and analyzes historical trajectory data to improve the quality of the results. In a similar context, *CrowdPlanner* [81] also employs historical information and recommends routes by taking into consideration the preferences of the users. For transportation networks, *Graphast* [57] is a framework which enables processing time-dependent spatio-temporal network queries [22]. Finally, *ISOGA* [43] enables the computation of isochrones on multimodal networks for reachability analysis.

[1] https://graphhopper.com

2.5 Summary

To sum up, there has been a lot of research work in the area of route planning. In particular, a huge variety of preprocessing-based methods for computing distance and shortest path queries on road networks have been proposed. State-of-the-art methods for distance queries offer exceptional query times, but they do not provide any efficient retrieval mechanism for the shortest path. In contrast, state-of-the-art methods for shortest path queries show relatively poor performance for distance queries. However, many applications require efficient query processing for both types of queries. Our ParDiSP approach (cf. Chapter 3) aims at filling this gap in existing research as it provides exceptional query times for both distance and shortest path queries.

In the area of alternative routing, existing literature has approached the computation of alternative paths from different perspectives. Most current approaches either do not propose a formal result set and, hence, provide no guarantees regarding the quality of the recommended paths, or they propose alternative paths based solely on their individual similarity to the shortest path, which results in alternative paths that are very similar to each other. In contrast to these approaches, our k-SPwLO query (cf. Chapter 4) aims at computing paths that are sufficiently dissimilar to each other *and* as short as possible.

CHAPTER 3

Partition-based Shortest Path Query Processing

Our preliminary investigation in [19] showed that in order to boost distance and shortest path query processing using a single approach is not enough. Bounded-hop methods are exceptionally fast for distance queries. Reducing the evaluation of distance queries to a number of lookups is clearly the best approach. For shortest path queries, we observed that in order to optimize the evaluation, the utilization of some form of shortcuts is required. It has been shown in [86] that the most efficient way to generate shortcuts is to follow a hierarchical approach.

In this chapter, we present the *Partition-based framework for Distance and Shortest Path* queries (ParDiSP) on road networks. ParDiSP combines ideas from both bounded-hop and hierarchical methods in a novel way, taking the best of both worlds, and efficiently supports both distance and shortest path queries. During preprocessing, ParDiSP precomputes the distances between any node in a component and the border nodes of the same component, the pairwise distances between all border nodes, and the union of the shortest paths between all border nodes. ParDiSP answers distance queries by combining distances from exactly three precomputed distance tables. For shortest path queries, the information in the distance tables allows to identify two border nodes that are traversed by the shortest path, thereby decomposing the path into three segments which can be computed in parallel. In a comprehensive experimental evaluation, we demonstrate the efficiency of ParDiSP on distance queries, shortest path queries and mixed workloads containing both types of queries.

3.1 Preprocessing for ParDiSP

In the preprocessing phase, ParDiSP partitions the input road network into *components* and computes the following data structures: the *extended component* of each component, which extends the component with all shortest paths between its border nodes; the *transit network*, which is composed of the shortest paths between the border nodes of each component together with the connecting edges; *in-component distance tables* (IDT), which store for every node in a component the distance from and to the border nodes of the component; and the *component distance matrix* (CDM), where each entry stores the distance between any pair of border nodes of any two components. Finally, both for the preprocessing of the aforementioned structures and to further improve the retrieval of shortest paths over the transit network, we employ Contraction Hierarchies (CH) [36], a state-of-the art method for shortest path queries.

3.1.1 Road Network Partitioning

Partitioning the road network is the first step of the preprocessing phase of every partition-based method. A *node-based partition* of a road network $G = (N, E)$ is a set $P(G) = \{C_1, \ldots, C_k\}$ of connected non-overlapping sub-networks $G_i = (N_i, E_i)$, termed *components*, such that:

- $N_i \cap N_j = \emptyset$ and $E_i \cap E_j = \emptyset$ for $i \neq j$,
- $N_1 \cup \ldots \cup N_k = N$, and
- $E_1 \cup \ldots \cup E_k \cup E_{con} = E$,

where E_{con} is the set of *connecting edges*, i.e., the set of all edges for which the source and target nodes belong to different components. All edges (n_i, n_j), where n_i and n_j are in the same component, are assigned to the same component as n_i and n_j as well. Furthermore, a node $n_i \in C_i$ is a *border node* of C_i if there exists a connecting edge $(n_i, n_j) \in E_{con}$ (or $(n_j, n_i) \in E_{con}$). For each component C_i, we represent the set of border nodes as $B(C_i)$.

Figure 3.1 illustrates a road network which is partitioned into four components, i.e., $P(G) = \{C_1, C_2, C_3, C_4\}$. The filled nodes are the border nodes. For instance, $B(C_1) = \{n_2, n_3, n_4\}$ and $B(C_3) = \{n_{16}, n_{19}\}$ are the border nodes of the components C_1 and C_3, respectively. There is a total of five connecting edges $E_{con} = \{(n_2, n_6), (n_3, n_7), (n_4, n_8), (n_{12}, n_{16}), (n_{18}, n_{19})\}$.

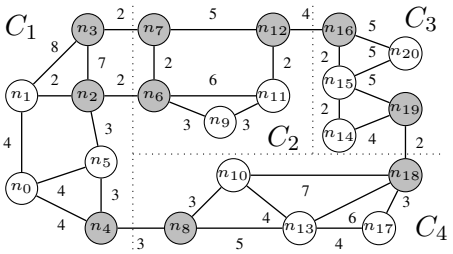


Figure 3.1: Road network partitioned into four components.

As the problem of road network partitioning is out of the scope of this thesis, we use METIS [47] to partition the road network, which is a multilevel graph partitioning method. Multilevel graph partitioning has been the most successful heuristic for partitioning large graphs, and METIS has been described as the fastest and best known system that implements this partitioning method [15]. METIS has also been adopted by other partition-based methods for spatial-network query processing, e.g., the *PTree* [84].

There exist graph partitioning methods, such as PUNCH [27] and Spatial Partition Clustering (SPC) [42], which take advantage of road network characteristics in order to provide a more efficient partitioning, i.e., reduce the number of border nodes even further. However, these algorithms do not provide much flexibility and produce partitions based on some objective criteria related to the structure of the network, over which the user has no control, i.e., natural cuts. For instance, if the best partition according to the objective criteria consists of two components, only this partition can be produced. Such a partition would not be suitable for our approach. In contrast, METIS guarantees that the generated partition has exactly the same number of components as specified by the user.

To partition an input road network, METIS requires the number of components k as an argument and attempts to compute a partition while minimizing the total number of border nodes. We also provide additional parameters to ensure that the components of the partition will always be contiguous. To store the partition of a road network, we simply add a tag/number to each node, which indicates the component of the partition the node belongs to.

3.1.2 Extended Components

For each component C of a partitioned road network, we define the *extended component* C^* of C which contains the nodes and edges of C plus all nodes and edges that are located on a shortest path between two border nodes of C.

Definition 1 (*Extended Component*)**.** Let $G = (N, E)$ be a road network with partition $P(G) = \{C_1, \ldots, C_k\}$. Given a component $C_i = (N_i, E_i)$, its *extended component* is defined as a sub-network $C_i^* = (N_i^*, E_i^*)$ with

$$N_i^* = N_i \cup \{v \mid (u, v) \in p_s(b_i \rightarrow b_j) \wedge b_i, b_j \in B(C_i)\},$$
$$E_i^* = E_i \cup \{(u, v) \mid (u, v) \in p_s(b_i \rightarrow b_j) \wedge b_i, b_j \in B(C_i)\}.$$

Since the shortest path between two border nodes of a component C does not necessarily lie entirely in C, the extended component C^* may contain nodes and edges that are not in C. In this case, the shortest path $p_s(s \rightarrow t)$ can be expressed as a concatenation of three shortest paths $p_s(s \rightarrow b_i) \circ p_s(b_i \rightarrow b_j) \circ p_s(b_j \rightarrow t)$, where b_i and b_j are border nodes of C. By the definition of C^*, $p_s(b_i \rightarrow b_j)$ is covered by C^*, while $p_s(s \rightarrow b_i)$ and $p_s(b_j \rightarrow t)$ are covered by C. Therefore, in order to determine the shortest path between two nodes s and t that are located in the same component C, only nodes on the extended component C^* need to be accessed.

Figure 3.2 shows the extended component C_1^* of C_1. The shortest path between the two border nodes n_2 and n_3 contains nodes n_6 and n_7 which are not in C_1. Hence, C_1^* is formed by all nodes in C_1 plus nodes n_6 and n_7 located in C_2 and all edges in C_1 plus the edges (n_2, n_6), (n_6, n_7) and (n_7, n_3) along the shortest path between n_2 and n_3. For the other components we have $C_2^* = C_2$, $C_3^* = C_3$ and $C_4^* = C_4$.

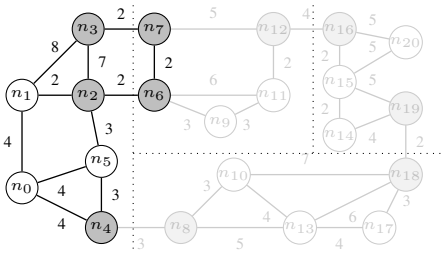


Figure 3.2: Extended component C_1^* of C_1.

3.1.3 Transit Network

The *transit network* G_T of a road network G consists of all shortest paths between all border nodes of each component C_i plus the connecting edges $E_{con}(G)$.

Definition 2 (*Transit Network*)**.** Let G be a road network with partition $P(G) = \{C_1, \ldots, C_k\}$. The *Transit Network* $G_T = (N_T, E_T)$ is a sub-graph of G with

$$N_T = \bigcup_{C \in P(G)} (B(C) \cup \{u \mid (u,v) \in p_s(b_i \rightarrow b_j) \wedge b_i, b_j \in B(C)\})$$

$$E_T = E_{con}(G) \cup \bigcup_{C \in P(G)} \{(u,v) \mid (u,v) \in p_s(b_i \rightarrow b_j)\}.$$

In practice, the transit network covers all shortest paths between any two border nodes of different components as shown in the following lemma.

Lemma 1. Let G be a road network with partition $P(G) = \{C_1, \ldots, C_k\}$ and transit network $G_T = (N_T, E_T)$. The shortest path, $p_s(b_i \rightarrow b_j)$, between any pair of border nodes $b_i \in B(C_i)$ and $b_j \in B(C_j)$ with $C_i \neq C_j$ is entirely in G_T.

Proof. We do a proof by contradiction. Let $e{=}(u,v)$ be an edge in E such that $e \in p_s(b_i \rightarrow b_j)$, but $e \notin E_T$. Since all connecting edges are in G_T, e cannot be a connecting edge, but must be located inside some component $C_i \in P(G)$ crossed by $p_s(b_i \rightarrow b_j)$. The only way for a path to cross C_i is to enter through a border node $b'_i \in B(C_i)$ and to exit through a border node $b'_j \in B(C_i)$. Since $p_s(b_i \rightarrow b_j)$ passes through border nodes b'_i and b'_j, there exists a sub-path $p(b'_i \rightarrow b'_j) \subseteq p_s(b_i \rightarrow b_j)$ which contains e and is the shortest path between b'_i and b'_j. By definition, the transit network G_T contains the shortest paths between all border nodes of all components, hence also $p(b'_i \rightarrow b'_j)$. Since e is in $p(b'_i \rightarrow b'_j)$, it is also in G_T. This leads to a contradiction. □

Figure 3.3 shows the transit network of our running example. It contains all connecting edges between components and, for each component, all shortest paths between its border nodes.

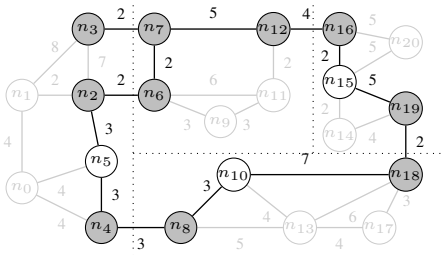


Figure 3.3: Transit Network of the example road network.

3.1.4 Distance Tables and Component Distance Matrix

For each node in a component, we define an *in-component distance table* (IDT). The IDT of a node $v \in C_i$ stores the distances from v to every border node of C_i.

Definition 3 (*In-component Distance Tables*)**.** Let G be a road network with partition $P(G) = \{C_1, \ldots, C_k\}$. The *in-component distance table* of a node $v \in C_i$ is defined as

$$T(v) = \{\langle b_i, d(v, b_i)\rangle \mid b_i \in B(C_i)\}.$$

Figure 3.4 shows two IDTs. $T(n_0)$ stores the distances between n_0 and the border nodes n_2, n_3 and n_4 of component C_1, whereas $T(n_{20})$ stores the distances between n_{20} and the border nodes n_{16} and n_{19} of component C_3.

$T(n_0)$	
n_2	6
n_3	12
n_4	4

$T(n_{20})$	
n_{16}	5
n_{19}	10

M			
$\emptyset$	M_{12}		M_{14}
M_{21}	$\emptyset$	M_{23}	M_{24}
M_{31}	M_{32}	$\emptyset$	M_{34}
M_{41}	M_{42}	M_{43}	$\emptyset$

M_{13}		
n_2	n_{16}	13
n_3	n_{16}	11
n_4	n_{16}	19
n_2	n_{19}	20
n_3	n_{19}	18
n_4	n_{19}	15

Figure 3.4: IDTs for node n_0 and n_{20} and CDM with entry (C_1, C_3).

Next, we compute the *component distance matrix* (CDM) of a partitioned road network. Given a partitioned road network $G = (N, E)$, the CDM contains all the distances between border nodes of any pair of components.

Definition 4 (*Component Distance Matrix*)**.** Let G be a road network with partition $P(G) = \{C_1, \ldots, C_k\}$. The *component distance matrix* M is a $k \times k$ matrix, where each element is defined as

$$M_{ij} = \begin{cases} \{\langle u, v, d(u, v)\rangle \mid u \in B(C_i) \wedge v \in B(C_j)\} & i \neq j, \\ \emptyset & i = j. \end{cases}$$

Algorithm 1: Preprocessing(G, k)

Input: road network $G = (N, E)$, number k of components
Output: Extended components C^*, transit network G_T, in-component distance tables T, component distance matrix M

```
// Create partition
1  Partition G into k components P(G) = {C_1, ..., C_k};
2  Determine E_con(G), and ∀C_i ∈ P(G) determine B(C_i);
// Create extended components, G_T and IDTs
3  E_T ← E_con(G);
4  foreach C ∈ P(G) do
5      C* ← C;
6      foreach b_i ∈ B(C) do
7          Run Dijkstra with b_i as source;
8          Update IDT T(v) for each v in C;
9          foreach path p_s(b_i→b_j) with b_j ∈ B(C) found by Dijkstra do
10             Add nodes and edges on p_s to C*;
11             Add nodes on p_s to N_T and edges to E_T;
// Create component distance matrix M
12 B_cdm ← B(C_1) ∪ ... ∪ B(C_k);
13 d_matrix ← CH_many_to_many(G_T, B_cdm, B_cdm);
14 foreach b_i ∈ B_cdm do
15     foreach b_j ∈ B_cdm \ {b_i} do
16         C_m ← C ∈ P(G) that contains b_i;
17         C_n ← C ∈ P(G) that contains b_j;
18         if C_m ≠ C_n then
19             M_{m,n} ← M_{m,n} ∪ {d_matrix[b_i][b_j]};
```

Each cell M_{ij} in the CDM is associated with a pair (C_i, C_j) of components and stores the set of all distances from every border node in C_i to every border node in C_j. The cardinality of this set is $|B(C_i)| \cdot |B(C_j)|$. The diagonal elements are empty since the distances are already stored in the IDTs. The CDM entry $M_{1,3}$ in Figure 3.4 stores all pairwise distances between the border nodes n_2, n_3 and n_4 of C_1 and the border nodes n_{16} and n_{19} of C_3.

3.1.5 Preprocessing Algorithm

To summarize the entire preprocessing phase, Algorithm 14 illustrates the step-by-step computation of all index structures employed by ParDiSP. The first step is the partitioning of the input road network. The partition is computed using METIS configured to aim at minimizing the number of connecting edges (Line 1). We elaborate more on how we choose the number k of components in Sections 3.3 and 3.4.2.

The second step is to compute for each component C its extended component C^* along with the IDTs and the transit network G_T (lines 3-11). After initializing the edges of G_T to the set of connecting edges (Line 3), we iterate through all components C. First, for each component C we initialize C^* to C (Line 5). Then, for each border node $b_i \in B(C)$ we run Dijkstra's algorithm with b_i as source and all other nodes $v \in C$ (including border nodes) as targets (Line 7). From this we extract the distances to update $T(v)$ (Line 8) and the shortest

paths to update C^* and G_T (lines 9-11).

Finally, for the computation of the component distance matrix M we construct CH [36] over the G_T and execute the many-to-many variant (Line 13). All border nodes of all components are used both as sources and as targets. The distance between every pair of border nodes is then added to the respective entry of M (lines 14-19).

3.2 Query Processing with ParDiSP

3.2.1 Processing Distance Queries

To compute distance queries from a source node $s \in C_s$ to a target node $t \in C_t$, we distinguish two cases:

1. s and t are in the same component
2. s and t are in different components.

First, if s and t are in the same component C_i, we employ the ALT algorithm [38] over the extended component C_i^*, which is essentially a bidirectional A^*-search with landmarks. To compute a lower bound from any node n to the target node t, ALT hinges on the triangle inequality $dist(n,t) \geq |dist(n,l) - dist(t,l)|$ for some landmark node l. We use the border nodes of each component C_i as landmarks, since the precomputed IDTs store all required distances to determine the lower bounds. Note that in case the A^*-search expands a node that is not in C (but in C^*), the lower bound is set to 0. Such a strategy eagerly expands the nodes that are in C^* but not in C.

Second, the distance between a source node s and a target node t that are located in different components C_s and C_t, respectively, is composed of three distances: the distance from s to a border node b_s of C_s, the distance from a border node b_t of C_t to t, and the distance from b_s to b_t, i.e.,

$$d(s,t) = \min_{\substack{b_s \in B(C_s) \\ b_t \in B(C_t)}} \{d(s,b_s) + d(b_s,b_t) + d(b_t,t)\}.$$

Since the three partial distances have already been precomputed and stored in the IDTs and in the CDM, the distance query amounts to joining three distance tables, regardless of how far away the source and target nodes are. Figure 3.5 illustrates the three distance tables $T(n_0)$, M_{13} and $T(N_{20})$ that are required to determine the distance from node $n_0 \in C_1$ to node $n_{20} \in C_{20}$, along with the part of the road network that is accessed, i.e., $C_1^* \cup G_T \cup C_3^*$.

The algorithm to compute distance queries is shown in Algorithm 16. First, the components C_s and C_t of the source node s and target node t, respectively, are determined (lines 1-2). If $C_s = C_t$, we execute ALT over the extended component C_s^* (Line 4). Otherwise, if $C_s \neq C_t$, we join the precomputed distance tables $T(s)$, M_{C_s,C_t}, and $T(t)$ (lines 6-15). The first loop joins $T(s)$ and the entry M_{C_s,C_t} of the component distance matrix, which yields the distances between s and each border node of C_t (lines 9-12). The second loop joins the intermediate result in T_{tmp} with $T(t)$ and determines the overall minimum distance, which is the distance from s to t (lines 13-15). Finally, the algorithm return the distance d from s to t in Line 16.

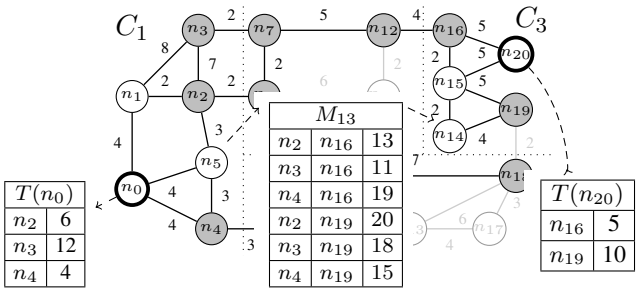


Figure 3.5: IDTs/CDM entry for a distance query from n_0 to n_{20}.

Algorithm 2: Distance_Query(G, s, t)

Input: road network G, source node s, target node t
Output: distance $d(s,t)$

```
1  C_s ← C ∈ P(G) that contains s;
2  C_t ← C ∈ P(G) that contains t;
3  if C_s = C_t then
4      d ← distance_ALT(C*_s, s, t);
5  else
6      T_tmp ← empty array of size |T(t)|, initialized to ∞;
7      d ← +∞;
8      k ← −1;
9      for i = 0 to |M_{C_s,C_t}| − 1 do
10         j ← i mod |T(s)|;
11         if j = 0 then k ← k + 1;
12         T_tmp[k] ← min(T_tmp[k], T(s)[j] + T_2[i]);
13     for i = 0 to |T(t)| − 1 do
14         d' ← T_tmp[i] + T(t)[i];
15         if d' < d then d ← d';
16 return d;
```

To achieve an efficient implementation of the joins, we keep for each component a sorted list of border nodes. By doing so, the distance tables need to store only the distances (and not the node ids) if they are sorted in the same order. Similarly, the entries of M store only distances. The entries are sorted first by the id of the target node and then by the id of the source node. This implementation technique saves space by omitting the storage of node ids in the distance tables, and permits efficient joins of the tables since the indices of matching elements can be computed directly.

Example 1. The execution of the algorithm is illustrated in Figure 3.6. The tables $B(C_1)$ and $B(C_3)$ determine the sort order of the distance tables. The first entry of $T(n_0)$ stores the distance from n_0 to n_2, whereas the first entry of $T(n_{20})$ stores the distance from n_{16} to n_{20}. The first loop joins $T(n_0)$ and M_{13} and produces the intermediate table T_{tmp}, which contains the distances from n_0 to the two border nodes n_{16} and n_{19} of C_3. In the second loop, we sum one-to-one the elements of T_{tmp} with the elements of $T(n_{20})$ and keep the minimum, i.e., $19 + 5 = 24$ is the distance between n_0 and n_{20}. This is possible since T_{tmp} and $T(t)$ are ordered in the same way.

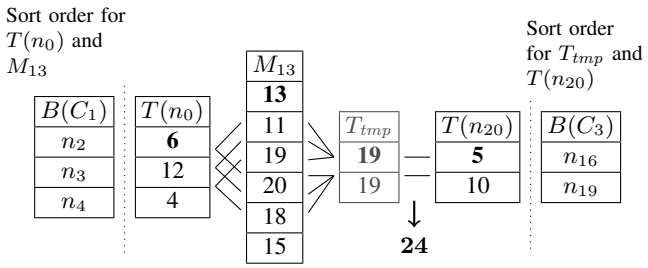


Figure 3.6: Joining distance tables to compute $dist(n_0, n_{20})$.

3.2.2 Processing Shortest Path Queries

For the computation of shortest paths queries we follow the same strategy as for distance queries and distinguish two cases. First, if the source node s is in the same component C as the target node t, we execute the ALT algorithm [38] over the extended component C^*. Second, if the source and the target nodes are in different components, we decompose the shortest path into three disjoint sub-paths, as shown in the following corollary:

Corollary 2. Let G be a road network with partition $P(G) = \{C_1, \ldots, C_k\}$. The shortest path from a source node $s \in C_s$ to a target node $t \in C_t$, where $C_s \neq C_t$, can be expressed as the concatenation of three shortest paths as follows:

$$p_s(s \rightarrow t) = p_s(s \rightarrow b_s) \circ p_s(b_s \rightarrow b_t) \circ d(b_t \rightarrow t),$$

where $b_s \in B(C_s)$ is a border node of C_s and $b_t \in B(C_t)$ is a border node of C_t.

The corollary follows directly from Lemma 1 and forms the basis to compute shortest path queries when source and target nodes are in different components. The key idea is to first determine the border nodes b_s and b_t and then run three independent shortest path queries. It is easy to see that the first (last) segment of the shortest path lies entirely in the extended component C_s^* (C_t^*), whereas the intermediate segment lies entirely in the transit network G_T.

Algorithm 11 shows our shortest path algorithm. If source and target are in the same component, the ALT algorithm is called over C^* (Line 4). Otherwise, we first call a slightly modified version of our Distance_Query algorithm in Line 6, which along with the distance returns also the two border nodes b_s and b_t that split the shortest path into three segments. Then, the first and the last segment of the shortest path are retrieved from the respective extended components in Line 7 and Line 9 respectively, by running an A^* search that uses the distance to (from) b_s (b_t) as a lower bound. In contrast to the path retrieval mechanism for bounded hop techniques, no distance queries are executed since all distances are in the respective IDT of each node. To compute the middle segment, we run a shortest path query from b_s to b_t over G_T (Line 8). For this query, we employ the same CH scheme that was constructed to compute the CDM. The query time is further improved by computing the three segments in parallel. Finally, the algorithm constructs the final path p by concatenating partial paths p_1, p_2 and p_3, and return p in Line 11.

Note that although we use the precomputed CH scheme for processing shortest path queries over the transit network, CH can be replaced with any other state-of-the-art method.

Algorithm 3: Shortest_Path_Query(G, s, t)

Input: road network G, source node s, target node t
Output: shortest path $p_s(s \rightarrow t)$

```
1  C_s ← C ∈ P(G) that contains s;
2  C_t ← C ∈ P(G) that contains t;
3  if C_s = C_t then
4  |  p ← shortest_path*(C*_s, s, b_s) // ALT over C*_s
5  else
6  |  ⟨b_s, b_t, d(s,t)⟩ ← Distance_Query*(s, t);
7  |  p_1 ← shortest_path*(C*_s, s, b_s) // A* over C*_s
8  |  p_2 ← shortest_path(G_T, b_s, b_t) // CH over G_T
9  |  p_3 ← shortest_path*(C*_t, b_t, t) // A* over C*_t
10 |  p ← p_1 ∘ p_2 ∘ p_3;
11 return p;
```

3.3 Theoretical Analysis

In this section, we discuss some theoretical aspects of our ParDiSP approach with regard to the size of the transit network, the space requirements of the precomputed structures and the query processing cost.

ParDiSP takes full advantage of the distance tables and the transit network if the source and target nodes are in different components C_s and C_t. Given a road network partitioned into k components, the probability that $C_s = C_t$ is $1/k^2$. Therefore, the more the components a road network is partitioned into, the higher the chance that source and target are in different components, which allows a more efficient use of the precomputed structures. However, the number of components and the border nodes influence not only the query processing but also the space requirements and the preprocessing time, which both increase with the number of border nodes.

In general, we characterize a partition as good if it minimizes the total number of connecting edges between the components, and hence the total number of border nodes. As road network partitioning is *NP*-hard, an optimal solution is out of question [33]. However, we can estimate the number of border nodes by employing the $\sqrt{n}$-Separator Theorem [53], which has also been used in [84]. According to this theorem, the partition of a (undirected) graph $G = (N, E)$ into k components generates $O(\sqrt{\frac{|N|}{k}})$ border nodes per component.

Transit Network Size. The worst case for the size of the transit network $G_T = (N_T, E_T)$ is when all nodes of the road network $G = (N, E)$ are in the transit network, i.e., $N_T = N$. The best case occurs if only the border nodes of the components are in the transit network, i.e., $N_T = \bigcup_{C_i \in P(G)} B(C_i)$. By applying the $\sqrt{n}$-Separator Theorem, we obtain the following bounds for the size of the transit network:

$$k\sqrt{\frac{|N|}{k}} \leq |N_T| \leq |N|.$$

The transit network takes advantage of the spatial coherence of road networks, a notion that has been analyzed and used in [74]. Since the transit network is the union of the shortest paths

between the border nodes, it is expected to be much smaller than the original network. This is empirically verified in Section 3.4.2.

Memory Overhead. The memory overhead of ParDiSP is dominated by the number of precomputed distances stored in the IDT of each node and in the CDM. The space to store the partition, the extended components, and the transit network is negligible. The IDT associated with a node v in a component C_i contains the distances to/from all border nodes of C_i. Following the $\sqrt{n}$-Separator Theorem, this gives $\sqrt{\frac{|N|}{k}}$ distances for each node and a cumulative size of all IDTs of

$$\textit{size}(\textit{IDT}) = |N| \sqrt{\frac{|N|}{k}}. \tag{3.1}$$

The component distance matrix M stores for each pair of distinct components C_i, C_j the pairwise distances between all border nodes of C_i and C_j. Thus, each entry in M stores $\sqrt{\frac{|N|}{k}} \cdot \sqrt{\frac{|N|}{k}} = \frac{|N|}{k}$ distances. We exclude the distances between border nodes of the same component as these distances are stored in the IDTs. Furthermore, we need to store the distances for each pair of components C_i, C_j only once as $M_{i,j} = M_{j,i}$. Hence, the total number of populated entries in M is $(k^2 - k)/2$, yielding a total size of

$$\textit{size}(M) = \frac{k^2 - k}{2} \frac{|N|}{k} = \frac{(k-1)\,|N|}{2}. \tag{3.2}$$

The total space overhead required by the index structures of ParDiSP is obtained by summing up Equation 3.1 and 3.2, i.e.,

$$size = size(IDT) + size(M) = |N| \sqrt{\frac{|N|}{k}} + \frac{(k-1)\,|N|}{2}.$$

Query Processing Cost. For processing distance queries between two nodes located in different components, we join three distance tables: two IDTs and one entry of CDM (cf. Algorithm 16). The first loop joins one IDT with the CDM entry and produces an intermediate table. This step performs $\frac{|N|}{k}$ summations since the CDM entry is of size $\frac{|N|}{k}$. The second loop joins the intermediate table with the second IDT, which requires $\sqrt{\frac{N}{k}}$ summations, corresponding to the size of the IDT. Therefore, the total cost for processing a distance query q_d is

$$\textit{cost}(q_d) = \frac{|N|}{k} + \sqrt{\frac{|N|}{k}} = O(\frac{|N|}{k}).$$

The complexity of shortest path queries with source and target node in the same component C is the same as for the ALT algorithm [38], taking into consideration that the algorithm runs only over the extended component C^* and not over the entire graph. To process a shortest path query $q_p(s \to t)$ with s and t in different components, we first have to process the distance query $q_d(s, t)$. The retrieval of the first and the last segment (in the source and target components, respectively) has linear cost in the size of the segment. The most expensive part is the retrieval of the middle segment from the transit network G_T. The cost for retrieving the middle segment depends on the the size of G_T and the runtime cost of CH [21].

3.4 Experimental Evaluation

In this section, we describe the results of a detailed experimental evaluation of ParDiSP on real-world road networks. We analyze the runtime, the preprocessing cost and the memory consumption of ParDiSP in comparison to state-of-the-art methods for distance and shortest path queries. Furthermore, we provide some additional measurements to back our claims regarding the effect of road network partitioning to the preprocessing cost and the performance of our ParDiSP approach.

3.4.1 Setup and Datasets

In our experimental evaluation we use four real-world public available datasets from the *9th DIMACS Challenge*[1], each of which corresponds to a part of the road network in US. These datasets are summarized in Table 3.1).

Table 3.1: Datasets.

Name	Region	# Nodes	# Edges
NY	New York City	264,346	733,846
FL	Florida	1,070,376	2,712,79
CA	California-Nevada	1,890,815	4,657,742
E	Eastern USA	3,598,623	8,778,114

We performed three sets of experiments in total. In the first set of experiments (Figures 3.7-3.11 and 3.13) we evaluate the performance of ParDiSP in terms of preprocessing cost, space overhead and query processing time for different partitions of the same dataset varying the numbers of components k from 128 to 1024.

In the second set of experiments (Figures 3.12, 3.14–3.16), we compare ParDiSP to three different algorithms: Arterial Hierarchy (AH) [94], which has been shown to be the fastest approach for shortest path queries [86]; Contraction Hierarchies (CH) [36], which we also employ for the computation of the CDM; and Pruned Highway Labeling (PHL) [7], a state-of-the-art method for distance queries.

To determine the number of components k for ParDiSP for the second set of experiments, we use the insight we gained from the measurements of the first set. We observed that for distance queries ParDiSP becomes faster as the number of components k is increasing while, for shortest path queries, ParDiSP shows the best performance for $k = \{256, 384, 512\}$. Since shortest path queries are more time consuming than distance queries we run our experiments for all datasets with $k = 384$. For the selected value of k, ParDiSP is adequately fast for distance queries too.

Finally, in the third set of experiments (Figure 3.16), we analyze the query performance for mixed query sets that contain both distance and shortest path queries. The ratio of distance to shortest path queries in each query set varies between $20\%/80\%$, $40\%/60\%$, $60\%/40\%$ and $80\%/20\%$. We compare ParDiSP with two combined approaches using state-of-the-art solutions for each query type: PHL for distance queries and CH and AH, respectively, for shortest path queries, denoted as PHL-CH and PHL-AH.

[1] http://www.dis.uniroma1.it/challenge9/

For the first set of experiments we report the average runtime over $100,000$ randomly generated queries for each road network. For the second and the third set of experiments, we follow previous works [84, 86, 94] and generate for each road network ten query sets, $Q_1, \ldots Q_{10}$, as follows. We select $10,000$ pairs of nodes for each query set Q_i for $i = 1, \ldots, 10$ such that the distance between the nodes is in the range $[\frac{i \cdot d_{max}}{10}, \frac{(i+1) \cdot d_{max}}{10}]$, where d_{max} is the maximum distance between two nodes in the data set. Hence, the distance between any pair of nodes in Q_i is always smaller than the distance between any pair of nodes in Q_{i+1}. For each road network, we report the average runtime of all methods for all ten query sets separately.

All algorithms were implemented in C++ using the C++11 standard and compiled using GNU G++ compiler (version 4.8.1). The source code for PHL and CH is available on the web, while the source code of AH has been provided by the authors. Also, we implemented the shortest path retrieval mechanism for PHL ourselves as no such mechanism was provided, while we used CH with additional runtime optimizations provided by the authors. All experiments were executed on an Ubuntu Linux server with 4 Intel Xeon X5550 (2.67GHz) processors and 48GB of RAM.

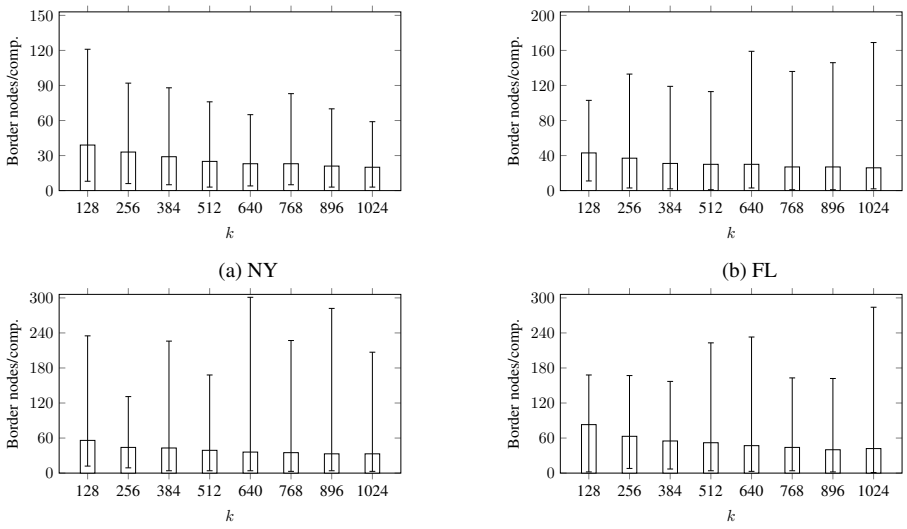
3.4.2 Graph Partitioning

In Section 3.3 we showed through a theoretical analysis how the number of components has an impact on the space overhead and the performance of ParDiSP. In what follows, we present measurements from our first set of experiments on attributes directly connected to the partition of the input road network, i.e., the average number of border nodes per component, the average size of each component and extended component, and the size of the transit network for each partition.

Figure 3.7 shows the average number of border nodes with min and max values. In all figures we notice that the number of border nodes decreases with the increase of k in a way that follows the trend of the $\sqrt{n}$-Separator Theorem [53]. We also observe that the minimum and the maximum number of border nodes per component show significant fluctuation. For NY (Figure 3.7a) the maximum number of border nodes is decreasing, while for the other three datasets, no conclusion can be drawn regarding the maximum and the minimum number of border nodes, as the behavior illustrated in Figures 3.7a-d) is irregular. This observation also backs our claim that result of the graph partitioning process cannot be predicted accurately.

Figure 3.8 shows the average size, i.e., the average number of nodes, of components and extended components per partition with min and max values. Naturally, the size of components is decreasing as k increases. Furthermore, by observing the minimum and the maximum values we notice that with the increase of k the components become more and more equally sized. With regard to the extended components, we observe that they are slightly bigger than the associated components per partition. However, we also observe a much greater difference in the minimum and maximum values, especially when k is small. For all datasets, when k is 128 the maximum size of an extended component is significantly higher than the average. Hence, while the nodes per component are balanced, that does not guarantee that the nodes per extended component will be balanced as well.

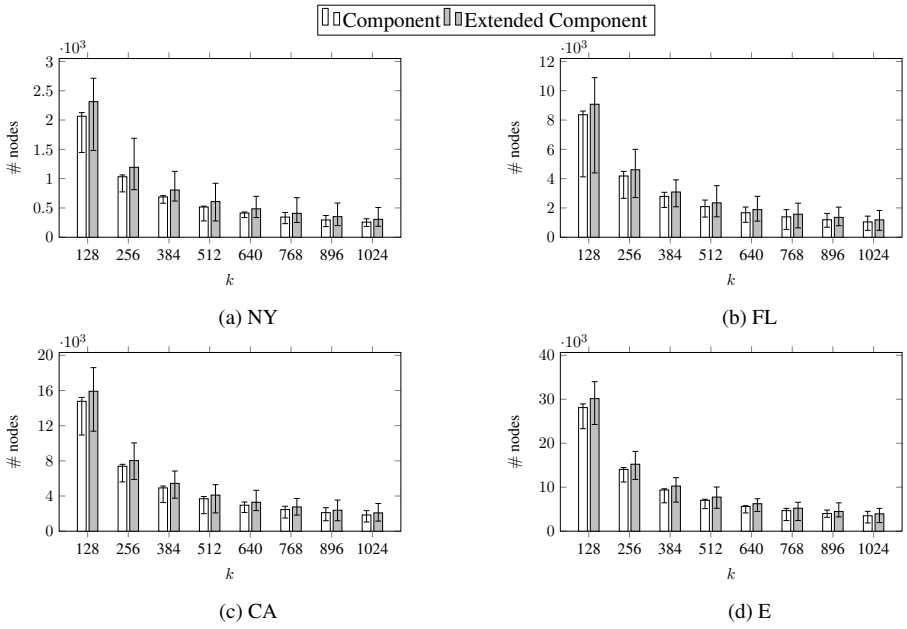
In general, by observing both Figures 3.7 and 3.8, we conclude that while METIS [47] provides quality partitioning, it also provides no particular guarantees regarding the size of the components and the extended components. Such a result is to be expected as METIS aims to minimize the average number of border nodes per component. Due to the unpredictability



(a) NY (b) FL

(c) CA (d) E

Figure 3.7: Border nodes per partition.



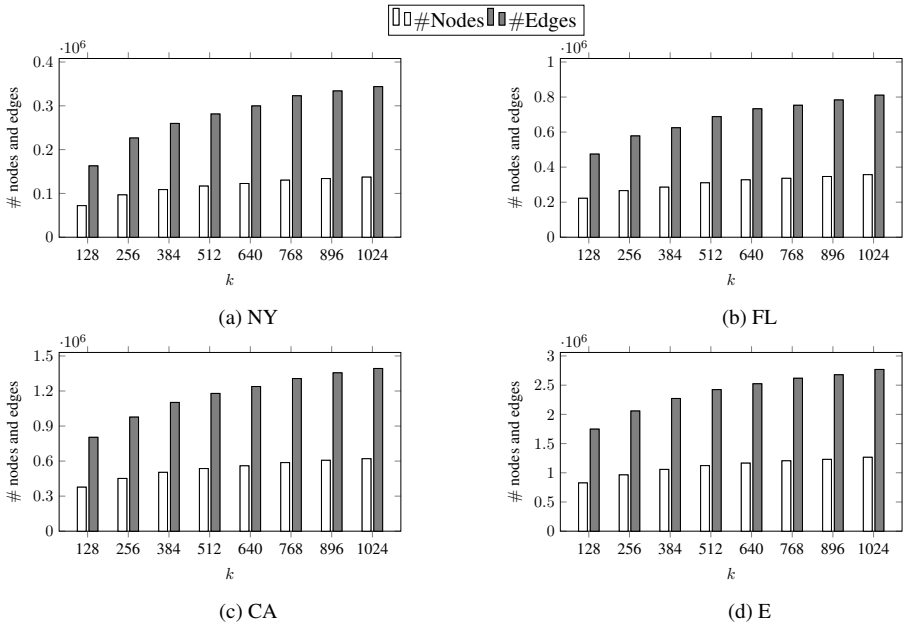
(a) NY (b) FL

(c) CA (d) E

Figure 3.8: Component and extended component size per partition.

of graph partitioning, it is impossible to minimize the size of the extended components unless both the partition and the extended components are computed. On the other hand, minimizing the average number of border nodes per component results in smaller IDTs and, hence, faster processing of distance queries.

Figure 3.9 analyzes the number of nodes and edges of the resulting transit network. The size of the transit network increases with k, yet it is significantly smaller (approximately 2-5 times) than the the original road network. Hence, any query between border nodes can be



(a) NY (b) FL

(c) CA (d) E

Figure 3.9: Transit network size per partition.

computed by accessing only a part of the road network which is only $1/2$ to $1/5$ the size of the original. Furthermore, the space overhead of CH, which we build over the transit network, is expected to decrease along with the number of components k.

For efficient query processing, apart from the size of the transit network, the density of the network plays an important role too. The CH method that we use for queries over the transit network works best for sparse networks. We observe though that the transit network is denser than the original road network. The density of a road network is given by:

$$D(G) = \frac{|E|}{|D| \cdot (|D| - 1)}.$$

For example, the density of the road network of NY is $D(NY) \simeq 10.5 \cdot 10^{-6}$ while the density of the the transit network varies from $D(G_T) \simeq 31 \cdot 10^{-6}$ for $k = 128$ to $D(G_T) \simeq 18 \cdot 10^{-6}$ for $k = 1024$. In general, we observe that the density of the transit network decreases with the increase of k. Hence, the more the components in a partition, the sparser the transit network gets.

3.4.3 Preprocessing

Figure 3.10 analyzes the preprocessing time of ParDiSP varying the number k of components. We distinguish between the time for precomputing a) the extended components (C^*) b) the IDTs and the transit network ($G_T + IDTs$), and c) the CDM including the construction of the CH scheme (CDM). In all figures, we observe that the time required to precompute the extended components decreases as k increases, whereas the time to precompute the CDM increases with k. Apparently, although more extended components need to be computed, since the size of the components and the number of border nodes decrease with k, each computation of of an extended component requires much less time; thus the overall preprocessing time

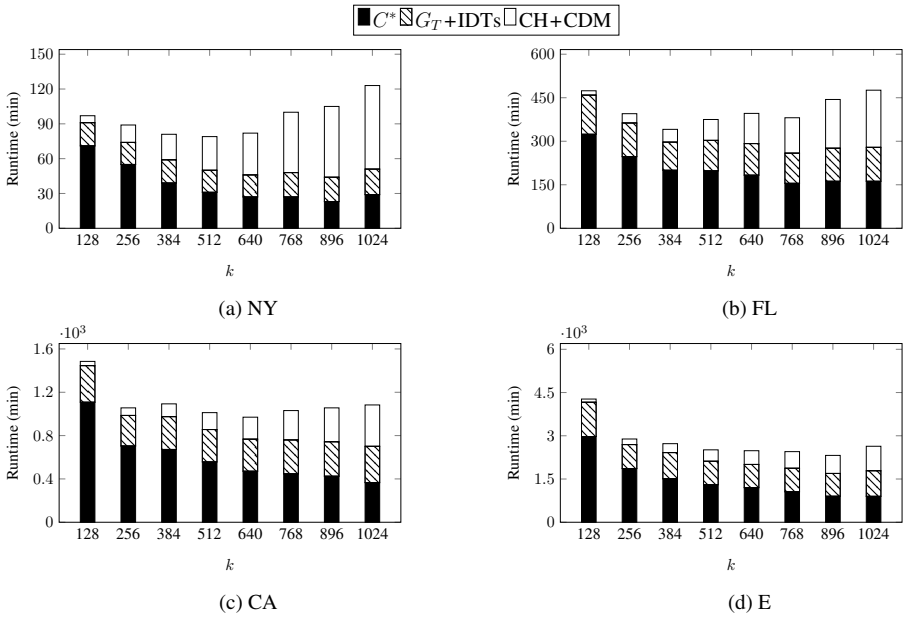


Figure 3.10: Preprocessing time in ParDiSP.

drops. Next, we observe that the time required to compute the IDTs and the transit network is similar for all values of k. On the contrary, the time required to compute the CDM increases with k, since the transit network becomes larger too. Also, the total number of border nodes increases and, hence, there are more sources-targets in the many-to-many search. Finally, we observe that for all datasets the preprocessing time shows a local minimum; for NY the preprocessing time is minimum for $k = 512$, for FL it is minimum for $k = 384$, for CA it is minimum for $k = 640$ and for E it is minimum for $k = 896$.

Figure 3.11 shows the space overhead of ParDiSP varying the number k of components. We distinguish between the memory required to store a) the IDTs b) the CDM and c) the CH scheme over the transit network. The space required to store the extended components and the transit network is negligible. With an increasing number k of components, the space required by the CDM increases, whereas the space for the IDTs decreases. The space required to store the CH scheme over the transit network slightly increases with k but it is negligible in comparison to the memory requirements for storing the IDTs and the CDM. Overall, with the exception of small values of k, the memory requirements are dominated by the CDM.

Figure 3.12 compares the preprocessing time and space overhead of ParDiSP (with $k = 384$) to AH, CH and PHL for all datasets. Figure 3.12a shows that CH outperforms ParDiSP and the other competitors in terms of preprocessing time on all datasets. For NY, PHL and ParDiSP have similar preprocessing time while AH is the slowest. For FL and CA, PHL, AH and ParDiSP have similar preprocessing time while, for E, AH is the fastest method after CH, ParDiSP is slightly slower and PHL is the slowest method. Figure 3.12b analyzes the space overhead. For NY all approaches have similar space requirements except for ParDiSP which has slightly more. For all the other datasets, CH is shown to consume the least space, followed by AH. PHL is the method which inflicts the highest space overhead by far. ParDiSP requires significantly less space than PHL, but more than CH and AH.

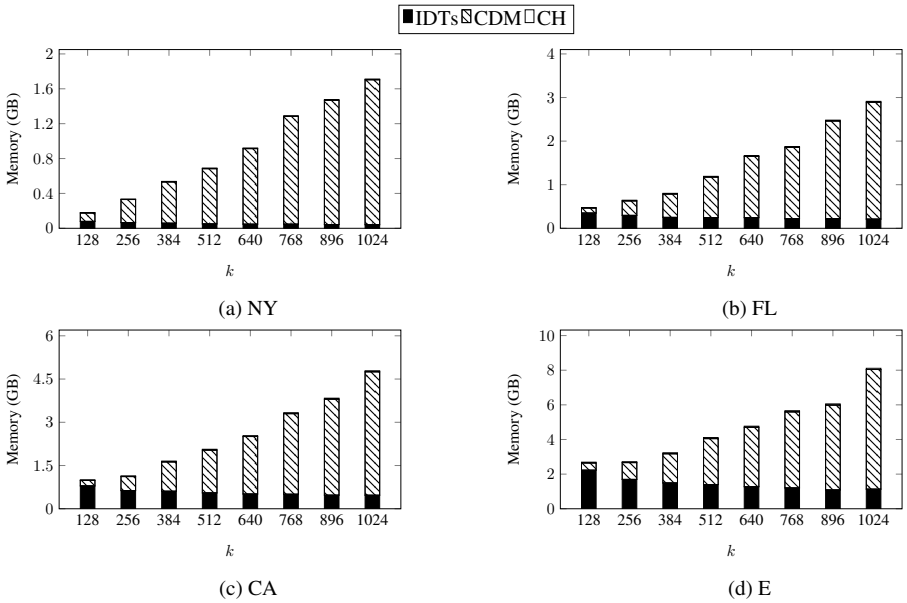


Figure 3.11: Space overhead in ParDiSP.

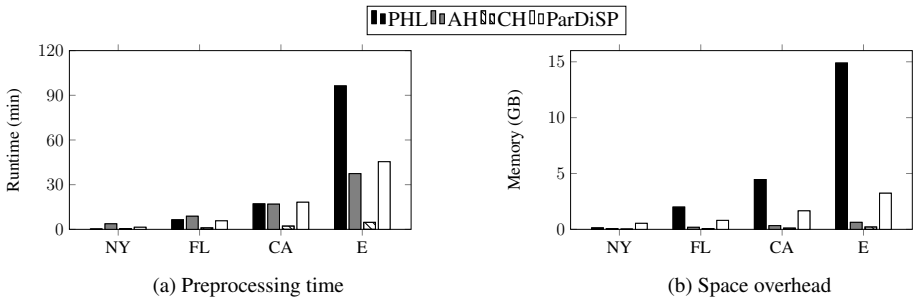


Figure 3.12: Preprocessing cost for PHL, AH, CH and ParDiSP.

3.4.4 Query Processing

Figure 3.13 presents the results of our first set of experiments on query processing with ParDiSP. More specifically, the figure analyses the query processing time of ParDiSP for distance and shortest path queries varying the number k of components. First, we observe that, for all datasets, the processing of distance queries becomes faster with an increasing number of components. This result verifies our claim from observing Figure 3.7 that larger values of k lead to few border nodes per component, thus smaller distance tables. Furthermore, the larger the value of k is the less probable that a query will be an in-component one. Even if that is the case though, the size of the extended component becomes smaller with an increasing k and the processing time of in-component queries drops as well. With regard to shortest path queries, we observe that the processing time does not change much with k. In fact, the performance of ParDiSP for shortest path queries shows a local minimum for all datasets; for NY and FL the processing time is minimum for $k = 256$, for CA it is minimum for $k = 512$ and for E it is minimum for $k = 384$. This observation lead us to choose the number of components $k = 384$

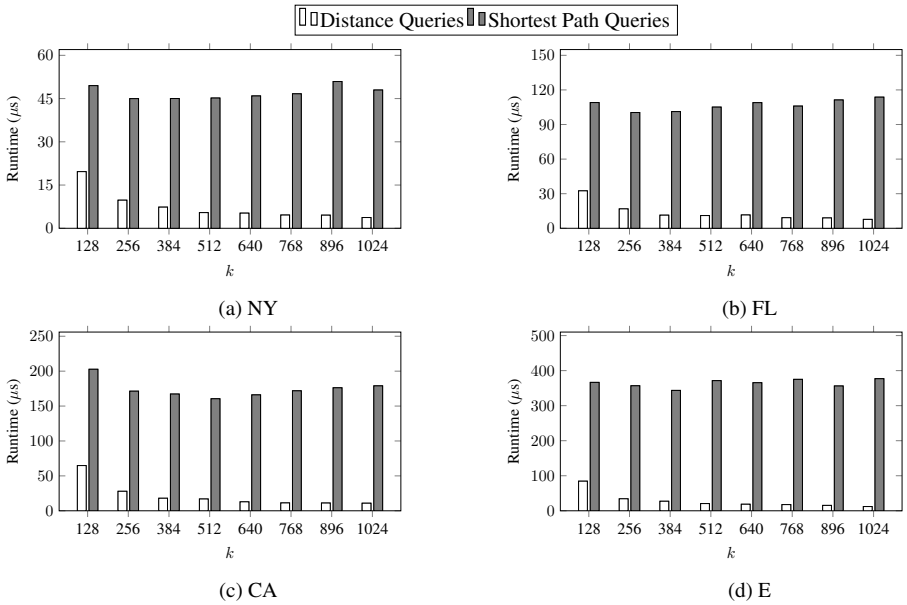


Figure 3.13: Cost of query processing in ParDiSP.

for our second set of experiments.

Figures 3.14 and 3.15 report the results of our second set of experiments on the processing time of distance and shortest path queries. In particular, Figure 3.14 compares the processing of distance queries with ParDiSP, AH, CH and PHL for all four datasets. In all datasets, the runtime of ParDiSP is constant, except for query set Q_1, where the rate of in-component queries is comparably high. Clearly, processing in-component queries with ParDiSP is more expensive than processing cross-component queries. While PHL is the fastest solution for all four datasets, ParDiSP is very close to this state-of-the-art solution. CH is approximately one order of magnitude slower than both ParDiSP and PHL and alt least two times slower than AH, especially for query sets Q_5 to Q_{10} Finally, we observe that, in all road networks, the performance of AH deteriorates in the beginning, but improves significantly after Q_4, i.e., while the distance between query points increases. It appears that AH is better at processing queries where the query points are far apart. However, even for Q_{10} where AH shows the best performance, it is still slower than our ParDiSP approach.

Figure 3.15 compares the processing of shortest path queries with ParDiSP, AH, CH and PHL for all four datasets. First, for the NY dataset we observe in Figure 3.15a that, although PHL is the fastest method for distance queries, it is extremely inefficient for shortest path queries. The reason for this poor performance is the large number of distance queries that need to be processed for retrieving the shortest path. Naturally, the performance of PHL deteriorates even more for query sets where the query points are distant. For presentation purposes, we omit PHL in Figures 3.15b–3.15d.

With regard to the other algorithms, we observe that, for NY, AH and ParDiSP have similar performance (with AH being sightly faster) while CH is clearly the slowest approach among the three. ParDiSP is the fastest approach in FL, CA and E, outperforming both CH and AH for all query sets, with the exception of Q_1 for the CA road network. Since the source and the target points of queries in Q_1 are very close, the number of in-component queries that ParDiSP

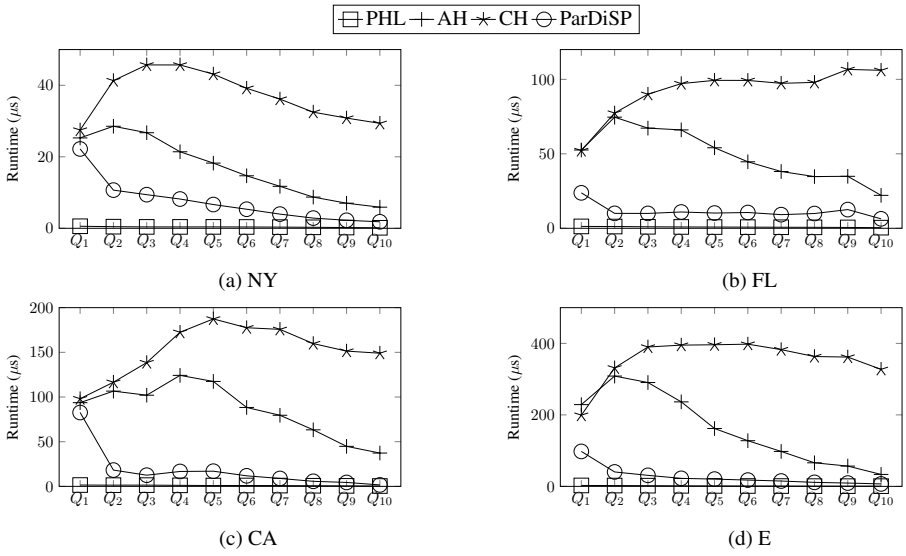


Figure 3.14: Performance for distance queries.

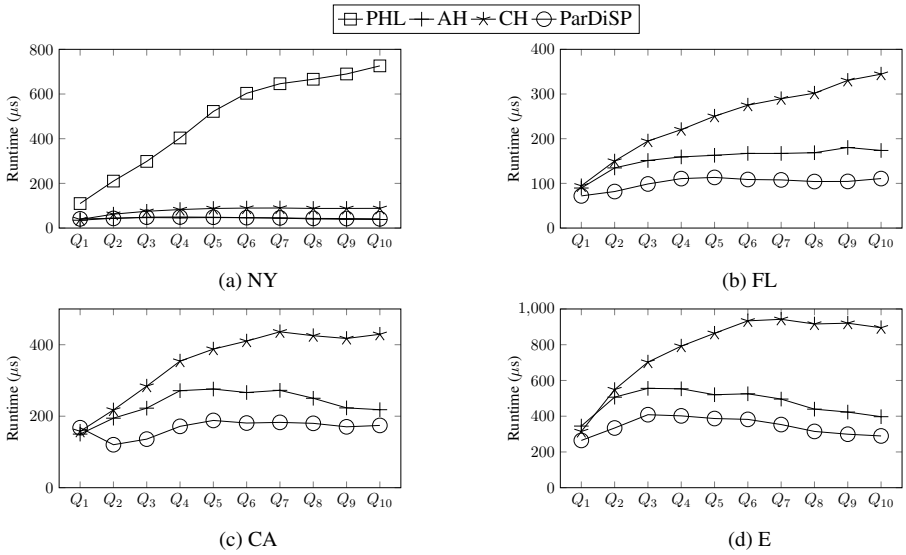


Figure 3.15: Performance for shortest path queries.

has to process is quite high. Clearly, The efficiency of ParDiSP drops when it has to process a large number of in-component queries.

In addition, we observe that the performance of CH is influenced by the distance between the source and target query points. For instance, for FL the performance of CH steadily deteriorates as the distance between the query points increases. We observe a similar tendency for CA and E too. However, the performance of CH seems to stabilize for query sets Q_7 to Q_{10}. In contrast to CH, the performance of AH and ParDiSP seems to be much less influenced by the distance between the query points. The performance of AH varies from network to network; in FL, the performance of AH seems to deteriorate as the distance between query points grows,

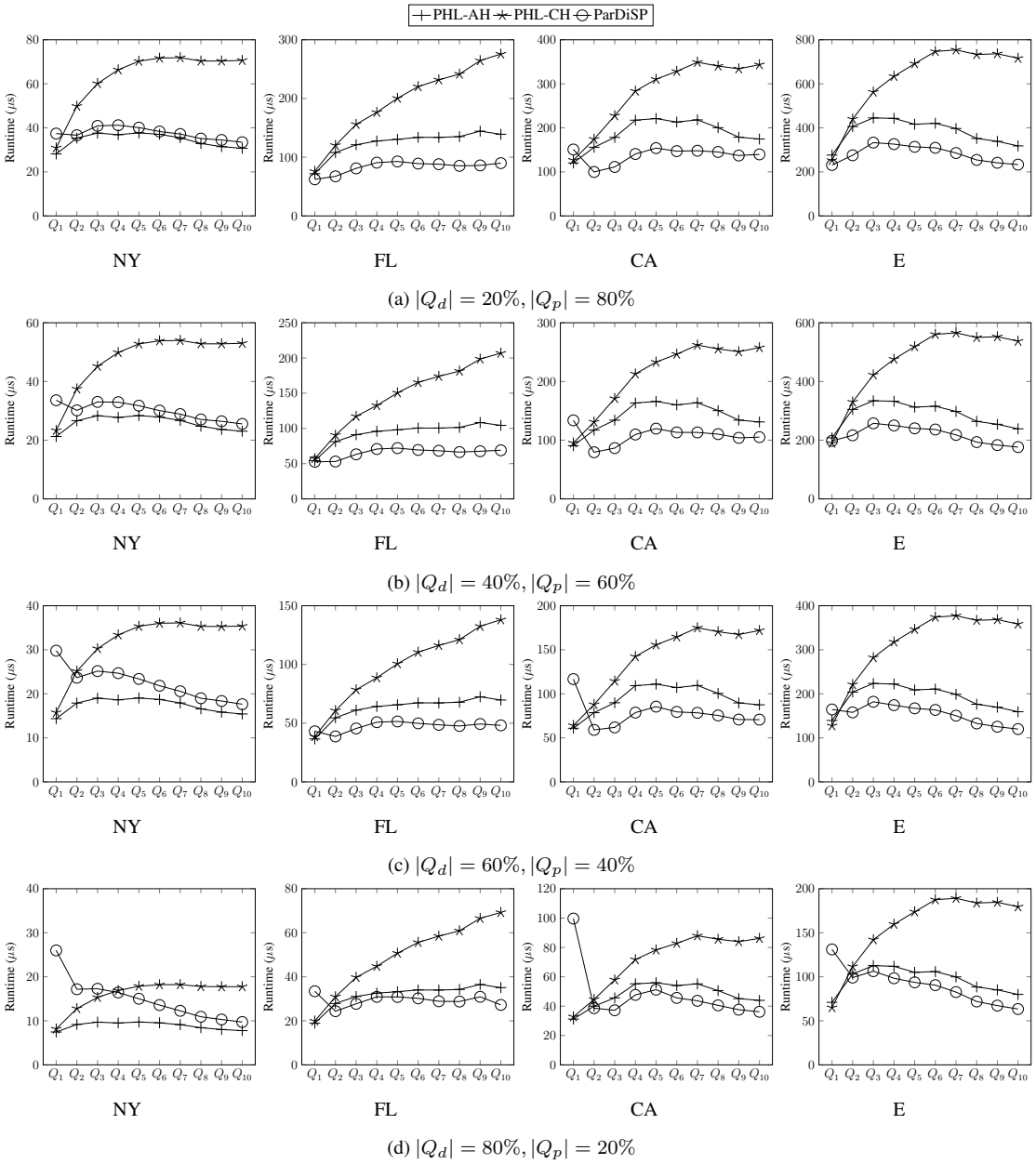


Figure 3.16: Performance for mixed query sets.

in CA, the performance does not deteriorate and appears to be most efficient for query sets Q_2 and Q_3, while in E, the performance deteriorates from Q_1 to Q_3 but then, from Q_3 to Q_{10} it improves steadily. The performance of ParDiSP is similar for all query sets.

Finally, Figure 3.16 shows the results of our third set of experiments with mixed query sets. For NY, PHL-AH is the fastest approach outperforming ParDiSP. For Q_1 of CA, the performance of ParDiSP displays a similar behavior as in our previous experiments, meaning that the number of in-component queries is quite high. However, in all settings apart from NY and Q_1 of CA, ParDiSP clearly outperforms both PHL-CH and PHL-AH. We also observe that

the gap between ParDiSP and the two competitors gets smaller with the increase of distance queries in the query load. When more than 95% of the queries are distance queries, the three approaches have comparable performance for FL, CA and E. Finally, we observe that PHL-AH outperforms PHL-CH for all settings. Such a result is expected as both methods use PHL for distance queries, while AH is faster than CH in shortest path queries.

3.5 Summary

We presented ParDiSP, a framework for the efficient processing of both distance and shortest path queries on road networks. Given an input road network partitioned into components, ParDiSP precomputes a set of index structures which are utilized to optimize query processing. ParDiSP answers distance and shortest path queries by running the ALT algorithm if source and target are in the same component, using the precomputed distances to the border nodes to compute lower bounds. Otherwise, for distance queries, ParDiSP combines distances from three precomputed tables, whereas shortest path queries are decomposed into the retrieval of three sub-paths, which can be executed in parallel. Our experimental evaluation has shown that ParDiSP outperforms the state-of-the-art for shortest path queries, while being comparable to the state-of-the-art for distance queries. For mixed query sets with distance and shortest path queries, ParDiSP is, in most cases, more efficient even than a combination of the best state-of-the-art approaches for the two query types, while incurring less space overhead.

CHAPTER 4

k-Shortest Paths with Limited Overlap

In this chapter, we introduce a novel approach to tackle the problem of computing alternative paths on road networks. Our focus is to recommend a set of k paths (including the shortest path) such that every path in the result set is sufficiently dissimilar to all the other paths in the result set based on a user-specified similarity threshold and, at the same time, as short as possible. We formalize this form of alternative routing as the *k-Shortest Paths with Limited Overlap* (k-SPwLO) problem. Furthermore, we propose three algorithms for evaluating k-SPwLO queries. The baseline algorithm BSL computes first a (necessarily large) set of candidate paths and applies similarity filtering in a second step. OnePass traverses the network once and expands only those paths that qualify the similarity constraint. Finally, MultiPass extends and improves OnePass by employing an additional pruning criterion and computes k-SPwLO queries by traversing the network $k-1$ times.

4.1 Alternative Paths

Given a road network $G = (N, E)$, let P be a set of paths from a source node $s \in N$ to a target node $t \in N$. We call any path $p(s \to t)$ alternative to P if p is sufficiently dissimilar to every path $p' \in P$. More formally, the similarity of p to every path $p' \in P$ is determined by their overlap ratio:

$$Sim(p, p') = \frac{\sum_{(n_x, n_y) \in p \cap p'} w_{xy}}{\ell(p')}, \tag{4.1}$$

where $p \cap p'$ denotes the set of edges shared by p and p'. For the overlap ratio we have that $0 \leq Sim(p, p') \leq 1$, where $Sim(p, p') = 0$ holds if p shares no edge with p', and $Sim(p, p') = 1$ holds if $p \equiv p'$. Since we consider only simple paths, i.e., cycle-free, the similarity between different paths is strictly lower than 1. We formalize the concept of alternative paths in the following definition.

Definition 5 (*Alternative Path*)**.** Let P be a set of paths from s to t and $\theta \in [0, 1)$ be a similarity threshold. A path p is alternative to P *iff* (a) p is also from s to t and (b) $\forall p_i \in P : Sim(p, p_i) \leq \theta$.

Consider the road network in Figure 4.1. 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 $Sim(p_1, p_0) = (w_{s,3} + w_{3,5})/\ell(p_0) = 6/8 = 0.75$, 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.

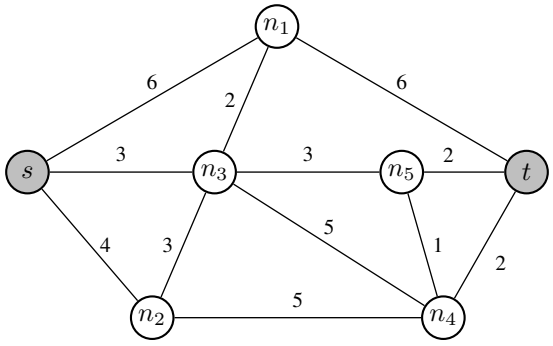


Figure 4.1: Running example.

Note that the asymmetric similarity metric of Equation (4.1) allows us to exclude needlessly long paths. Following up on our previous example, consider the shortest path p_0 and the paths $p_3 = \langle (s, n3), (n3, n4), (n4, t) \rangle$ and $p_4 = \langle (s, n3), (n3, n2), (n2, n4), (n4, t) \rangle$ with $\ell(p_3) = 10$ and $\ell(p_4) = 13$, respectively. The use of a symmetric similarity metric, such as the Jaccard coefficient, 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 (4.1) we avoid such cases. Furthermore, the similarity metric of Equation (4.1) guarantees the pairwise dissimilarity of p and p', as long as $\ell(p) \geq \ell(p')$. This observation is captured by the following lemma.

Lemma 3. Given two paths p, p' where $\ell(p) \geq \ell(p')$, and a similarity threshold θ, iff $Sim(p, p') < \theta$ then also $Sim(p', p) < \theta$.

Proof. Following Equation (4.1) for the similarity between p and p' and given that $\ell(p) \geq \ell(p')$, we have:

$$\frac{\sum_{(n_x,n_y)\in p\cap p'} w_{xy}}{\boldsymbol{\ell(p)}} \leq \frac{\sum_{(n_x,n_y)\in p\cap p'} w_{xy}}{\boldsymbol{\ell(p')}} \Rightarrow Sim(p', p) \leq Sim(p, p').$$

Therefore, iff $Sim(p, p') < \theta$ then also $Sim(p', p) < \theta$. □

4.2 k-Shortest Paths with Limited Overlap

We now introduce the problem of *k-Shortest Paths with Limited Overlap* (k-SPwLO). Given a source node s and a target node t, the goal is to recommend a set of k paths from s to t, sorted by length in increasing order such that (a) the shortest path $p_0(s{\rightarrow}t)$ is always included, (b) every path is dissimilar to its predecessors with respect to a similarity threshold θ, and (c) all k paths are as short as possible. Intuitively, this task can also be seen as progressively recommending k paths to the user starting from the shortest path p_0. Every path p_i recommended next is alternative to the set of paths already recommended and as short as possible. We formalize the k-SPwLO problem in the following definition.

Definition 6 (*k-SPwLO*)**.** Given a road network $G = (N, E)$, a source node $s \in N$, a target node $t \in N$, a requested number of paths k and a similarity threshold θ. A $k{-}SPwLO(s, t, \theta, k)$ query returns a set $\mathcal{P}_{\mathcal{LO}} = \{p_0, \ldots, p_{k-1}\}$ of k paths from s to t, for which the following holds:

- p_0 is the shortest path from s to t,
- $\forall p_i, p_j \in \mathcal{P}_{\mathcal{LO}}$ with $i \neq j$: $Sim(p_i, p_j) \leq \theta$, and
- $\forall p \notin \mathcal{P}_{\mathcal{LO}}$ one of the following two conditions holds:
  - $\ell(p) \geq \ell(p_i)$ holds $\forall p_i \in \mathcal{P}_{\mathcal{LO}}$
  - $\exists p_i \in \mathcal{P}_{\mathcal{LO}}$ with $\ell(p_i) \leq \ell(p)$ and $Sim(p, p_i) > \theta$.

The first bullet of Definition 6 guarantees that the shortest path $p_0(s{\rightarrow}t)$ is always in the result set $\mathcal{P}_{\mathcal{LO}}$. The second bullet assures that the recommended paths in $\mathcal{P}_{\mathcal{LO}}$ are sufficiently dissimilar to each other. Finally, the third bullet guarantees that $\mathcal{P}_{\mathcal{LO}}$ contains the shortest among all paths qualifying the previous constraints. More specifically, every path p that is not part of the k-SPwLO, is either longer than all the paths in the set (Condition 1), or there is a shorter path p_i in the set which causes p to violate the similarity constraint (Condition 2).

A naïve approach for computing k-SPwLO queries is to iterate over all paths from the source node s to the target node t and compute their pairwise similarity. However, such a solution is impractical; for real-world road networks, it is impossible to offline precompute and store all paths connecting any pair of nodes as the enumeration of all possible paths is a $\#P$-complete problem [83]. In what follows, we present three algorithms for processing k-SPwLO queries (Sections 4.3-4.5) which examine paths in increasing length order and built the result set progressively.

4.3 Baseline Algorithm

Our baseline algorithm, denoted by BSL, directly implements the idea of examining paths in increasing order of their length and works as follows. First, the shortest path $p_0(s \rightarrow t)$ is naturally considered and added to the result set $\mathcal{P}_{\mathcal{LO}}$. For every path p_c examined next, BSL investigates whether it is alternative to the already recommended paths. If yes, $\mathcal{P}_{\mathcal{LO}}$ is updated by adding p_c, otherwise, BSL proceeds to the next path in order until the result set $\mathcal{P}_{\mathcal{LO}}$ contains exactly k paths or all possible paths from s to t have been examined. In practice, BSL employs Yen's algorithm initially proposed in [89] and further optimized in [41, 59] to efficiently generate paths in length order.

Algorithm 4 illustrates the pseudocode of BSL. The result set $\mathcal{P}_{\mathcal{LO}}$ is initialized with p_0, i.e., the shortest path from s to t (Line 1). At each iteration (Lines 3-6), the loop of Yen's algorithm is employed in Line 3 to access the next path p_c from s to t in length order. Note that Yen's algorithm does not run from scratch; instead, it continues from its previous state. In Line 5, BSL checks whether the current path p_c is alternative to the already recommended paths and updates $\mathcal{P}_{\mathcal{LO}}$ in Line 6, if needed. Finally, the $\mathcal{P}_{\mathcal{LO}}$ result set is returned in Line 7.

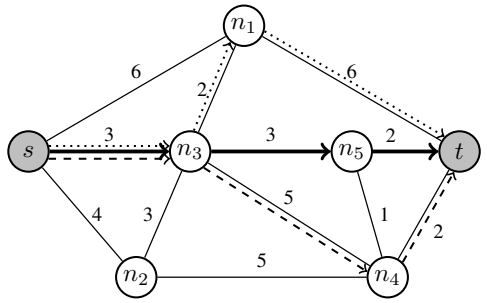
Algorithm 4: BSL

Input: Road network $G = (N, E)$, source node s, target node t, # of results k, sim. threshold θ
Output: Set $\mathcal{P}_{\mathcal{LO}}$ of k paths

```
1 initialize P_LO ← ∅;
2 p_c ← NULL;
3 while P_LO contains less than k paths and p_c not null do
4     p_c ← NextShortestPath(G, s, t);                    ▷ Use Yen's alg.
5     if Sim(p_c, p_i) ≤ θ for all paths p_i ∈ P_LO then
6         add p_c to P_LO;                                ▷ Update result set
7 return P_LO;
```

Example 2. We demonstrate BSL using the road network illustrated in Figure 4.2 and the k-SPwLO$(s, t, 0.5, 3)$ query. The first path constructed by BSL is the shortest path $p_0 = \langle s, n_3, n_5, t\rangle$ which is added to the result set. The next path examined by BSL is $p_1 = \langle (s, n_3), (n_3, n_5), (n_5, n_4), (n_4, t)\rangle$. Path p_1 is not added to the result set as $Sim(p_1, p_0) = 6/8 = 0.75$ which exceed the similarity threshold. Next, BSL examines path $p_2 = \langle (s, n_3), (n_3, n_4), (n_4, t)\rangle$. Since the similarity $Sim(p_2, p_0) = 3/8 = 0.375$ does not violate the similarity constraint, p_2 is added to the results set. BSL continues its execution until it examines path $p_4 = \langle (s, n_2), (n_2, n_4), (n_4, t)\rangle$ and adds it to the result set. Then BSL returns the result set $\mathcal{P}_{\mathcal{LO}} = \{p_0, p_2, p_4\}$.

Despite its simplicity, BSL cannot be used even for small networks; our tests in Section 4.7 clearly verify this argument. As the algorithm does not employ any pruning techniques, a large number of paths need to be examined in length order. For this task, BSL builds upon an already expensive operator, i.e., the computation of K-shortest paths. In practice, returning the next path in length order requires a number of intermediate/candidate paths to be constructed first. In the worst case scenario, BSL has to construct all paths connecting source node s and target node t, a problem which, as we already mentioned, is $\#P$-complete [83].



P_{all}		
p_0	$\langle(s,n_3),(n_3,n_5),(n_5,t)\rangle$	8
p_1	$\langle(s,n_3),(n_3,n_5),(n_5,n_4),(n_4,t)\rangle$	9
p_2	$\langle(s,n_3),(n_3,n_4),(n_4,t)\rangle$	10
p_3	$\langle(s,n_3),(n_3,n_4),(n_4,n_5),(n_5,t)\rangle$	11
p_4	$\langle(s,n_3),(n_3,n_1),(n_1,t)\rangle$	11
p_5	$\langle(s,n_2),(n_2,n_4),(n_4,t)\rangle$	11
p_6	$\langle(s,n_2),(n_2,n_4),(n_4,n_5),(n_5,t)\rangle$	12
p_7	$\langle(s,n_2),(n_2,n_3),(n_3,n_5),(n_5,t)\rangle$	12
p_8	$\langle(s,n_1),(n_1,t)\rangle$	12
p_9	$\langle(s,n_3),(n_3,n_2),(n_2,n_4),(n_4,t)\rangle$	13
(...)		

⟶ shortest path (p_0) - -→ 1st alt. (p_2) ⋯→ 2nd alt. (p_4)

Figure 4.2: Computation of k-SPwLO (s,t,0.5,3) query with BSL.

4.4 OnePass Algorithm

OnePass, our second algorithm, traverses the road network starting from source node s expanding only paths that satisfy the similarity constraint θ. Hence, the number of paths examined by OnePass for the computation of a k-SPwLO query is significantly lower than the number of paths examined by BSL. In what follows, we formally introduce this pruning criterion and we present the OnePass algorithm which employs it.

4.4.1 Pruning Overlaping Sub-paths

Let $p(s{\rightarrow}n)$ be a path connecting source node s to some node n, and $p_i(s{\rightarrow}t) \in \mathcal{P}_{\mathcal{LO}}$ 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 with p_i is at least equal to the similarity of p with p_i, i.e., $Sim(p', p_i) \geq Sim(p, p_i)$. Hence, given a similarity threshold θ, if there exists $p_i \in \mathcal{P}_{\mathcal{LO}}$ such that $Sim(p, p_i) \geq \theta$, path p can be safely discarded; no extension of p can possibly be an alternative path. This pruning criterion is formally captured by the following lemma:

Lemma 4. Let $\mathcal{P}_{\mathcal{LO}}$ be the set of already recommended paths. If p is an alternative path to $\mathcal{P}_{\mathcal{LO}}$ with respect to a similarity threshold θ then $Sim(p', p_i) \leq \theta$ holds for every subpath p' of p and any $p_i \in \mathcal{P}_{\mathcal{LO}}$.

Proof. Let p be an alternative path to $\mathcal{P}_{\mathcal{LO}}$, i.e., $\forall p_i \in \mathcal{P}_{\mathcal{LO}} : Sim(p, p_i) \leq \theta$. Consider a subpath p' of p and denote by $E_{p-p'}$ the set of edges contained in p but not in p'. Following Equation (4.1), for the overlap ratio of p and p' and $\forall p_i \in \mathcal{P}_{\mathcal{LO}}$ we have:

$$Sim(p, p_i) = Sim(p', p_i) + \frac{\sum_{(n_x,n_y)\in E_{p-p'}} w_{xy}}{\ell(p_i)}$$

from which we get

$$Sim(p', p_i) \leq Sim(p, p_i) \Rightarrow Sim(p', p_i) \leq \theta.$$

□

Lemma 4 also demonstrates the monotonicity of the similarity function of Equation 4.1. Let a path $p_i \in \mathcal{P}_{\mathcal{LO}}$, a path p and a path p' which is created by extending p, hence p is a subpath of p'. According to Lemma 4 the similarity $Sim(p', p_i)$ can only be larger or equal to $Sim(p, p_i)$ and it will not decrease under any circumstances.

4.4.2 The OnePass Algorithm

Next, we present a label-setting algorithm, termed OnePass, which employs the pruning criterion of Lemma 4 to evaluate k-SPwLO queries. The algorithm has the following key features. OnePass traverses the road network expanding every path from source node s that qualifies the pruning criterion of Lemma 4. Similar to all label-setting algorithms, OnePass maintains a set of labels $\Lambda(n)$, where each label $\langle n, p(s{\rightarrow}n)\rangle$ represents a path from s to n[1]. The paths are examined in increasing order of their length. Each time a new path $p(s{\rightarrow}t)$ is added to $\mathcal{P}_{\mathcal{LO}}$, an update procedure takes place for all remaining paths p' in $\Lambda(n)$. More specifically, for each path p' in $\Lambda(n)$, OnePass computes the similarity of p' with the newly found $p(s{\rightarrow}t)$ path and removes all paths p' from $\Lambda(n)$ for which the similarity with p exceeds the similarity threshold θ (Lemma 4). Finally, the algorithm terminates when either k paths are recommended or all paths from s to t qualifying Lemma 4 have been examined.

Algorithm 5 illustrates the pseudocode of OnePass. The result set $\mathcal{P}_{\mathcal{LO}}$ is initialized with p_0, i.e., the shortest path from s to t, in Line 1. The algorithm uses a min-priority queue $\mathcal{Q}$ (initialized with $\langle s, \emptyset\rangle$ in Line 2) to traverse the road network. In between Lines 4 and 15, OnePass examines the contents of $\mathcal{Q}$ until either k paths are recommended or the queue is depleted. At each round, current label $\langle n, p_n\rangle$ is popped from $\mathcal{Q}$ (Line 5). If node n is the target t then p_n is recommended, i.e., added to $\mathcal{P}_{\mathcal{LO}}$ (Line 7). Next, between Lines 8-10, for each label $\langle n_q, p_q\rangle$ in $\mathcal{Q}$, OnePass computes the similarity ratio of p_q with the newly recommended path p_n and determines whether p_q qualifies the pruning criterion of Lemma 4; in particular, if $Sim(p_q, p_n) > \theta$ 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 12-15), provided that the new path $p_c \leftarrow p_n \circ (n, n'')$ qualifies the pruning criterion of Lemma 4 (Line 14). Finally, the result set $\mathcal{P}_{\mathcal{LO}}$ is returned in Line 16.

To achieve an efficient implementation, OnePass stores for each label $\langle n, p_n\rangle$ a vector V_{Sim} containing the overlap ratio of p_n with all paths that were in $\mathcal{P}_{\mathcal{LO}}$ at the time when the label was created. Due to the monotonicity of Equation 4.1, 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 OnePass performs lazy updates for $\mathcal{Q}$. In order to consider results in $\mathcal{P}_{\mathcal{LO}}$ that are added after the creation of the label, each time a label is popped, OnePass compares the size of V_{Sim} stored in the popped label to $|\mathcal{P}_{\mathcal{LO}}|$. If the size of V_{Sim} is smaller than the number of paths currently in $\mathcal{P}_{\mathcal{LO}}$, OnePass computes the missing overlaps and updates V_{Sim} accordingly.

Example 3. We demonstrate OnePass using the road network of Figure 4.3 and the k-SPwLO$(s, t, 0.5, 2)$ query. During the initialization phase, 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 the result set $\mathcal{P}_{\mathcal{LO}}$. Starting from s, the first path examined by OnePass is p_1. The similarity $Sim(p_1, p_0) = 3/8 = 0.375$ is below

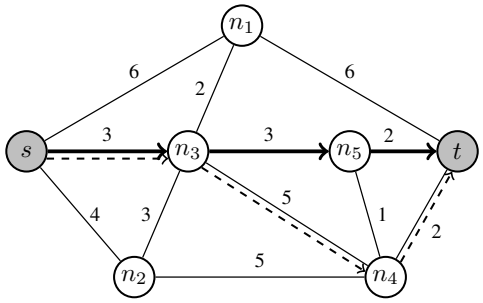
[1] In practice, OnePass stores only the predecessor of each label during the expansion. By tracing backwards each step of the expansion, the actual path can be retrieved at any time.

Algorithm 5: OnePass

Input: Road network $G = (N, E)$, source node s, target node t, # of results k, sim. threshold θ
Output: Set $\mathcal{P}_{\mathcal{LO}}$ of k paths

```
1  P_LO ← {shortest path p_0(s → t)};
2  initialize min-priority queue Q with ⟨s, ∅⟩;
3  ∀n ∈ N : Λ(n) ← ∅;
4  while P_LO contains less than k paths and Q not empty do
5      ⟨n, p_n⟩ ← Q.pop();                                   ▷ Current path
6      if n = t then
7          P_LO ← P_LO ∪ {p_n};                              ▷ Update result set
8          foreach label ⟨n', ℓ(p_n')⟩ in Q do
9              if Sim(p_n', p_i) > θ, ∀p_i ∈ P_LO then
10                 remove ⟨n', ℓ(p_n')⟩ from Q ;              ▷ Lemma 4
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
15                 Q.push(⟨n_c, p_c⟩);
16 return P_LO;
```

the similarity threshold $\theta = 0.5$; hence 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 $Sim(p_3, p_0) = 3/8 = 0.375$ and $Sim(p_4, p_0) = 0$ do not exceed the similarity threshold. On the contrary, for path p_5 the similarity $Sim(p_5, p_0) = 6/8 = 0.75$ exceeds the similarity threshold of 0.5 and so, path p_5 is pruned. OnePass continues its execution until the alternative path p_{14} with $\ell(p_{14}) = 10$ is found and subsequently added to $\mathcal{P}_{\mathcal{LO}}$.



Examined paths		
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	$\langle(s, n_3), (n_3, n_5)\rangle$	6
$(\ldots)$		
p_{12}	$\langle(s, n_2), (n_2, n_3), (n_3, n_5)\rangle$	10
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

$\longrightarrow$ shortest path (p_0) $\dashrightarrow$ 1st alternative (p_{14})

Figure 4.3: Computation of the k-SPwLO (s,t,0.5,2) query with OnePass.

Complexity analysis. OnePass can be viewed as an extension of *Fox's algorithm* [35] 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, and 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. OnePass terminates when either k paths are recommended or all paths from s to t qualifying Lemma 4 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.

4.5 MultiPass Algorithm

Despite employing the pruning criterion of Lemma 4, 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.

4.5.1 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 4.4. 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, p_j are extended to reach target t following the same path $p(n{\rightarrow}t)$, 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 similarity of p_i with p_0 is equal or lower than the similarity of p_j with p_0. Due to the monotonicity of the similarity function (Equation (4.1)), any extension of p_i to n will have the same or less similarity 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 similarity with p_0; thus, p_j can be pruned.

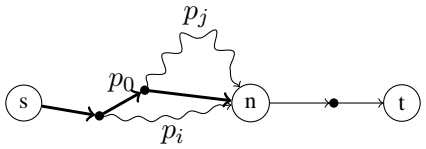


Figure 4.4: Pruning paths with Lemma 5.

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 similarity $Sim(p_i, p)$ is lower or equal to $Sim(p_j, p)$. This pruning criterion is formally captured by the following lemma:

Lemma 5. 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 $\forall p \in P : Sim(p_i, p) \leq Sim(p_j, p)$ 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 an alternative path, $Sim(p'_j,p) \leq \theta$ holds $\forall p \in P$, and following Lemma 4 $Sim(p_j,p) \leq \theta$ also holds $\forall p \in P$. Furthermore, due to the $\forall p \in \mathcal{P}_{\mathcal{LO}}$: $Sim(p_i,p) \leq Sim(p_j,p)$ assumption of Lemma 5, 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 5 can be utilized to compute the shortest alternative path 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 $p'(s{\rightarrow}n)$ 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 similarity $Sim(p',p)$ is higher than $Sim(p_n,p)$ 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 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.

The aforementioned pruning criterion cannot be employed directly for the computation of k-SPwLO queries. Consider again the example in Figure 4.4. Let p_0 be the only path in the set of currently recommended alternative paths P. If during the search for the alternative path 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 path, i.e., the alternative path to $P = \{p_0, p_1\}$, p_i may be pruned much earlier by Lemma 4. 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 path from the beginning.

4.5.2 The MultiPass Algorithm

Next, we present MultiPass, a label-setting algorithm which employs both pruning criteria of Lemma 4 and Lemma 5 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 $\langle n, p(s{\rightarrow}n)\rangle$[2]. 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 4) and (b) its extension can possibly lead to the shortest alternative path during the current iteration (Lemma 5). 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) anymore. As soon as a path to target t is found, MultiPass terminates current round, discards all stored labels, and re-traverses the

[2]Similar to OnePass, MultiPass stores only the predecessor of each label during the expansion.

Algorithm 6: MultiPass

Input: Road network $G = (N, E)$, source node s, target node t, # of results k, similarity threshold θ
Output: Set $\mathcal{P}_{\mathcal{LO}}$ 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 4
15                 else if ∃⟨n_c, p'_c⟩ ∈ Λ(n_c) : p'_c ≺_{P_LO} p_c then
16                     continue;                                 ▷ Pruned by Lemma 5
17                 else
18                     remove from Q and Λ(n_c) all ⟨n_c, p'_c⟩ : p_c ≺_{P_LO} p'_c;   ▷ Lemma 5
19                     Q.push(⟨n_c, p_c⟩);
20                     Λ(n_c) ← Λ(n_c) ∪ {⟨n_c, p_c⟩};
21 return P_LO;
```

network from source s. The algorithm terminates after k paths are added to result set $\mathcal{P}_{\mathcal{LO}}$ or the last round 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.

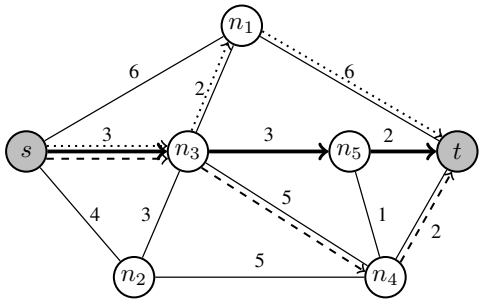
Algorithm 6 illustrates the pseudocode of MultiPass. The result set $\mathcal{P}_{\mathcal{LO}}$ is initialized with the shortest path $p_0(s \to t)$ (Line 1). The algorithm employs a min priority queue $\mathcal{Q}$ to traverse the road network. Before each traversal round, $\mathcal{Q}$ is initialized to $\langle s, \emptyset \rangle$ (Line 3) and each node n is associated with a (initially empty) set of labels $\Lambda(n)$ (Line 4). At each round in between Lines 5 and 20, MultiPass 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 $\mathcal{P}_{\mathcal{LO}}$ and the round terminates (Lines 7–9). Otherwise, MultiPass expands the current path p_n considering all outgoing edges (n, n_c) (Lines 10-16). Each new path $p_c \leftarrow p_n \circ (n, n_c)$ (Line 12) is evaluated against the pruning criteria of Lemma 4 (Lines 13-14) and Lemma 5 (Lines 15-16). 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 $p_c \prec_{\mathcal{P}_{\mathcal{LO}}} p'_n$ (Line 19). Finally, the new label is added to $\mathcal{Q}$ (Line 19) and $\Lambda(n_c)$ (Line 20) and the next label is popped from $\mathcal{Q}$. The loop terminates after k paths are added to $\mathcal{P}_{\mathcal{LO}}$ or the last round failed to find an alternative path. Finally, the $\mathcal{P}_{\mathcal{LO}}$ result set is returned in Line 21.

To achieve an efficient implementation, similar to OnePass, MultiPass also stores for each label $\langle n, p_n \rangle$ a vector V_{Sim} containing the overlap ratio of p_n with all paths that were in $\mathcal{P}_{\mathcal{LO}}$ at the time when the label was created. The overlap ratios stored in V_{Sim} are updated incrementally. When a new label is added to $\mathcal{Q}$, MultiPass performs lazy updates for $\mathcal{Q}$. Finally, each time a label is popped, MultiPass compares the size of V_{Sim} stored in the popped label to

$|\mathcal{P}_{\mathcal{LO}}|$ and computes any missing overlaps.

Example 4. We demonstrate MultiPass using the road network of Figure 4.5 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 the result set $\mathcal{P}_{\mathcal{LO}}$. Starting from s, the first path examined by MultiPass is p_1. The similarity $Sim(p_1,p_0) = 3/8 = 0.375$ is below the similarity threshold $\theta = 0.5$; hence p_1 is not pruned. The same holds for p_2 the next path examined MultiPass. Subsequently, MultiPass examines paths p_3, p_4 and p_5. Path p_3 is not pruned as the similarity $Sim(p_3,p_0) = 3/8 = 0.375$ does not exceed the similarity threshold. For path p_4 the similarity $Sim(p_4,p_0) = 0.375$ also does not exceed the similarity threshold. Since node n_1 has already been visited by path p_3 though, we also check Lemma 5. We have $Sim(p_3,p_0) > Sim(p_4,p_0)$ and for the length $\ell(p_3) < \ell(p_4)$. Therefore, Lemma 5 cannot be applied and path p_4 is not pruned. On the contrary, for path p_5 the similarity $Sim(p_5,p_0) = 6/8 = 0.75$ exceeds the similarity threshold of 0.5 and so, path p_5 is pruned by Lemma 4. 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 $\mathcal{P}_{\mathcal{LO}}$.

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 $\mathcal{P}_{\mathcal{LO}}$.



Examined paths - round 1		
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_{12}	$\langle(s,n_2),(n_2,n_3),(n_3,n_5)\rangle$	10
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		
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'_{12}	$\langle(s,n_2),(n_2,n_4),(n_4,n_5)\rangle$	10
p'_{13}	$\langle(s,n_3),(n_3,n_1),(n_1,t)\rangle$	11

⟶ shortest path (p_0) - - → 1st alternative (p_{15}) ⋯→ 2nd alternative (p_4)

Figure 4.5: Computation of the k-SPwLO (s,t,0.5,3) query with MultiPass.

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 5, the number of paths that MultiPass has to examine (including the paths examined multiple times) is lower than the 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 5 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 the number of requested paths k or the similarity threshold θ. For the computation of $p_0(s{\rightarrow}t)$

any shortest path algorithm can be employed, e.g., Dijkstra's algorithm [32] which requires $\mathcal{O}(|E| + |N| \cdot \log |N|)$.

To find each subsequent alternative path, all paths from source s that qualify the pruning criterion of Lemma 4 are expanded. However, as the value of the similarity threshold θ approaches 1, the number of paths pruned by Lemma 4 significantly drops. In such a case, the result set is the K-Shortest paths. Furthermore in practice, there exists no formula for estimating the number of paths pruned by the pruning criterion of Lemma 5. Hence, each round of MultiPass becomes equivalent to Fox's algorithm [35] 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 $k-1$ rounds are required to compute a k-SPwLO query, the total runtime complexity of MultiPass is $\mathcal{O}(k(|E| + K \cdot |N| \cdot \log |N|))$ where $K >> k$. As we explained in Section 4.2, the number of paths that have 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 5 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 though, in practice it is much more efficient than OnePass.

4.6 Optimization

To further improve the performance of OnePass and MultiPass we employ 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, both algorithm traverse the network in an A^*-like fashion and direct 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 [71]. In practice, at the beginning of the execution of both OnePass and MultiPass, 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.

4.7 Experimental Evaluation

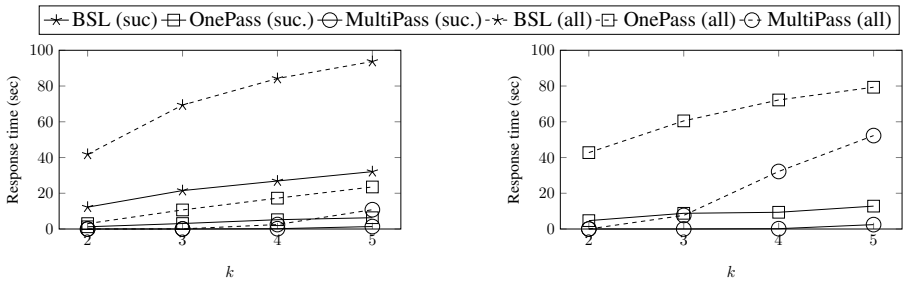
4.7.1 Experimental Setup

To demonstrate the efficiency of our algorithms, we measure the performance of BSL, OnePass and MultiPass using real road networks. 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, the similarity threshold θ and the distance between the query points in a similar fashion as in Section 3.4.1. In the first two experiments, we vary one of the two parameters and fix the other to its default value: 3 for k and 0.5 for θ; in the third experiment we fix both k and θ to their default values. All algorithms were implemented in C++ using the C++11 standard and compiled using GNU G++ compiler (version 4.8.1). All experiments were executed on an Ubuntu Linux server with 4 Intel Xeon X5550 (2.67GHz) processors and 48GB of RAM.

Due to the high execution time of BSL and 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. Each query that fails to execute within the predefined timeout is considered failed.

4.7.2 Performance

The continuous lines in Figures 4.6-4.8 show the time for the queries for which all algorithms finished their execution in less than 120 seconds (successful queries), whereas the dashed lines show the time for all 1,000 queries including those which did not finish within 120 seconds (timed-out/failed queries). In both figures we observe in that BSL is clearly impractical. In fact, without the timeout, the response time of BSL would have been several orders of magnitude higher than MultiPass (even with timeout MultiPass is from one to almost 4 orders of magnitude faster). Naturally, this is due to the large number of paths examined by BSL in length order, i.e., the paths returned by Yen's algorithm. Hence, for presentation purposes, we report the results of BSL only for Oldenburg.



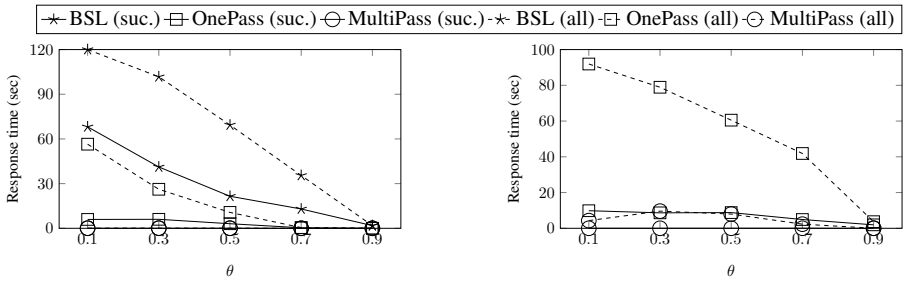
(a) Oldenburg (θ=50%) (b) San Joaquin (θ=50%)

Figure 4.6: Performance comparison varying requested paths k (θ=50%).

Next, 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. In most cases, MultiPass is at least two times faster. Another interesting observation is that the performance curve of OnePass is almost linear, that is each iteration requires approximately the same time, e.g., for Oldenburg (Figure 4.6a) 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 Figure 4.7 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 alternative paths can be very similar. Furthermore, the fact that the performance of OnePass and MultiPass deteriorates as θ increases, indicates that the pruning power of Lemma 4 deteriorates. However at the same time, we observe that for small values of θ the performance of MultiPass improves. This result indicates that for small values of θ the pruning power of Lemma 5 increases.

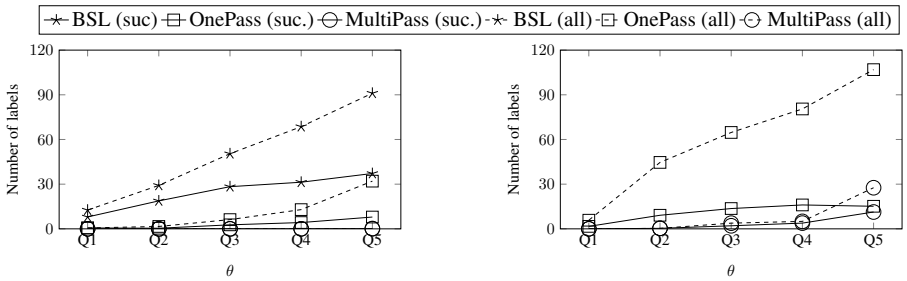
Finally, Figure 4.8 compares the response time of all algorithms varying the distance between query points. For all algorithms we observe that their performance deteriorates as the



(a) Oldenburg (k=3) (b) San Joaquin (k=3)

Figure 4.7: Performance comparison varying similarity threshold θ (k=3).

distance between query points increases. Once again it is clear that BSL is the slowest algorithm, OnePass is faster than BSL, while MultiPass is clearly the fastest algorithm as it beats both BSL and OnePass. However, in Figure 4.8b we also observe that, even though MultiPass is the fastest algorithm, for Q_5, where the distance between query points is the highest, MultiPass requires more than ten seconds to compute the k-SPwLO; thus, for larger road networks, MultiPass may not be very practical.



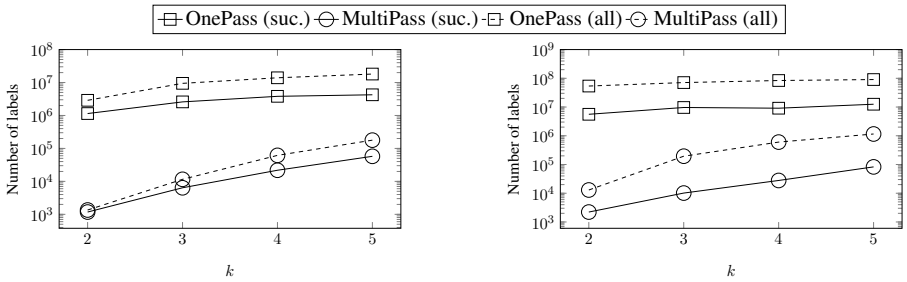
(a) Oldenburg (b) San Joaquin

Figure 4.8: Performance comparison varying distance between s and t nodes (k=3, $\theta = 50\%$).

4.7.3 Memory Consumption

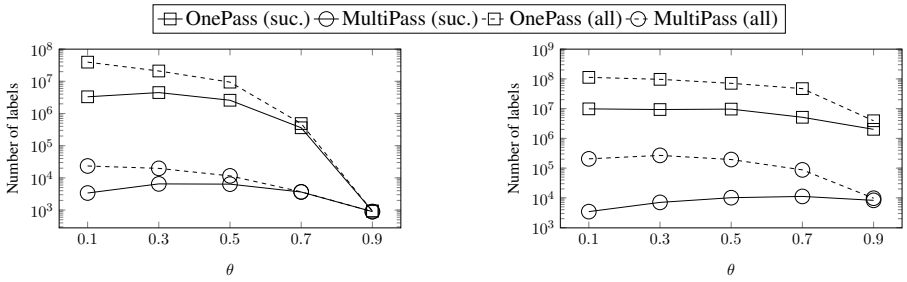
To provide a better insight on the performance of MultiPass and OnePass, we report in Figures 4.9 and 4.10 the number of labels each algorithm needs to examine before returning the k-SPwLO result. We observe that, in all scenarios, MultiPass examines significantly fewer paths than OnePass (even though 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, the result verifies our claim in Section 4.7.2 regarding the pruning power of Lemmas 4 and reflm:prune2. As θ increases, the pruning power of Lemma 4 deteriorates, and both OnePass and MultiPass construct more paths. 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 5 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.



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

Figure 4.9: Comparison of examined paths varying requested paths k (θ=5 − %).



(a) Oldenburg (k=3) (b) San Joaquin (k=3)

Figure 4.10: Comparison of examined paths varying similarity threshold θ (k=3).

4.7.4 Failed Queries

Finally, in Table 4.1 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 BSL is clearly the highest. Also, 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<0.5. For the road network of San Joaquin, apart from the case where k=3 and θ=0.9, the failure rate of OnePass is more than 30%, even when the timeout is set to 120 seconds. On the contrary, the failure 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%, except for the cases where k>3. However, even in cases where the failure rate of MultiPass is the highest (θ=0.5 and k>3), it is still significantly lower than the failure rate of OnePass and BSL.

4.8 Summary

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. We formalized this task using the k-SPwLO query and designed three algorithms to evaluate such queries: BSL, a baseline solution based on Yen's algorithm for the $K-$shortest paths; OnePass, a label-setting algorithm which traverses the network once and prunes partial paths that cannot lead to a solution; and our best algorithm, MultiPass, which progressively computes the result

Table 4.1: Failure rate (%) for timeout set to 30, 60 and 120 sec.

road network	θ	k	BSL			OnePass			MultiPass		
			30	60	120	30	60	120	30	60	120
Oldenburg	0.1	3	98.1	95.8	93.6	46.9	45.4	44.5	0	0	0
	0.3	3	83.9	76.5	74.2	22.8	20.5	17.8	0	0	0
	0.5	3	61.3	54.7	48.5	9.1	7.7	6.6	0	0	0
	0.7	3	31.4	25.6	21	0.4	0.1	0.1	0	0	0
	0.9	3	0.4	0	0	0	0	0	0	0	0
	0.5	2	36.6	31.3	27.4	2.7	0.2	1.5	0	0	0
	0.5	4	74.5	67.6	61.5	15.2	12.8	10.7	2.6	1.4	0.7
	0.5	5	82.1	76.3	70.1	20.4	17.5	15	9.6	7.4	6.2
San Joaquin	0.1	3	99.3	99.1	98.7	77.5	76	74.6	3.2	1.8	1.3
	0.3	3	97.5	96.6	96.2	66.8	65.2	63.2	8.1	5.6	4.4
	0.5	3	93.8	93.1	91.3	52.3	49.4	46.6	6.8	5.1	3.5
	0.7	3	86.2	83.4	81.3	35.8	33.6	32.1	2.1	0.9	0.3
	0.9	3	61.3	54.1	46.7	3.1	2.3	1.6	0	0	0
	0.5	2	85.1	82.6	80.6	36.5	34.1	33.2	0	0	0
	0.5	4	96.2	94.7	93.8	61	59.3	56.8	28.9	25.5	22.5
	0.5	5	97.2	96.3	95.5	67.5	64.8	62.3	45.2	42	39.1

set after traversing the network $k-1$ times, and reduces the search space by employing two powerful pruning criteria. Through an extensive experimental evaluation using real road networks, we demonstrated that MultiPass is superior to both BSL and OnePass; hence, MultiPass is the fastest exact algorithm for computing k-SPwLO queries.

CHAPTER 5

Heuristic Algorithms for k-SPwLO Queries

In the previous chapter, we showed that MultiPass is the most efficient algorithm for computing k-SPwLO. However, despite employing two pruning criteria, MultiPass still has to examine a large number of paths, which essentially renders the algorithm not applicable on large road networks. In view of this, in this chapter, we investigate heuristic algorithms which examine only a fraction of the paths required to process k-SPwLO queries. We first discuss a baseline algorithm termed SVP$^+$, which builds on top of existing literature. Then, we propose two novel heuristic algorithms: OnePass$^+$ employs the same pruning criteria as MultiPass, but traverses the network only once. Thereby, 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.

5.1 Baseline Heuristic Algorithm

Our baseline algorithm, denoted by SVP$^+$, builds upon the notion of *single-via paths*, which was introduced as an alternative routing technique in [4]. Given a road network $G = (N, E)$, a source node s and a target node t the single-via path $p_{svp}(n)$ of a node $n \in N \setminus \{s, t\}$ is the concatenation of the shortest path $p_s(s \to n)$ and the shortest path $p_s(n \to t)$. Naturally, only simple single-via paths are considered. The method compares each single-via path with the shortest path using a set of objective criteria, i.e., local optimality and stretch. However, the method considers the similarity of single-via paths only with the shortest path and disregards the pairwise dissimilarity of all results; thus it cannot be employed directly for computing k-SPwLO queries.

Instead of employing the objective criteria proposed in [4], our SVP$^+$ algorithm iterates over the set of single-via paths aiming to find a subset of k paths that 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 4.2. However, instead of iterating over all possible $s{-}t$ paths and computing their pairwise similarity, SVP$^+$ iterates over the much smaller set of single-via paths.

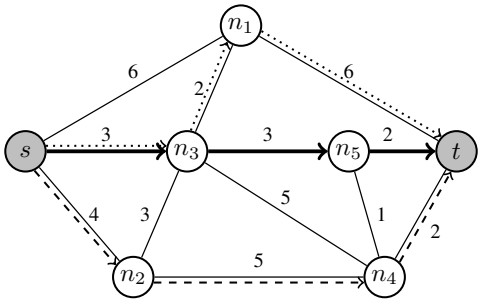
Algorithm 7 illustrates the pseudocode of SVP$^+$. First, the algorithm computes two shortest path trees, one from s to every node n of G (Line 2) and a reverse one from every node n of G to t (Line 3). During this step all distances $d(s, n)$ and $d(n, t)$ are computed. The algorithm orders the nodes based on the sum $d(s, n)+d(n, t)$, 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 $\mathcal{P}_{\mathcal{LO}}$ 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 $\mathcal{P}_{\mathcal{LO}}$; if so, p_n is added to $\mathcal{P}_{\mathcal{LO}}$ (Line 11). The algorithm terminates and returns the $\mathcal{P}_{\mathcal{LO}}$ set when either k paths have been added to $\mathcal{P}_{\mathcal{LO}}$ or there exist no more single-via paths to examine, i.e., queue $\mathcal{Q}$ is depleted.

Algorithm 7: SVP$^+$

Input: Road network $G = (N, E)$, source node s, target node t, # of results k, sim. threshold θ
Output: Set $\mathcal{P}_{\mathcal{LO}}$ 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 $\mathcal{Q}.push(\langle n, d(s, n)+d(n, t)\rangle)$;
6 $\mathcal{P}_{\mathcal{LO}} \leftarrow \{$shortest path $p_0(s \to t)\}$;
7 **while** $\mathcal{P}_{\mathcal{LO}}$ *contains less than* k *paths* ***and*** $\mathcal{Q}$ *not empty* **do**
8 $\langle n, d(s, n)+d(n, t)\rangle \leftarrow \mathcal{Q}.pop()$;
9 $p_n \leftarrow$ `RetrieveSingleViaPath`$(T_{s \to N}, T_{N \to t}, n)$;
10 **if** $Sim(p_n, p) \leq \theta$ *for all* $p \in \mathcal{P}_{\mathcal{LO}}$ **then**
11 add p_n to $\mathcal{P}_{\mathcal{LO}}$; ▷ `Update result set`
12 **return** $\mathcal{P}_{\mathcal{LO}}$;

Example 5. We demonstrate SVP$^+$ using the road network of Figure 5.1 and the k-SPwLO$(s, t, 0.5, 3)$ query. First, SVP$^+$ adds the shortest path $p(s{\rightarrow}t) = \langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$ to the result set. Then, SVP$^+$ iterates over the set of single-via paths in length order. The table in Figure 5.1 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 with the shortest path exceed the similarity threshold θ. Single-via paths $p_{svp}(n_3)$ and $p_{svp}(n_5)$ are the same with the shortest path, hence $Sim(p_{svp}(n_3), p) = Sim(p_{svp}(n_5), p) = 1$. In fact, the single-via path of every node on the shortest path is the shortest path. 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 single-via path $p_{svp}(n_2)$ for which the similarity with the shortest path is $Sim(p_{svp}(n_2), p) = 0$. Hence, $p_{svp}(n_2)$ is recommended, i.e., 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.



Single-via paths		*length*
$p_{svp}(n_3)$	$\langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$	8
$p_{svp}(n_5)$	$\langle (s, n_3), (n_3, n_5), (n_5, t) \rangle$	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

$\longrightarrow$ shortest path (p_0) $\dashrightarrow$ 1st alternative (p_{15}) $\cdots\rightarrow$ 2nd alternative (p_4)

Figure 5.1: Example of SVP$^+$.

Notice that in Example 5, SVP$^+$ fails to find the exact result for the given k-SPwLO query. In particular, path $p = \langle (s, n_3), (n_3, n_4), (n_4, t) \rangle$, which is in the exact k-SPwLO result, is not a single-via path; hence, p is not examined by SVP$^+$.

5.2 The OnePass$^+$ algorithm

Our first heuristic algorithm, denoted by OnePass$^+$, combines the feature of OnePass to scan the road network only once with the pruning power of Lemma 5, which is also used by MultiPass. OnePass$^+$ has the following key features. Given a source node s and a target node t, OnePass$^+$ traverses the road network expanding every path $p(s{\rightarrow}n)$ from source s to a node n that qualifies both Lemma 4 and Lemma 5. This procedure is the same with each distinct round of MultiPass. In contrast to MultiPass though, each time a new path is added to the result set $\mathcal{P}_{\mathcal{LO}}$, an update procedure takes place for all remaining incomplete paths $p(s{\rightarrow}n)$. In particular, for every incomplete path $p(s{\rightarrow}n)$, OnePass$^+$ computes the overlap of p with the newly found result and, then, p is checked against Lemma 4 with respect to the updated $\mathcal{P}_{\mathcal{LO}}$. The same update procedure is also employed by OnePass. The algorithm terminates

when either k paths are added to the result set or all paths from s to t qualifying both Lemma 4 and Lemma 5 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. In Section 4.5, we explained that for the MultiPass algorithm to find the exact solution, the restart is required. 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$^+$ and, hence, OnePass$^+$ cannot guarantee that the exact solution is found. Nevertheless, this case applies to only a small subset of the examined paths. The average length of paths in the result set of OnePass$^+$ is expected to be close to the optimal one.

Algorithm 8 illustrates the pseudocode of OnePass$^+$. The algorithm employs a min priority queue $\mathcal{Q}$ (initialized with source s in Line 1) to traverse the road network. Result set $\mathcal{P}_{\mathcal{LO}}$ is initialized with p_0, i.e., the shortest path from s to t (Line 2). In between Lines 4 and 21, OnePass$^+$ examines the contents of $\mathcal{Q}$ until either k paths are found and added to $\mathcal{P}_{\mathcal{LO}}$ 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 (Line 6), then p_n is added to $\mathcal{P}_{\mathcal{LO}}$ (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 4 (Lines 14-15) and Lemma 5 (Lines 16-17) and updates $\mathcal{Q}$ and $\Lambda(n_c)$ accordingly. 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 $\mathcal{Q}$. Finally, the result set $\mathcal{P}_{\mathcal{LO}}$ is returned in Line 20.

Similar to OnePass and 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 $\mathcal{P}_{\mathcal{LO}}$ 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 $\mathcal{Q}$.

Complexity Analysis 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. Naturally, the cost of this step is independent of the number of requested paths k or the similarity threshold θ. 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 4 and Lemma 5. 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

Our second heuristic algorithm, denoted by ESX, computes k-SPwLO by iteratively excluding edges from the road network. The idea behind ESX is the following. Given a road network $G = (N, E)$, a source node s and a target node t, a requested number of paths k and a

Algorithm 8: OnePass$^+$

Input: Road network $G = (N, E)$, source node s, target node t, # of results k, sim. threshold θ
Output: Set $\mathcal{P}_{\mathcal{LO}}$ of k paths

```
1  initialize min-priority queue Q with ⟨s, ∅⟩;
2  P_LO ← {shortest path p_0(s → t)};
3  ∀n ∈ N : Λ(n) ← ∅;
4  while P_LO contains less than k paths and Q not empty do
5      ⟨n, p_n⟩ ← Q.pop();                                   ▷ Current path
6      if n = t then
7          P_LO ← P_LO ∪ {p_n};                              ▷ Update result set
8          foreach label ⟨n', ℓ(p_n')⟩ in Q do
9              if Sim(p_n', p_i) > θ, ∀p_i ∈ P_LO then
10                 remove ⟨n', ℓ(p_n')⟩ from Q ;              ▷ Lemma 4
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
15                 continue;                                 ▷ Pruned by Lemma 4
16             else if ∃⟨n_c, p_c'⟩ ∈ Λ(n_c) : p_c' ≺_{P_LO} p_c then
17                 continue;                                 ▷ Pruned by Lemma 5
18             else
19                 remove from Q and Λ(n_c) all ⟨n_c, p_c'⟩ : p_c ≺_{P_LO} p_c';   ▷ Lemma 5
20                 Q.push(⟨n_c, p_c⟩);
21                 Λ(n_c) ← Λ(n_c) ∪ {⟨n_c, p_c⟩};
22 return P_LO;
```

similarity threshold θ, the algorithm first adds the shortest path p_0 to the result set $\mathcal{P}_{\mathcal{LO}}$, 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[1]. If the similarity of path p_c with p_0 does not exceed the similarity threshold θ, then p_c is added to the result set $\mathcal{P}_{\mathcal{LO}}$. Otherwise, the algorithm proceeds with removing more edges from the road network. If $\mathcal{P}_{\mathcal{LO}}$ contains more than one paths, ESX removes an edge from path $p \in \mathcal{P}_{\mathcal{LO}}$ for which the similarity $Sim(p_c, p)$ is the highest. At each iteration, ESX removes only one edge from some path in $\mathcal{P}_{\mathcal{LO}}$. 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 from the road network at any iteration.

The order in which we remove the edges from the road network affects both the quality of

[1]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.

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 to the next result faster. As it is also prohibitively expensive to compute the all-pairs shortest paths and count the number of shortest paths crossing each edge, ESX performs a local check. Given an edge $e(a, b)$ on some path $p \in \mathcal{P}_{\mathcal{LO}}$, let $E_{inc}(a)$ be the set of all incoming edges $e(n_i, a)$ to a from some nodes $n_i \in N \backslash \{b\}$ and $E_{out}(b)$ be the set of all outgoing edges $e(b, n_j)$ from b to some nodes $n_j \in N \backslash \{a\}$. 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 6. Consider our running example in Figure 5.2, 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 $\mathcal{P}_{\mathcal{LO}}$. 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). 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 $prio(s, n_3)$=0 and $prio(n_5, t)$=0.

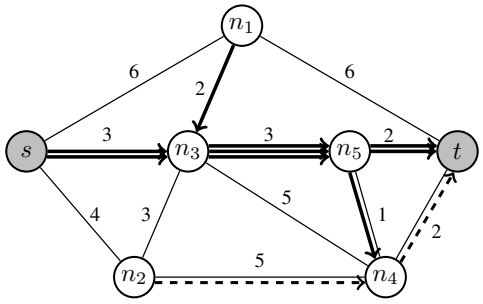


Figure 5.2: Computing the priority of edge (n_3, n_5).

Algorithm 9 illustrates the pseudocode of ESX. First, $\mathcal{P}_{\mathcal{LO}}$ is initialized with the shortest path p_0 (Line 1) and a max-heap H_0 is created (Line 2). H_0 is associated with p_0 and all the edges of p_0 are enheaped and sorted based on their priority. Note that, every time a path p_i is added to the result set, a max-heap H_i is created and stores the edges of p_i in priority order. The set E_{DNR} of non-removable edges is also initialized in 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 road network. Next, the algorithm sets p_c to the last result found and enters the inner loop (Line 6). At each iteration p_{max} is chosen as the path in $\mathcal{P}_{\mathcal{LO}}$ which has the maximum overlap $Sim(p_c, p_{max})$ and contains edges in H_{max} that can be removed from the road network. Then, ESX 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 road network (Line 11). In Lines 12-15 p_{tmp} is checked whether it is a valid path from s to t, and, if not, e_{tmp} is re-inserted to the road network and marks it as non-removable. Otherwise, ESX sets

p_c to p_{tmp} and proceeds to the next iteration. When the inner loop is finished, the algorithm checks if p_c is a valid alternative path by checking its similarity with all paths currently in $\mathcal{P}_{\mathcal{LO}}$ (Line 17). If p_c is a valid alternative, it is added to $\mathcal{P}_{\mathcal{LO}}$ and a new max-heap H_c associated with p_c is initialized with the edges of p_c (Lines 18-19). Finally, after the outer loop is finished, the result set $\mathcal{P}_{\mathcal{LO}}$ is returned in Line 20.

Algorithm 9: ESX

Input: Road network $G(N, E)$, source node s, target node t, # of results k, sim. threshold θ
Output: Set $\mathcal{P}_{\mathcal{LO}}$ 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;        ▷ 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;
```

Example 7. We demonstrate the functionality of ESX using the road network of Figure 5.2 and the 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 $\mathcal{P}_{\mathcal{LO}}$. Next, 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 road network. 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 $\mathcal{P}_{\mathcal{LO}}$.

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 $\mathcal{P}_{\mathcal{LO}}$ 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.

Finally, in our implementation of ESX we employed the optimization using lower bounds described in Section 4.6, which reduces the cost for retrieving shortest paths and optimizes the

performance of ESX even further.

5.4 Experimental Evaluation

5.4.1 Experimental Setup

In order to measure the performance of our heuristic algorithms we use seven different road networks shown in Table 5.1. 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. As a reference, we also include MultiPass in our experiments, the fastest exact algorithm for processing k-SPwLO queries, presented in the previous chapter.

Table 5.1: Road networks.

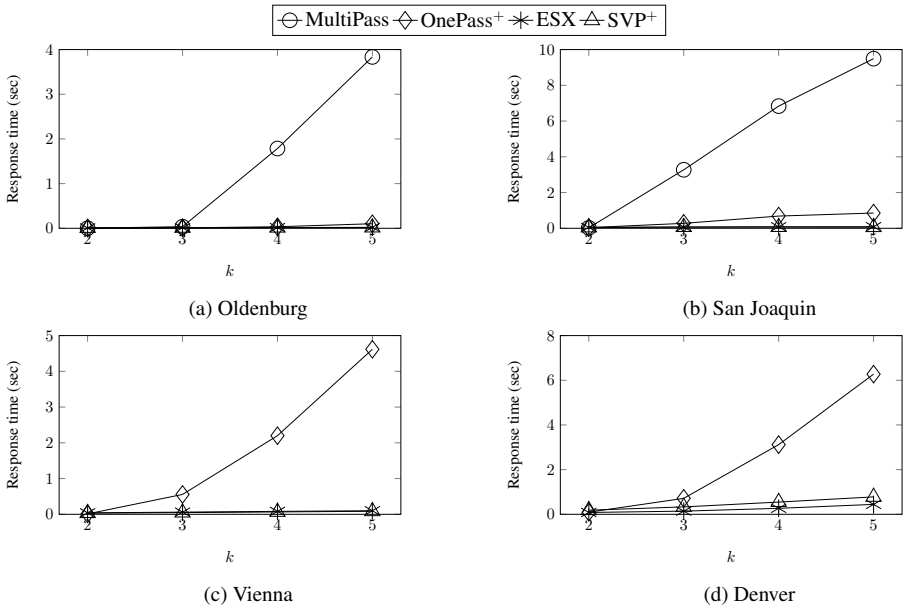
road network	# nodes	# edges
Oldenburg	6,105	14,058
San Joaquin	18,263	47,594
Vienna	19,826	54,918
Denver	73,166	196,630
San Francisco	174,956	443,604
New York City	264,346	730,100
Colorado	435,666	1,057,066

Similar to Section 4.7, due to the high execution time of MultiPass and OnePass$^+$, we consider a timeout of 120 seconds for each query. Each query that fails to execute within the predefined timeout is considered timed-out/failed. While we have already reported on the ratio of failed queries for MultiPass in Section 4.7, the ratio of failed queries for OnePass$^+$ was below 10% in all experiments. For SVP$^+$ and ESX, all queries were executed within 120 seconds. For presentation purposes, the results shown in this section consider only those queries which were successfully completed by all algorithms. In addition, we report experiments on the quality of the results computed by the heuristic algorithms. Given the number k of requested paths, we measure (a) the number of paths returned by each 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++ using the C++11 standard and compiled using GNU G++ compiler (version 4.8.1). All experiments were executed on an Ubuntu Linux server with 4 Intel Xeon X5550 (2.67GHz) processors and 48GB of RAM.

5.4.2 Performance

The first experiment in Figure 5.3 compares the exact algorithm MultiPass with the heuristic algorithms SVP$^+$, OnePass$^+$ and ESX for a constant similarity threshold θ=50% varying the requested number of paths k. In all figures we observe that the runtime of all algorithms increases with the number k of requested paths. As expected, the runtime of the heuristic algorithms increases only slightly, whereas the exact solution MultiPass deteriorates for large



(a) Oldenburg (b) San Joaquin

(c) Vienna (d) Denver

Figure 5.3: Performance comparison of algorithms varying requested paths k (θ=50%).

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$^+$. Clearly, MultiPass cannot scale and, hence, we exclude it from our measurements on Vienna and Denver (Figures 5.3c-d). Furthermore, in Figures 5.3c-d we also observe that the performance of OnePass, SVP$^+$ and ESX is similar. However, for increasing values of k both SVP$^+$ and ESX clearly outperform OnePass$^+$ up two one order of magnitude.

The second experiment compares the exact algorithm MultiPass with the heuristic algorithms SVP$^+$, OnePass$^+$ and ESX for a constant requested number of paths k=3 varying the similarity threshold θ. Figure 5.4 shows that for θ<70%, MultiPass is much slower than the heuristic algorithms: 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 heuristic algorithms (for θ=90% MultiPass is even faster than ESX and SVP$^+$). We also 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.

Another interesting observation in Figures 5.3c–d and 5.4c–d is that the performance of MultiPass and OnePass$^+$ shows a local maximum for θ=30%, which indicates the following important trade-off. As θ increases, the pruning power of Lemma 4 deteriorates, and both MultiPass and OnePass$^+$ construct more paths. 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 5 increases and more partial paths can be pruned. This explains the behavior of MultiPass and OnePass$^+$, where the response time initially increases, but after θ=30% the runtime of both algorithms goes down again.

To sum up, the heuristic algorithms clearly outperform the correct algorithm MultiPass. Comparing the heuristic algorithms, we observe that SVP$^+$ and ESX have a similar performance and are the clear winners for small and medium size datasets. Apparently, OnePass$^+$ is

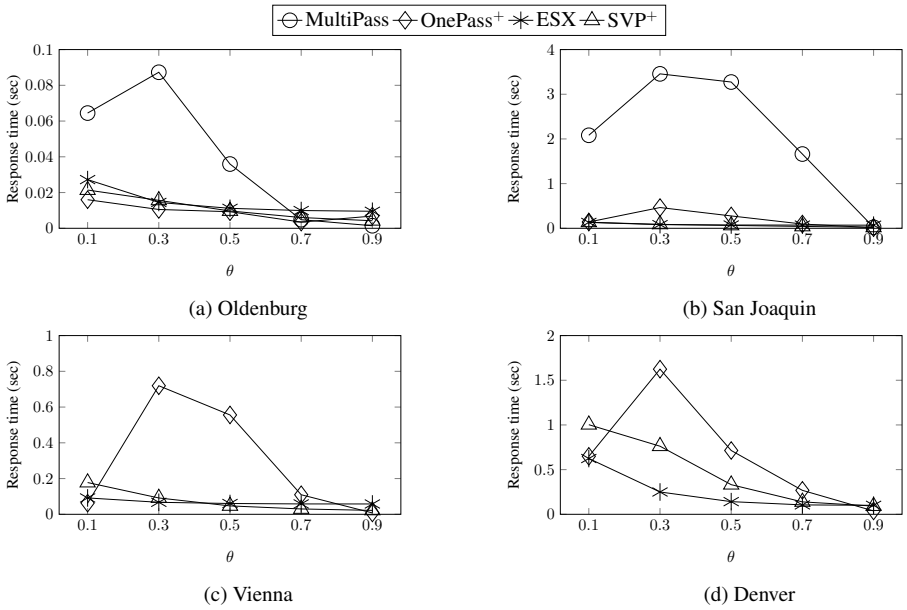


Figure 5.4: Performance comparison of algorithms varying similarity threshold θ (k=3).

not practical for large road networks and/or values of k>3.

5.4.3 Scalability

From the previous experiments it is clear that both MultiPass and OnePass$^+$ are not scalable. For values of k>3 both algorithms are prohibitively expensive. However, the same does not apply for ESX and SVP$^+$. To demonstrate their scalability, we present in Figure 5.5 the results of an experiment using larger values of $k \in \{4, 8, 12, 16\}$ and including also 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 5.2 and the discussion in Sec. 5.4.4). 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$^+$.

5.4.4 Result Quality and Completeness

In Figure 5.6, we present our measurements on the quality of the computed results. We consider all queries for which each algorithm returned exactly k paths and compute the average length of the returned paths. Then we compare the average length of each result set to the shortest path. We show how much longer, on average, are the alternative paths with respect to the shortest path. Apparently, the exact k-SPwLO result contains the shortest alternative paths. Looking at the heuristic algorithms, OnePass$^+$ produces clearly the best alternative paths, which are very close to the paths in the exact k-SPwLO result. Both ESX and SVP$^+$

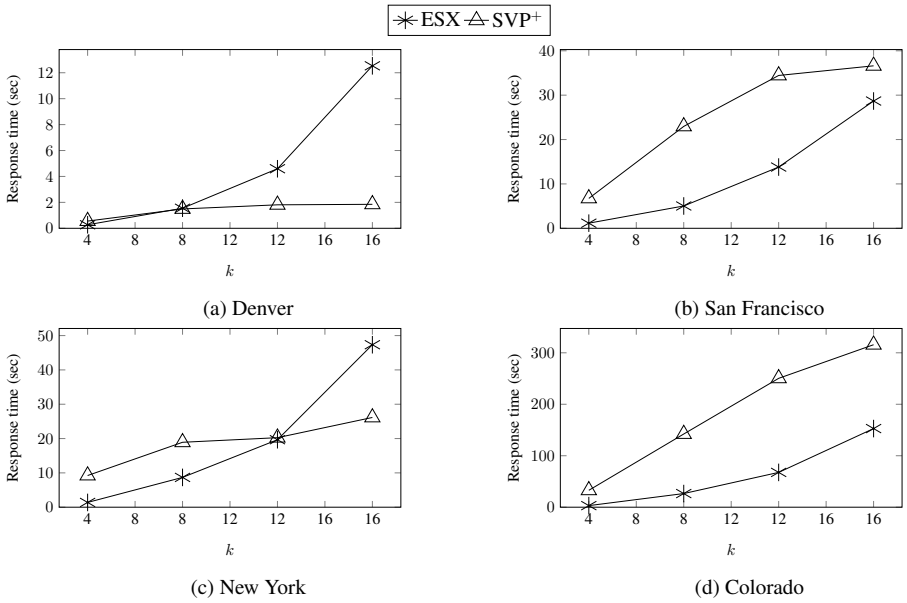


Figure 5.5: Performance comparison of SVP$^+$ and ESX for $k \in \{2, 4, 8, 16\}$ and $\theta = 50\%$.

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

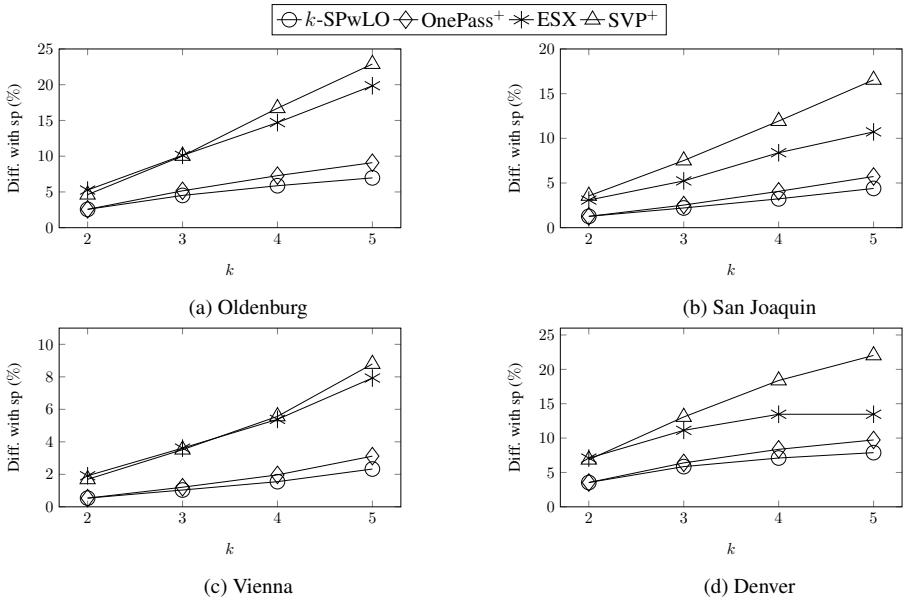


Figure 5.6: Result quality of algorithms varying requested paths k ($\theta = 50\%$).

Next, we measure the completeness of the result sets. Table 5.2 reports for each algorithm the percentage of queries for which exactly k alternative paths were found. Apparently, the exact solution k-SPwLO has the highest completion ratio. The completion ratio OnePass$^+$ is

Table 5.2: Average completeness ratio (%) per query varying k ($\theta = 50\%$).

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

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.

Finally, in Table 5.3 we compare 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 clear 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. Hence, SVP$^+$ returns more and more incomplete results for an increasing k.

Table 5.3: Average returned results per query varying k ($\theta = 50\%$) for SVP$^+$ and ESX.

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

5.5 Summary

We studied heuristic algorithms to compute k-SPwLO queries. First, we presented SVP$^+$, a baseline heuristic algorithm based on existing literature. We also introduced two novel heuristic algorithms. OnePass$^+$ employs ideas from both OnePass and MultiPass, presented in the previous chapter, and achieves to compute a set of relatively short dissimilar paths. ESX, our second heuristic algorithm, computes alternative paths by incrementally removing edges from the road network and running shortest path queries. Through a comprehensive experimental evaluation we showed that (a) OnePass$^+$ is significantly faster than MultiPass, the most efficient exact algorithm, while its result set is close to the exact solution, and (b) ESX is scalable, i.e., it computes a result set of dissimilar paths for large road networks and large values of k.

CHAPTER 6

MoTrIS: A System for Multimodal Route Planning

In this chapter, we present MoTrIS, a **M***ultim***o***dal* **Tr***ansport* **I***nformation* **S***ystem*. MoTrIS differs from existing frameworks as it not only offers route planning services over road networks, but also tackles the challenge of combining different types of networks into a single multimodal network. MoTrIS allows shortest path and distance query processing on multimodal transportation networks. Such fundamental queries can be employed as basic building blocks for creating advanced solutions to different types of problems on multimodal networks, e.g., journey and itinerary planning. Selected algorithms presented in the previous chapters of this thesis for route planning on road networks, have been implemented and integrated into MoTrIS as well.

6.1 MoTrIS Framework

MoTrIS is a service-oriented framework designed to support applications that depend on multimodal transportation networks. The framework allows users/developers to create customized services over specific regions. First, the users choose a road network and select the modes of transportation which will be associated with the selected road network. Then, MoTrIS creates a new instance/service where the selected street and transportation networks are combined into a single multimodal network. To enable easy access to query services, MoTrIS provides an API which allows users to employs the customized routing services directly in their applications.

6.1.1 System Overview

Figure 6.1 illustrates the architecture of MoTrIS, which is composed of four core modules: the *network* module represents the multimodal network; the *timetable* module handles the transportation network data, i.e., the schedule and the availability of each transportation mode; the *query processing* module includes algorithms for processing queries either on the multimodal network or on each of its individual components; and the *visualization* module produces the results of each query execution.

Apart from the core modules, Figure 6.1 also illustrates the *data import*, which is responsible for importing road and transportation network data into PostGIS[1], a spatial-enabled RDBMS, as well as the *web application*, which allows users to create new services and run sample queries to test the services.

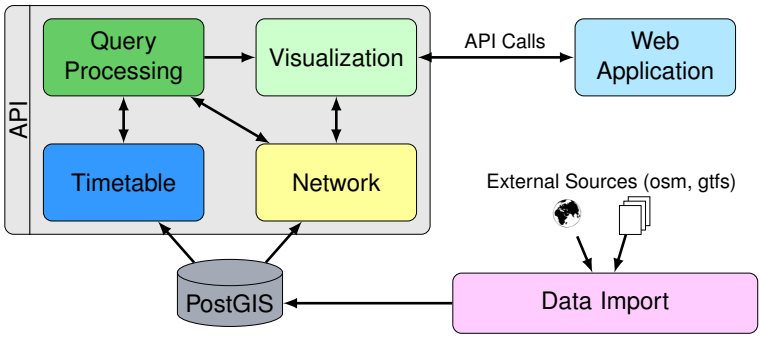


Figure 6.1: System architecture.

6.1.2 Data Import and PostGIS

The *data import* module is responsible for importing road and transportation network data from various sources into PostGIS. In particular, we obtain road network data from *OpenStreetMap*[2] (OSM). The road network data from OSM is then transformed into a routable graph format and stored in a single relational table in PostGIS. Each tuple represents an edge of the road network along with related information about the edge and its two adjacent nodes, i.e.,

[1] http://postgis.net
[2] https://www.openstreetmap.org/

edge length, source and target node location, etc. Together with the road network, a polygon that marks the boundaries of the region associated with the road network is stored in PostGIS.

Routing services can be employed over any imported region directly. To employ services over smaller regions though, the module allows users to specify sub-regions of already imported regions. Polygons representing sub-regions are stored in PostGIS in a hierarchical fashion, as illustrated in Figure 6.2. Note that in order to define a sub-region, the polygon bounding the sub-region must entirely inside the polygon bounding an already imported region. After specifying a polygon that represents a sub-region, the module extracts the edges of the road network that lie inside the given polygon. Then, PostGIS creates a view which models only the part of the road network that is bounded by the given polygon[3].

Figure 6.2 illustrates an example, where we have the entire network of Italy obtained from OSM and stored in table *italy_streets*. The polygons representing various cities and states are stored in table *road_network*. To obtain the road network for the city of Bolzano, all the edges that lie entirely inside the respective polygon are extracted. Then, the view *bolzano_view* is created which contains only the edges on the sub-network for the city of Bolzano.

Note that the extracted edges may not always form a connected road network but may form several ones. Since sub-regions are not always defined with the road network in mind, it is possible that, in order to reach certain parts of the bounded region, the user has to travel outside the region first. To address this problem, the data import module comes with an optional verification process which extracts only the largest sub-network and, hence, ensures that the road network of every sub-region is a connected graph.

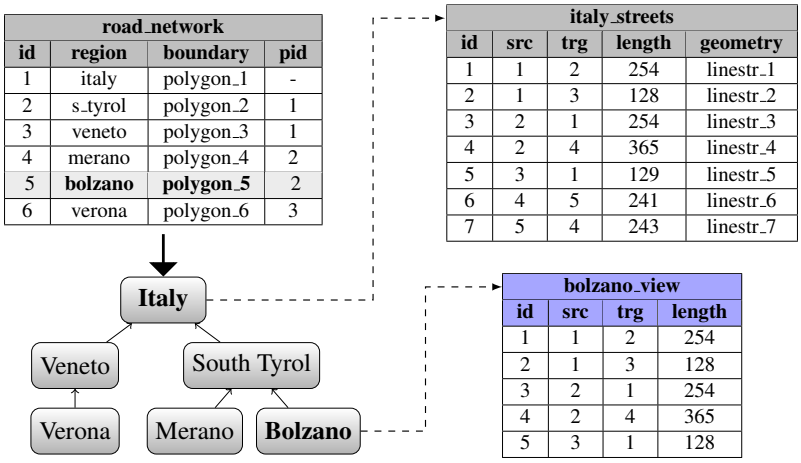


Figure 6.2: Sub-region road network extraction.

For transportation network data, we first create a set of relational tables in PostGIS that match the General Transit Feed Specification[4] (GTFS). We outline the key GTFS entities for building a transportation network: *stops*, which are individual locations where vehicles pick up or drop off passengers and can be associated with several routes; *routes*, which are groups of trips that are displayed to riders as a single transportation service, i.e., a bus line; *trips*, which

[3]In practice, instead of creating a view, we add a column of Bit type to the table storing the road network to mark the edges that are on the road network of the sub-region.

[4]https://developers.google.com/transit/gtfs/reference

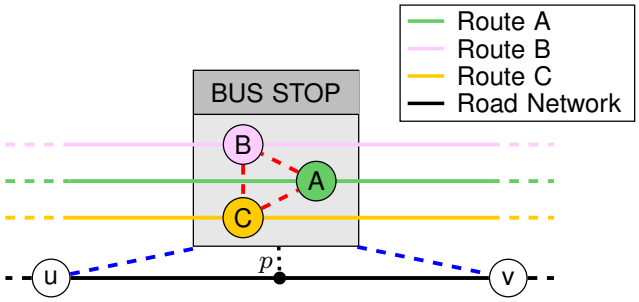


Figure 6.3: Multimodal network.

store sequences of two or more stops that are on the same route; and *stop_times*, which store the time that a vehicle arrives at and departs from individual stops for each trip. The module creates additional columns for spatial data types. For example, the location of a stop is stored both as two separate float numbers (representing longitude and latitude as dictated by GTFS) and as a *Point* data type.

Finally, the data import module associates each imported transportation network with an already imported (or extracted) road network. The associated road network is either provided manually or is determined automatically as the network bounded by the smallest polygon which also bounds the given transportation network. For each stop in the transportation network, the closest point p of the road network (the black point along the road network in Figure 6.3) is computed and edge $e(u, v)$ which contains p is retrieved. Weights $w_u(u, p)$ and $w_v(p, v)$, which represent the length of the segment of e from u to point p and the length of the segment of e from point p to v, respectively, are determined. Finally, two links for each bus stop are stored as a single tuple $\langle stop\_id, e\_id, u\_node\_id, v\_node\_id, w_u, w_v \rangle$ in PostGIS.

Since the regions are stored hierarchically, the transportation network associated with a given region can be directly employed for services on all the ancestors of the given region in the hierarchy. For example, given the polygon hierarchy in Figure 6.2, a transportation network associated with Bolzano can be directly employed for services over South Tyrol and Italy.

6.1.3 Network Model

The *network* module is responsible for constructing and maintaining the multimodal network graph for each service in main memory. To construct a multimodal network, we first choose a region and the module obtain from the database the road network for the selected region. Then, the module loads from PostGIS the transportation networks selected by the user and models each transportation network following the time-dependent model [69]. For all routes, the stops that each route serves are extracted from the trips. For each distinct pair of a route and a stop, a node is created and added to the transportation network graph. For example, in Figure 6.3 there are three routes passing through the bus stop. Hence, there are three nodes A, B and C, one for each route, added to the transportation network graph. Subsequently, the module extracts from the trips all pairs of consecutive stops. An edge connecting nodes that are associated with consecutive stops and are on the same route, is added to the transportation network graph. Each edge is assigned with a travel time function, which is executed on query time and determines the weight, i.e., the time required to cross the edge.

Next, transfer link edges between nodes of the transportation network are generated to allow transfers. Apparently, a stop can be associated with more than one nodes. The number of nodes associated with a given stop is the same as the number of routes that pass through that stop. The module adds transfer link edges between all nodes associated with the same stop (red dashed lines in Figure 6.3). The weight of each link edge represents the time required to switch from one route to another. By default the transfer time is zero, unless it is specified explicitly in the input GTFS timetable.

Finally, for each bus stop, the module loads from the database the precomputed links to nodes u and v of the road network. For each node n associated with a stop, two link edges $e(u, n)$ and $e(n, v)$ with weights w_u and w_v, respectively, are added to the network graph, thereby connecting the transportation networks with the road network. In Figure 6.3, each of the two blue/dashed lines represents three link edges, each connecting a road network node with one of the nodes A, B, C of the transportation network, all associated with the same stop.

6.1.4 Timetable

The *timetable* module is responsible for storing in main memory and querying the schedule of each transportation network. Given a timestamp and an edge of some transportation network, the timetable determines the weight of the edge, i.e., the time needed to cross the edge at the given timestamp. For example, given a bus network, the timetable module stores for each edge e a set of entries $\{n_s, n_d, t_s, t_d\}$, where n_s is the source node, n_d is the target node, t_s is the departure time of the bus from the stop associated with n_s and t_d is the arrival time of the bus at the stop associated with n_t. Given a timestamp t, i.e., arrival time of the user at the stop associated with n_s, the timetable determines the bus that the user should take to reach n_d. The selected bus is the one which arrives at the stop after t, i.e., $t_s \geq t$ and the travel time $t_d - t$ required to reach node n_d, which also includes the waiting time, is minimum.

6.1.5 Query Processing

The *query processing* module executes queries over the multimodal network. Shortest path queries are of the form $q(G, p_s, p_t, t)$, where G is a multimodal network, p_s and p_t are the source and destination query points on the map and t is the starting time of the journey. First, the module maps the query points to the multimodal network in the same way bus stops are mapped during the construction of the network. To compute the shortest path on the multimodal network, a modified version of Dijkstra's algorithm is used. During the expansion of edges on the road network, the fixed weight is used, whereas during the expansion of edges on some transportation network, the module queries the timetable to obtain the weights of the edges.

Apart from route planning on multimodal networks, the query processing module can also process queries over the road network by simply ignoring transportation and link edges. For processing queries over the road network, we have implemented and integrated into MoTrIS two of our algorithms: (a) ParDiSP, for processing distance and shortest path queries, and (b) ESX, for recommending alternative routes.

We have designed the query processing module in a way that it is highly extensible; new algorithmic implementations can be integrated easily into our framework. Algorithms in the query processing module can be accessed via a public API interface, which provides the users with all the required calls for submitting queries and retrieve results. The multimodal network

is accessed in an abstract way and cannot be modified by the algorithms that are implemented in the query processing module. By designing the module with extensibility in mind, we aim for MoTrIS to become a solid framework for further research that is not limited to route planning services on multimodal networks. More algorithms for processing different types of queries can be added to our framework, as long as the implementation of these algorithms does not require the modification of the network model.

6.1.6 Visualization

The *visualization* module is responsible for producing the results in a visualizable format after a query is successfully executed. Given the result of a shortest path query, first the module retrieves from the network model the geometries for all the edges contained in the shortest path. Then a response in GeoJSON[5] format is generated, which can be visualized directly by most external map services, e.g., Google Maps, Mapbox, OpemStreetMap, etc. Apart from the route itself, the module includes in the GeoJSON response additional information about the result, such as distance, trip duration for each mode, cost of the trip, etc., provided that the information is available in the dataset.

6.1.7 Web Application

The *web application* allows users to create, modify and test services. Users can create a new service by selecting the region/road network on which their service will be applied, the transportation modes that will be supported by the new service and the queries that will be available. The application allows the visualization of the resulting multimodal network. For each route in the transportation network, the application also displays the links connecting the transportation network stops with the street network.

Moreover, to get a clear picture of the query results, a testing interface which uses the API for submitting queries and visualizing the results is provided. The testing interface provides all the necessary tools to allow users to explore the functionality of the public API, along with instructions on how to access the API directly from external applications. For the visualization the module employs Mapbox[6].

6.2 Use-cases

This section present the main use-cases of our MoTrIS framework. We split the users of MoTrIS into two groups:

- the administrators who manage existing data and import new data into PostGIS, and
- the developers who want to use our framework for building customized routing services and employ them in their applications

In what follows, we present use-cases for both user groups.

[5] http://geojson.org
[6] https://www.mapbox.com

6.2.1 Administrator tasks

We have implemented a set of command line scripts for the system administrators. Hence, an administrator can perform any of the following tasks simply by running the respective script.

Importing new road networks. To import new road network data, the administrators need to obtain first data directly from OpenStreetMap[7] in OSM or PBF format. In addition, they need to obtain or extract from the input file, the polygon which bounds the associated region. Finally, the administrators execute the script providing a name and a description for the imported road network. The road network is assigned with an *id* and it becomes the root of a new hierarchy.

Defining sub-regions. Given that a root region already exists in PostGIS, the administrators can define and extract sub-regions. The administrators run the respective script providing the id of the root region from which they want to extract the road network of a sub-region, the polygon of the sub-region, a name and a description. MoTrIS checks automatically whether the input polygon lies entirely inside the polygon of the root region. If yes, then the sub-region is extracted as we described in Section 6.1.2 and the polygon is inserted at the appropriate level of the hierarchy. Otherwise, MoTrIS stops the import process and prints an error message indicating that the input polygon does not define a proper sub-region.

Importing transportation data. Finally, administrators can also import data in gtfs format by running the respective command line script. Administrators need to provide the location on which all the GTFS files are located, along with a name and a description for the transportation network. Additionally, administrators may provide the id of the region with which they want the imported transportation network to be associated with. If no such id is provided, MoTrIS matches the input transportation network with the smallest possible region and generates links accordingly. Either when an id is provided or not, if no matching region or sub-region for the transportation network is found, e.g. some stops of the transportation network are outside the polygon of any region or sub-region stored in PostGIS, then MoTrIS stops the import process and prints a message indicating the error.

6.2.2 User/Developer

Any user of MoTrIS can use the web application to deploy and test customized services. Figures 6.4- 6.6 illustrate an example in which we create a new multimodal routing service for the province of South Tyrol.

Create New Service. To create a new service, the user provides a name and a description for the service, and defines the region/road network over which he/she wants the new service to be functional. In the example illustrated in Figure 6.4 the user selects the road network of the province of South Tyrol as the road network for the new service.

Next, the user selects transportation modes from a list of available transportation networks. Figure 6.5 illustrates the web interface which enables the user to select one or more transportation networks. Note that the interface displays only those transportation networks that

[7] `http://download.geofabrik.de`

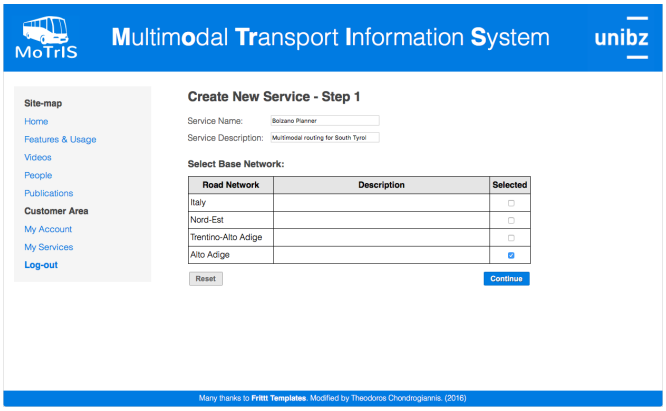


Figure 6.4: Selection of road the transportation networks.

are compatible with the selected road network. Therefore, the only available transportation network shown in Figure 6.5 is that of SASA, the local city-bus company.

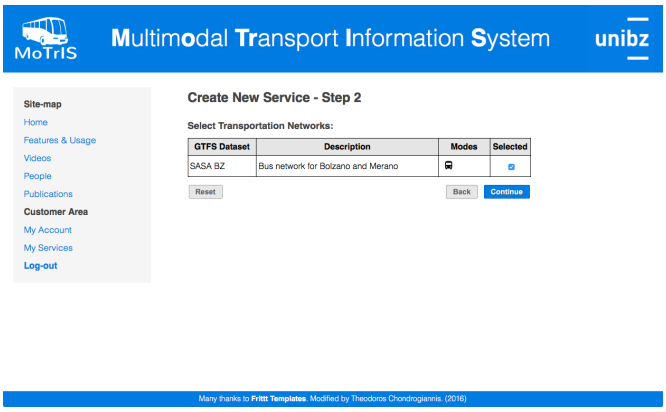


Figure 6.5: Selection of road the transportation networks.

After the user has verified his selection, MoTrIS extracts from the database the road network for the requested region as well as the timetables for the requested transportation networks, and constructs the multimodal network for the new service. The new service is loaded into the main memory and is ready to be accessed using the API.

Run Sample Queries. Next, we show how the user can submit shortest path queries to MoTrIS and visualize the results using the web interface. The application asks the user to define a starting location, a destination, the date and the starting time (or the desirable arrival time) of his trip. Figure 6.6 illustrates the web interface of MoTrIS for testing query services. More specifically, the visualization of a shortest path query over a multimodal network for the city of Bolzano is shown. The blue lines represent the parts of the route that the user crosses on foot while the orange lines represent the parts that are crossed by bus.

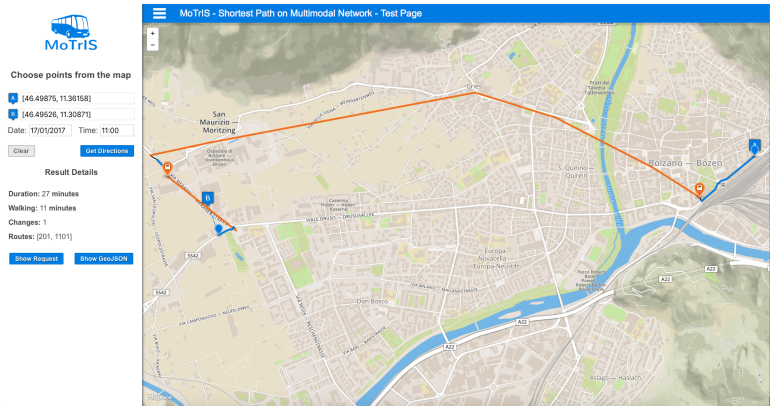


Figure 6.6: Sample query result visualization.

6.3 Summary

We presented MoTrIS, a service-oriented framework which provides support for building applications on multimodal networks. Our framework combines street and transportation networks into a single multimodal network and provides query services via a public API. Apart from route planning on multimodal transportation networks, MoTrIS can also be employed for route planning on road networks. Selected algorithms presented in the previous chapters of the thesis for route planning and alternative routing, have been integrated into MoTrIS as well. MoTrIS comes with a web-interface which enables users to create, modify and test their own customized services. Last but not least, MoTrIS was implemented with extensibility in mind; thus, more algorithms for processing different types of queries can be implemented and integrated into our framework easily.

CHAPTER 7

Conclusion

7.1 Summary

In this thesis, we studied, formalized and proposed algorithmic solutions for two problems related to route planning on road networks: the processing of distance and shortest path queries and the computation of dissimilar yet short alternative paths.

For processing distance and shortest path queries on road networks, we presented ParDiSP, a framework which supports the efficient processing of both types of queries. We showed how ParDiSP partitions the network into components and precomputes a set of index structures. Distance and shortest path queries are processed by running the ALT [38] algorithm if source and target are in the same component. Otherwise, for distance queries ParDiSP combines distances from three precomputed tables, whereas shortest path queries are decomposed into the retrieval of three sub-paths, which can be executed in parallel. An experimental evaluation has shown that ParDiSP outperforms the state-of-the-art for shortest path queries, while being comparable to the state-of-the-art for distance queries. For mixed query sets containing both distance and shortest path queries, ParDiSP is even more efficient than a combination of the best state-of-the-art approaches for the two query types, while incurring less space overhead.

The second problem we studied is that 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. We formalized this task using the k-Shortest Path with Limited Overlap query. We also designed and implemented three algorithms for processing k-SPwLO queries: BSL builds upon the computation of the $K-$shortest paths; OnePass traverses the network once expanding only paths that do not violate the similarity constraint; and MultiPass progressively computes the result set after traversing the network $k-1$ times, while employing two powerful pruning criteria. Through an extensive experimental evaluation, we demonstrated that MultiPass is the most efficient algorithm for processing k-SPwLO queries outperforming its competitors and, in most cases, by a large margin.

To achieve scalability though, we also introduced two heuristic algorithms. OnePass$^+$ employs ideas from both OnePass and MultiPass and achieves to compute a set of dissimilar

paths which, in terms of average length, is very close to the exact solution. ESX computes alternative paths by incrementally removing edges from the road network and running shortest path queries. Through a comprehensive experimental evaluation we showed that OnePass$^+$ is significantly faster than MultiPass while its result set is close to the exact solution, and, in contrast to the other algorithms, ESX is scalable and can compute k-SPwLO queries for large road networks and large values of k.

Finally, we have presented MoTrIS, a service-oriented framework which provides support for building applications on multimodal networks. Apart from integrating our algorithms for route planning on road networks, MoTrIS combines street and transportation networks into a single multimodal network and provides query services over such networks via a public API. In addition, MoTrIS comes with a web-interface which enables the users to create, modify and test their own customized services. Last but not least, MoTrIS is highly extensible. New algorithms can be easily integrated and extend the capabilities of the platform.

7.2 Future Work

Future work points in different directions. First, regarding the computation of distance and shortest path queries, we plan to extend ParDiSP to support more dynamic scenarios, i.e., when the weights of the edges of a road network change over time. In addition, as both distance and shortest path queries are used as building blocks for processing more complex queries, we will study the problem of the efficient processing of such queries in batches, as well as the efficient computation of other spatial network queries, such as kNN queries and distance joins.

Second, with regard to alternative routing, we plan to extend and improve our algorithms by employing preprocessing-based methods such as Contraction Hierarchies [36]. From a theoretical point of view, we will investigate approximation algorithms with error guarantees. We also plan to extend the definition of alternative routing by considering additional criteria and constraints besides the overlap between paths and their length. Last but not least, we will investigate the computation of multiple dissimilar paths on different types of networks such as social networks and web graphs.

Third, we will study the problem of route planning on multimodal transportation networks. We plan to modify ParDiSP to support distance and shortest path query processing on multimodal transportation networks. Furthermore, we would like to investigate alternative routing on multimodal transportation networks and the computation of dissimilar trips using appropriate similarity metrics, e.g. means of transport used, walking distance, journey cost etc.

Finally, we plan to extend MoTrIS with additional modules, such as a module to monitor the timetable information and support real-time updates on the schedule. Moreover, we intend to use MoTrIS as basis for further research on more efficient algorithms for various problems on multimodal networks, such as the analysis and evaluation of transportation networks. Such tools will support transportation scientists to identify problems and evaluate the usability of existing transportation systems, and will support the development of more efficient solutions.

Bibliography

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

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A Hub-based Labeling Algorithm for Shortest Paths in Road Networks. In *Proceedings of the 10th International Symposium on Experimental Algorithms*, SEA'11, pages 230–241, 2011.

[3] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical Hub Labelings for Shortest Paths. In *Proceedings of the 20th Annual European Conference on Algorithms*, ESA'12, pages 24–35, 2012.

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

[5] E. Ahmadi and M. A. Nascimento. K-Closest Pairs Queries in Road Networks. In *Proceedings of the 17th IEEE International Conference on Mobile Data Management*, MDM'16, pages 232–241, 2016.

[6] V. Akgun, E. Erkut, and R. Batta. On Finding Dissimilar Paths. *European Journal of Operational Research*, 121(2):232–246, 2000.

[7] T. Akiba, Y. Iwata, K. Kawarabayashi, and Y. Kawata. Fast Shortest-path Distance Queries on Road Networks by Pruned Highway Labeling. In *Proceedings of the 16th SIAM Workshop on Algorithm Engineering and Experiments*, ALENEX'14, pages 147–154, 2014.

[8] J. Arz, D. Luxen, and P. Sanders. Transit Node Routing Reconsidered. In *Proceedings of the 12th International Symposium on Experimental Algorithms*, SEA'13, pages 55–66, 2013.

[9] R. Bader, J. Dees, R. Geisberger, and P. Sanders. Alternative Route Graphs in Road Networks. In *Proceedings of the 1st International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems*, TAPAS'11, pages 21–32, 2011.

[10] H. Bast, M. Brodesser, and S. Storandt. Result Diversity for Multi-modal Route Planning. In *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems*, ATMOS'13, pages 123–136, 2013.

[11] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route Planning in Transportation Networks. Technical Report MSR-TR-2014-4, Microsoft Research, 2014.

[12] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In Transit to Constant Time Shortest-Path Queries in Road Networks. In *Proceedings of the 9th SIAM Workshop on Algorithm Engineering and Experiments*, ALENEX'07, pages 45–59, 2007.

[13] A. Bernstein and D. Karger. A Nearly Optimal Oracle for Avoiding Failed Vertices and Edges. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, STOC'09, pages 101–110, 2009.

[14] P. Bolzoni, S. Helmer, K. Wellenzohn, J. Gamper, and P. Andritsos. Efficient Itinerary Planning with Category Constraints. In *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS'14, pages 203–212, 2014.

[15] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. *Recent Advances in Graph Partitioning*, volume 9220, pages 117–158. Springer International Publishing, 2016.

[16] V. Ceikute and C. S. Jensen. Vehicle Routing With User-Generated Trajectory Data. In *Proceedings of the 16th IEEE Conference on Mobile Data Management*, MDM'15, pages 14–23, 2015.

[17] Z. Chen, H. T. Shen, and X. Zhou. Discovering Popular Routes from Trajectories. In *Proceedings of the 27th IEEE International Conference on Data Engineering*, ICDE'11, pages 900–911, 2011.

[18] T. Chondrogiannis, P. Bouros, J. Gamper, and U. Leser. Exact and Approximate Algorithms for Finding k-Shortest Paths with Limited Overlap. In *Proceedings of the 20th International Conference on Extending Database Technology*, EDBT'17, pages xxx–xxx, (to appear).

[19] T. Chondrogiannis and J. Gamper. Exploring Graph Partitioning for Shortest Path Queries on Road Networks. In *Proceedings of the 26th GI-Workshop Grundlagen von Datenbanken*, GvDB'14, pages 71–76, 2014.

[20] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and Distance Queries via 2-hop Labels. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, SODA'02, pages 937–946, 2002.

[21] T. Columbus. On the Complexity of Contraction Hierarchies. Master's thesis, Karlsruhe Institute of Technology, 2009.

[22] C. F. Costa, M. A. Nascimento, J. A. F. Macedo, and J. C. Machado. A^*-based Solutions for KNN Queries with Operating Time Constraints in Time-Dependent Road Networks. In *Proceedings of the 15th IEEE Conference on Mobile Data Management*, MDM'14, pages 23–32, 2014.

[23] V. T. de Almeida and R. H. Güting. Using Dijkstra's Algorithm to Incrementally Find the k-Nearest Neighbors in Spatial Network Databases. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing*, SAC'06, pages 58–62, 2006.

[24] D. Delling, J. Dibbelt, T. Pajor, D. Wagner, and R. F. Werneck. Computing multimodal journeys in practice. In *Proceedings of the 12th Symposium on Experimental Algorithms*, SEA'13, pages 260–271, 2013.

[25] D. Delling and W. Dorothea. Pareto Paths with SHARC. In *Proceedings of the 8th International Symposium on Experimental Algorithms*, SEA'09, pages 125–136, 2009.

[26] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In *Proceedings of the10th International Symposium on Experimental Algorithms*, SEA'11, pages 376–387, 2011.

[27] D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *Proceedings of the 35th International Parallel & Distributed Processing Symposium*, IPDPS'11, pages 1135–1146, 2011.

[28] D. Delling, T. Pajor, and D. Wagner. Accelerating Multi-modal Route Planning by Access-Nodes. In *17th European Symposium on Algorithms*, ESA'09, pages 587–598, 2009.

[29] U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. TransDec: A Spatiotemporal Query Processing Framework for Transportation Systems. In *Proceedings of the 26th IEEE International Conference on Data Engineering*, ICDE'10, pages 1197–1200, 2010.

[30] K. Deng, X. Zhou, H. T. Shen, S. Sadiq, and X. Li. Instance Optimal Query Processing in Spatial Networks. *The VLDB Journal*, 18(3):675–693, 2009.

[31] J. Dibbelt, T. Pajor, and D. Wagner. User-Constrained Multimodal Route Planning. *Journal of Experimental Algorithmics*, 19:1.1–1.19, 2015.

[32] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[33] A. E. Feldmann and L. Foschini. Balanced Partitions of Trees and Applications. In *29th Symposium on Theoretical Aspects of Computer Science*, volume 14 of *STACS'12*, pages 100–111, 2012.

[34] R. A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, 1974.

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

[36] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction Hierarchies : Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th International Workshop on Experimental Algorithms*, WEA'08, pages 319–333, 2008.

[37] A. V. Goldberg. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proceedings of the 7th SIAM Workshop on Algorithm Engineering and Experiments*, ALENEX'07, pages 129–143, 2007.

[38] A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A^* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'05, pages 156–165, 2005.

[39] R. Gutman. Reach-based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proceedings of the 6th SIAM Workshop on Algorithm Engineering and Experiments*, ALENEX'04, pages 100–111, 2004.

[40] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[41] J. Hershberger, M. Maxel, and S. Suri. Finding the K Shortest Simple Paths: A New Algorithm and Its Implementation. *ACM Transactions on Algorithms*, 3(4):26–36, 2007.

[42] Y. W. Huang, N. Jing, and E. A. Rundensteiner. Effective Graph Clustering for Path Queries in Digital Map Databases. In *Proceedings of the 5th Conference on Information and Knowledge Management*, CIKM'96, pages 215–222, 1996.

[43] M. Innerebner, M. Böhlen, and J. Gamper. ISOGA: A System for Geographical Reachability Analysis. In *Proceedings of the 12th International Conference on Web and Wireless GIS*, W2GIS'13, pages 180–189, 2013.

[44] 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.

[45] N. Jing, Y. W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: an optimal model and its performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):409–432, 1998.

[46] S. Jung and S. Pramanik. HiTi Graph Model of Topographical Roadmaps in Navigation Systems. In *Proceedings of the 12th IEEE International Conference on Data Engineering*, ICDE'96, pages 76–84, 1996.

[47] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[48] E. Köhler, R. H. Möhring, and H. Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In *Proceedings of the 9th DIMACS Implementation Challenge*, 2006.

[49] M. Kolahdouzan and C. Shahabi. Voronoi-based K Nearest Neighbor Search for Spatial Network Databases. In *Proceedings of the 30th International Conference on Very Large Data Bases*, VLDB'04, pages 840–851, 2004.

[50] H.-P. Kriegel, M. Renz, and M. Schubert. Route skyline queries: A multi-preference path planning approach. In *Proceedings of the 26th IEEE International Conference on Data Engineering*, ICDE'10, pages 261–272, 2010.

[51] P. H. Li, M. L. Yiu, and K. Mouratidis. Discovering historic traffic-tolerant paths in road networks. *GeoInformatica*, 21(1):1–32, 2017.

[52] 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.

[53] R. J. Lipton and T. R.E. A Separator Theorem for Planar Graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[54] W. Luo, H. Tan, L. Chen, and L. M. Ni. Finding Time Period-Based Most Frequent Path in Big Trajectory Data. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD'13, pages 713–724, 2013.

[55] D. Luxen, N. Gmbh, and C. Vetter. Real-Time Routing with OpenStreetMap data Categories and Subject Descriptors. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS'11, pages 513–516, 2011.

[56] D. Luxen and D. Schieferdecker. Candidate Sets for Alternative Routes in Road Networks. *Journal of Experimental Algorithmics*, 19:1–28, 2015.

[57] R. P. Magalhães, J. Machedo, C. Ferreira, L. Cruz, G. Coutinho, and M. Nascimento. Graphast : An Extensible Framework for Building Applications on Time-dependent Networks. In *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS'15, pages 4–7, 2015.

[58] E. Q. V. Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operation Research*, 26(3):236–245, 1984.

[59] E. Q. V. Martins and M. M. B. Pascoal. A New Implementation of Yen's Ranking Loopless Paths Algorithm. *4OR: A Quarterly Journal of Operations Research*, 1(2):121–133, 2003.

[60] J. Maue, P. Sanders, and D. Matijevic. Goal-directed Shortest-path Queries Using Precomputed Cluster Distances. *Journal on Experimental Algorithms*, 14(2):2–27, 2010.

[61] K. Mouratidis, Y. Lin, and M. lu Yiu. Preference Queries in Large Multi-cost Transportation Networks. In *Proceedings of the 26th IEEE International Conference on Data Engineering*, ICDE'10, pages 533–544, 2010.

[62] T. Pajor. *Multi-modal route planning*. Master's thesis, UniversitaÃàt Karlsruhe (TH), 2009.

[63] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *Proceedings of the 20th IEEE International Conference on Data Engineering*, ICDE'04, pages 301–312, 2004.

[64] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate Nearest Neighbor Queries in Spatial Databases. *ACM Transactions on Database Systems*, 30(2):529–576, 2005.

[65] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query Processing in Spatial Network Databases. In *Proceedings of the 29th International Conference on Very Large Data Bases*, VLDB'03, pages 802–813, 2003.

[66] A. Paraskevopoulos and C. Zaroliagis. Improved Alternative Route Planning. In *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, ATMOS'13, pages 108–122, 2013.

[67] I. S. Pohl. *Bi-directional and Heuristic Search in Path Problems*. PhD thesis, Stanford University, Stanford, CA, USA, 1969.

[68] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast Shortest Path Distance Estimation in Large Networks. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management*, CIKM'09, pages 867–876, New York, New York, USA, 2009.

[69] E. Pyrga, F. Schulz, D. Wagner, and C. Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *Journal of Experimental Algorithmics*, 12(2.4):2.4:1–2.4:39, 2008.

[70] L. Qi and M. Schneider. Trafforithm: A traffic-aware shortest path algorithm in real road networks with traffic influence factors. In *1st International Conference on Geographical Information Systems Theory, Applications and Management*, GISTAM'15, pages 1–8, 2015.

[71] D. Sacharidis and P. Bouros. Routing Directions: Keeping it Fast and Simple. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS'13, pages 164–173, 2013.

[72] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable Network Distance Browsing in Spatial Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD'08, pages 43–54, 2008.

[73] P. Sanders and D. Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th European Symposium on Algorithms*, ESA'05, pages 568–579, 2005.

[74] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient Query Processing on Spatial Networks. In *Proceedings of the 13th Annual ACM International Workshop on Geographic Information Systems*, GIS'05, pages 200–209, 2005.

[75] J. Sankaranarayanan and H. Samet. Distance Oracles for Spatial Networks. In *Proceedings of the 25th IEEE International Conference on Data Engineering*, ICDE'09, pages 652–663, 2009.

[76] J. Sankaranarayanan and H. Samet. Query Processing Using Distance Oracles for Spatial Networks. *IEEE Transactions on Knowledge and Data Engineering*, 22(8):1158–1175, 2010.

[77] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path Oracles for Spatial Networks. *Proceedings of the VLDB Endowment*, 2(1):1210–1221, 2009.

[78] D. Schultes and P. Sanders. Dynamic Highway-Node Routing. In *Proceedings of the 6th International Workshop on Experimental Algorithms*, WEA'07, pages 66–79, 2007.

[79] M. Shekelyan, G. Jossé, and M. Schubert. Linear path skylines in multicriteria networks. In *Proceedings of the 31st IEEE International Conference on Data Engineering*, ICDE'15, pages 459–470, 2015.

[80] C. Sommer. Shortest-path Queries in Static Networks. *ACM Computing Surveys*, 46(4):1–31, 2014.

[81] H. Su, K. Zheng, J. Huang, H. Jeung, L. Chen, and X. Zhou. CrowdPlanner : A Crowd-Based Route Recommendation System. In *Proceedings of the 30th IEEE International Conference on Data Engineering*, ICDE'14, pages 1144–1155, 2014.

[82] M. Thorup. Compact Oracles for Reachability and Approximate Distances in Planar Digraphs. *Journal of the ACM*, 51(6):993–1024, 2004.

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

[84] J. Wang, K. Zheng, H. Wang, N. Zheng, and Z. X. Cost-efficient Spatial Network Partitioning for Distance-based Query Processing. In *Proceedings of the 15th IEEE International Conference on Mobile Data Management*, MDM'14, pages 13–22, 2014.

[85] L.-Y. Wei, Y. Zheng, and W.-C. Peng. Constructing Popular Routes from Uncertain Trajectories. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, SIGKDD'12, pages 195–203, 2012.

[86] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest Path and Distance Queries on Road Networks: An Experimental Evaluation. *Proceedings of the VLDB Endowment*, 5(5):406–417, 2012.

[87] 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.

[88] J. Xu, L. Guo, Z. Ding, X. Sun, and C. Liu. Traffic aware route planning in dynamic road networks. In *Proceedings of the 17th International Conference on Database Systems for Advanced Applications*, DASFAA'12, pages 576–591, 2012.

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

[90] M. L. Yiu, N. Mamoulis, and D. Papadias. Aggregate Nearest Neighbor Queries in Road Networks. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):820–833, 2005.

[91] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse Nearest Neighbors in Large Graphs. *IEEE Transactions on Knowledge and Data Engineering*, 18(4):540–553, 2006.

[92] J. Yuan, Y. Zheng, C. Zhang, and W. Xie. T-Drive : Driving Directions Based on Taxi Trajectories. In *Proceedings of the 18th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS'10, pages 99–108, 2010.

[93] R. Zhong, G. Li, K.-L. Tan, and L. Zhou. G-tree: An Efficient Index for kNN Search on Road Networks. In *Proceedingsof the 22nd ACM Conference on Information and Knowledge Management*, CIKM'13, pages 39–48, 2013.

[94] 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 *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD'13, pages 857–868, 2013.