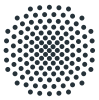


Universität Stuttgart

Personalized Route Planning: On Finding your Way in Theory and Practice

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Florian Benjamin Ihle

aus Heilbronn

Hauptberichter:	Prof. Dr. Stefan Funke
Mitberichter:	Prof. Dr. Peter Sanders
Tag der mündlichen Prüfung:	28.06.2022

Institut für Formale Methoden der Informatik

2022

Contents

Zusammenfassung

Personalisierung ist ein wichtiger Trend in der heutigen digitalen Welt. Im Bereich der Routenplanung hatte dieser jedoch noch keine starken Auswirkungen. Diese Dissertation beschäftigt sich mit dem multikriteriellen Routenplanungsansatz, der “personalized route planning model” genannt wird. Das Modell optimiert Konvexkombinationen von Routenplanungskriterien wie z.B. Reisezeit, Distanz oder Straßentyp und berechnet unterschiedliche Routen basierend auf den Präferenzen des Nutzers. Wir nennen die optimalen Pfade dieses Modells *personalisierte Pfade*. Das Angeben solcher Präferenzen ist keine einfache Aufgabe für einen Nutzer. Daher haben wir uns im praktischen Teil unserer Forschung auf Anwendungen konzentriert, die die Präferenzen im Backend verarbeiten ohne den Nutzer mit ihnen zu konfrontieren. Wir stellen drei praxisorientierte Anwendung und eine theoretische Analyse des Models vor. Für jede der Anwendungen entwickelten wir effiziente Algorithmen und verifizierten die Qualität und Laufzeit experimentell.

Die erste Anwendung findet große Mengen von *Alterantivrouten*, die nicht zu sehr überlappen. Dazu entwickeln wir einen Algorithmus, der alle personalisierten Pfade aufzählen kann. Dieser wird dann erweitert um optional das Berechnen von Pfaden zu vermeiden, die zusätzliche Optimalitätskriterien nicht erfüllen. Darüber hinaus zeigen wir ‘erfundene’ Routenplanungskriterien, die die Ergebnisse noch weiter diversifizieren. In unseren Experimenten beobachteten wir große Mengen an sinnvollen Alternativrouten.

Im Gegensatz zur ersten Anwendung sind die nächsten beiden Anwendung nicht an Endnutzer gerichtet, sondern dienen zum Lernen aus bestehenden Trajektorien. Die zweite Anwendung *identifiziert Zwischenziele* in Trajektorien. Wir nehmen an, dass Fahrer auf längeren Fahrten mit mehreren Zielen personalisierte Pfade zwischen den einzelnen Zielen wählen. Daher verwenden wir einen Trajektoriensegmentierungsansatz, der die Trajektorien in personalisierte Pfade aufteilt. Unser Ansatz konnte ca. 60% aller bekannten Zwischenziele in einem echt Welt Datensatz exakt erkennen und 90% waren sehr nahe an den errechneten Zwischenzielen.

Unsere letzte Anwendung hat das Ziel *Trajektorien nach Präferenzen zu clustern*. Hier ist die Idee eine Menge an Trajektorien durch eine kleine Menge Präferenzen zu erklären, so dass jede Trajektorie für eine der Präferenzen optimal ist. Das erlaubt es wichtige Präferenzen für einzelne Fahrer zu finden oder eine Menge von Fahrern zu kategorisieren. Wir berechnen den Präferenzbereich für alle Trajektorien und lösen eine darauf basierende geometrische “Hitting Set” Instanz. Obwohl beide Schritte im schlimmsten Fall teuer zu berechnen sind, stellten wir fest, dass der gesamte Prozess häufig in Sekunden bis Minuten berechnet werden kann für Instanzen mit tausenden Trajektorien.

Im letzten Teil der Dissertation analysieren wir wie viele personalisierte Pfade es abhängig von der

Graphgröße zwischen zwei Knoten geben kann. Unsere Analyse basiert auf zwei speziellen Graphfamilien und wird dann auf alle Graphen generalisiert. Wir beweisen, dass die Anzahl der personalsierten Pfade subexponentiell in der Graphgröße und exponentiell in der Zahl der Routenplanungskriterien ist.

Abstract

Personalization is an important trend in today's digital world. One area where this trend has not yet had deep impact is route planning. This dissertation discusses the multi-criteria route planning approach called personalized route planning model. The model optimizes convex combinations of route planning criteria, e.g., travel time, distance, road type, etc., and gives different results based on user preferences. We call the resulting optimal paths *personalized paths*. Specifying preferences is not an easy task for users. Therefore, we focus on applications that handle the preferences in the back end without exposing it to users, in the practical parts of our research. We present three different practical applications and a theoretical analysis of the model. For each application, we provide efficient algorithms and experiments to verify result quality and running time.

The first application deals with finding large numbers of *alternative routes* which do not overlap too much. To this end, we develop an algorithm that is able to enumerate all personalized paths. This algorithm is then extended to optionally avoid computing paths that do not meet certain stronger optimality conditions. Furthermore, we present 'invented' routing criteria to diversify the results even more if needed. In our experiments, we observed large sets of meaningful alternative routes for the users to choose from.

While our first application is aimed at end users, the next two are focused on learning from existing trajectories. The second one is about *identifying intermediate destinations* in trajectories. We assume that on long trips with intermediate destinations drivers will take personalized paths between destinations. So, we employ a trajectory segmentation approach, which partitions trajectories into personalized paths. Our approach recovered about 60% of all known intermediate destinations from real-world trajectories exactly and 90% were in close neighborhood to our predicted destinations.

The final application considers *clustering trajectories by preference*. Here, the idea is to explain a set of trajectories by a small number of preferences, such that each trajectory is optimal for one preference. This allows finding important preferences for a single driver or classifying multiple drivers into categories, e.g., cruisers and speeders. We approach this by computing the preference space of all trajectories and solving a geometric hitting set instance based on the preference spaces. While both of these steps are expensive in the worst case, we found that the process can often be computed in seconds to minutes for instances with thousands of trajectories.

In the last part of this dissertation, we analyze how many personalized paths can exist between two vertices depending on the graph size. We base our analysis on two specific graph families and generalize to all graphs. We proved that the number of personalized paths is subexponential in the graph size and exponential in the number of criteria used.

Acknowledgements

For this thesis coming together, I have many people to thank. I will try my best to not forget anyone and to give credit for the support I have received. I know already, that my words cannot do justice to the wonderful people around me and I am nearly sure I will miss someone. Just know you are much appreciated.

I want to thank my supervisor Prof. Dr. Stefan Funke. His lectures introduced me to the field of Algorithm Engineering, sparked my enthusiasm for it and made me seriously consider pursing a PhD for the first time. Thank you for providing me the opportunity and for your support throughout my time at the FMI. I am grateful for the suggestions, insights and pointers I received in our conversations as well as for the time and trust you gave me to figure things out on my own.

Next up, I want to thank my colleagues at FMI: Daniel Bahrdt, Filip Krumpe, Thomas Mendel, Claudius Proissl, Tobias Rupp and Felix Weitbrecht. I thank you for the brainstorming sessions, the discussions, the table soccer sessions, the coffee and cake breaks, for showing up in the online space when social distancing was necessary and most of all for making our time together not only about research and teaching, but also about joy and companionship. An extra thanks for proof reading this document goes to Claudius Proissl, Tobias Rupp and Felix Weitbrecht.

I want to thank my coauthors of the papers we published together: Stefan Funke, Claudius Proissl, Jan Rapp, Tobias Jepsen Skovgaard and Sabine Storandt. It was a pleasure working with you and developing ideas into lemmas, theorems, algorithms and research papers.

The last but surely not least important group to thank consists of my “chosen family”, that supported me in the private space. Most important here is Ilona Ihle who will be my wife by the time this document is published. She always believed in me and supported me even when the work at a paper with an impending deadline ate large parts of my weekend time. Thank you, I am truly fortunate to receive your love and support. I am proud for having you at my side! Finally, I want to mention my closest friends who have probably listened to me talking about route planning algorithms way too much. Thank you for your interest, your patience, your understanding and most of all your friendship Fabian Friz, Igor Hörner and Jan Rapp.

Chapter 1
Introduction

1.1. Motivation

Navigation system are common for basically every mode of transportation. No matter if you plan a trip by foot, bicycle, car or public transportation, systems which are specialized for those purposes are available. Usually, these tools will yield routes or trips that are optimized for one - potentially selectable - criterion like travel time or number of transfers. Typically, there are only few options to customize the routing results to ones liking like adding intermediate destination or avoiding certain types of roads. In this work, we explore a routing model which uses multiple criteria for optimal paths and makes the importance of these criteria configurable per query. So customization cannot only happen per user but in theory a user can tweak every route to their liking by tuning the importance of the criteria. In day-to-day use, finding the right configuration for every query can be difficult and time consuming. So instead of requiring a user to do all the work, such a system should propose good options to the user or learn his preferences. We present three practical applications for this model, which showcase capabilities a personalized navigation system might have. These include finding suitable alternative routes for the user to choose from but also gaining insight in the users routing preferences by analyzing existing trajectories.

The rest of this chapter, contains basic definitions and building blocks used in the later chapters and an overview of the rest of the work.

1.2. Routing

Routing is the process of finding a path between two given points. More specifically in this thesis, we are talking about finding an optimal (or shortest) path in a road network represented by a graph $G = (V, E)$.

Definition 1.1
A graph $G = (V, E)$ is a pair of a set of vertices V and a set of edges E. An edge $e \in E$ is a pair of vertices (v_1, v_2) with $v_1, v_2 \in V$.

The vertices of G represent intersections of the road network, while the edges represent the roads connecting the intersections. The edges of a graph can either be *undirected* or *directed*. If $e = (v, u)$ is

undirected, it means that edge can be used along both direction, from u to v and from v to u. In the directed case, traversal is only possible from v to u. All graphs in this work are directed. The cost of traveling along such an edge is usually modeled by a cost function $c : E \to \mathbb{R}_0^+$. The cost function assigns each edge some cost, such as seconds to traverse (travel time) or meters (distance). In the following section, we will use distance as the example representative for any cost type. A path π in a graph is a sequence of edges $e_1, e_2, \ldots, e_j$ such that each two consecutive edges $e_i = (u, v), e_{i+1} = (v, w)$ share a vertex v between them. One algorithm to find a shortest path, given a graph G a cost function c (which assigns non-negative costs) and source and destination vertices s and t is Dijkstra's Algorithm.

1.2.1. Dijkstra's Algorithm

The main idea of Dijkstra's algorithm [Dij59] is to explore the graph from s in ascending distance order until t is reached. One can picture this by an expanding circle centered in s. The vertices inside the circle have a known shortest path distance. The vertices on the boundary or frontier of the circle are adjacent to vertices inside the circle but their shortest path distance is not yet known. Finally, the vertices outside of the circle were not yet observed by the algorithm. To facilitate this, the algorithm keeps a heap of the current frontier of vertices which were seen but not yet explored. In every iteration, the vertex with the shortest distance in the frontier is taken from the heap and all its neighbors inspected. Taking a vertex from the heap is called *settling* the vertex as its shortest path distance is now known. Inspecting an (outgoing) edge of a vertex is called *relaxing* the edge. If the neighbors can be reached with a better distance than previously known, they are added to or updated in the heap with this distance. The algorithm computes all shortest path distances from s in running time $O((|E| + |V|) \log |V|)$ when using a binary heap and $O(|E| + |V| \log |V|)$ when using a Fibonacci heap [MS08]. It is the fastest-known algorithm for the single source shortest path problem on directed graph with non-negative costs in terms of the asymptotic worst case running time. A trivial lower bound for the problem would be the size of the input graph $\Omega(|E| + |V|)$. It is not known if this lower bound for the problem can be achieved or if Dijkstra's algorithm has an optimal running time. Because Dijkstra's algorithm processes nodes in ascending distance order, thereby implicitly sorting them, an alternative algorithm to compute shortest paths would need to not process nodes in this order or have at least $\Omega(|V| \log |V|)$ running time for the sorting of nodes. In practice, on a country sized road network a shortest path query takes a few seconds to complete.

An important variation of the algorithm for this work is the *bidirectional* Dijkstra. Here the graph is explored both from s and from t. When the first vertex has been settled from both searches, all vertices on the shortest path have been processed by at least one of the searches. By starting from both sides, the algorithm has to explore a smaller number of vertices in most queries and in practice runs in about half the time of a regular Dijkstra.

1.2.2. Contraction Hierarchies

On top of Dijkstra's Algorithm, speed up techniques have been developed. These techniques usually improve the shortest path query time by using a data structure that was computed in a preprocessing phase. One important approach in practice is called Contraction Hierarchies (CH) [GSSV12]. The high level idea of this approach is to use the inherent hierarchy in road networks. Each vertex in the graph is assigned a rank signifying its importance. The query is a modified bidirectional Dijkstra which only considers edges leading to higher ranked vertices. To ensure that queries still yield the

same shortest paths, shortcut edges are added to the graph. On road networks, this approach typically yields a query speed up of about 1,000 compared to a regular Dijkstra.

Preprocessing In the preprocessing phase, the ranks for the vertices are assigned and shortcuts are added to the graph. The shortcuts are created in a process called *node contraction*. In this process, one vertex v gets contracted and assigned a rank. Vertices contracted later have higher rank, i.e., they are "more important". For every pair of neighboring vertices u and w of v a shortcut is created, if the only shortest path from u to w is of the form $(u,v),(v,w)$. The shortcut from u to w will have cost equal to the shortest path distance and store a reference to the pair of edges it replaced. An example of this is shown in Figure 1.1. Afterwards, the contracted vertex and the edges adjacent to it are removed from the graph for the rest of the preprocessing phase. The order in which vertices are contracted is important for the efficiency of the final data structure, as it determines the ranks of the vertices and the amount of created shortcuts. The goal would be to create only a small number of shortcuts and have vertices that are part of many shortest paths have high rank. Unfortunately, finding the optimal contraction order is NP-hard [BCK+10]. Hence, different heuristic strategies have been developed and are used in practice.

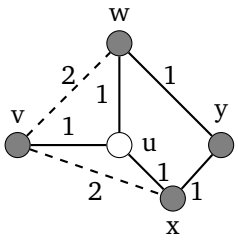


Figure 1.1.: Example Node contraction of vertex u. Shortcuts (dashed lines) have to be created for vertex pairs vw and vx but not for wx because there is a path via y which has the same distance as the one via u. [GSSV12]

Query The CH-query is a special bidirectional Dijkstra query. Both searches only relax edges that lead to vertices with higher rank. By construction the vertex v that minimizes $d_s(v)+d_t(v)$ where d_s is the distance calculated by the CH-query rooted in s and d_t respectively for t, is on the shortest path from s to t. In contrast to the "normal" bidirectional Dijkstra, the CH-query can only stop when either all reachable nodes are settled or each search $x \in \{s,t\}$ has found some vertex w_x such that $d_x(w_x) > d_s(v)+d_t(v)$ where v is the vertex with the shortest combined distance found so far. While the distances computed via this query are the same as via Dijkstra the actual paths now contain shortcut edges which need to be resolved. This can be done by recursively unpacking the shortcuts in the path via the references to the replaced edges until all edges in the path are edges from the original graph.

Uncontracted Core There are some graphs for which a full contraction is not advantageous. This usually happens for graph with complicated edge costs, e.g., in time-dependent route planning [Bat14] or personalized route planning. With such edge costs, the number of shortcut increases tremendously at some point. When the number of shortcuts gets too large, the additional memory used and the number of edges needed to be visited per vertex negatively affect performance. Which means contraction only makes sense up to a certain level. In these cases, leaving a small *core* of vertices

uncontracted is often a performance benefit for query and preprocessing times. To keep the queries correct, the core vertices are assigned the highest rank in the CH graph and the query will now consider neighbors with greater or equal rank to the current vertex. The effect of this is, that CH-query will behave as normal for non-core vertices and like a standard Dijkstra for core vertices.

1.3. Linear Programming

Vanderbei [Van20] describes Linear Programming as an approach to find the optimal solution for a linear objective function under a set of linear constraints. A linear program (LP) is given as follows:

$$\text{Maximize} \qquad c^T \cdot x \tag{1.1}$$

$$\text{Subject to} \qquad A \cdot x \leq b \tag{1.2}$$

Here Equation (1.1) is the objective function. It consists of the two vectors $c, x \in \mathbb{R}^n$ for an LP with n variables. Here, c contains the coefficients of the objective function as given by the instance and x contains the variables which need to be optimized. In Equation (1.2) m constraints are given via a matrix $A \in \mathbb{R}^{m \times n}$, which contains the coefficients of the constraints, and a vector b which contains the limits of the constraints. So the j-th constraint looks as follows:

$$\sum_{i=1}^{n} a_{ij} \cdot x_i \leq b_i$$

An LP can be visualized as the intersection of the halfspaces defined by its constraints. If the intersection is non-empty, the intersection of the halfspaces is called feasible region. In such a picture, the objective function can be shown as a vector pointing into the direction of the optimal solution. If the optimal solution is bounded and unique, it is the vertex formed by at least n halfspaces which is the furthest into the direction of the objective function vector.

Figure 1.2 shows an example for this.

The simplex algorithm for solving LPs was first developed by Dantzig [Dan51]. It is an iterative method that starts at some feasible vertex formed by n constraints and then goes to a neighboring vertex with a better objective function value until no better vertex can be found and the optimal solution is reached. This moving to a neighboring vertex is called the *pivoting step* and multiple *pivoting rules* exist. The algorithm is efficient in practice but [KM72] showed that LPs for which the simplex algorithm with a deterministic pivoting rule takes exponential running time (in the number of variables and constraints) exist.

Another algorithm to solve LPs is the ellipsoid method [GLS81]. The idea is to start with an ellipse that is guaranteed to contain the whole feasible region and iteratively shrink the ellipse until the optimal solution is found or the feasible region is proven empty. To shrink the ellipse a *separation oracle* is employed which given some point p returns a constraint of the LP that is not fulfilled by p which will then be used to shrink the ellipse in "the right direction". In contrast to the simplex algorithm, for the ellipsoid method one can prove a polynomial worst case running time. At the same time, this method performs worse on most instances in practice.

Furthermore, *interior-point methods* have delivered a different approach to solving LPs. Some of these methods, notably one presented by Karmarkar [Kar84], not only have a theoretical guarantee of a polynomial running time, but also perform well in practice. [PW00]

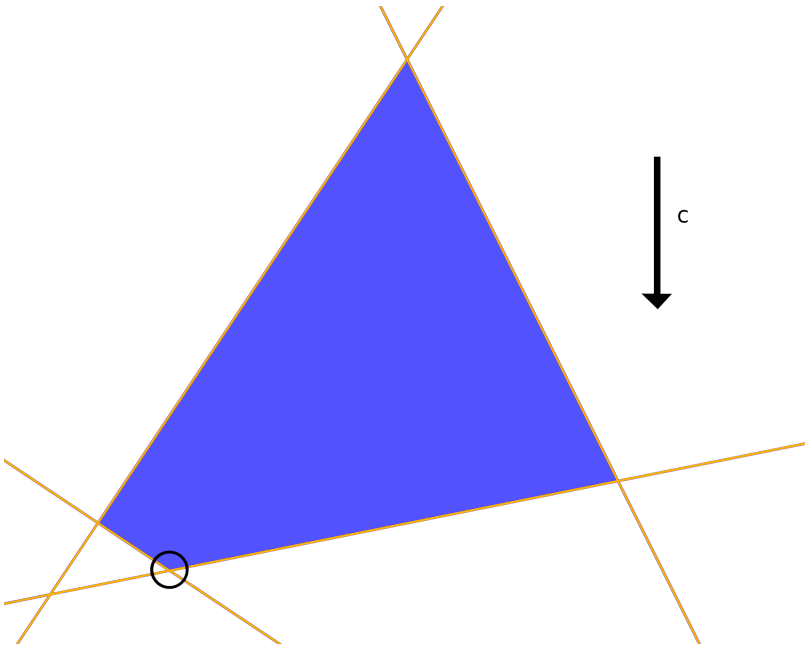


Figure 1.2.: Visualization of a linear program with 2 variables. On the right side, the objective function vector c is shown. The feasible region is colored in blue and the vertex that represents the optimal solution is circled.

1.4. Personalized Route Planning Model

The personalized route planning model is a way to represent different driving behaviors in routing [FS15]. Instead of minimizing one kind of cost, e.g., travel time, the model considers d cost types. A cost function $c : E \to \mathbb{R}_+^d$ assigns each edge a vector of costs of length d. The driving behaviors are modeled via preference vectors $\alpha \in [0,1]^d$ whose entries sum up to one, i.e., $\sum \alpha_i = 1$. The cost for an edge e under preference α is then defined as $c(e,\alpha) := \sum_i c(e)_i \cdot \alpha_i$ and similarly the cost for a path π is $c(\pi,\alpha) := \sum_{e\in\pi} c(e,\alpha)$. To decide the optimal path between two vertices s,t with preference α one needs to find the path π from s to t which minimizes $c(\pi,\alpha)$. Clearly, such an optimal path can be found via a modified Dijkstra's algorithm, which takes a preference α as additional input and calculates $c(e,\alpha)$ on the fly. We call a path that is optimal for some α a *personalized path*. In the literature, the model is also known as the *parametric shortest path problem* and usually discussed with fixed d, see for example [Gus80].

1.4.1. Personalized Contraction Hierarchies

The Contraction Hierarchies technique was adapted to the personalized route planning model by [FLS17]. The key difference to the single cost case is that it is no longer possible to easily tell if some path is the optimal path at preprocessing time because it is dependent on the preference used at query time. Therefore, shortcuts need to be added for any path that is optimal for some α in the node contraction step. As there exists an infinite amount of preferences, a structured approach is needed to determine whether a path is a personalized path. We know that for a personalized path π^*

which is optimal for preference α^* and is part of the set P of all paths from s to t the following holds:

$$\forall \pi \in P : \qquad c(\pi^*, \alpha^*) \leq c(\pi, \alpha^*) \tag{1.3}$$

This can be used as a set of constraints for an LP to test whether some path π^* is a personalized path. If one defines the α to be the variable vector and adds constraints to make the result a valid preference as seen in Equations (1.4) and (1.5), this LP yields either an α for which π^* is optimal or is infeasible if π^* is not an optimal path.

$$\sum_i \alpha_i = 1 \tag{1.4}$$

$$i = 1, \ldots, d : \qquad \alpha_i \geq 0 \tag{1.5}$$

Unfortunately, the path set P might be exponentially large which means creating the whole LP might be infeasible in practice. Instead of enumerating all paths an incremental approach is possible, when solving the LP via the ellipsoid method. This method does not need to know all constraints to solve the LP instead it needs a separation oracle. In this case, the oracle gets as input a certain α and should output one of two things. Either, a certificate that π^* is optimal for α or a constraint that is violated by α. The adapted Dijkstra's algorithm is a perfect fit for this. A query with the α in question will return either that π^* is optimal or another path π' that has less cost then π^* for this α. Hence, $c(\pi^*, \alpha) \leq c(\pi', \alpha)$ is not fulfilled. In summary, the following procedure efficiently answers the question "Is π^* a personalized path?" in the node contraction process.

1. Initialize LP with Equations (1.4) and (1.5).
2. Solve the LP using ellipsoid method. If the LP is infeasible return false. Otherwise let the solution be α^*.
3. Run Dijkstra with α^*. If the optimal path is π^* return true. Otherwise, add the constraint $c(\pi^*, \alpha) \leq c(\pi', \alpha)$ to the LP where π' is optimal for α^*. Go to 2.

Funke, Laue, and Storandt [FLS17] call this procedure the *LP-Oracle*. Let P be the set of paths returned by the separation oracle so far, then the LP looks as follows in every iteration:

$$\begin{aligned}
&\text{Maximize} && \alpha_1 \\
&\text{Subject to:} && \\
&\forall \pi \in P : && c(\pi^*, \alpha^*) \leq c(\pi, \alpha^*) \\
& && \sum_i \alpha_i = 1 \\
&i = 1, \ldots, d : && \alpha_i \geq 0
\end{aligned}$$

Note that the objective function was chosen arbitrarily and is not important for the purpose of the LP-Oracle.

1.5. Outline

In Chapter 2 we show how to use this model to find a diverse set of alternative routes. Furthermore, we show how to enumerate only those routes that are not "too bad" in important metrics and diversify

the results even further by inventing new metrics. The next part focuses on explaining existing trajectories with this model. Chapter 3 presents how to split long trajectories into smaller parts to identify intermediate destinations. Instead of looking at individual trajectories, Chapter 4 shows how a set of trajectories can be clustered, such that each cluster contains only trajectories that can be explained by the same preference. Chapter 5 explores the theoretical underpinnings of the model. It focuses on finding upper bounds for the number of optimal paths in the model and shows properties of the preferences used in the model. Finally, Chapter 6 concludes the dissertation and gives some ideas on possible future work.

Part I.

Personalized Paths as alternative Routes

Chapter 2

Enumerating personalized Routes efficiently

2.1. Introduction

This chapter is based on joint work with Stefan Funke and Sabine Storandt. Its contents were published in the 2019 proceedings of the 21st Workshop on Algorithm Engineering and Experiments (ALENEX) [BFS19] (up to and including Section 2.5) and the proceedings of the 12th ACM SIGSPATIAL International Workshop on Computational Transportation Science [BF19] (from Section 2.6 onward). I contributed to all parts of this chapter.

Route planning in road networks is an intensively studied field. While Dijkstra's algorithm takes several seconds to find the shortest route for a distance metric in a country sized network, multiple speed up techniques have been developed in recent years. These techniques usually employ some form of preprocessing to build data structures which later enable query times for shortest path queries of only a few *milli-* or even *microseconds*. See [BDG+15] for a comprehensive survey.

With the single weight case being well understood, people have started to consider more complex route planning tasks, e.g., scenarios where the edge weights are functions dependent on time, several weight values associated with each edge, or dynamically changing edge weights. One of the more complex route planning tasks is the computation of not only one optimal route, but a selection of several, hopefully reasonable, but sufficiently distinct routes. This problem is of practical importance, since routes that people actually take are often not simply the quickest or shortest routes but take into account additional aspects like quietness, road surface properties, or scenicness. As most of the time these criteria are unknown to a route planner, modern route planning systems like Google Maps always propose several *alternative routes* for a given query from a source s to a target t, hoping that one of them is to the user's liking.

This chapter explores finding alternative routes via a personalized routing model. For a given source and target node combination s, t, each preference might yield a different optimal path. The hope is that there are many different optimal routes which also appeal to different users. A user preferences like "a short travel time is most important, but the gas price should also not be to high" might e.g., be captured by a preference that values travel time at 70% and gas price at 30%. We

evaluate the quantity of routes found in our experiments and do a qualitative examination on queries of different types.

2.2. Related Work

This section reviews previous works on computing alternative routes. In Sections 2.2.1 to 2.2.3, we review approaches where the only constraint on the alternative routes created – apart from being relatively disjoint – is that they must be 'almost' optimal with respect to the single edge weight that is considered. Sections 2.2.4 and 2.2.5 presents prior work that uses multiple edge weights. Finally, Section 2.2.6 contains an approach to measure the quality of alternative routes created by any approach based on a single edge weight. An overview of the different approaches in regard to run time and number of found alternative paths can be found in Table 2.1.

2.2.1. k-Shortest Paths

Probably the first approach that comes to mind when thinking about the computation of alternative routes is the computation of the k shortest paths from s to t. This means calculating the shortest path, the 2nd shortest path up to the kth-shortest path. Eppstein [Epp94] studied this problem for digraphs with non-negative edge weights which can contain cycles, self-loops and multi-edges. They use an implicit representation of the shortest paths to ensure running times of $O(|E| + |V| \log |V| + k)$ for finding k-shortest paths.

The implicit path representation uses the fact that every s-t path p can be represented by the sequence of its edges S_p which are not part of the single-destination shortest path tree rooted in t. The representation is a heap of paths where each path p has a parent path prefix(p) and an edge e such that $S_p = S_{\text{prefix}(p)} + \{e\}$. The root of the heap is the shortest path with $S = \emptyset$. Using this heap of paths, it is possible to provide some properties of each path, e.g., the length, in constant time and build an explicit representation in time proportional to the number of edges of the path.

Unfortunately, the computation is complex and the resulting paths are typically almost identical, differing in only few edges.

The approach by [CBGL15] is in fact based on the k-shortest path idea and addresses the "similarity" downside of the k-shortest path approach. It essentially enumerates shortest paths in increasing distance and discards paths that are too similar. They also introduce the OnePass algorithm which checks the similarity of the paths in construction (a subpath of a potential k-shortest path) with paths already found. Thereby pruning candidates early in the search and improving the running time in comparison to their baseline.

Yet, it seems to be only viable for very small networks of few thousand nodes, as Figure 3 in their paper shows running times in the order of tens of seconds for city-sized networks.

2.2.2. Edge Usage Penalty Approach

More successful attempts (in particular w.r.t. to practicality) have been based, e.g., on the *penalty approach*. Here, one starts with the optimal shortest path but then additively penalizes its edges before recomputing another s-t shortest path in the reweighted network. Bader et al. [BDGS11] note that a fixed penalty per edge, penalizes paths with many edges more than those with few edges. They instead propose a *penalty-factor* by which each edge weight should be increased. Furthermore,

they consider the case of alternative paths that only make short detours from the shortest path and join it again, which is not desirable in most situations. To counter this behavior, the authors utilize a *rejoin-penalty* to edges that leave/join the shortest path (or any of the previously computed alternative paths). The rejoin-penalty should be based on the aforementioned penalty-factor as well as the length of the shortest $s-t$-path.

This approach suffers from the need of careful tuning of the reweighting parameters to obtain good results. The authors stated that penalty factors between 0.3 and 0.4 yielded the best results but did not report how they obtained these values.

2.2.3. Via Nodes

Another approach is based on so-called *via nodes*. Here the idea is to compute a selection of $s-t$ paths that are all of the form 'shortest path from s to some other node v' followed by a 'shortest path from v to t'. Abraham et al. [ADGW13] first define admissibility criteria for an alternative route P and then develop exact and heuristic algorithms to find via nodes. Lastly, the found nodes are sorted by an objective function and (approximate) admissibility is verified.

Admissibility Criteria The criteria are limited sharing, local optimality and uniformly bounded stretch (UBS). Limited sharing restricts the length of P that can be shared with the optimal path Opt. Local optimality means that subpaths of P below a length T must be optimal paths. UBS limits the length of each subpath P' from s' to t' to be at most an ϵ-factor longer than the optimal path from s' to t': $l(P') \leq (1+\epsilon)l(Opt(s',t'))$.

Checking the local optimality and UBS naively takes $O(|P|^2)$ time which makes the checks infeasible in practice. The authors therefore provide heuristic checks which only require a single point to point query each. The local optimality check uses that the property can only be violated around the via node v. By checking the subpath from x to y for optimality where x and y are the closest nodes to v with distance T on either side of v, one finds that either P is T-locally optimal or that it is not $2T$-locally optimal. The authors call this procedure the T-test and go on to prove that if P has a stretch of $(1+\epsilon)$ and passes the T-test for $T = \beta \cdot \text{dist}(s,t)$ (with $0 < \epsilon < \beta < 1$) then P is a $\frac{\beta}{\beta-\epsilon}$ UBS path.

The BDV-Algorithm Family The BDV algorithm is based on bidirectional Dijkstra and differs from it only in the stopping condition. The two searches continue until they find a node that is more than $(1+\epsilon)l(Opt)$ from its origin (No admissible path can be longer than that). Then all nodes reached from both searches can be checked for approximate admissibility and the best alternative path according to the objective function is returned. The CHV and REV algorithms are created with the same approach based on contraction hierarchies [GSSV12] and reach for A^* [GKW09], respectively. The authors state that these algorithms find too many potential via nodes to be practically useful and therefore designed "X-" variants. These variants aggressively prune nodes that are not likely to be good via nodes.

Evaluation All algorithms were evaluated on a continent sized network. On long range queries, the X-CHV and X-REV have fast running times (3ms and 20ms respectively) and moderate to good success rates (58% and 91%). The success rates can be improved by pruning less strictly at the cost of running time. In mid range queries, the success rate drops to about 60% for X-REV.

The approach yields alternative routes that have good qualities with respect to the single metric used and has fast running times. The main disadvantage are the low success rates for finding alternative routes for mid and short range queries which are arguably the most important query sizes in continent sized road networks.

2.2.4. Pareto-optimal Routes

If several criteria are to be considered, one natural idea is to compute the set of all Pareto-optimal solutions. These are all routes which are non-dominated by other routes, which means the routes are better or equal than all other routes in at least one criterion. Delling and Wagner [DW09] augmented the SHARC speed-up technique to work for the multi-criteria case. To achieve this they replace three basic operations of SHARC with multi-criteria variants: shortest path queries, arc-flags computation and contraction.

Augmenting SHARC For the shortest path queries, they use a multi-criteria Dijkstra query which assigns lists of labels to each node instead of a single label.

Arc-flags are used to reduce the number of edges that need to be considered in a shortest path query. The nodes of the graph are partitioned into cells and each edge contains a flag for each cell. In the single criteria case, if an edge is contained in a shortest path into a cell, then the flag for that cell is set to true. For the multi-criteria case, the authors set an arc-flag to true if there exists at least one Pareto path into the cell which contains the edge.

The contraction step of SHARC works similarly to the node contraction in CH. Nodes and their edges get temporarily removed and shortcut edges which maintain the shortest path property are inserted. Note that not all nodes are contracted but a core of uncontracted node is left. To adapt the step to the multi-criteria setting, the authors allow multi-edges and augment the edge reduction phase which removes unnecessary shortcuts. The latter is done by running a multi-criteria Dijkstra from all core nodes u to all their neighbors v. Then all edges from u to v whose labels are dominated by one of the labels of v can be removed.

For a graph with 77k nodes and *two* metrics only, they report running times around 200ms. They stated in the paper explicitly that more than 2 metrics are only considered for very strongly correlated travel time metrics. For a network of 18Mio nodes they state: “however, it turns out this input is too big for finding all Pareto routes”.

Heuristically reducing the Number of Paths The authors also present an algorithm based on a tightened definition of dominance. For this definition an important criterion is determined (travel time is used in the paper) and paths are only allowed to be an $(1+\epsilon)$-factor worse in this criterion. This new definition allows to reduce the running time as well as the number of returned paths significantly. Given a small enough ϵ (≤ 0.05), this approach is also viable on the continent sized road networks.

Skyline Queries In [KRS10] an approach inspired by skyline queries, which also yields Pareto-optimal routes, is presented. They show two pruning strategies that allow to recognize dominated (sub-)paths during exploration to reduce runtime. The first is based on the A^*-search which uses lower bounds to direct the path search. They use these lower bounds for the path costs to do a dominance check on sub-paths. Thereby pruning sub-paths which cannot be Pareto-optimal without exploring

the whole path. The second strategy relies on the fact that sub–paths of Pareto-optimal paths are Pareto-optimal as well. The authors expand their algorithm to keep track of the Pareto-optimal path costs found for each node and prune sub-paths that are dominated at their current node. The approach also turned out to be feasible only for very small graphs, e.g., a query time of 40s on a graph with 6k nodes and three metrics was reported.

2.2.5. Triangle Splitting

In the master's thesis [Bar18], we developed an algorithm for finding alternative routes based on the personalized route planning model (see Section 1.4). For the case of three criteria, the preference space forms a triangle in the plane. The corners of this triangle are the preferences and respective paths that each only consider one of the criteria. The idea is to split this triangle successively into smaller triangles and gain new alternative routes with every split. Each split is performed at the center of some triangle whose coordinates form the preference used for the next alternative path. Also, with this center point three new triangles are formed which each consist of two corners from the original triangle and the split point as third corner. See Figure 2.1 for an example. The paths obtained this way are optimal for their preference and also Pareto-optimal. One can show that a triangle that is formed by preferences belonging to the same path cannot yield new paths and therefore does not need to be split further. In general, this algorithm does not converge to a situation where all triangles are of this form and therefore needs to rely on heuristics like a limited refinement depth to ensure termination.

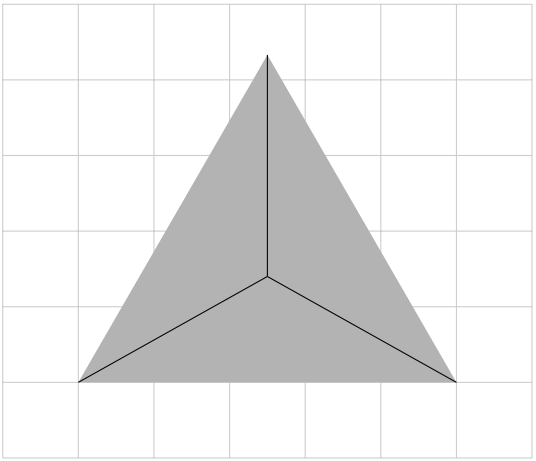


Figure 2.1.: Triangle split into three parts [Bar18]

2.2.6. Quality Measures for Alternative Paths

Somewhat orthogonal to these approaches, Bader et al. in [BDGS11] propose to not only consider the set of alternative routes as routes but also consider their overlay, which they call *alternative graph*. An alternative graph $H = (V', E')$ can be built for a graph $G = (V, E)$ and source and target nodes s and t. Its nodes are a subset of the graph nodes $V' \subseteq V$ and it edges are representations of paths in G. For every edge $e = (u, v) \in E'$, there exists a path p from u to v in G with $cost(e) = dist(u, v)$ and there exists a simple s-t-path which contains p. Such an H can be created iteratively by starting with the shortest path and adding new alternative paths if they fulfill some constraints or improve the

Algorithm	Graph Size #nodes	Metrics	#Alternative Paths	Runtime
Via-Nodes [ADGW13]	18M	1	3	4ms
Pareto-SHARC [DW09]	77k	2	100	200ms
Pareto-SHARC (tightened dominance definition) [DW09]	18M	2	23	400ms
Triangle Splitting [Bar18]	10M	3	20	5s
Skyline-queries [KRS10]	6k	3	250	40s
k-shortest paths [CBGL15]	18k	1	5	120s
Edge Penalty [BDGS11]	not reported			

Table 2.1.: Running times of Related work implementations. Note that this overview is not a fair comparison as the experimental settings as well as the concrete problems the algorithms solve are very different. Some of the approaches require preprocessing of the graph while others have a low success rate.

quality measures. The authors develop three main quality measures for alternative graphs: total distance, average distance and number of decision edges.

$$\text{totalDistance} := \sum_{e=(u,v)\in E'} \frac{cost(e)}{dist(s,u)+cost(e)+dist(v,t)}$$

$$\text{averageDistance} := \frac{\sum_{e\in E'} cost(e)}{dist(s,t)\cdot \text{totalDistance}}$$

$$\text{decisionEdges} := \sum_{v\in V'\setminus\{t\}} \text{outdegree}(v)-1$$

The total distance measures how much the alternative paths are nonoverlapping. It increases up to k for k disjoint paths. The average distance, on the other hand, describes the average stretch of the alternative paths. Finally, the number of decision edges measure the complexity of the alternative graph. The authors suggest to limit the number of decision edges and the average distance and maximize the objective function $\text{totalDistance} - \alpha(\text{averageDistance} - 1)$ for some parameter α.

This approach is a general way to represent and measure the quality of a set of alternative routes mainly in regard to the used distance criterion. It suffers a bit from its definition of admissibility which depends on the insertion order of the routes.

2.3. Parameter Space Exploration via Partial Convex Hull Construction

This section contains the main contribution of this chapter. It provides an algorithm that efficiently enumerates optimal paths in the personalized route planning model and gives proofs for correctness and termination of this algorithm.

For a given source s, target t and a graph G with d weights on every edge, we are interested in finding a large set $\{\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(k)}\}$ with $\alpha^{(j)} \in [0,1]^d$ and $\sum_{i=1}^{d} \alpha_i^{(j)} = 1$ for all $j = 1, \ldots, k$, such that the respective optimal paths $\pi_1, \pi_2, \ldots, \pi_k$ are 'sufficiently' different. Our approach will be based on a parameter space exploration by partially computing the convex hull of a set of suitable points in $\mathbb{R}^d$.

2.3.1. Exhaustive Exploration

Let us first try to explore *all* paths from s to t that are optimal for some α ignoring overlap between the paths. In the following, we will explain our approach for the general case of d metrics, but illustrate our concepts for the case of two metrics. Later on, our approach is evaluated for a concrete application scenario with three metrics.

Consider d metrics (weights on each edge) of the graph that are to be minimized, e.g., for $d = 2$ distance and (positive) height difference in case of a simple route planner for bicycles. Every s-t-path π (also a suboptimal one) has aggregated costs $c_i(\pi)$ for $i = 1, \ldots, d$ in each of these metrics. We associate the point $(c_1(\pi), c_2(\pi), \ldots, c_d(\pi)) \in \mathbb{R}^d$ with the path π. Let P be the set of points corresponding to all possible s-t-paths; note that P resides in the positive orthant of $\mathbb{R}^d$. Consider Figure 2.2 for the example of $d = 2$. Here, every point in $\mathbb{R}^+_0$ corresponds to the two costs c_1 and c_2 associated with an s-t-path. Clearly, some of the paths are not really of interest, e.g., if a path has higher or equal cost than another path in each dimension (and strictly higher cost in one dimension), the former is said to be *dominated* by the latter. The set of non-dominated paths is called the *Pareto set*. In our concrete example for $d = 2$, the path corresponding to point $(3.5, 5)$ under no circumstances is preferrable to the path $(1.5, 4.5)$. So $(3.5, 5)$ is *dominated* by $(1.5, 4.5)$. In Figure 2.2 all non-dominated paths are marked in blue. Clearly, a dominated path is not of interest for our purpose, since it can always be replaced by a path dominating it. Note though that it is not the case that for every non-dominated (or *Pareto-optimal*) path there exists an α such that the path is the only optimum under this preference α, in Figure 2.2, $(2.5, 4)$ is such an example. In fact, only the Pareto-optimal paths that are extreme points of the convex hull of P are unique optima for some α:

Lemma 2.1
An s-t-path π is a unique optimum for some $\alpha \in \mathbb{R}^d_{\geq 0}$ if and only if it is Pareto-optimal and an extreme point of the convex hull of P.

Proof If a point p is an extreme point of the convex hull of P, there exists a halfspace $a^T x \geq b$ which has p on its boundary ($a^T p = b$) and all other points p' strictly on one side ($a^T p' > b$ for all $p' \in P - \{p\}$). Since p is Pareto-optimal, such a, b must exist with $a, b \geq 0$. Normalizing a yields the desired α. On the other hand, if a path π is an unique optimum for some α, it has to be Pareto-optimal and for all other paths π', their weighted cost under this α is higher. This again translates into a respective half space certifying being an extreme point of the convex hull. □

In Figure 2.2, the paths extreme for the convex hull are additionally circled in red, the respective convex hull edges are also drawn in red. Note that there are paths which are on the hull boundary but not extreme points. These points are surrounded by a dashed circle. There is no α for which $(6, 1.5)$ is the only optimal path. Only for $\alpha = (\frac{1}{3}, \frac{2}{3})$ it is optimal together with the neighboring extreme points. In general, the set of Pareto-optimal paths can have exponential size and has turned out to be too big to handle in practice even for moderately-sized networks with uncorrelated metrics (see, e.g., [DW09]). The set of points/paths on the convex hull is considerably smaller and can be computed for moderately sized networks. See Chapter 5 for a theoretical analysis of the maximum number of optimal paths. But more importantly, there are natural ways of *partially* computing the boundary of the convex hull with its extreme points that are efficient enough for practical use and that lead to a good selection of alternative paths.

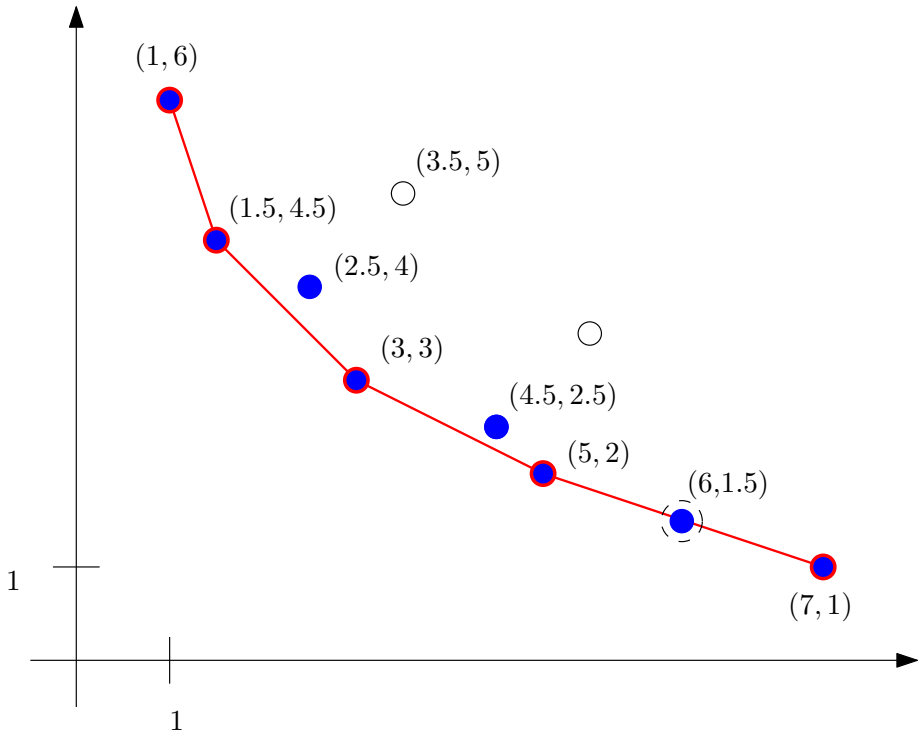


Figure 2.2.: Space of possible paths in terms of distance and height differences for $d = 2$.

2.3.1.1. Convex Hull Exploration

Conceptually, the complete exploration of the convex hull of all paths is quite simple. We start by computing optimal paths for each of the d metrics. This yields d points in $\mathbb{R}^d$ spanning a $d-1$ dimensional simplex in $\mathbb{R}^d$. We determine the preference α for which these d paths are equally valued, and then search for the optimum path for preference α. Let us illustrate this for our example in $d = 2$ in Figure 2.3. The optimum path for the first metric corresponds to point $(1, 6)$, the optimum path for the second metric to $(7, 1)$. We then search in direction indicated by the small arrow in Figure 2.3, top, which corresponds to searching for the optimum path for $\alpha = (5/11, 6/11)$. In this example, this yields a path with cost/point with coordinates $(3, 3)$ (see Figure 2.3, top right); by computing the convex hull of the now three points, we identify facets and directions in which to search further for optimum paths. In general, having determined an optimal path for α, we compute the part of the convex hull of the now $d+1$ points in $\mathbb{R}^d$ and inspect its facets which are again $(d-1)$-dimensional simplices. When inspecting a facet and searching for an optimal point for the respective α, two things can happen: (a) the resulting path has the same objective function (for this α) as the d points spanning the facet; in this case, the facet is indeed part of the relevant part of the final convex hull. (b) the resulting path has a better objective function value (like in the example above for $(3, 3)$); in this case we have to update the convex hull und explore newly created facets. Figure 2.3 depicts two more steps in the hull exploration where point/path $(1.5, 4.5)$ is found when inspecting the facet spanned by $(1, 6)$ and $(3, 3)$, and point/path $(5, 2)$ is found when inspecting the facet spanned by $(3, 3)$ and $(7, 1)$. At the very end, the relevant part of the convex hull has 4 facets, none of which yield new paths when optimizing for the respective α, hence the relevant part of the convex hull is complete and we have determined all paths that are optimal for some α value.

Note that for large distance queries, the complexity of the relevant part of the convex hull might still be too high to afford full computation, so we will refine the exploration in the following to be able to abort computation at any time but still obtain useful results.

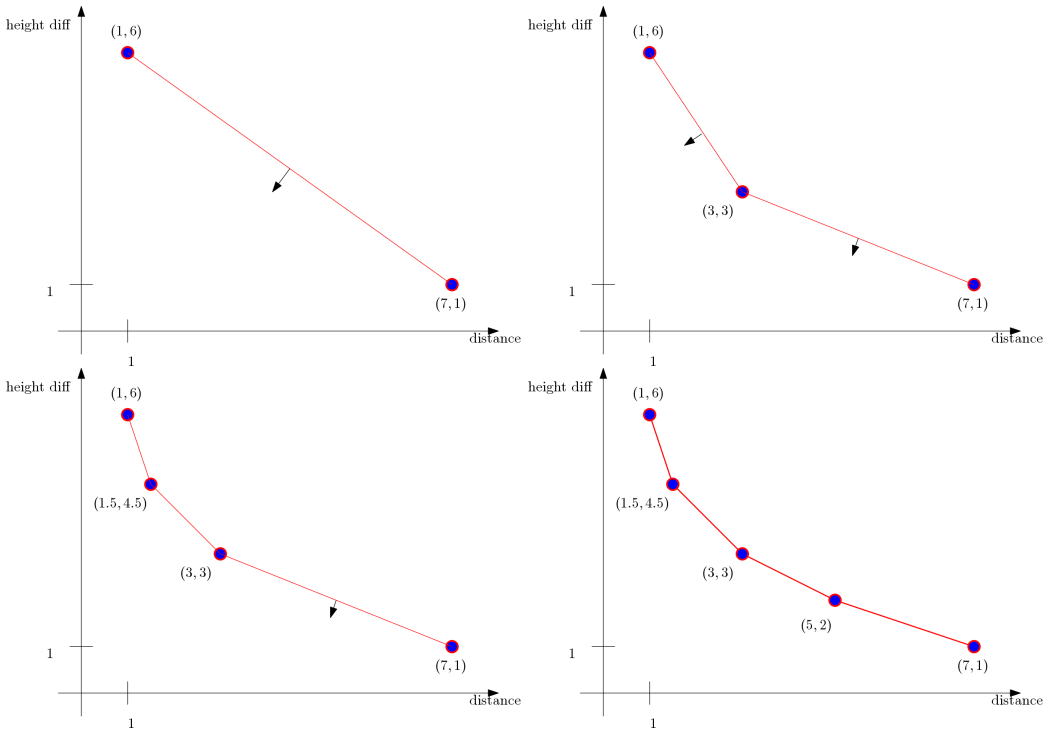


Figure 2.3.: Steps of exhaustive hull exploration for $d = 2$.

2.3.2. Bounded Exploration with Guidance

Depending on the scenario, full exploration of the relevant part of the convex hull might be too expensive, since the number of searches with some α is essentially the number of $d-1$ simplices therein.

So we will modify our exploration strategy to simply stop after a given number R of exploration steps. As we are interested in as diverse as possible s-t-paths at the end, we better guide the exploration and prefer inspecting facets that might lead to considerably different new paths.

To that end, we introduce a prioritization of facets which are candidates for refinement. Let us define for two paths π_1, π_2 the *overlap coefficient* $\kappa(\pi_1, \pi_2)$ as the number of edges shared by both paths divided by the smaller edge count of the two paths:

$$\kappa(\pi_1, \pi_2) := \frac{\#\text{edges}(\pi_1 \cap \pi_2)}{\min(\#\text{edges}(\pi_1), \#\text{edges}(\pi_2))}$$

Depending on our preferences, we fix a parameter $K \in [0, 1]$ and consider paths π_1, π_2 sufficiently different, if $\kappa(\pi_1, \pi_2) \leq K$. For any facet f with corners $p_1, \ldots, p_d$ consider the s-t-paths $\pi_1, \ldots, \pi_d$ corresponding to the respective corners. We count the number of already computed paths which are not sufficiently different from $\pi_1, \ldots, \pi_d$ ($|\{\pi \in P | \exists i \in \{1, \ldots, d\} : \kappa(\pi, \pi_i) \geq K\}|$). Intuitively, we give priority to simplices for which this number is small.

Having introduced the prioritization, we are now ready to state our final algorithm. The algorithm is called with an initial convex hull H, the maximum number of refinement steps R and a limit for the overlap coefficient. The initial convex hull is obtained by adding the d paths which belong to the d preferences that have exactly one non-zero component, as well as the path which is obtained by setting $\alpha_i = \frac{1}{d}$. In each round it refines the most promising facets according to their priority, see Algorithm 2.1 for the pseudo-code. As more than one facet is refined per round, some of the facets could be destroyed by the insertion of a new path. Therefore we check facets for validity in line 8.

Algorithm 2.1 BoundedChExploration

Input: Convex Hull H, Refinement Step Limit R, overlap limit K
Output: Path Set P

```
1:  PriorityQueue Q
2:  for all Facet f ∈ H do
3:      prio ←count routes p ∈ P, f_p ∈ f with κ(p, f_p) > K
4:      Q.insert(f, prio)
5:  end for
6:  while not Q.empty() ∧ R ≥ 0 do
7:      Facet f ← Q.pop_min()
8:      if not H.isValid(f) then
9:          continue
10:     end if
11:     R ← R − 1
12:     find α which equalizes costs for paths in f
13:     p ← dijkstra(s, t, α)
14:     f_p ← a path in f
15:     if c(p)^T α ≤ c(f_p)^T α then
16:         H.insert(p)
17:         for all Facet f ∈ H.incidentFacets(p) do
18:             prio ← count routes p ∈ P, f_p ∈ f with κ(p, f_p) > K
19:             Q.insert(f, prio)
20:         end for
21:     end if
22: end while
23: for all Vertex v ∈ H do
24:     P.insert(v.path)
25: end for
```

2.3.3. Making use of the Parameter Exploration: Path Extraction

Once the parameter space exploration terminates, we have found a set of α values which hopefully represent the variety of different $s-t$-paths well. Our goal is to extract a large set from the α values encountered during the parameter exploration, such that the respective paths are all pairwise sufficiently different, i.e., have $\kappa(.,.) \leq K$. We do so by constructing a conflict graph as follows:

- every α value (or the respective optimal path) corresponds to a node
- nodes v and w have an edge between them if and only if for the respective paths π_v, π_w we have $\kappa(\pi_v, \pi_w) > K$

The largest set of sufficiently different paths now corresponds to a maximum independent set in

this conflict graph. For medium sized candidate sets we can employ integer linear programming for solving this subproblem optimally, for larger sets, a simple greedy approach which repeatedly picks the node with minimum degree from the conflict graph (removing all neighbors of the picked node) results in very good results.

2.4. Improving the LP Oracle for Personalized Contraction Hierarchies

The LP oracle, as described in Section 1.4.1, wants to find a preference α such that a path π^* is optimal. We insert one constraint for every other optimal path π' found so far. It will always output an α for which at least two of its constraints are tight. If the constraint for π' is tight the costs of π^* and π' under α are equal. In the contraction phase of PCH, we want to ascertain that there exists an α^* such that π^* is the only optimal path for α^*. Therefore, we would like the oracle's output to be such an α^*. To that end, we introduce a new variable δ which represents the maximum cost difference $c(\pi', \alpha) - c(\pi^*, \alpha)$ and an objective function which tries to maximize δ.

$$\begin{aligned} \max \quad & \delta \\ & c(\pi^*, \alpha) - c(\pi', \alpha) + \delta \leq 0 \end{aligned}$$

For our concrete application scenario, where we quite frequently encounter many optimal paths, this together with a slightly more involved search for dominating paths, allowed us to reduce the number of created shortcuts and the preprocessing time and to increase the speed-up by a factor of about two compared to the implementation in [FLS17].

2.5. Experimental Results for Route Enumeration

Our implementations are all in C++ using g++ (version 7.3.0). As LP solver for the PCH as well as for the independent set computation via ILP we used GLPK in version 4.65-1. We used the CGAL package for arbitrary dimensional triangulations to compute the convex hulls in version 4.11-2 [DHJ17; The17]. The code was executed on an intel i7-3770 running Ubuntu Linux 18.04 with 32GB of RAM. Our experiments are based on data from the OpenStreetMap project [18]. We have extracted a road network of the German state of Baden-Württemberg which contains all roads, paths, and tracks passable by a bicycle. This results in a graph with 9,711,550 nodes and 20,192,264 edges, see Figure 2.4. Note that this graph contains almost 3 times as many nodes and edges as the respective network of roads passable by car. Baden-Württemberg is very suitable as a test case for a bicycle route planner due to its heterogeneous landscape with mountain ranges, river valleys and flat planes.

2.5.1. Edge Weight Generation

The length of the edges is calculated by using the haversine formula on the node coordinates and rounded to the next meter. The *highway* and *bicycle* tags (or respective attributes) in the OpenStreetMap data are used to determine the unsuitability of the edges. Edges with an explicit *bicycle* tag having a value other than "no", do only incur half the usual unsuitability cost. The unsuitability cost of other edges is determined by the value of their *highway* tag which represents the road type of the edge; larger roads have bigger unsuitability costs. Each *highway* tag is assigned a

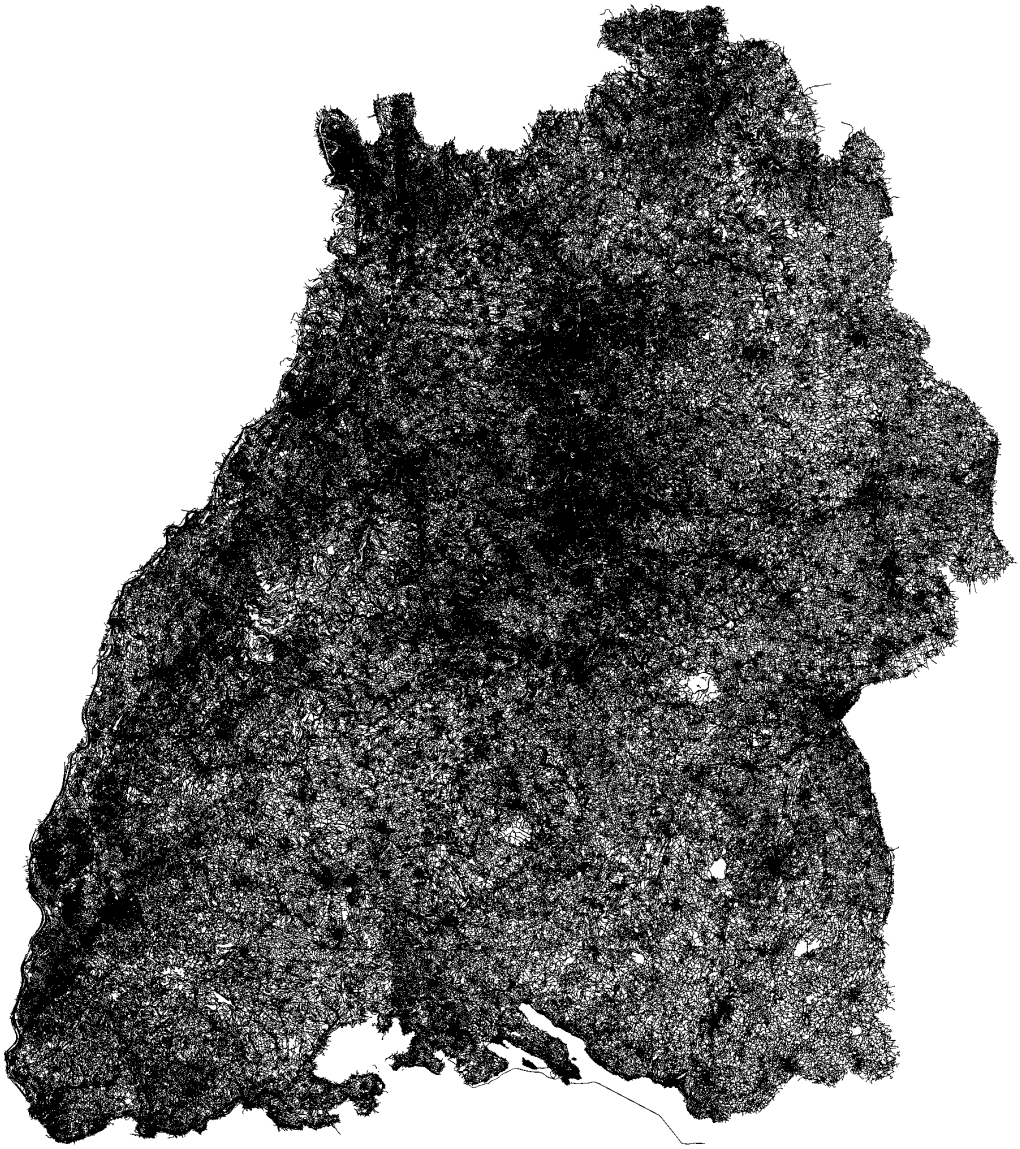


Figure 2.4.: Rendering of the considered network (9.7 million nodes, 20.2 million edges) of the German state of Baden-Württemberg (around 35,000 square kilometers).

factor which is then multiplied with the length of the road. This way of computing the costs takes into consideration that staying on a large road with the bicycle for a long distance is worse than only following it for a few meters. The complete mapping can be found in Table 2.2. The data from the Shuttle Radar Topography Mission (SRTM) is used to assign a height to each node and subsequently calculate the positive height difference traveled by traversing each edge [FRC+07], that is, only uphill edges bear some non-zero value.

2.5.2. Personalized Contraction Hierarchy Construction

We computed a PCH graph following the description in Section 1.4.1 with the improvements to the LP oracle stated in Section 2.4. However, we did not contract all nodes but left a so-called *core* of

Table 2.2.: Unsuitability costs assigned to different road types

Highway Tag	Unsuitability Cost Factor
cycleway	0.5
footway	0.75
path	0.75
pedestrian	0.75
platform	0.75
track	0.75
service	0.75
living_street	0.75
traffic_island	1.0
residential	1.0
unclassified	1.0
bridleway	1.25
road	1.25
tertiary_link	1.25
tertiary	1.25
secondary_link	1.5
secondary	1.5
primary_link	1.75
primary	1.75
other tags	2

uncontracted nodes. We only contracted 99.9% of all nodes and hence left a core of roughly 9,000 nodes. The contraction process took about four hours and resulted in a CH graph with 53,766,688 edges. This is an increase in the number of edges by almost a factor of about 2,7 compared to the original graph.

Averaged over 1,000 queries with a randomly selected source and target node as well as a random choice of α, the running time for a conventional Dijkstra is about 3.37 seconds in the original graph. Using Dijkstra on the CH-graph took 0.09 seconds on average, resulting in a speed-up of 36. The number of poll operations (number of node extractions from the priority queue used in Dijkstra's algorithm) decreased on average from 4,826,370 to 45,091 which is a reduction by factor 107. These numbers refer to our implementation of [FLS17] with the modifications mentioned in Section 2.4. Without these modifications, the speed-up was around 18 and preprocessing times as well as number of shortcuts was slightly higher.

Note that for road networks tested in [FLS17], the number of edges in the CH graph for personalized route planning with $d = 3$ metrics was only about twice the number of original edges and the speed-up for query answering was two orders of magnitude. There are several reasons why the performance is worse in our case: (1) Our input network is denser than a typical road network. In fact, we use a road network augmented with all segments that are suitable for cycling, which are not necessarily passable by car, hence we consider significantly more edges. (2) Our chosen metrics lead to a large diversification of optimal paths for different choices of α. This is a desired property for our application of computing useful sets of alternative routes. But for the CH computation, a large diversity of shortest paths also means that many shortcuts are necessary to represent them. In [FLS17], the three tested metrics were distance, travel time (for cars) and positive height difference. As distance and travel time are often closely correlated, these metrics usually induce rather similar shortest paths. In our scenario, where we replace travel time with unsuitability, the three metrics usually lead to very

different optimal path structures when considered each on their own. This explains the higher factor of inserted shortcut edges in our CH graph.

Next, we will discuss the experimental results for our complete pipeline for alternative cycle route computation.

2.5.3. Parameter Space Exploration and Route Recommendations

Let us now evaluate the performance of the actual parameter space exploration as well as the route recommendation. We will investigate three different scenarios in which we expect to search for route alternatives. For a typical *commuter ride*, we consider random source-target pairs that are within 10km beeline distance from the city center of a large metropolitan area (in our case Stuttgart) and are between 2km and 20 km apart from each other. But we also generate routes that mimic a common *day trip* by bicycle, by randomly picking source and target that are between 40km and 80km apart from each other. At last, we consider routes which are typically taken over several days by randomly picking source and target that are at least 120km apart from each other. We call these *vacation routes*.

2.5.3.1. Complete Enumeration

For the commuter and day trip routes a complete enumeration of all optimal routes for all possible values of α is feasible. For this experiment we modified the Algorithm 2.1 to omit the calculation of priorities as we will visit every facet anyway. We measured the number of routes as well as the running time. The result is displayed in Figure 2.5. For the rather short commuter routes, the number of routes on the relevant part of the convex hull ranges from four to 1,027. To compute these path sets the algorithm ran 8ms and 59.8s respectively. The average run for commuter routes was 10.1s long and yielded 255.2 routes. In comparison, the day trip route count ranges from 113 to 3,279 and took between 4.5s and 14.8min. On average exploring all paths for a day trip route took 3 minutes and resulted in 1,052 routes. The smoothed lines for the two data sets show that our method takes time roughly proportional to the number of routes it needs to enumerate. The different slopes of the lines are explained by the different run times per Dijkstra in the data sets.

From these numbers it becomes clear that in an interactive scenario a full exploration only makes sense for routes of moderate length. So in the following, our experiments will focus on bounded but guided exploration of the relevant part of the convex hull.

2.5.3.2. Bounded Exploration and Route Recommendation

For the bounded exploration, we consider the algorithm parameter *maximum number of refinement steps* $\mathcal{R}$ as well as the *maximum overlap coefficient* K from Section 2.3. We measure the time taken by the algorithm as well as the number of routes that are obtained. We do not report the time to compute the convex hull separately because it is dominated by the shortest path queries and only operates on a small point set.

The result of the parameter space exploration is a set of α values and respective optimal paths. Yet, this set is typically too large to be presented at once to a user, and many of the paths in this set exhibit a considerable pairwise overlap. So as described in Section 2.3.3, we extract a subset of paths, each pair of which does not exhibit an overlap coefficient of more than K. We compute the largest such set via an integer linear programming formulation and employing the ILP solver of

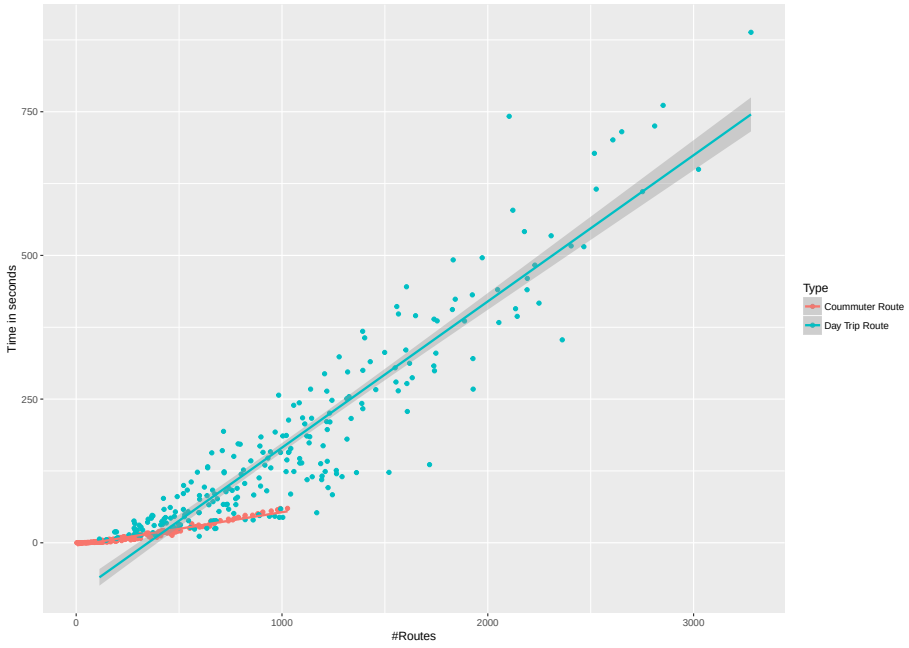


Figure 2.5.: Scatter plot showing the time needed to enumerate all optimal routes depending on the number of routes for commuter and day trip routes.

glpk [And12]. Additionally, we employ the greedy independent set algorithm. The running time of the greedy algorithm was not included into the tables below, as it always finished in less than 1ms.

In Table 2.3 we see the results for the commuter routes. For example, the line with $\mathcal{R} = 12$, $K = 0.5$ states that with 12 refinement steps, our parameter space exploration takes around 108ms and produces 12 different α values and respective paths on average. Determining a subset of paths which have pairwise overlap coefficient of less than 0.5 via our ILP formulation takes fewer than a millisecond resulting in 5.3 route alternatives on average, so considerably more than just the three paths one would get by simply optimizing for each of the three weights separately (often these paths also partly coincide, in particular for short distance queries).

In Table 2.4 we see the results for the day trip routes. Running times for parameter space exploration are higher due to the more expensive CH-Dijkstra computations. We get a considerable increase in the number of paths after pruning, e.g., for $\mathcal{R} = 48, K = 0.8$ we obtain on average 22.8 routes vs. 16.7 in the commuter scenario. This comes as no surprise since the longer the distance between source and target, the more options there are.

Finally, in Table 2.5 we see the results for the holiday routes. Running times have increased even further as have the number of routes after pruning.

Common to all three scenarios, the number of (unpruned) values for α (or the respective paths) is directly related to the number of allowed refinement steps. If many refinement steps are allowed, the running time naturally increases due to the increased number of shortest path computations. Apart from the largest value for the number of allowed refinement steps, the time to prune the α values via ILP to obtain diverse paths with not too much overlap is negligible and most of the time below one millisecond. Large values of $\mathcal{R}$ also increase the time for the parameter space exploration to a degree that is not acceptable for interactive use, in particular for the non-commuter-route scenarios. In the 9,000 runs of this experiment, there were 12 instances for which the ILP run time exceeded

Table 2.3.: *Commuter Route (2km-20km):* Statistics for parameter space exploration and route recommendation; averages over 3,750 runs.

$\mathcal{R}$	K	explore time (ms)	result size	ILP time (ms)	ILP size	Greedy size
3	0.5	33.1	4	0	3.4	3.4
	0.8	33.2	4	0	3.8	3.8
	0.9	33.2	4	0	3.8	3.8
6	0.5	51.0	6	0	4.3	4.3
	0.8	51.0	6	0	5.3	5.3
	0.9	51.1	6	0	5.5	5.5
12	0.5	108.0	12	0	5.3	5.3
	0.8	108.1	12	0	8.9	8.9
	0.9	108.5	12	0	9.9	9.9
24	0.5	229.1	23.8	2	6.2	6.1
	0.8	228.8	23.8	0.1	12.8	12.8
	0.9	231.4	23.8	0	16.6	16.6
48	0.5	505.4	47.4	175	6.9	6.7
	0.8	507.4	47.4	10.2	16.7	16.6
	0.9	503.5	47.4	0.8	24.6	24.6

Table 2.4.: *Day Trip Route (40km-80km):* Statistics for parameter space exploration and route recommendation; averages over 3,750 runs.

$\mathcal{R}$	K	explore time (ms)	result size	ILP time (ms)	ILP size	Greedy size
3	0.5	179.2	4	0	3.6	3.6
	0.8	179.6	4	0	3.9	3.9
	0.9	180.0	4	0	4.0	4.0
6	0.5	273.5	6	0	4.7	4.7
	0.8	273.5	6	0	5.5	5.5
	0.9	273.7	6	0	5.7	5.7
12	0.5	567.8	12	0.1	6.6	6.5
	0.8	574.4	12	0.1	10.1	10.1
	0.9	569.1	12	0	11.0	11.0
24	0.5	1192.0	24	2.1	7.9	7.8
	0.8	1188.5	24	0.3	16.2	16.2
	0.9	1190.4	24	0.2	19.8	19.8
48	0.5	2590.1	48	311.5	9.2	8.9
	0.8	2579.3	48	10.4	22.8	22.7
	0.9	2595.8	48	1.0	32.7	32.6

Table 2.5.: *Vacation Route (> 120km)*: Statistics for parameter space exploration and route recommendation; averages over 1,500 runs.

$\mathcal{R}$	K	explore time (ms)	result size	ILP time (ms)	ILP size	Greedy size
3	0.5	507.7	4	0	3.7	3.7
	0.8	508.9	4	0	4.0	4.0
	0.9	508.8	4	0	4.0	4.0
6	0.5	777.8	6	0	5.0	5.0
	0.8	776.9	6	0	5.7	5.7
	0.9	776.0	6	0	5.8	5.8
12	0.5	1599.0	12	0.2	7.9	7.9
	0.8	1604.2	12	0.1	10.6	10.6
	0.9	1602.3	12	0.1	11.2	11.2
24	0.5	3365.4	24	1.1	10.3	10.2
	0.8	3344.8	24	0.1	19.0	19.0
	0.9	3363.1	24	0.3	21.1	21.1
48	0.5	7321.2	48	263.0	12.4	12.0
	0.8	7315.6	48	4.1	29.9	29.8
	0.9	7297.3	48	2.5	38.3	38.3

the exploration time. All of these were started with the parameter combination $R = 48, K = 0.5$. This suggest that for larger R and lower K values the greedy algorithm might be a better fit for interactive use. Therefore, we will compare the results of the greedy algorithm to the ILP variant in the next section.

2.5.3.3. Comparison of Independent Set Algorithms

For most parameter combinations, the average result of the ILP and the greedy algorithm are identical. The biggest difference can be found in Table 2.5 for $R = 48, K = 0.5$ where the average differs by 0.4. Figure 2.6 illustrates the data in more detail. It contains a scatter plot showing the independent set size returned by the ILP on the x-axis and the difference between the ILP set size and the greedy set size on the y-axis. Combinations that occur more often are displayed as bigger dots. The blue line is a smoothed trend line. The graph shows that the greedy algorithm produced sets that contained up to three routes less than the ILP on two occasions. Which in one case meant a reduction in set size of more than 35% from eight to five. However, in 97% of the cases the results were identical, which is emphasized by the trend line.

Therefore, the greedy algorithm is a good fit for interactive use. It is always on par or faster than the ILP solver and produces equal results most of the time.

2.5.3.4. Comparison to naive Parameter Space Exploration

We now compare the results of our approach to a random and a naive exploration of the parameter space for the same s–t combinations as before. For the random exploration, we ran R Dijkstras with random configurations to get roughly the same number of routes. The naive exploration was conducted by fixing an $\epsilon < 1$ as discretization parameter. Then all choices of $\alpha_i = k \cdot \epsilon$ with $k = 0, 1, 2, \ldots, \lfloor \frac{1}{\epsilon} \rfloor$ and $i = 1, 2$ were explored. α_3 was uniquely determined by α_1 and α_2. We chose epsilon values to

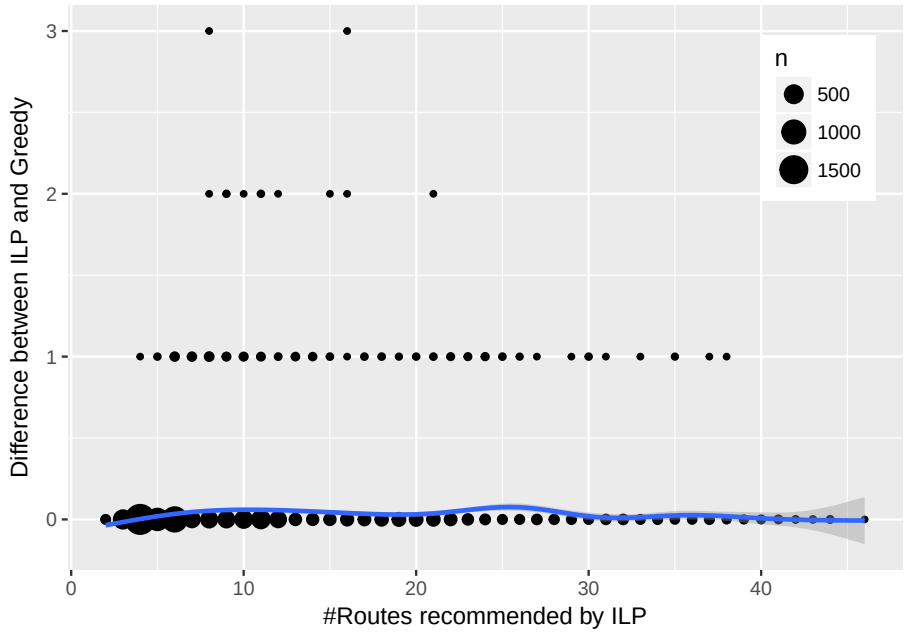


Figure 2.6.: Scatter plot showing the difference in ILP and Greedy result set size over the ILP set size.

reflect the number of Dijkstra runs our approach used for every R. Afterwards, we applied the same route recommendation procedure described in Section 2.3.3 and compared the results. Figure 2.7 displays a smoothed line of the results with confidence areas in grey and one graph per K value. In every case the number of selected routes is significantly higher for our approach.

2.5.4. Exemplary Queries

Let us illustrate the produced results for two exemplary queries.

Commuter Route Here we consider a route from the village of Scharnhausen (305m above sea level) to Vaihingen (438m) in the Stuttgart metropolitan area, see Figure 2.8. Essentially there are three different route categories. The first one (top, in red) is via the Degerloch and the Stuttgart TV tower (483m); this is characterized by a steady climb to the highest point followed by a mild descent; this route is found when optimizing for road unsuitability only. A considerable part of this path uses tracks blocked for cars and hence are favoured for bicycle routes. The second type of route follows right of the small river Körsch via Hohenheim. It never reaches an altitude comparable to the red route. Our algorithm finds several routes in that category (in varying shades of blue and purple) when favouring distance more than the other two criteria. Lastly the third category of routes is to the left of the river Körsch mostly staying on the Fildern plateau; these three paths were found by our algorithm when avoiding climbs. Our algorithm performed 15 refinements; from the resulting paths, the integer linear program selected 14 paths which have pairwise overlap coefficients of at most 42%. The whole computation took about 200ms, the only parameters provided were 15 (number of refinements) and 42% (degree of disjointness of the recommended paths).

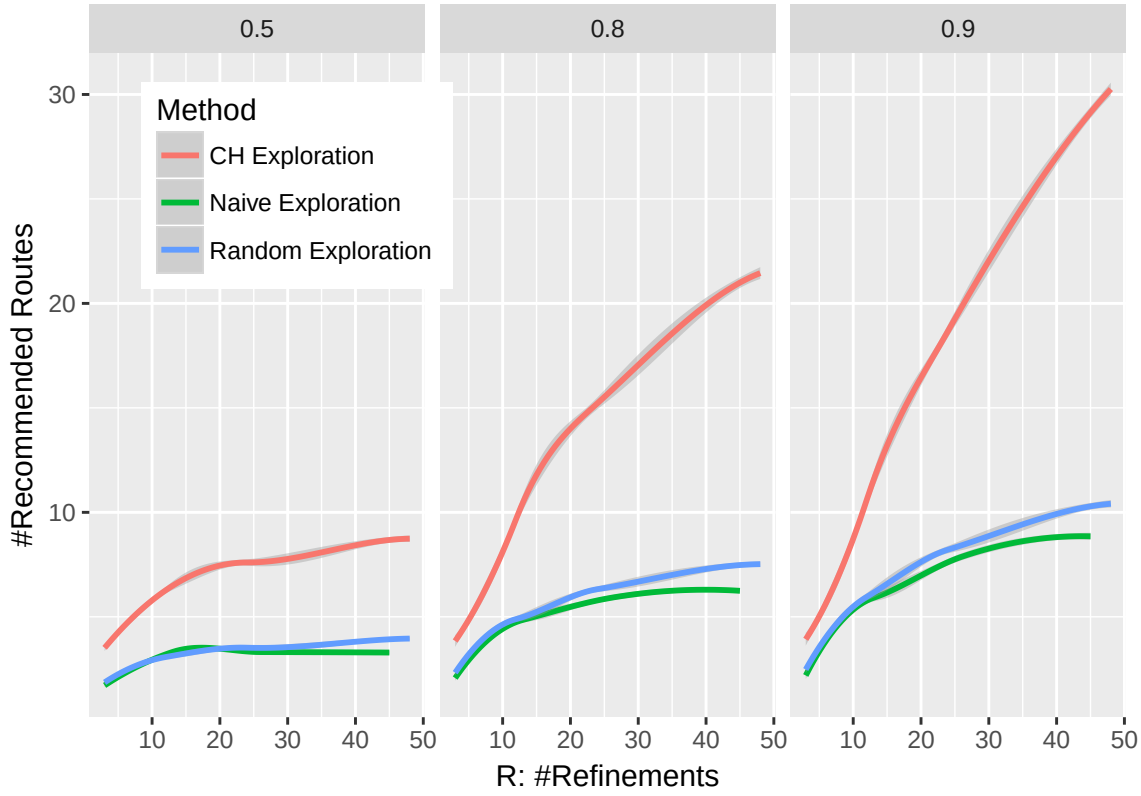


Figure 2.7.: Comparison of random sampling and naive exploration with our approach for different R and K values.

Vacation Route For this scenario we consider a tour across the state of Baden-Württemberg from the city of Ulm near the border to Bavaria in the east to the city of Offenburg at the French border in the west, see Figure 2.9. The direct routes with small distance values lead directly through the mountain range of the Schwäbische Alb and the Black Forest. The routes with small positive height difference try to minimize the intersection with the Schwäbische Alb and avoid the Black forest by passing north of it via Stuttgart and Karlsruhe and then going south in the valley of the river Rhein. The southernmost route (in red) is optimal for road unsuitability, ignoring the other metrics. The computation took less than three seconds with parameter choices $\mathcal{R} = 15$ and $K = 0.46$.

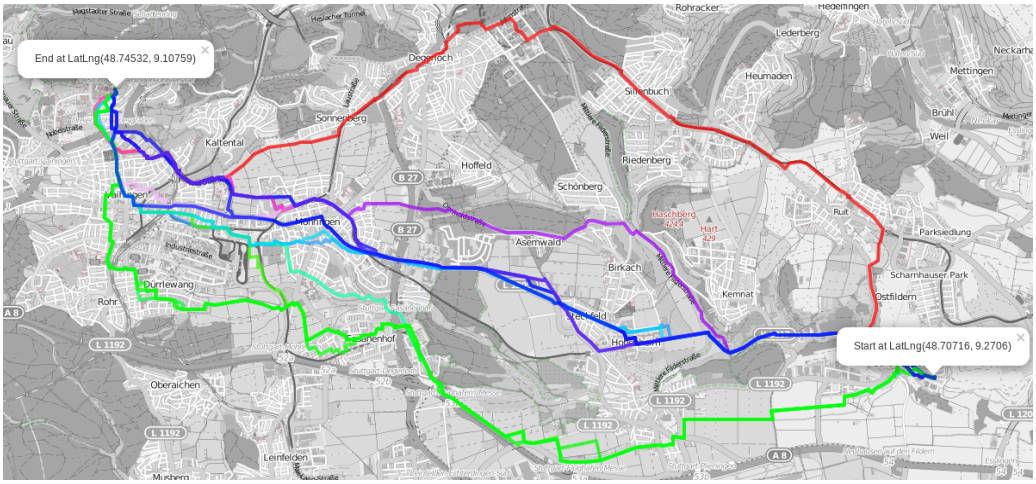


Figure 2.8.: Urban Commute Routes in the Stuttgart Metropolitan area from Scharnhausen to Vaihingen.

Figure 2.9.: Vacation routes across the state of Baden-Württemberg from Ulm to Offenburg.

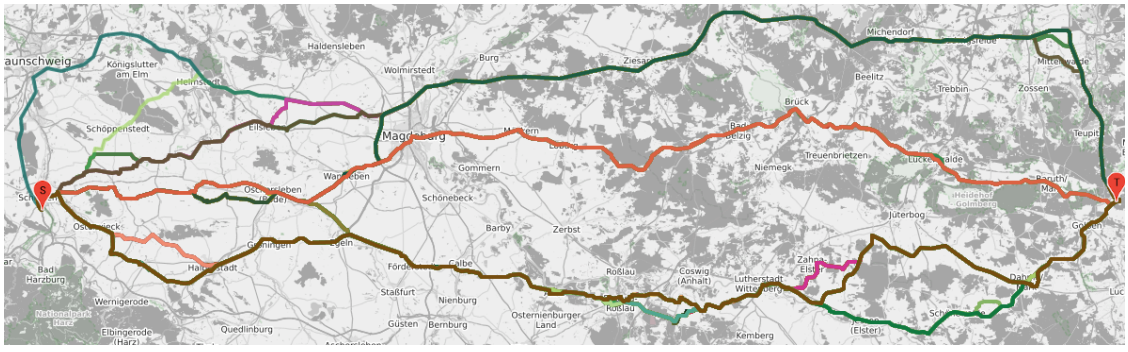


Figure 2.10.: Example for large set of alternative routes all within 30 % of the shortest route.

2.6. Restricting Enumeration and Metric Invention for Diversification

Inspired by [DDP19] from the public transit routing space, we show how to compute a *large* set of *reasonable* routes. In this work, the authors suggested that not any Pareto-optimal journey is desirable for users and focused on those journeys which are good in the travel time and number of trips metrics. The goal is that the set of routes should be diverse but at the same time, all routes inside the set should not be too far from the selfish optimum of reducing travel time. The former part of the goal should enable to distribute traffic in a fashion that reduces congestion. The latter part ensures that each individual vehicle is not "punished" too much when following one of the routes in the set. Our approach is based on restricted enumeration and metric invention.

2.6.1. Restricted Enumeration of Personalized Routes

Depending on the metrics used, the enumeration algorithm described in Section 2.3.1 often yields hundreds of routes for queries in large road networks with three dimensions or more. Furthermore, it opens up the possibility for using less conventional metrics like the number of traffic lights or left turns on the path. On their own, these metrics would be of limited use. While these new metrics lead to good routes that would previously not be found, the algorithm also finds all the routes which mainly focus on these metrics. This in turn might lead to rather strange routes that increase travel time or distance too much. This section explores how to prevent computing such "bad" routes. Therefore, only the *reasonable* routes remain in the resulting set.

2.6.1.1. Definitions

We formalize the notion of trade offs that are undesirable for a user by defining certain metrics as *important*. The *slack* for an important metric defines how much a route may exceed the cost of the cheapest route in that metric. This maximum excess may be specified as absolute or relative value.

Consider an example with $d = 2$ where the metrics are distance and travel time. Let there be four personalized routes between some s and t with costs (10km, 30min), (13km, 25min), (20km, 15min), (25km, 10min). If the user considers the distance of the trip important and assigns it a slack of $h_1 = 1.5$ then only routes with distance costs $\leq 10\text{km} * 1.5 = 15\text{km}$ conform to the users preferences. In this example, the routes with cost (10km, 30min) and (13km, 25min) are interesting to the user. We call these *admissible routes*. *The Restricted Personalized Routes Problem* is now defined as follows:

Given a street network $G(V, E)$, with a d-dimensional non-negative cost vector $c(e) \in \mathbb{R}^d$ for each edge $e \in E$; a query consists of source and target $s, t \in V$ and a set of important metrics $M = \{m_1, \ldots, m_j\}$ and their slacks $H = \{h_{m_1}, \ldots, h_{m_j}\}$ which represent how much of a detour is acceptable to the user. The goal is to compute all routes $\pi \in R$ from s to t in G which minimize $\sum_{e \in \pi} \alpha^T c(e)$ for some α and fulfill $\forall m_i \in M : c_{\text{opt},m_i} * h_{m_i} \geq c(\pi)_{m_i}$. Where c_{opt,m_i} is the minimum cost in metric m_i over all routes in R.

2.6.1.2. Restricting Enumeration

The performance goal for the modification of the algorithm is to prevent Dijkstra runs that yield inadmissible routes as much as possible. Given only the input preference α, it is not possible to predict

the admissibility of the route. Even an α which assigns a factor of zero to all important metrics might lead to an admissible route.

By the properties of personalized routes, we know that in every iteration of the algorithm all routes in R are extreme points of the partial convex hull as well as the final convex hull in cost space. With this information, we can compute a lower bound on the costs of the cheapest route that may be found by refining a facet f. Figure 2.11 depicts an example convex hull and the areas and points used by the algorithm to determine if a facet needs to be refined. The area A which contains all personalized routes that may be obtained by refining f, is defined by the intersection of the hyperplanes induced by the facets adjacent to f. To determine if A intersects with the area B that contains all admissible routes, we construct a lower bound coordinate p_{low}. It consists of the cheapest costs of the routes defining f:

$$p_{\text{low},i} = \min p_i : \forall p \in f$$

Clearly, p_{low} dominates all $p \in f$. Therefore, if $p_{\text{low}} \notin B$, no route that can be obtained from f can be admissible. The slack condition formulated in the previous section is used to determine if p_{low} lies in B. In this example the point p_{low} (black dot) lies inside the area B (beneath the black, dashed line) and the facet needs to be refined.

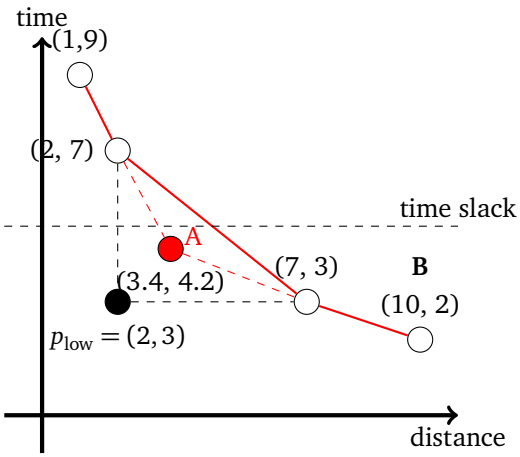


Figure 2.11.: Partial convex hull in cost space; Area A contains all potential routes which can be found by refining the central facet (red, dashed lines); Area **B** contains all admissible routes (beneath slack constraint); Lower bound p_{low} is used for determining if a facet may not yield admissible routes (black dot).

2.6.2. Metric Invention

An important factor for finding a large and diverse set of alternative routes with the above approach are the underlying metrics used in the graph. If the used metrics are very similar e.g., travel time for cars and trucks, which only differ on roads with high speed limit, only a small number of alternative routes can be found. Data for other natural metrics than distance and travel time are not always available in high quality. With the restriction on natural and important metrics in place, 'invented' metrics might be sufficient to diversify the result without producing bad routes. In this section, we describe two metrics we 'invented'.

2.6.2.1. Random Weights

An obvious way to avoid correlation with any given metric is to use random values from a uniform distribution. For this metric, we assign a cost value between 1 and 20 to every edge.

2.6.2.2. Chessboard Metric

For our second 'invented' metric, we overlay our graph with a rectangular grid. We consider the cells in the grid colored as if they were a chessboard. See Figure 2.12 for an illustration. For each edge that starts inside a white cell we assign a "high" cost of 20. Edges that start in black cells are assigned a "low" cost of 1. Assigning the costs like this should have little to no effects while moving inside a cell, as uniform edge costs are similar to the distance metric calculated from OpenStreetMap (OSM) data. On the other hand, traveling between different cells induces cost depending on the starting cell and the moving direction.

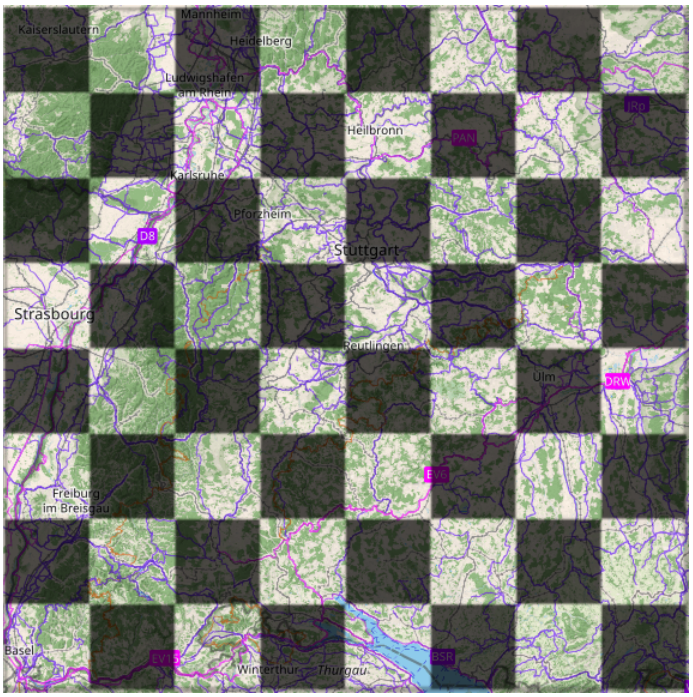


Figure 2.12.: Visualization of the chessboard metric grid overlaying a road network.

2.7. Experimental Results for constrained Enumeration and Metric Invention

Our implementations are all in C++ and were compiled using clang++ (version 7.0.0). We used the CGAL package for arbitrary dimensional triangulations to compute the convex hulls in version 4.11-2 [DHJ17; The17]. For solving of linear equations the version 3.3.4 of the Eigen3 library was used [GJ+10]. The code was executed on an AMD Ryzen Threadripper 1950X 16-Core Processor with 32 threads running Ubuntu Linux 18.04 with 128GB of RAM. In contrast to the description in Section 2.6.1.2, the facet refinement is executed in parallel in our implementation. Our experiments

are based on data from the OSM project [18] and SRTM [FRC+07]. We have extracted the road network of Germany as a graph with 26,084,719 nodes and 54,666,385 edges.

2.7.1. Used Metrics

For the experiments, we used five types of metrics. We view distance and travel time for cars as essential for routing car traffic and combined these two metrics with each one of the other metrics.

Distance The length of the edges is calculated by using the haversine formula on the node coordinates.

Travel Time for Cars and Trucks The speed limit per edge is determined by the OSM tag *maxspeed* or inferred by the *highway* tag. If the speed limit exceeds 120km/h for cars (80km/h for trucks), we cap the speed limit. The result is used for the travel time calculation together with the distance metric.

Height Ascent The data from SRTM is used to assign a height to each node and subsequently calculate the positive height difference traveled by traversing each edge, that is, only uphill edges bear some non-zero value. While height ascent may not be a very important criterion for car routing today, it is relevant for electric vehicles, which may charge their batteries when driving downwards.

Random Weights As described in Section 2.6.2.1.

Chessboard The chessboard metric from Section 2.6.2.2 was used with a 20×20 and a 200×200 grid.

2.7.2. Personalized Contraction Hierarchy Preprocessing

To improve Dijkstra query times we applied the PCH scheme described in [FLS17] and refined in Section 2.4. For every metric combination, the graph was contracted to 99.9% leaving an uncontracted core of about 24,500 nodes. In Table 2.6 the contraction times and speed-up for every metric combination is listed. The speed-up and preprocessing time are both linked to the "similarity" of the used metrics. The higher the similarity the faster the preprocessing can be done and the higher the speed-up will be. The speed-up is determined by running 200 random Dijkstra queries with and without the additional shortcuts in the graph. Processing time ranges from 13min to 146min while the speed-up goes from 93 to 245.

2.7.3. Experiment Design

We evaluate our implementation by using 100 randomly chosen s-t pairs with a distance of at least 100km. For each of these pairs, we compute all personalized routes, as well as the restricted route set for a slack of 1.1, 1.15 and 1.2 for distance and car travel time separately. We measure the execution time, the size of the route set and the average load per edge defined as $l(e) = \frac{\#\text{routes that contain e}}{\#\text{routes}}$.

Table 2.6.: Preprocessing time and average speed-up for Dijkstra queries for each of the metric combinations.

Additional Metric	Preprocessing	Speed-Up
-	14 min	245
Truck Travel Time	13 min	227
Height	29 min	121
Chess 20×20	146 min	110
Chess 200×200	61 min	93
Random	59 min	96

2.7.4. Results

Table 2.7 contains the average measurements for the distance restricted queries grouped by the additional metric and used slack value. The speed-up and the share of admissible routes are measured in comparison to a full enumeration of all personalized routes in the same query, i.e., for a given st-pair we perform both a full and a restricted enumeration and compare the query times and the result sets to obtain these values.

$$\text{Speed-up: } \frac{\text{Full Query Time}}{\text{Restricted Query Time}}$$

$$\text{Share of admissible routes: } \frac{|\text{Full Query result set}|}{|\text{Restricted Query Result set}|}$$

When no additional metric is used, the queries returned 28 or 29 routes on average and took about 0.3 seconds. Most of the personalized routes ($\geq 97\%$) for this metric combination did not exceed even the smallest slack condition used in the experiments. Therefore, there is no speed-up in comparison to the full enumeration with these metrics. The average edge load falls in the range 43% to 45%.

Adding the truck travel time metric more than doubles the output of the algorithm into the range 62 to 63, but at the same time nearly quadruples the running time to 1.19s. The number of excluded routes and the speed-up stay negligible with this addition.

The two chessboard metrics increase the number of personalized routes significantly. Depending on the used slack, the output set grew up to 362 routes for large cells (20×20) and 1062 for small cells (200×200). The running time rises respectively up to levels of ten to sixteen seconds (20×20) and 48s to 51s (200×200). Note that the portion of admissible routes sank for the large cell Chessboard to $\approx 25\%$ for a slack of 1.1 leading to a speed-up of five compared to a full enumeration. On the other hand, the small cell variant leads to a considerably smaller average edge load of $\approx 31\%$ or less.

In comparison to the small cell chessboard metrics, using random edge weights and a distance restriction leads to results that are relatively similar but a little lower in #routes and time but higher in edge load.

The largest results are achieved by using the height ascent metric. It produced up to 1416 routes on average and most of the time had less running time than the chessboard metric. It also produced the smallest edge load of all metrics ($\approx$ 22%-26%)

The second set of experiments which use a slack for the travel time are summarized in Table 2.8. For the German road network, travel time seems to be a more limiting restriction. For all metrics the share of admissible routes decreased and a speed-up compared to full enumeration was achieved. The flip side of this is that the result size is reduced substantially ranging from approximately ten to 685.

The best average load for the travel time restricted queries was achieved for the 20×20 chessboard metric with a slack of 1.2. This setting produced about 314 routes on average and each edge is only used by approximately 30% of these. The greatest speed-up of ≈ 13.6 was accomplished by using a slack of 1.1 with the height ascent metric. Which is directly related to the small share of admissible routes of ≈ 11.3%

Table 2.7.: Average results of the distance restricted queries

Added Metric	Distance Slack	#Routes	Time (s)	Speed-Up	Admissible Routes (%)	Load (%)
—	1.1	28.05	0.36	1.00	97.03	45.09
	1.15	28.78	0.37	1.00	99.34	43.38
	1.2	28.86	0.36	1.00	99.66	43.06
Travel Time for Truck	1.1	61.93	1.19	1.04	94.65	40.94
	1.15	63.14	1.16	1.00	97.20	39.81
	1.2	63.69	1.19	1.00	98.77	39.21
Chessboard 20×20	1.1	220.58	10.36	5.13	25.19	50.87
	1.15	293.56	13.54	3.74	32.70	44.77
	1.2	362.78	16.41	3.00	38.95	41.20
Chessboard 200×200	1.1	976.46	48.21	1.01	91.42	31.02
	1.15	1040.43	51.09	1.00	97.13	28.63
	1.2	1062.17	51.84	1.00	98.99	27.75
Random	1.1	943.93	45.82	1.02	90.26	31.80
	1.15	1010.74	49.01	1.00	96.21	29.23
	1.2	1029.49	49.76	1.00	97.94	28.15
Height Ascent	1.1	1111.39	41.59	1.56	70.65	26.32
	1.15	1308.09	48.96	1.23	82.56	23.69
	1.2	1416.61	52.93	1.14	89.08	22.66

2.7.5. Load Distribution

While the addition of the “invented” metrics effectively reduced the average load per edge, it is unclear whether this is sufficient to reduce congestion. If there are still few edges that are part of most routes these might easily become congested. For evaluating the potential of reducing such bottleneck edges in the road network by using alternative routes, we examine the distribution of routes onto edges via histograms. Figure 2.13 shows the histograms for the query shown in Figure 2.10. On the x-axis, the number of routes that contain an individual edge are categorized into bins. The y-axis shows how many edges fall into each bin. Edges that do not belong to any route are excluded. Figure 2.13 contains one chart for the case where no metrics were added as well as one for truck travel time, the chessboard metric (200×200) and the height ascent metric. The different configurations led to 12 (No additional metrics), 19 (Truck Travel Time), 283 (Chessboard 200×200) and 537 (Height Ascent) routes. When only distance and car travel time are used, most edges belong to seven of the twelve routes and over 100 edges are used by eleven routes or more. By adding the Truck Travel Time, the distribution shifts towards lower numbers. Especially, no edge is present in all routes anymore. Also, most edges are only shared by four or five routes.

As the number of routes increases immensely, the histograms get less detailed for the other two

Table 2.8.: Average results of the time restricted queries

Added Metric	Time Slack	#Routes	Time (s)	Speed-Up	Admissible Routes (%)	Load (%)
—	1.1	10.54	0.08	1.32	39.94	62.40
	1.15	13.83	0.10	1.23	51.11	56.18
	1.2	16.94	0.14	1.11	61.51	50.93
Travel Time for Truck	1.1	30.21	0.59	1.60	60.88	58.44
	1.15	41.18	0.76	1.26	76.12	53.69
	1.2	48.22	0.92	1.17	82.35	50.08
Chessboard 20×20	1.1	102.05	4.86	10.74	15.35	42.16
	1.15	188.98	8.61	5.85	25.23	35.00
	1.2	314.36	14.11	3.67	37.47	30.02
Chessboard 200×200	1.1	352.44	19.07	2.85	37.76	42.17
	1.15	535.69	27.90	2.00	54.22	36.03
	1.2	684.72	35.06	1.54	66.79	32.55
Random	1.1	333.41	18.01	2.92	37.79	42.34
	1.15	505.22	26.39	2.04	52.95	36.39
	1.2	652.56	33.20	1.58	65.41	32.41
Height Ascent	1.1	121.32	5.01	13.56	11.27	44.13
	1.15	211.52	8.13	8.91	19.04	37.61
	1.2	337.68	12.94	5.68	27.31	33.14

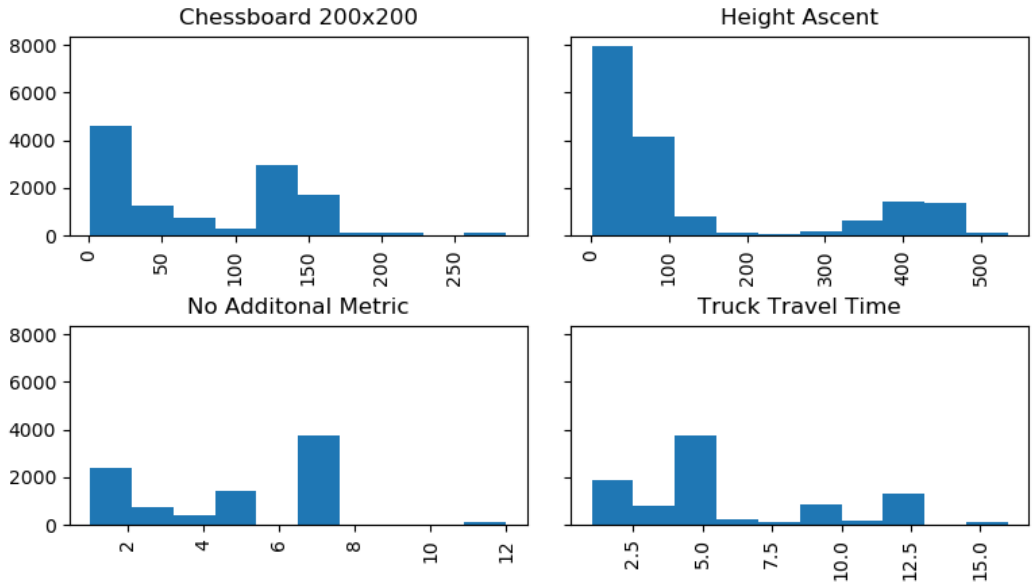


Figure 2.13.: Histograms showing the distribution of routes onto edges for different added metrics for the query of Figure 2.10.

metrics. Both metrics have in common that most edges fall into the first bin of the histogram, which includes more routes than were found in the first two metrics. For the chessboard metric, the first bin contains edges that are shared by up to 29 routes while for the height ascent metric it contains edges contained by up to 54 routes. There is a significant number of edges that are included by 115 to 171 routes of the chess board metrics which means half the routes use these nearly 4,700 edges. Furthermore, eleven edges are used by all routes. The Height Ascent metric does not have edges that are used by all routes, but about 3,000 of the 16,843 edges are used by more than 375 of the 537 routes.

2.7.6. Exemplary Queries

In this Section, two exemplary queries are described that were computed by using the distance and car travel time metrics only. To better follow the next paragraphs it is important to know that the German highways (autobahn) are identified by an 'A' and a number.

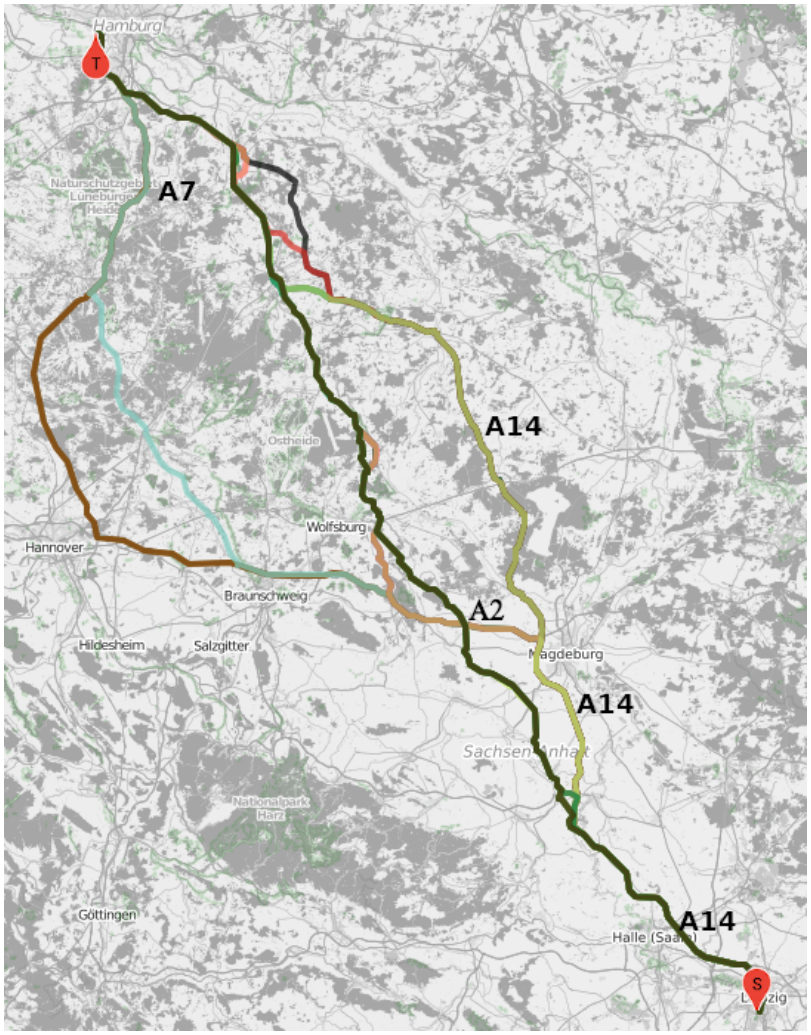


Figure 2.14.: Example query with a time slack of 1.2.

Figure 2.14 depicts all personalized routes from Leipzig to Hamburg with a time slack of 1.2. All routes begin on highway A14 (dark green, later light green), but split up after a few kilometers to

follow a state road which takes a more direct way (dark green). The dark brown routes takes the A2 over Hannover and merges on A7 with the cyan route. As the time is constrained in this example, the computed routes focus on highways or state roads which are considerably shorter than their faster alternative.

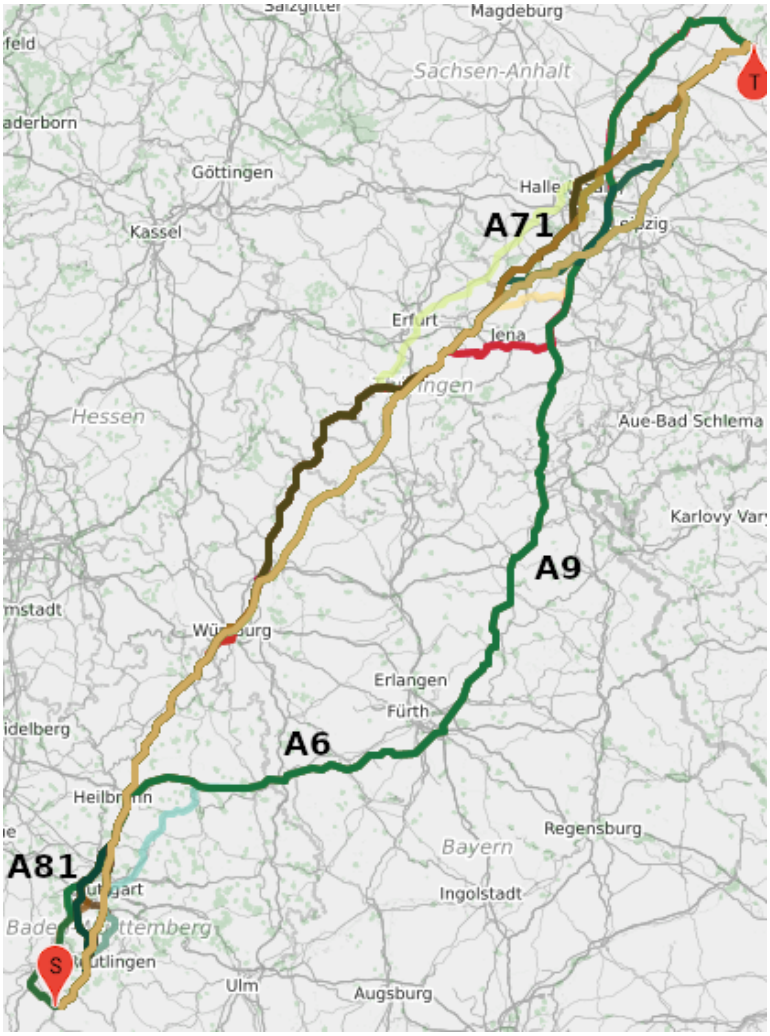


Figure 2.15.: Example query with a distance slack of 1.2.

The example query displayed in Figure 2.15 has a distance slack of 1.2 and connects a small city in Baden-Württemberg to a small city in Brandenburg. As we have seen in Section 2.7.4, distance restricted queries tend to have more diverse outputs. In the beginning, the dark green route moves directly onto highway A81, while the others use smaller streets until they find their way onto the same highway. After a short period, the dark green route changes to A6 and then to A9 while the light brown one stays on A81. For a longer stretch, all routes more or less follow these two highways until dark brown branches off onto A71. Closer to the destination, the routes begin to diverge and use a variety of state and country roads.

Those two examples show that the classical car routing metrics naturally encourage the use of the highways that can take the most load anyway. On the other hand, the routes emitted by the algorithm also use a lot of different smaller streets. The resulting routes are not necessarily interesting for a

driver who just wants to reach his destination fast but might be a step to reduce congestion if used by self-driving vehicles.

2.8. Conclusions

In this chapter, we have presented a novel way of computing alternative routes in case more than one set of edge weights is available in the road network. Our approach is based on a guided, partial exploration of the parameter space. With a standard Dijkstra on the multi-weighted graph, this exploration would be far too time-consuming, so we had to adopt a recently developed speed-up scheme for multi-criteria shortest paths [FLS17]. For the concrete example of bicycle route planning with alternatives, this allowed us to answer queries within few seconds, coming up with several reasonable alternative routes.

Existing approaches for generating alternative routes based on one metric only essentially generate suboptimal routes (according to that metric) avoiding paths with too much overlap. Our approach is fundamentally different in that respect as it always produces routes that are optimal for a convex combination of the metrics considered. Similar to the existing approaches, a filtering of routes with too much pairwise overlap takes place.

For our concrete example of route planning for bicycles, the two other edge weights apart from the distance – namely positive height difference and road unsuitability for cycling – were in fact very obvious choices. It remains to explore in future work whether in case of route planning for cars the consideration of additional edge weights like energy consumption or non-scenicness of the road lead to similarly good alternative routes.

We also expanded our algorithm to be able to only compute paths which do not diverge from important metrics such as distance or travel time too much and experimentally evaluated this in a route planning scenario for cars. Given an appropriate slack value, our algorithm efficiently skips inadmissible routes and achieves a considerable speed-up in contrast to a full enumeration of personalized routes. Especially, for the time restricted queries the average speed-up exceeded 10 in two experiment configurations. If none or only few routes are excluded, the overhead is small enough to not slow down the algorithm noticeably.

Using “invented” and untypical metrics for car routing did diversify the results with respect to the number of admissible routes as well as the average edge load. While the height ascent metric might work especially well for the German street network, the chessboard and random metrics should provide improvements that are independent of the underlying geography. The average load per edge did also decrease as was expected. For the time restricted case, the average load could be reduced from 51% to 30% for a slack of 1.2 by adding the chessboard 20×20 metric. A more detailed inspection of the edge load distribution, showed that most of the edge load did get spread to more edges. Still, there is a small number of edges that are used by most routes, which might lead to congestion in these areas. To determine the potential for congestion reduction the described algorithm and metrics should be used in a traffic simulation and compared to selfish routing. The idea would be to assign one of the restricted routes at random to each simulated car and compare total and individual travel times of all cars.

Computing so many routes has a major impact on running time, which should be addressed in future work. Of course, one obvious way to improve running time is to use less slack and reducing the number of routes. But this probably affects the average load negatively. A simulation as described

above can be used to empirically search for efficient and effective parameters. Another way could be to use the current average load to determine whether more routes need to be computed or not.

Part II.

Understanding real world trajectories

Chapter 3

Identifying Intermediate Destinations in Real World Trajectories

3.1. Introduction

This chapter is based on joint work with Stefan Funke, Tobias Skovgaard Jepsen and Claudius Proissl. It was published in the proceedings of the 9th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data [BFJP20]. My contribution to this work was focused on the trajectory segmentation and the generation of routing cost types. Part of this work which proposes an approach for identifying robust driving preferences is not included in this dissertation as I contributed little to it.

The ubiquity of mobile devices with position tracking capabilities via GPS or localization using WiFi and mobile networks continuously generate vast streams of location data. Such data may be used in a variety of ways. Mobile networks providers and many companies, such as Google or Apple, use the location data of their customers to improve their services, e.g., by monitoring of traffic flow or detection of special events. Location data sharing platforms such as Strava, GPSies, and OSM allow their users to share their location data with their community. In all of these cases, location measurements are considered collectively as sequences, each reflecting the movement of a person or a vehicle. Such sequences can be map-matched to paths in an underlying transportation network—in our case a *road network*—using appropriate methods [Zhe15]. We refer to such map-matched sequences as *trajectories* throughout the chapter.

A common assumption is that most of the time, users travel on 'optimal' routes towards a (possibly intermediate) destination, where optimality is understood as the shortest path w.r.t. suitable scalar *traversal costs* of each road section in the underlying road network. For instance, route planners and navigation systems often use travel times as traversal costs. However, in practice, drivers seldom travel on such 'optimal' routes due to complex traversal costs, e.g., time-dependent and uncertain travel times [PYJ20], a (possibly unknown) combination of several traversal costs [DGG+15], or due to changing intentions/destinations during a trip. We therefore investigate analysis techniques that

do not rely on a fixed criterion but are capable of identifying a suitable combination of given criteria.

The high-level goal of this chapter is to develop a trajectory segmentation approach to enable a better understanding of the semantics of trajectory data.

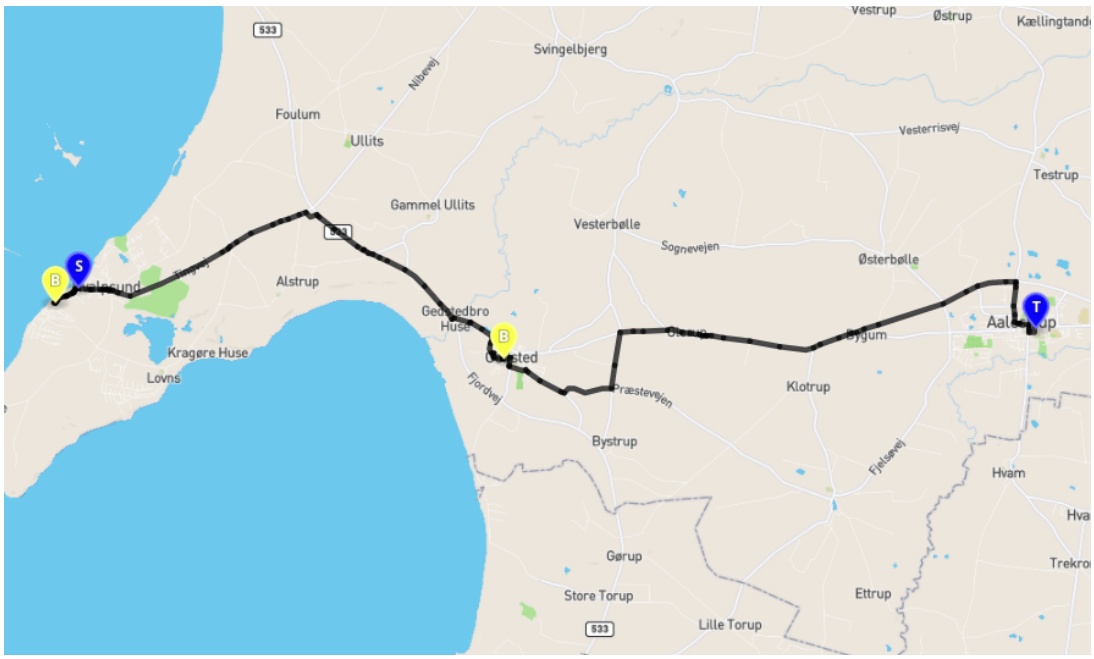


Figure 3.1.: An example of a trajectory going from S to T with two intermediate stops labeled B.

Trajectory Segmentation

A trajectory is often not just the manifestation of someone going from location S to location T following an optimal route w.r.t. some criteria, but rather determined by a sequence of activities/intentions. For instance, Figure 3.1 shows a trajectory from S to T with two intermediate stops labeled B. The driver starts at location S but rather than taking the fastest routes, decides to drive southwest and makes a stop. Then, the driver backtracks and takes the fastest route from S to T but decides to make a stop on the way. In this chapter, we present a trajectory segmentation approach that can identify intermediate stops or other points of interest in a trajectory and divide it into subtrajectories accordingly.

In contrast to previous work on trajectory segmentation [ABB+14; BDVS11; EJH+19; JTR+18; SMT+15], our approach solely relies on traversal costs and the structure of the road network. No additional information such as time stamps are required. Thus, compression techniques for efficient trajectory storage [KJT16; SSZZ14] are applicable. However, despite not utilizing temporal information, our experiments show that our trajectory segmentation approach can recover such information through a structural analysis of the trajectory. In addition, our trajectory segmentation approach uses a driving preference model to segment trajectories into subtrajectories. To the best of our knowledge, this is the first trajectory segmentation approach to do so.

3.2. Related Work

In this section, we present approaches from literature for supervised (3.2.1) and unsupervised (3.2.2) trajectory segmentation, as well as Driving Preferences Modelling (3.2.3).

3.2.1. Supervised Trajectory Segmentation

Supervised trajectory segmentation relies on predefined criteria and parameters for the segmentation. These have to be chosen and customized for the respective data set.

Buchin et al. [BDVS11] presented a general framework for supervised trajectory segmentation into a minimal number of segments for decreasing monotone criteria. As we leverage this framework, it is described in more detail in Section 3.3.3.

In a follow up work [ABB+14], the authors considered not only decreasing monotone criteria, but boolean combinations of criteria which can be increasing and decreasing monotone. More concretely, they worked with criteria that do not change often along the trajectory and called this property *stable*. Furthermore, they allowed for classification of the subtrajectories into *movement states* to further refine and constrain the resulting segmentation. The main data structure to enable this is the *compressed start stop matrix*. Such a matrix stores the allowed segments of a trajectory for a stable criterion in a space $O(n)$ and can be computed in $O(n \log n)$. One can also compute them for conjunctions and disjunctions of criteria.

This approach allows for a finely customized segmentation of trajectories with many rules and criteria. At the same time, the need for careful parameter tuning is even higher than for less sophisticated supervised approaches, because of the large parameter space.

3.2.2. Unsupervised Trajectory Segmentation

In contrast to supervised approaches, unsupervised trajectory segmentation does not need predetermined criteria or parameters. Instead, suitable criteria and parameters are selected e.g., based on a cost function.

GRASP-UTS Soares Júnior et al. [SMT+15] introduced an algorithm named *Greedy Randomized Adaptive Search Procedure for Unsupervised Trajectory Segmentation (GRASP-UTS)*. The algorithm works by first setting a number of landmarks (points of the trajectory representing segments) and then optimizing their placement. Their approach aims to generate segmentations with low distortion and high compression. Where low distortion means that segments have high homogeneity, which is measured by the euclidean distance between the landmark and all points inside the segment in all trajectory features. Compression, on the other hand, is measured in the number of segments. To balance these two goals, they introduced a cost function based on the minimum description length principle. The cost function prioritizes landmarks that create segments with high homogeneity but different consecutive movement behaviors. The high level idea of the GRASP-UTS algorithm is to repeatedly, randomly generate a feasible solution and optimize it according to the cost function via local search.

Octal Window Segmentation The octal window segmentation algorithm from Etemad et al. [EJH+19] uses an interpolation based approach. Concretely, the trajectory is processed with a sliding window

containing seven trajectory points. The first and last three points inside a window are used to interpolate the position of the central point from both sides. The haversine distance between the midpoint of the two interpolations and the actual central point is then called the error signal. A high error signal indicates a change in movement behavior and therefore a good candidate for beginning a new segment.

Although the authors claim that their algorithm classifies as unsupervised, it uses an ϵ parameter to split trajectories at points where the error signal is greater than ϵ, which the authors determine with a subset of each data set.

3.2.3. Driving Preferences Models

Driving preference models try to explain why a driver choose a certain route over another. They can also be used to create personalized routing recommendations.

Weighting of Cost Types Yang et al. [YGMJ15] modeled driving preferences by a preference vector w that contains one nonnegative weight w_i for each travel cost c_i. A vector w prefers route R_1 over R_2 if $\sum_i w_i \cdot c_i(R_1) < \sum_i w_i \cdot c_i(R_2)$. With this definition only the driving preferences for trajectories which are non-dominated by other possible routes can be optimal. To be able to assign a preference to dominated trajectories, the authors used what they call positive and negative personalized *skyline routes*. A personalized skyline route is a route which is not dominated by any other route. Positive skyline routes are such routes that dominate the trajectory and therefore are routes the driver should have taken instead. On the other hand, negative skyline routes are routes that do not dominate the trajectory and therefore there exists preferences where the trajectory is preferable to the negative route. For each pair of a negative R_j and a positive skyline route R_i, two training features for a support vector machine are created. The features $f_{i,j} = c(R_i) - c(R_j)$ is classified as *good* and the feature $f_{j,i} = c(R_j) - c(R_i)$ is classified as *bad*. The output of the support vector machine learning a linear classification is a preference vector w for the trajectory.

This simple model catches trade-offs between different cost types well. But it can suffer from cost types that strongly differ in value range in general as well as in localized regions of the road network (e.g., mountain areas vs. plains).

Preference Ratios Dai et al. [DYGD15] modeled driving preferences by a vector P that represents desired ratios between different cost types of a trajectory. They defined the *preference ratio* between the i-th and the j-th cost type of a trajectory as $pr_{i,j} = \frac{c_i}{c_j}$. The vector P contains $M = \binom{N}{2}$ entries if N cost types are used. Each entry p_i is a random variable which describes the preference ratio between two costs. The driving preference for a specific driver are derived from their historical trajectories. To do this, the individual preference ratios are calculated and consolidated into a Gaussian Mixture Model. The *Personalized Satisfaction Score* $\mathcal{F}(\mathcal{T}, P)$ measures how a satisfied a driver is with a given trajectory $\mathcal{T}$ according to his preferences P. The higher the value of $\mathcal{F}(\mathcal{T}, P) = \sum_{i=0}^{M} \int_{\hat{\mathcal{T}}_i - \Delta}^{\hat{\mathcal{T}}_i + \Delta} p_i(c) dc$ is, the better the trajectory fits to the driver preferences. Here $\hat{\mathcal{T}}_i$ is the preference ratio of $\mathcal{T}$ that corresponds to p_i and Δ is a small real value to form a neighborhood.

In contrast to the weighting model, this model has more information at hand to use. This might lead to a better explanation for trajectories, but also means there is greater potential for overfitting. Especially, because the size of a preference increases superlinear with the number of cost types used. Furthermore, the model in itself does not enforce some form of optimality in the trajectory. This

means that non-optimal trajectories might be explained, but also that trajectories with irregular parts like circling and looking for a parking space can be overemphasized.

Function Parameters The work of Delling et al. [DGG+15] represented the preference of a driver as a set of parameters β to a complex cost function F. The cost function used β to determine the travel time of each edge according to the attributes of the edge. The parameters can include for example average driving speed for different kinds of roads, boolean variables which exclude certain road types and time penalties for different kinds of turns. To arrive at a specific parameter set β^* for a driver, the authors used a combination of *local search* to approach local optima, *perturbation* to explore regions in the search space that do not lead to strictly better solutions and *specialised sampling* to shift the attention to trajectories which are not yet well explained by the current β.

A cost function and its parameters can catch fine details of the behavior of drivers. As described by the authors, those parameters influence the travel time or the availability of an edge for a certain driver. This means other factors like the crowdedness or scenicness of a route are hard to include into such a cost function.

3.3. Preliminaries

3.3.1. Data Set

3.3.1.1. Road Network Data

We use a directed graph representation of the Danish road network [JJN20] $G = (V, E)$ that has been derived from data provided by the Danish Business Authority and the OSM project. In this graph representation, V is a set of nodes, each of which represents an intersection or the end of a road, and E is a set of edges, each of which represents a directed road section. The graph representation of the Danish road network contains the most important roads and has a total of $583,816$ intersections and $1,291,171$ road sections. In addition, each road section has attributes describing their length and type (e.g., motorway) and each intersection has attributes that indicate whether they are in a city area, a rural area, or a summer cottage area. The data is further augmented with a total of $163,044$ speed limits combined from OSM data and speed limits provided by Aalborg Municipality and Copenhagen Municipality [JJNT18].

3.3.1.2. Trajectory Data

We use a set of $1,306,392$ vehicle trajectories from Denmark collected between January 1st 2012 and December 31st 2014 [AKT13]. The trajectories have been map-matched to the graph representation of the Danish road network s.t. each trajectory is a sequence of traversed road sections $T = (e_1, \ldots, e_n)$ where $e_i \in E$ for $1 \leq i \leq n$. In addition, each segment is associated with a time stamp and a recorded driving speed whenever the GPS data is sufficiently accurate. In this data set, a trajectory ends after its GPS position has not changed more than 20 meters within three minutes. See [AKT13] for more details.

Trajectory Stitching A vehicle trajectory in the trajectory data set ends when the vehicle has not moved more than 20 meters within three minutes. However, in practice, a driver may choose a trajectory with several intermediate stops, for instance when visiting multiple supermarkets to go

grocery shopping. We are interested in examining such trajectories. We therefore stitch temporally consecutive trajectories from the same vehicle together if there is less than 30 minutes difference between the end of the current trajectory to the start of the next. Each stitch thus indicates the end of a 3 to 33 minutes break in movement. We call these stitches *break points* that mark a temporal gap in the trajectory.

In many cases temporally consecutive trajectories are not connected due to imprecision or lack of GPS data. In such cases, we compute the shortest route from the destination of the current trajectory to the start of the next. If the shortest route is shorter than 200 meters or consists of at most one road section, we stitch the trajectories. We continue attempting to join the stitched to the next trajectory until the next trajectory does not meet the stitching criteria.

From the original 1,306,392 trajectories we obtain 260,190 combined trajectories. Of these trajectories, 190,199 trajectories are stitched and contain break points.

3.3.2. Routing Cost Types

From the data sets described in Section 3.3.1, we derive a number of criteria that are a measure of the expected cost of taking a route. In our experiments, we use the following four cost types: travel time, congestion, crowdedness, and number of intersections. We normalize the average value of each cost type to one.

Travel Time Each road section is associated with a fixed value that represents the time it takes to traverse the road section. To derive travel time, we combine historical traversal data from the trajectory data set with travel time estimates from a pre-trained machine learning model [JJN20]. The vehicle trajectories in our trajectory set have the tendency to be concentrated on a few popular segments, as such, many road sections have few or no traversals in the trajectory set. We therefore require a means of estimating travel times for such road sections. To this end, we use a pre-trained machine learning model to provide travel time estimates. However, for road sections with an abundance of traversal data the model's estimates may be inaccurate. Inspired by previous work [FJ17], we therefore combine travel time estimates with travel times of historical traversals s.t. when the driving speed estimate of a road section becomes increasingly less influential the more historical traversals the road section is associated with.

We compute the travel time t_e for a road section e as $t_e = \frac{k\hat{t_e}+n\bar{t_e}}{k+n}$ where $\hat{t_e}$ is the estimate of the mean travel time, $\bar{t_e}$ is the mean travel time of the historical traversals, n is the number of historical traversals of segment e in the trajectory dataset, and k represents the confidence in $\hat{t_e}$. We use $k = 10$ in our experiments.

We use a pre-trained Relational Fusion Network (RFN) [JJN20] to provide travel time estimates $\hat{t_e}$ for each road section $e \in E$. Specifically, we use the best performing RFN from [JJN20] which has been trained on the Danish Municipality of Aalborg using trajectories within the municipality that occurred between January 1st 2012 and June 30th 2013. Despite having been trained only on a subset of the network, the model generalizes well to unseen areas of the road network [JJN20]. However, in a few cases the network would give very low values. We therefore modify the output s.t. the estimated driving speed on any road section cannot be below 5 km/h.

Congestion We derive the congestion level on a particular road section based on how close to the speed limit people tend to drive. The closer to the speed limit, the less congestion. Many road

sections do not have a speed limit in our speed limit data set. In such cases, we use a simple OSM routing heuristic.We assign a congestion level to road section e depending on the speed limit s_e on the segment in km/h, the length of l_e of the segment in km, and the travel time t_e in hours. Let $\tau_e = l_e/s_e$ denote the travel time on road section e if a vehicle is driving at exactly the speed limit. Formally, we assign road section e the congestion level $c_e = \max\{1 - \frac{t_e}{\tau_e}, 0\}$ s.t. a value of 0 indicates that it is possible to drive at (or above) the speed limit and a value of 1 indicates that the road section is not traversable.

The value of τ_e relies on the speed limit of road section e. We use a speed limit data set that combines OSM speed limits with speed limits provided by Aalborg Municipality and Copenhagen Municipality [JJNT18]. This data set contains 163 044 speed limits, thus leaving many road sections without a known speed limit. In such cases, we use an OSM routing heuristic[1] which in Denmark assigns a speed limit of 130 km/h to motorways, a speed limit of 50 km/h in cities, and a speed limit of 80 km/h on other types of segments. For our data, we count a road section as in a city if either the source or destination intersection is in a city according to its attributes.

Crowdedness This criterion measures how 'crowded' the surroundings along a vehicle trajectory are. We derive a crowdedness value for each road sections from the number of nearby road sections and points of interest OSM nodes. We use all OSM nodes in Denmark from a 2019 data set regardless whether they represent a road, a building or some other point of interest. To calculate it, we first overlay our graph with a grid and count the OSM nodes within each cell. For each road section, we locate the OSM nodes that are part of its geometry in the grid. The cost per road section is then the sum of the cell counts of its (geometry) nodes. We use a grid of 2000 by 2000 resulting in a cell size of roughly 209 m x 177 m.

Number of Intersections The number of intersections visited in a trajectory, excluding the source intersection.

3.3.3. Trajectory Segmentation

In this section, we discuss the definition of the trajectory segmentation problem and a general algorithmic framework for it.

3.3.3.1. Trajectory Segmentation Problem

The segmentation of a trajectory $T = v_0 v_1 \dots v_{k-1}$ is a sequence of B trajectory segments $S_1 = v_0 v_1 \dots v_{b_1}$, $S_2 = v_{b_1} v_{b_1+1} \dots v_{b_2}$, $S_3 = v_{b_2} v_{b_2+1} \dots v_{b_3}$, up to $S_B = v_{b_{B-1}} v_{b_{B-1}+1} \dots v_{b_B}$. We refer to the common node of two consecutive trajectory segments, e.g., S_1 and S_2, as a *segmentation point*. For instance, v_{b_1} is a segmentation point because it is at the end of S_1 and the start of S_2. Buchin et al. [BDVS11] define the segmentation problem as finding a (minimal) number of segments for a trajectory such that each segment fulfills a criterion. They provide a general algorithmic framework for arbitrary segmentation criteria.

[1] See https://wiki.openstreetmap.org/wiki/OSM_tags_for_routing/Maxspeed.

3.3.3.2. Trajectory Segmentation Framework

In the framework of Buchin et al. [BDVS11], one has to provide a test procedure which verifies if a given segment meets the desired criterion. This test procedure is then used to repeatedly, greedily find the longest prefix that meets the criterion. The authors prove that this approach leads to an optimal, i.e., minimal, segmentation for (decreasing) *monotone* criteria in $O(T(k)\log k)$ time if the test procedure takes $T(m)$ time for a segment of length m. They define monotonicity for a criterion as follows. If any segment $S \subseteq T$ satisfies the criterion, then any segment $S' \subseteq S$ also satisfies the criterion. Even though Buchin et al. [BDVS11] focus on self-similarity criteria (criteria that do not change too much inside a segment) for the segments, this definition adapts well to the optimality criterion we introduce in Section 3.4.1.

3.4. Multi-Criteria Trajectory Segmentation

In this section, we present our trajectory segmentation approach designed to find all interesting via-points along a trajectory. A driver may have several via-points on a trip. Sometimes, these via-points may be linked to a point-of-interest such as a gas station or may be marked by an extended parking duration, but that is not necessarily the case. Therefore, our algorithm does not rely on such features of the trajectory but focuses on the path taken by the driver.

In brief, our approach assumes that drivers choose personalized paths between their via-points. Any deviation from their personalized path along a trajectory that goes from s to t indicates some interesting point p in the trajectory. The point of deviation p is marked as the end of the first trajectory segment and the beginning of the next. This process is repeated on the remaining subtrajectory going from p to t and so forth.

3.4.1. The Personalized Path Criterion

For trajectory segmentation in the framework of [BDVS11], described in Section 3.3.3.2, we propose a type of criteria that uses only the underlying graph and does not need any predetermined parameters. The *optimal path criterion* requires each trajectory segment S of a trajectory T to be an optimal path according to the traversal costs in the underlying graph. This criterion is monotone as defined in Section 3.3.3 because it requires S to be a "shortest path" and subpaths of shortest paths are also shortest paths. As a test procedure for a segment, we use a Dijkstra query.

The optimal path criterion can be generalized to the *personalized path criterion*. The personalized path criterion requires each trajectory segment to be a personalized path with respect to some driver preference α. This criterion is satisfied if there exists a solution to the LP described in Section 1.4.1. Note that the α for each trajectory segment can differ.

Fixing Corner Cases It is possible that there exists a minimal trajectory segment $S \subseteq T$ (consisting of a single edge) which is not a personalized/optimal path. One edge (u, v) might be more expensive in every traversal cost type than another path from u to v. This indicates that the used traversal cost types cannot explain driver behavior for taking such a road section. For the personalized path criterion, this can be remedied by including a cost type for which each edge is a personalized path between its source and target intersections. In general, a unit cost type (every edge e has $c_i(e) = 1$) has this property. This guarantees segmentability for arbitrary trajectories and makes the personalized

path more robust. In our experiments, the number of intersections cost type fulfills this role, as the road network only contains vertices that represent intersections.

This does, however, not fix the special case of self-loop edges, which in our data set typically represent road sections that allow traversals in parking lots. Such road section can never be optimal because the optimal path from the source intersection to itself remains at the intersection. Self-loop edges can either be dealt with by deleting them from the trajectories, if they do not cover significant areas in the road networks, or by representing such road section as two edges that each represent partial traversal of the self-looping road section.

3.4.2. Experiments

We now investigate the capabilities of the trajectory segmentation method to identify via-points in trajectories on the basis of the trajectory data set described in Section 3.3.1. In particular, we use the stitched trajectory set to evaluate our trajectory segmentation approach, Personalized Path Trajectory Segmentation (PPTS), to the Optimal Path Trajectory Segmentation (OPTS) baseline which uses only a single cost type to check for the optimal path criterion. We consider the four variants OPTS-TT, OPTS-Con, OPTS-Int, and OPTS-Cro, that use the travel time, congestion level, number of intersections, and crowdedness, respectively, as the single cost type.

We compare PPTS's ability to segment trajectories to that of the baselines. Our comparison is both in terms of the number of trajectories that can be segmented and the ability of the trajectory segmentation algorithms to recover the break points in the stitched trajectory set. As mentioned in Section 3.3.1.2, these break points indicate a break of 3 to 33 minutes and are therefore likely to indicate a via-point within the trajectory. We discard the self-loop edges within each trajectory to increase segmentability, as described in Section 3.4.1. Typically, these self-loop edges represent road sections that allow driving around parking lots.

All algorithms used in our experiments are implemented in the Rust programming language[1]. We make the implementation of our method, the used graph and some example trajectories publicly available[2]. We use PCH [FLS17] to speed up the Dijkstra queries by orders of magnitude.

3.4.2.1. Evaluation Functions

We use several evaluation functions to evaluate our trajectory segmentation method and for comparison with the baselines.

Segmentability Score The segmentability score, or simply S-score, measures the proportion of trajectories that are segmentable by a trajectory segmentation algorithm. Ideally, the S-score is 100% indicating that all trajectories in the data set could be segmented by the used trajectory segmentation algorithm.

Break Recovery Rate The Break Recovery Rate (BRR) is a measure of how good a trajectory segmentation algorithm is at placing segmentation points s.t. they coincide with known break points in the stitched trajectories. Let BP denote the set of known break points in a trajectory T and let SP

[1] https://www.rust-lang.org/
[2] https://github.com/Lesstat/ppts

denote the set of segmentation points output by a trajectory segmentation algorithm that has been given trajectory T as input. Then, the BRR of trajectory T is

$$\mathrm{BRR}(T) = \frac{|RBP|}{|BP|}$$

where $RBP = BP \cap SP$ is the set of recovered break points.

Segmentation Rate A trajectory segmentation algorithm can achieve a high BRR by simply segmenting a trajectory into trajectory segments consisting of one road section each. Although such a segmentation is guaranteed to recover all break points, it is also very likely to contain a lot of noise in the form of many false positives. To measure such noise, we use the Segmentation Rate (SR) which measures the number of segmentation points per break point:

$$\mathrm{SR}(T) = \frac{|SP|}{|BP|}$$

Ideally, the SR should be 1 for a trajectory segmentation that recovers all break points, i.e., has a BRR of 100%.

Segmentation Quality Score The segmentation quality score, or simply SQ-score, is a summary score to measure the overall quality of a trajectory segmentation. It combines the BRR and SR as follows:

$$\mathrm{SQ}(T) = \frac{\mathrm{BRR}(T)}{\mathrm{SR}(T)}$$

Note that the unit of the SQ-score is recovered break points per segmentation point and should ideally be 1.

3.4.2.2. Results

The results of our experiments are shown in Table 3.1.

Segmentability As shown in Table 3.1, PPTS and OPTS-Int are both capable of segmenting all trajectories and achieve an S-score of 100%. This result is not too surprising, since both algorithms use a unit cost type—the number of intersections—which guarantees that any trajectory can be segmented by these approaches, as discussed in Section 3.4.1. The remaining algorithms cannot segment a large portion of the trajectories (more than half in the case of OPTS-Con) and therefore achieve comparatively low S-scores. Thus, the inclusion of additional cost types can increase segmentability.

Segmentation Quality As shown in Table 3.1, PPTS and OPTS-Int achieve similar BRRs that are substantially higher than the remaining OPTS variants. However, for a fair comparison that ignores the ability of the algorithms to segment trajectories, we have computed BRRs, SRs, and SQ-scores on the subset of trajectories that are commonly segmentable, i.e., the trajectories that can be segmented by all algorithms. On this subset, the BRRs of all algorithms are comparable. This suggests that the superior BRR when considering all trajectories for PPTS and OPTS-Int can largely be attributed to their greater capability for segmenting trajectories.

Table 3.1.: Mean algorithm performance on all stitched trajectories (ALL) and the **60,249** commonly segmentable trajectories (CS) that can be segmented by all algorithms.

	ALL		*CS*		
Algorithm	*BRR*	*S-score*	*BRR*	*SR*	*SQ-score*
PPTS	57.98%	**100.0%**	56.29%	**2.39**	**0.235**
OPTS-TT	34.62%	58.16%	**58.35%**	2.83	0.206
OPTS-Con	29.32%	49.61%	57.49%	3.99	0.144
OPTS-Int	**59.19%**	**100.0%**	57.61%	4.37	0.132
OPTS-Cro	35.86%	61.11%	58.10%	5.62	0.103

Although the BRRs are quite similar on the commonly segmentable trajectories, the SRs are quite different, as shown in Table 3.1 In particular, the OPTS-Con, OPTS-Int, and OPTS-Cro algorithms have, respectively, 67%, 83%, and 135% more segmentation points per break point than PPTS. The SR of OPTS-TT algorithm is just 18% higher than that of PPTS.

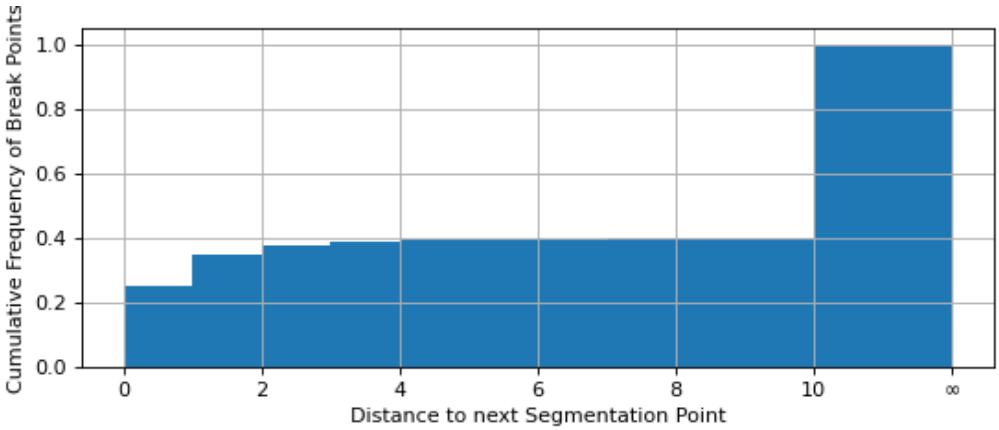
The higher SRs of the single-cost-type baselines compared to PPTS suggest that the inclusion of driving preferences and multiple criteria reduces the number of false positives. The PPTS achieves the lowest BRR on the commonly segmentable trajectories, but, as shown in Table 3.1, PPTS achieves the best overall trajectory segmentation quality with an SQ-score of 0.235, since it introduces the fewest false break points and thus has the lowest SR. Conversely, OPTS-TT achieves the highest BRR score on the same data subset, but introduces more false break points than PPTS. As a result, OPTS-TT achieves only the second-highest segmentation quality with an SQ-score of 0.206. Still, our results support the wide-spread use of the travel time cost type in many routing services, but also show that taking additional cost types and driving preferences into account can lead to better trajectory segmentation.

For the sake of brevity, we present only the comparison between PPTS and the best-performing baseline, OPTS-TT, in the remainder of this section.

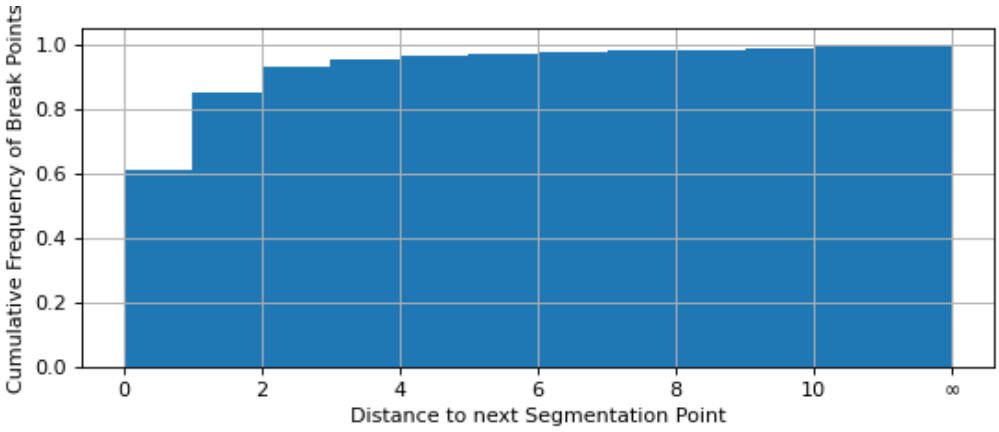
Segmentation Point Accuracy We have thus far only considered exact break point recovery, but a segmentation may still be useful if it indicates that a break point is near. Figure 3.2 shows the percentage of break points which is within a certain (hop) distance to the next segmentation point for OPTS-TT and PPTS. PPTS places segmentation points considerably more accurate than OPTS-TT. PPTS places more than 60% of the break points within one road section of the nearest segmentation point and over 80% are within two road sections of the next segmentation point. OPTS-TT achieves less than half of PPTS's performance. However, the performance disparity illustrated in Figure 3.2 is largely due to better segmentability of trajectories when using PPTS. If the analysis is restricted to break points with distance $d < \infty$ to the nearest segmentation point, i.e., trajectories that are segmentable by OPTS-TT, their distributions are comparable.

Qualitative Segmentation Assessment A good trajectory segmentation marks break points (or other interesting points) along a trajectory with a segmentation point. However, a good trajectory segmentation should also avoid too many false positives.

The PPTS and OPTS-TT, respectively, have an SR of 2.39 and 2.83 segmentation points per break point, respectively. However, although these numbers suggest that there are more false break points when using OPTS-TT, our data only contains positive examples of interesting behavior within the



(a) OPTS-TT



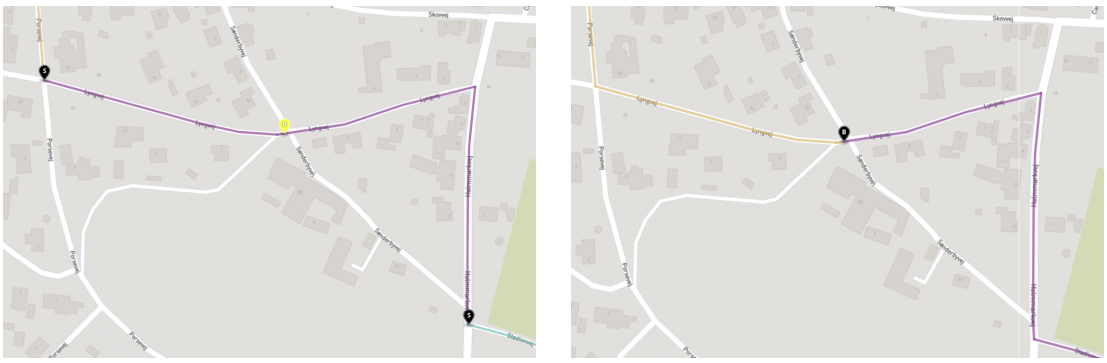
(b) PPTS

Figure 3.2.: Distribution of distance between a break point and the next segmentation point for (a) OPTS-TT and (b) PPTS. Break points in trajectories without any segmentation point are assigned distance ∞.

trajectory, i.e., the break points in the stitched trajectories. As result, we cannot quantitatively determine whether the segmentation points that do not match a break point are indeed false positives or mark interesting, but unknown, behavior during the trajectory. We therefore qualitatively assess the validity of the segmentation of a few trajectories.

Figure 3.3 shows a break point (marked with 'B' in yellow) in a segmented trajectory. The segmentation by OPTS-TT shown in Figure 3.3a places two segmentation points (marked with 'S' in black) around the break point. These segmentation points fail to recover the break point but are both within a distance of two road sections of the break point. Thus, the OPTS-TT segmentation appear to detect the presence of the break point, but fails to place the segmentation points exactly. The PPTS segmentation shown in Figure 3.3b, is a better segmentation and recovers the break point exactly (indicated by the black marker labeled 'B').

Figure 3.4 shows another part of the trajectory shown in Figure 3.3. Here, OPTS-TT places a segmentation point without comparable segmentation points in the PPTS segmentation. This additional segmentation point has no apparent meaning, and, upon detailed inspection, appears to



(a) OPTS-TT (b) PPTS

Figure 3.3.: A break point in a trajectory and the segmentation points for (a) OPTS-TT and (b) PPTS. Yellow markers labeled 'B' indicate a break point and black markers labeled 'S' a segmentation point. A black marker labeled B indicates a break point that is recovered by a segmentation point.

occur due to inaccuracies in the estimated travel time in the area. This suggests that PPTS may be more robust than OPTS-TT to noise in the traversal cost data.

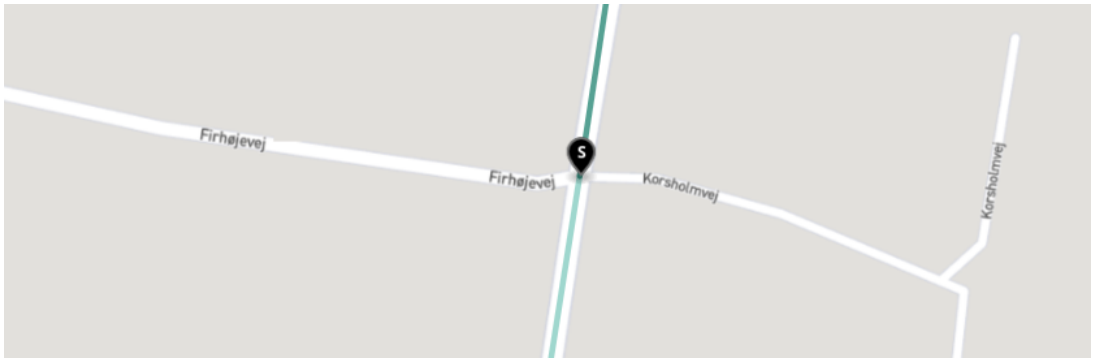


Figure 3.4.: A segmentation point with no obvious event occurring.

For the purposes of quantitative evaluation, our method attempts to recover breaks of 3 to 33 minutes from trajectories. However, our trajectory segmentation approach can discover interesting behavior beyond these known breaks. For instance, Figure 3.5 shows a segmentation point marking a detour to a gas station. This segmentation point is placed by both OPTS-TT and PPTS.

3.4.2.3. Processing Time

While using personalized routing does improve break recovery, it comes with an increase in processing time of trajectory segmentation. The increase in processing time for the personalized path variant is mostly driven by the PCH-Dijkstra queries being slower.

The trajectory segmentation process is trivially parallelizable, since each trajectory can be processed independently, making segmentation of even billions of trajectories feasible. In our experiments, we parallelized the trajectory segmentation process across 64 cores, each with a clock speed of 2.3 GHz. The time to process the 190,199 stitched trajectories for single-criteria and multi-criteria trajectory segmentation is, respectively, 1 and 5 hours in total, and 19 and 95 milliseconds per trajectory on

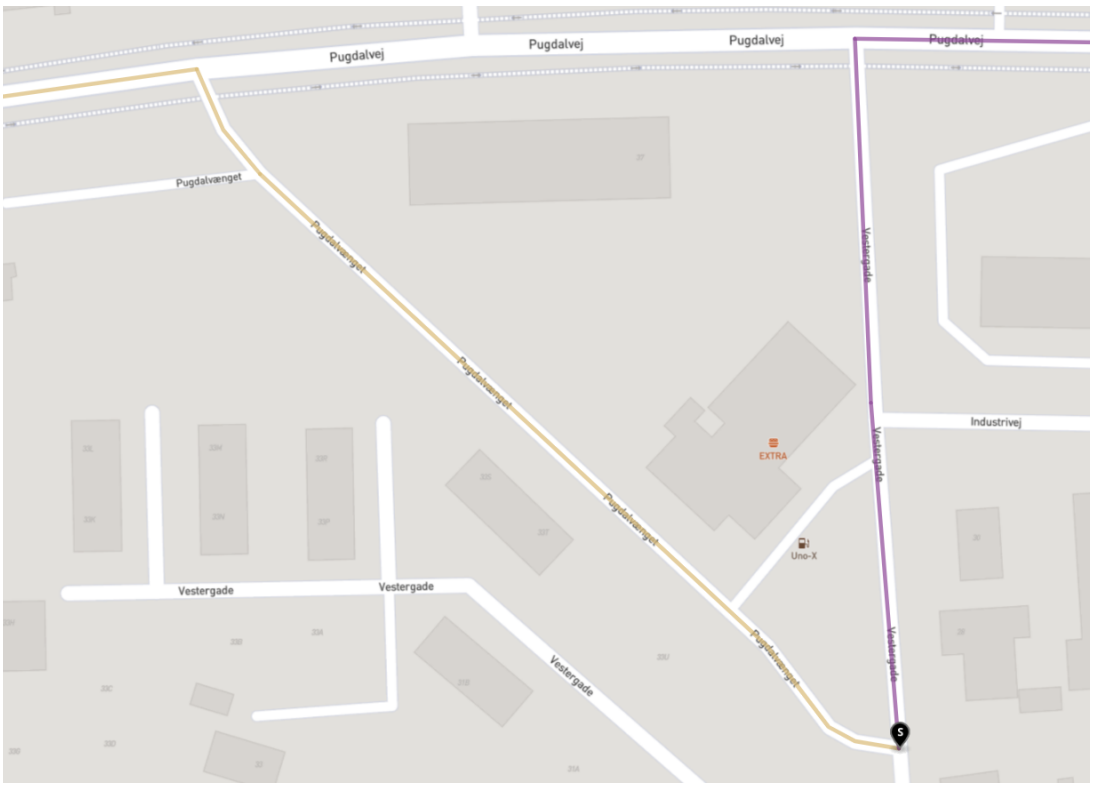


Figure 3.5.: A segmentation point recovers a detour to a gas station that is not marked as a break in our data set.

average. The total processing time took about half an hour in wall-clock time.

3.4.3. Discussion

Overall, PPTS achieves the highest trajectory segmentation quality in our experiments, followed by OPTS-TT. The results suggest that PPTS has two primary advantages over the baselines in our experiments. The use of multiple cost types and driving preferences makes PPTS capable of (1) segmenting more trajectories and (2) explaining driving behavior better, resulting in fewer false break points being placed. Our qualitative assessment of the OPTS-TT and PPTS segmentations support the conclusion that the segmentation points placed by PPTS are less likely to be false positives. In addition, PPTS discovered a detour to a gas station that is not indicated by a break point in our trajectory data set.

Even in the case where a break point is not recovered, PPTS is likely to place a segmentation point near the break point. PPTS places a segmentation point within a distance of 3 road sections for 95% of break points, but OPTS-TT for less than 40% of the break points. Although, this difference is largely explained by PPTS being capable of segmenting more trajectories, it suggests that PPTS's segmentation points are likely to indicate some interesting part of a trajectory. Although the increase in performance of PPTS over OPTS-TT comes at a factor 5 increase in processing time, it is still capable of segmenting a trajectory in a fraction of a second on average.

Stitching Parameters Changing the parameters for the stitching process has only very little effect on our results and does not affect our conclusions. The results are virtually invariant to changes to the temporal stitching threshold. However, they are sensitive to changes to the stitch length threshold, although the effect is minor. The longer the stitches are allowed to be, the worse break recovery performance for both OPTS-TT and PPTS. This is likely because more noise is introduced when longer stitches are allowed.

3.5. Conclusion

In this chapter, we presented a technique for large scale trajectory segmentation. We have shown experimentally that our proposed trajectory segmentation approach is a useful tool for understanding the semantics of a trajectory, e.g., the driver's intentions or changing destinations. Our technique can be implemented efficiently in practice and is trivially parallelizable. Thus, it scales to very large trajectory sets consisting of millions or even billions of trajectories.

Future Directions Our trajectory segmentation approach relies on linear combinations of costs. However, relationships between costs may be more complex and present an interesting opportunity for future work. In addition, driver preferences can be highly context-dependent and depend on, e.g., the time of day [YGMJ15]. Extending our approach to utilize such contextual information is an important future direction.

Chapter 4

Clustering Trajectories by Preference

4.1. Introduction

This chapter is based on joint work with Stefan Funke and Claudius Proissl. It was published in the proceedings of the 32nd International Symposium on Algorithms and Computation (ISAAC) [BFP21]. I contributed to all parts of this chapter, but to the Corner Cutting Approach in Section 4.4.1.2 I made only secondary contributions. The corresponding parts are included for completeness.

It is well observable in practice that drivers' preferences are not homogeneous. If we have two alternative paths π_1, π_2 between a given source-target pair, characterized by 3 costs/metrics (travel time, distance, and ascent along the route) each, e.g., $c(\pi_1) = (27min, 12km, 150m)$, and $c(\pi_2) = (19min, 18km, 50m)$, there are most likely people who prefer π_1 over π_2 and vice versa. Moreover, a single driver might choose differently depending on the context of his trip like, e.g., time of day, remaining gas in the tank, or being late. While typical real-world trajectories are not necessarily optimal in a the personalized route planning model, they are usually decomposable into very few optimal subtrajectories, as shown in Chapter 3.

Since the *preferences* are hard to specify as a user of a personalized route planning system, methods have been developed which infer the preferences from paths that the user has traveled before. The larger a set of paths is, though, the less likely it is that a single preference/weighting exists which explains all paths, i.e., for which all paths are optimal. One might, for example, think of different driving styles/preferences when commuting versus leisure trips through the country side.

So a natural question to ask is, what is the minimum number of preferences necessary to explain a set of given paths in a road network with multiple metrics on the edges. This can also be interpreted as a trajectory clustering task where routes are to be classified according to their purpose.

Note that we regard trajectories in this chapter as paths and disregard any timestamps attached to them. We are using the expected travel times rather than the actual instance based travel times because we are more interested in the preference that led to choosing a certain route compared to the result of choosing said route. In our example, one might be able to differentiate between commute and leisure. Or in another setting, where routes of different drivers are analyzed, one might be able

to cluster them into speeders and cruisers depending on the routes they prefer.

The goal of this chapter is the formulation of this natural optimization problem in the context of multicriteria routing and investigate the theoretical and practical challenges of solving this problem.

4.2. Related work

The linear preference model in the navigation context has been used in numerous works, e.g., [FLS17; FNS16; FS15; GKS10], to allow for personalized route planning services.

Fewer papers deal with the inference of personal preferences like [ONH17], [FLS16].

Oehrlein, Niedermann, and Haunert [ONH17] try to explain user trajectories in a bicriteria model which behaves like the personalized route planning model with $d = 2$. The cost function of their model is $c(e, \alpha) = (1-\alpha) \cdot c_0 + \alpha \cdot c_1$. Their goal is to find a preference α such that an input trajectory π of length n decomposes into the minimal number of α-optimal paths. To find such a decomposition, they first define the *optimality range problem* (OR) which given some path π asks for the range $I \subseteq [0;1]$ of α values for which the path is optimal. The authors developed an algorithm which solves this via a binary search approach if the costs are natural numbers. For the original problem, the idea is to solve OR for all subpaths of π to obtain a set R of intervals and then decompose π with each of the $O(n^2)$ α values in R. This leads to a total running time of $O(n^3 + n^2 \cdot o)$ where o is the running time for solving OR. This approach for inferring driving preferences has the advantage that it can work with trajectories that are not themselves optimal paths in the given model. While this makes successfully inferring a preference more likely, it does not guarantee it. A trajectory with a subpath that is not optimal for any α cannot be processed with this approach. Furthermore, it is limited to integer costs and only two cost types.

The other paper is most related to this work as it introduced preference inference based on a linear programming formulation, which we will also instrument in our approach. Note though that while [FLS16] already talked about the problem of minimizing the number of preferences to explain a set of trajectories, only a greedy algorithm is presented. The algorithm considers the trajectories in some order and only considers consecutive paths to be grouped under one preference. The quality of the output depends mainly on the order of the input and can be arbitrarily bad as we show in the following (one preference per trajectory). Consider two sets of trajectories T_1, T_2. Let all trajectory of T_1 be optimal for preference α_1 and those of T_2 for α_2. Also, let there be no two trajectories $t_1 \in T_1, t_2 \in T_2$ such that they are optimal for the same preference. Clearly, the minimal number of preferences needed to explain all trajectories in $T_1 \cup T_2$ is two. If we use the sweep algorithm and order the trajectories such that those of T_1 alternate with those of T_2, the algorithm will output $|T_1 \cup T_2|$ preferences instead.

4.3. Preliminaries

4.3.1. Geometric Hitting Set

The classic Minimum Hitting Set problem is defined as follows: Given a universe U with n elements $\{u_1, u_2, \ldots, u_n\}$ and a set $\mathcal{S}$ of subsets of U, find a (hitting) set H of elements of U such that $\forall S \in \mathcal{S}, \exists u \in H : u \in S$ and $|H|$ is minimized. In the geometric variant of the problem the universe U

becomes the set of all points in d-space $\mathbb{R}^d$, while the subsets contained in $\mathcal{S}$ now become geometric shapes, e.g., polyhedra.

4.3.2. Minimum Geometric Hitting Set Hardness

The minimum hitting set problem (or equivalently the set cover problem) in its general form is known to be NP-hard and even hard to approximate substantially better than a $\ln n$ factor, see [AMS06]. A simple greedy algorithm yields a $O(\log n)$ approximation guarantee, which in the general case is about the best one can hope for. For special instances, e.g., when the instance is derived from a geometric setting (as ours), better approximation guarantees can sometimes be achieved. While the exact solution remains still NP-hard even for seemingly simple geometric instances [FPT81], quite strong guarantees can be shown depending on the characteristics of the objects to be hit. Sometimes even PTAS are possible, see, e.g., [MR09]. Unfortunately, apart from convexity, none of the favourable characterizations seem to be applicable in our case. We cannot even exclude infinite VC dimension in hope for a $O(\log OPT)$ approximation [BG95], as the preference polyhedra might have almost arbitrarily many corners.

4.4. Driving Preferences and Geometric Hitting Sets

The LP-Oracle from Section 1.4.1 can easily be extended to decide for a *set* of paths (with different source-target pairs) whether there exists a single preference α for which they are optimal (i.e., which explains this route choice). It does not work, though, if different routes for the same source-target pair are part of the input or simply no single preference can explain all chosen routes. The latter seems quite plausible when considering that one would probably prefer other road types on a leisure trip on the weekend versus the regular commute trip during the week. So the following optimization problem is quite natural to consider:

> Given a set of (optimal) trajectories T in a multiweighted graph, determine a set A of preferences of minimal cardinality, such that each $\pi \in T$ is optimal with respect to at least one $\alpha \in A$.

We call this problem *preference-based trajectory clustering* (PTC).

For a concrete problem instance from the real world, one might hope that each preference in the set A then corresponds to a driving style like speeder or cruiser. Also, note that a real-world trajectory often is not optimal for a single α (prime example for that would be a round trip), yet, in Chapter 3 we showed that it can typically be decomposed into very few optimal subtrajectories if multiple metrics are available.

We aim to find practical ways to solve PTC optimally as well as approximately with good quality guarantees. Our (surprisingly efficient) strategy is to explicitly compute for each trajectory π in T the polyhedron of preferences for which π is optimal and to translate PTC into a geometric hitting set problem.

Fortunately, the formulation as a linear program as described in Section 1.4.1 already provides a way to compute these polyhedra. The constraints in the LP exactly characterize the possible values of α for which one path π is optimal. These values are the intersection of half-spaces described by the optimality constraints and the non-negativity constraints of the LP. We call this (convex) intersection *preference polyhedron*. A preference polyhedron P of path π is $d-1$ dimensional, where d is the

number of metrics. This is because the LP's scaling constraint reduces the dimension by one and we can substitute α_d by $1 - \sum_{i<d} \alpha_i$. In the remainder of this Section, we discuss how to construct preference polyhedra from given paths and reformulate PTC as a minimum geometric hitting set problem.

4.4.1. Exact Polyhedron Construction

A straightforward, yet quite inefficient way of constructing the preference polyhedron for a given st-path π is to actually determine *all* simple st-paths and perform the respective half-space intersection. Even when restricting to Pareto-optimal paths and real-world networks, their number is typically huge. So we need more efficient ways to construct the preference polyhedron.

4.4.1.1. Boundary Exploration

With the tool of linear programming at hand, one possible way of exploring the preference polyhedron is by repeated invocation of the LP described in Section 1.4.1 but with varying objective functions. Note that this is strongly related to the enumeration of personalized paths via convex hull exploration in Section 2.3. For sake of simplicity let us assume that the preference polyhedron is full (that is, $d-1$) dimensional[1]. We first determine $d-1$ distinct extreme points of the polyhedron to obtain a $d-1$-simplex as first (inner) approximation of the preference polyhedron. We then repeatedly invoke the LP with an objective function corresponding to the normal vectors of the facets of the current approximation of the polyhedron. The outcome is either that the respective facet is part of the final preference polyhedron, or a new extreme point is found which destroys this facet and induces new facets to be investigated later on. If the final preference polyhedron has f facets, this approach clearly requires $O(f)$ invocations of the LP solver (and some effort to maintain the current approximation of the preference polyhedron as the convex hull of the extreme points found so far). While theoretically appealing, in practice this approach is not very competitive due to the overhead of repeated LP solving.

4.4.1.2. Corner Cutting Approach

We developed the following *Corner Cutting Approach (CCA)* which in practice turns out to be extremely efficient to compute for a given path π its preference polyhedron, even though we cannot bound its running time in terms of the complexity of the produced polyhedron.

CCA is an iterative algorithm, which computes the preference polyhedron via a sequence of half-space intersections. Let P_π be the preference polyhedron of path π. In the i-th iteration CCA computes a polyhedron P_i with $P_\pi \subseteq P_i \subseteq P_{i-1}$. P_0 is the entire preference space with a number of corners equal to the number of metrics. Each of these corners is initially marked as unchecked.

In iteration i CCA takes one corner $\alpha^{(i)}$ of P_{i-1} that has not been checked before. If no such corner exists, CCA terminates and returns $P_{CCA} := P_{i-1}$. Otherwise, it marks the corner $\alpha^{(i)}$ as checked and computes the optimal path $\pi_{\alpha^{(i)}}$. If $c\left(\pi, \alpha^{(i)}\right) - c\left(\pi_{\alpha^{(i)}}, \alpha^{(i)}\right) = 0$, the corner $\alpha^{(i)}$ belongs to P_π and there is nothing to do. Otherwise, the constraint $c(\pi, \alpha) - c(\pi_{\alpha^{(i)}}, \alpha) \leq 0$ is violated by a part of P_{i-1}. We call this part P'_i. It is clear that $\alpha^{(i)} \in P'_i$. Finally, the polyhedron $P_i := P_{i-1} \backslash P'_i$ is computed by intersecting P_{i-1} with the half-space $c(\pi, \alpha) - c(\pi_{\alpha^{(i)}}, \alpha) \leq 0$. This intersection may introduce new corners, which are marked as unchecked. Afterwards, the next iteration starts.

[1] after elimination of α_d

We prove that CCA indeed computes P_π. Let P_{CCA} be the output of CCA. We first show that $P_\pi \subseteq P_{CCA}$. $P_\pi \subseteq P_0$ is trivially true. Furthermore, in each iteration i it is clear that $P_i' \cap P_\pi = \emptyset$. Hence, for each iteration i, we have $P_\pi \subseteq P_i$ and therefore $P_\pi \subseteq P_{CCA}$. The other direction $P_{CCA} \subseteq P_\pi$ follows from the fact that P_π is convex and that all corners of P_{CCA} belong to P_π as they are marked as checked.

CCA also terminates for finite graphs. Each optimal path $\pi_{\alpha^{(i)}}$ can add finitely many corners to P_i only once. Since the number of optimal paths is finite the number of corners to be considered is finite as well.

The great advantage of CCA compared to the boundary exploration approach is the avoidance of the linear programming solver.

4.4.2. Minimum Geometric Hitting Set

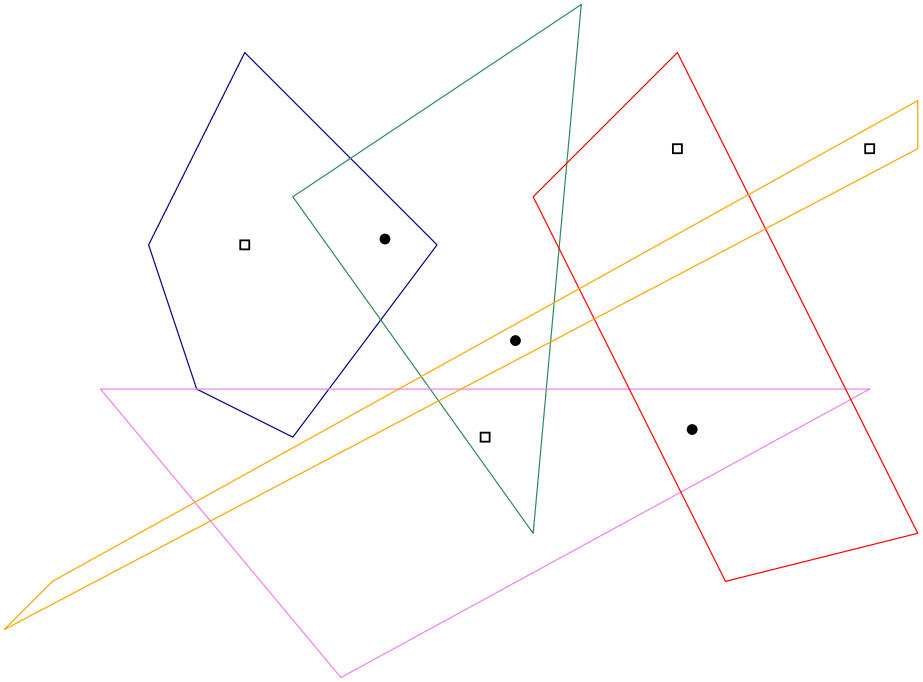


Figure 4.1.: Example of a geometric hitting set problem as it may occur in the context of PTC. Two feasible hitting sets are shown (white squares and black circles).

Using the preference polyhedra we are armed to rephrase our original problem as a *geometric hitting set (GHS)* problem. In an instance of GHS we typically have geometric objects (possibly overlapping) in space and the goal is to find a set of points (a *hitting set*) of minimal cardinality, such that each of the objects contains at least one point of the hitting set. Figure 4.1 shows an example of how preference polyhedra of different optimal paths could look like in case of three metrics. In terms of GHS, our PTC problem is equivalent to finding a hitting set for the preference polyhedra of minimum cardinality, and the 'hitters' correspond to respective preferences. In Figure 4.1 we have depicted two feasible hitting sets (white squares and black circles) for this instance. Both solutions are minimal in that no hitter can be removed without breaking feasibility. However, the white squares (in contrast to the black circles) do not describe a minimum solution as one can hit all polyhedra with less points.

While the GHS problem allows to pick arbitrary points as hitters, it is not hard to see that it suffices to restrict to vertices of the polyhedra and intersection points between the polyhedra boundaries, or

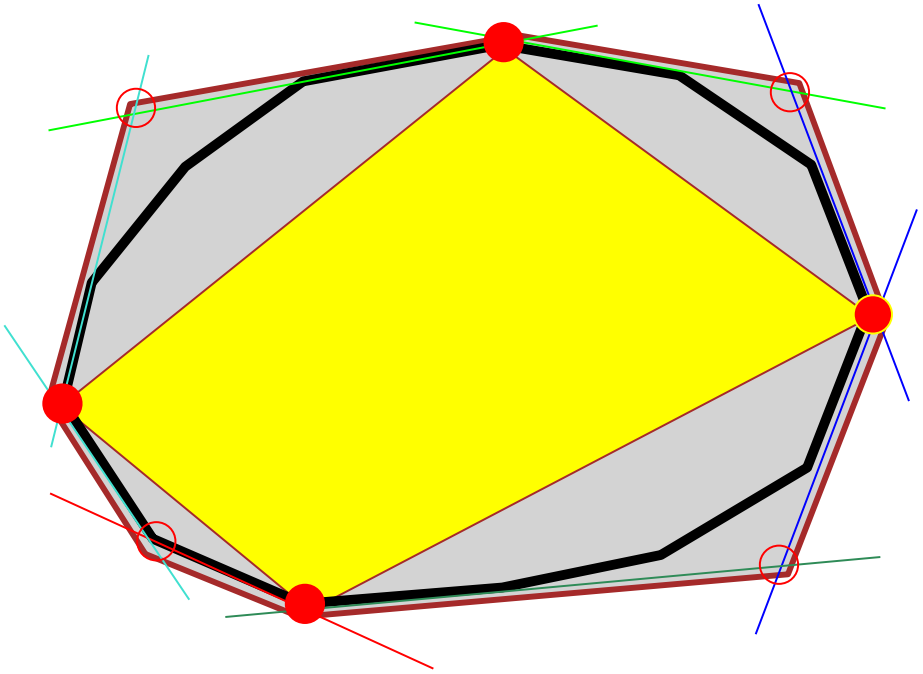


Figure 4.2.: Inner (yellow) and outer approximation (grey) of the preference polyhedron (black).

more precisely vertices in the arrangement of preference polyhedra. We describe the computation of these candidates for the hitting sets in Section 4.4.3.

The GHS instance is then formed in a straightforward manner by having all the hitting set candidates as ground set, and subsets according to containment in respective preference polyhedra. For an exact solution we can formulate the problem as an integer linear program (ILP). Let $\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(l)}$ be the hitting set candidates and $\mathcal{U} := \{P_1, P_2, \ldots, P_k\}$ be the set of preference polyhedra. We create a variable $X_i \in 0, 1$ indicating whether $\alpha^{(i)}$ is picked as a hitter and use the following ILP formulation:

$$\begin{aligned} \min \sum_i X_i \\ \forall\, P \in \mathcal{U} : \sum_{\alpha^{(i)} \in P} X_i \geq 1 \\ \forall\, i : X_i \in \{0, 1\} \end{aligned}$$

While solving ILPs is known to be NP-hard, it is often feasible to solve ILPs derived from real-world problem instances even of non-homeopathic size.

4.4.3. Hitting Set Instance Construction via Arrangements of Hyperplanes

To obtain the actual hitting set instance, we overlay the individual preference polyhedra. This can be done via construction of the arrangement of the hyperplanes bounding the preference polyhedra. Each vertex in this arrangement then corresponds to a candidate for the hitting set. If N is the total number of hyperplanes bounding all preference polyhedra in D-dimensional space, then this arrangement has complexity $O(N^D)$ and can be computed by a topological sweep within the same time bound [Ede87]. For K polyhedra with overall N bounding hyperplanes we obtain a hitting set

instance with K sets and $O(N^D)$ potential hitter candidates.

4.4.4. Challenges

While our proposed approach to determine the minimum number of preferences to explain a set of given paths is sound, it raises two major issues. First, the complexity of a single preference polyhedron might be too large for actual computation, so just writing down the geometric hitting set instance becomes infeasible in practice. Second, solving a geometric hitting set instance to optimality is far from trivial. In the following we will briefly discuss these two issues and then in the next section come up with remedies.

4.4.4.1. Preference Polyhedron Complexity

In this section, we show that the number of facets of a preference polyhedron can be as high as the number of optimal paths. For this section, we use a tighter definition of optimal. Namely, that a path is a unique optimal path if and only if its preference polyhedron has a non-zero volume. See Chapter 5 for a more thorough explanation of the concept. We do this in two steps. First, we show that we can construct a graph with 2-dimensional costs with k optimal paths between s and t for any k. Then, we show how to transform such a graph into a graph with 3-dimensional costs and $k+1$ optimal paths, such that one preference polyhedron has at least k facets.

Lemma 4.1
It is possible to construct a graph with 2-dimensional costs and $k+1$ optimal paths between two nodes for arbitrary k.

Proof Let us assume that there are $k+1$ st-paths $\pi_0, \pi_1, \ldots, \pi_k$ with cost vectors $c(\pi_i) := \left[i^2, (k-i)^2\right]$. This could be easily realized with one-edge paths. A preference is in this case a tuple $[1-\alpha, \alpha]$ with $0 \leq \alpha \leq 1$.

For any $0 \leq i < k$ we have $c(\pi_i) - c(\pi_{i+1}) = [-2i-1, 2(k-i)-1]$. With $\alpha = \frac{2i+1}{2k}$ we get

$$\begin{aligned}
&[1-\alpha, \alpha]^T (c(\pi_i) - c(\pi_{i+1})) \\
=&\left[\frac{2(k-i)-1}{2k}, \frac{2i+1}{2k}\right]^T [-2i-1, 2(k-i)-1] \\
=&0
\end{aligned}$$

Hence, for $0 < i < k$ the path π_i is optimal for the range $\alpha \in \left[\frac{2i-1}{2k}, \frac{2i+1}{2k}\right]$. Path π_0 is optimal for the range $\alpha \in \left[0, \frac{1}{2k}\right]$ and path π_k is optimal for the range $\alpha \in \left[\frac{2k-1}{2k}, 1\right]$. □

Lemma 4.2
Given a 2-dimensional cost graph with k optimal paths between two nodes, one can construct a graph with 3-dimensional costs and $k+1$ optimal paths where one path has a preference polyhedron with at least k 1-dimensional facets for arbitrary $k > 1$.

Proof Let the k optimal paths be called π_1 to π_k. We refer to the j-th entry of cost vector $c(\pi_i)$ with $c(\pi_i)_j$. Let $x := \max_{1 \leq i \leq k} c(\pi_i)_1 + c(\pi_i)_2$ be the maximum of the sums of the cost vectors.

We now introduce a third metric with a constant cost of $3x$ for each path. It is clear that each of the k optimal paths is optimal for the preference $\alpha_3 = [0, 0, 1]$. Hence, each preference polyhedron is now a triangle as shown in Figure 4.3a.

(a) Before inserting new optimal path

(b) After inserting new optimal path

Figure 4.3.: Example of preference polyhedra of optimal paths with the same source and target and with equal cost in the third metric

Finally, we create a new optimal path π_{k+1} with the cost vector $c(\pi_{k+1}) := [2x, 2x, 2x]$. This path is clearly optimal for the preference α_3. Since $\alpha_3^T (c(\pi_{k+1}) - c(\pi_i))$ is strictly less than zero for each $1 \leq i \leq k$ the volume of the preference polyhedron of π_{k+1} is non-zero. Furthermore, restricted to the first two metrics π_{k+1} is not optimal. This directly follows from the definition of x. Hence, the preference polyhedron of π_{k+1} shares a constraint with each of the k other optimal paths as shown in Figure 4.3b. □

The arguments of Lemmas 4.1 and 4.2 can be extended to arbitrary dimension. As we will see in Chapter 5, the number of optimal st-paths is subexponential in the graph size. Therefore, it follows that there are no preference polyhedra with exponential complexity.

4.5. Polynomial-Time Heuristics with Instance-based Lower Bounds

The previous section suggests that if we require worst-case polynomial running time, we have to resort to approximation of some kind. Both, generation of the geometric hitting set instance as well as solving of the instance might not be possible in polynomial time. We address both issues in this section.

4.5.1. Approximate Instance Generation

It appears difficult to show polynomial bounds on the size of a single preference polyhedron, so approximation with enforced bounded complexity seems a natural approach. There are well-known techniques like coresets [AHV+05] that allow arbitrarily (specified by some ϵ) accurate approximation of convex polyhedra in space polynomial in $1/\epsilon$ (which is independent of the complexity of the original polyhedron). We follow the coreset approach and also make use of the special provenance of the polyhedron to be approximated.

For d metrics, our polyhedron lives in $d-1$ dimensions, so we uniformly ϵ-sample the unit $(d-2)$-sphere using $O((1/\epsilon)^{d-2})$ samples. Each of the samples gives rise to an objective function vector for our linear program, we solve each such LP instance to optimality. This determines $O((1/\epsilon)^{d-2})$ extreme points of the polyhedron in equally distributed directions. Obviously, the convex hull of

these extreme points is *contained within* and with decreasing ϵ converges towards the preference polyhedron. Guarantees for the convergence in terms of ϵ have been proven before, but are not necessary for our (practical) purposes. We call the convex hull of these extreme points the *inner approximation* of the preference polyhedron.

What is interesting in our context is the fact that each extreme point is defined by $d-1$ half-spaces. So we can also consider the set of half-spaces that define the computed extreme points and compute their intersection. Clearly, this half-space intersection *contains* the preference polyhedron. We call this the *outer approximation* of the preference polyhedron.

Let us illustrate our approach for a graph with $d = 3$ metrics, so the preference polyhedron lives in the 2-dimensional plane, see the black polygon/polyhedron in Figure 4.2. Note that we do not have an explicit representation of this polyhedron but can only probe it via LP optimization calls. To obtain inner and outer approximation we determine the extreme points of this implicitly (via the LP) given polyhedron, by using objective functions $\max \alpha_1, \max \alpha_2, \min \alpha_1, \min \alpha_2$. We obtain the four solid red extreme points. Their convex hull (in yellow) constitutes the inner approximation of the preference polyhedron. Each of the extreme points is defined by 2 constraints (halfplanes supporting the two adjacent edges of the extreme points of the preference polyhedron). In Figure 4.2, these are the light green, blue, dark green, and cyan pairs of constraints. The half-space intersection of these constraints form the *outer approximation* in gray.

4.5.1.1. Sandwiching the Optimum Hitting Set Size

If – by whatever means – we are able to solve geometric hitting set instances optimally, our inner and outer approximations of the preference polyhedra yield upper and lower bounds to the solution size for the actual preference polyhedra. That is, if for example the optimum hitting set of the instance derived from the *inner* approximations has size 23 and the optimum hitting set of the instance derived from the *outer* approximations has size 17, we know that the optimum hitting set size of the actual exact instance lies between 17 and 23. Furthermore, the solution for the instance based on the inner approximations is also feasible for the actual exact instance. So in this case we would have an instance-based (i.e., not apriori, but only aposteriori) approximation guarantee of $23/17 \approx 1.35$. If for the application at hand this approximation guarantee is not sufficient, we can try improving by refining the inner and outer approximations. In the limit, inner and outer approximations coincide with the exact preference polyhedra.

4.5.2. Approximate Instance Solving

In this section, we discuss approximation algorithms to replace the ILP shown in Section 4.4.2, which is not guaranteed to run in polynomial time.

The geometric objects to be hit are the preference polyhedra of the given set of paths, the hitter candidates and which polyhedra they hit are computed via the geometric arrangement as described in Section 4.4.3.

4.5.2.1. Naive Greedy Approach

The standard greedy approach for hitting set iteratively picks the hitter that hits most objects, which have not been hit before. We call this algorithm *Naive Greedy* or short NG. It terminates as soon as all objects have been hit. The approximation factor of this algorithm is $O(\log n)$, where n is the number

of objects to be hit, see [Joh74]. The information which preference hits which polyhedron comes from the arrangement described in Section 4.4.3.

After computing the cover, NG iterates over the picked hitters and removes them if feasibility is not violated. In this way the computed hitting set is guaranteed to be minimal (but not necessarily minimum).

4.5.2.2. LP-guided Greedy Approach

While naive greedy performs quite well in practice, making use of a precomputed optimal solution to the LP relaxation of the ILP formulation from Section 4.4.2 can improve the quality of the solution. We call this algorithm *LP Greedy* or LPG. It starts with an empty set S^* and iterates over a random permutation of the cover constraints of the LP. Note that each of these constraints C_i corresponds to a preference polyhedron P_i.

At each constraint C_i LPG checks if P_i is hit by at least one preference in S^*. If not, one of its hitters is randomly picked based on the weights of the LP solution and added to S^*. To be more precise, the probability of a hitter α to be drawn is equal to its weight divided by the sum of the weights of all hitters of P_i.

After iterating over all constraints it is clear that S^* is a feasible solution. Finally, LPG iterates over S^* in random order removing elements if feasibility is not violated. This ensures that S^* is minimal. This process is repeated several times and the best solution is returned.

4.6. Experimental Results

In this section, we assess how real-world relevant the theoretical challenges of polyhedron complexity and NP-hardness of the GHS problem are, and compare exact solutions to our approximation approaches.

4.6.1. Experimental Setup

We run our experiments on a server with two intel Xeon E5-2630 v2 running Ubuntu Linux 20.04 with 378GB of RAM. The running times reported are wall clock times in seconds. Some parts of our implementation make use of all 24 CPU threads. We cover two different scenarios by extracting different graphs from OpenStreetMap of the German state of Baden-Württemberg. This first graph is a road network with the cost types *distance, travel time for cars* and *travel time for trucks*. It contains about 4M nodes and 9M edges. The second graph represents a network for cyclists with the cost types *distance, height ascent*, and *unsuitability for biking*. The latter metric was created based on the road type (big road ⇒ very unsuitable) and bicycle path tagging. The cycling graph has a size of more than double of the road graph with about 11M nodes and 23M edges. The Dijkstra separation oracle was accelerated using a precomputed personalized contraction hierarchy. For the Sections 4.6.3 to 4.6.5, we used a set of 50 preferences chosen u.a.r. per instance and created different quantities of paths with those preferences. We therefore know an apriori upper bound for the size of the optimal hitting set for our instances. In Section 4.6.6, we show that the number of preferences used does not change the characteristics of our approaches.

4.6.2. Implementation Details

Our implementation consists of multiple parts. The routing and the computation of the (approximate) preference polyhedra of paths is implemented in the rust programming language (compiled with rustc version 1.51) and uses GLPK [And12] (version 4.65) as a library for solving LPs. We intentionally refrained from using non-opensource solutions like CPLEX or GUROBI, as they might not be accessible to everyone. The preference polyhedra are processed in a C++ implementation (compiled with g++ version 10.2) which uses the CGAL library [The20; WBF+20] (version 5.0.3) to compute the arrangements with exact arithmetic and output the hitting set instances. We transform the hitting set instances into the ILP formulation from Section 4.4.2 and solve this ILP formulation and its LP relaxation with GLPK. Finally, we implemented the two greedy algorithms described in Sections 4.5.2.1 and 4.5.2.2 which solve the hitting set instances in C++. Source code and data under `https://doi.org/10.17605/osf.io/4qkuv`.

4.6.3. Geometric Hitting Set Instance Generation

First, we assess the generation of the GHS instances via preference polyhedra construction and computation of the geometric arrangement. In our tables, we refer to the corner cutting approach as "exact" and the approximate approach as "inner-k"/ "outer-k" where k is the number of directions that were approximated. Since the hitter candidates from the geometric arrangement contain a lot of redundancies (which slows down in particular the (I)LP solving), e.g., some hitters being dominated by others, we prune the resulting candidate set in a straightforward set minimization routine. The preference polyhedra construction as well as the set minimization routine are the multithreaded parts of our implementation. Table 4.1 shows run times and the average polyhedron complexity for a problem instance with 10,000 paths. In this instance, approximating in twelve directions takes more time than the exact calculation with CCA. This is due the polyhedron complexity being very low in practice. It shows that CCA is a valid approach in practice. Set minimizing is particularly expensive for the inner approximations since considerably more candidates are not dominated.

Table 4.1.: Statistics about instance generation: average number of polyhedron corners, polyhedra construction time (multithreaded), construction time for arrangement, time for set minimization (multithreaded). Car graph with 10,000 paths. Time in seconds.

Algo.	Polyh. Compl	Polyh. Time	Arr. Time	SetMin Time
Inner-12	3.8	70.6	352.3	1450.0
Exact	4.7	54.9	356.7	414.8
Outer-12	4.5	70.6	361.4	473.0

4.6.4. Geometric Hitting Set Solving

We now compare the two greedy approaches from Section 4.5.2 with the ILP formulation of Section 4.4.2. For the ILP solver, we set a time limit of 1 hour after which the computation was aborted and the best found solution (if any) was reported. The results for two instances on the bicycle graph are shown in Table 4.3. The naive greedy approach has by far the smallest run time but it also reports worse results than the LP-based greedy which was able to find the optimal solution for exact and outer approximations of the two instances. The ILP solver always reports the optimal solution but it

Table 4.2.: Instance generation and solving for various polyhera approximations. Car graph with 1,000 paths. Time in seconds.

Algo.	Polyh. Time	Arr. Time	ILP Sol.	ILP Time
Inner-4	6.7	4.4	110	>3600
Inner-8	8.4	5.5	50	>3600
Inner-16	11.3	4.8	45	15.5
Inner-32	16.5	5.0	42	3.7
Inner-64	26.9	4.9	39	1.8
Inner-128	48.8	5.0	36	2.3
Exact	6.5	5.2	36	0.7
Outer-128	48.8	5.1	36	0.8
Outer-64	26.9	5.1	36	0.8
Outer-32	16.5	5.0	36	0.6
Outer-16	11.3	5.2	36	1.8
Outer-8	8.4	5.0	36	1.6
Outer-4	6.7	5.9	35	11.5

might take a very long time to do so. Especially, the inner approximations seem to yield hard GHS instances before they converge to the exact problem instance. A state of art commercial ILP solver might improve run time drastically.

Table 4.3.: Comparison of GHS solving algorithms on two instances on the bicycle graph. The times (given in seconds) for the LP-Greedy and ILP algorithm include solving the LP-relaxation first.

Algorithm	Paths	Greedy Solution	Greedy Time	LP-Greedy Solution	LP-Greedy Time	ILP Solution	ILP Time
Inner-12	5000	210	4.2	181	40.8	-	>3600.0
Exact	5000	54	2.9	48	24.8	48	68.4
Outer-12	5000	57	3.1	48	30.5	48	95.5
Inner-12	10000	421	10.9	399	144.6	-	>3600.0
Exact	10000	58	7.6	50	73.5	50	188.9
Outer-12	10000	59	8.1	50	81.1	50	114.6

4.6.5. Varying Polyhedron Approximation

In Table 4.2 we show that increasing the number of directions in the approximation approach makes its results converge to the exact approach. Interestingly, the outer approximation converges faster than the inner approximation. It typically yields good lower bounds already for four approximation directions and higher. In Table 4.2 it yields a tight lower bound for eight directions while the inner approximation only achieves a tight upper bound with 128 directions.

4.6.6. Dependence on the number of preferences

So far, we have only considered PTC problem instances that were derived from 50 initial preferences. To ensure that the fixed number of preferences does not bias our results, we also ran instances

generated with 10, 20, 100 and 1000 preferences. Tables 4.4 and 4.5 hold the results for instances with 1,000 paths on both graphs. As expected, we found larger optimal solutions as the number of preferences increased. Although, the solution size did increase only to 53 and 177 for the car and bicycle graph, respectively. We attribute this to the probably bounded inherent complexity of the graphs and cost types used. We also note a slight increase in ILP run times with spikes when computing the solution to the inner approximation as discussed in Section 4.6.4. Otherwise, there is no performance difference.

Table 4.4.: Run times and optimal solution of instances generated with varying number of preferences. The instances each consist of 1.000 paths on the car graph.

Algorithm	α	Polygon Time	Arrangement Time	Set Minimization Time	ILP Solution	ILP Time
Inner-8	10	7.9	4.8	1.2	23	6.5
Exact	10	5.9	4.5	0.4	10	0.3
Outer-8	10	7.9	4.4	0.4	10	0.2
Inner-8	20	7.8	7.7	2.8	28	96.6
Exact	20	5.2	6.0	0.6	17	0.8
Outer-8	20	7.8	6.1	0.6	17	0.8
Inner-8	100	8.3	5.5	1.7	57	3601.0
Exact	100	6.3	5.8	0.6	49	23.1
Outer-8	100	8.3	5.6	0.6	49	21.1
Inner-8	1000	8.0	5.7	2.1	61	1166.5
Exact	1000	5.9	5.3	0.5	53	2.1
Outer-8	1000	8.0	5.2	0.5	53	21.9

Table 4.5.: Run times and optimal solution of instances generated with varying number of preferences. The instances each consist of 1.000 paths on the bicycle graph.

Algorithm	α	Polygon Time	Arrangement Time	Set Minimization Time	ILP Solution	ILP Time
Inner-8	10	47.9	3.0	0.6	45	0.9
Exact	10	57.9	2.9	0.4	10	0.1
Outer-8	10	47.9	3.0	0.5	10	0.1
Inner-8	20	42.4	2.9	0.7	69	3600.0
Exact	20	47.7	3.1	0.5	20	0.3
Outer-8	20	42.4	3.1	0.6	20	0.3
Inner-8	100	43.8	2.0	0.3	125	398.3
Exact	100	50.1	2.1	0.2	85	0.2
Outer-8	100	43.8	2.1	0.3	83	0.3
Inner-8	1000	43.9	2.0	0.4	193	1.2
Exact	1000	49.0	2.2	0.3	177	1.6
Outer-8	1000	43.9	2.2	0.3	177	3.7

4.7. Conclusions

We have exhibited an example for a real-world application where theoretical complexities and hardness do not prevent the computation of optimal results even for non-toy problem instances. The presented method allows the analysis of large trajectory sets as they are, for example, collected within the OpenStreetMap project. Our results suggest that exact solutions are computable in practice, even more so if commercial ILP solvers like CPLEX or GUROBI are employed. An open problem is to find an explanation for the observed average preference polyhedra complexity, which is surprisingly low. On a more abstract level, our approach of approximating the hitting set problem with increasing precision by refining the inner and outer approximations until an optimal solution can be guaranteed could also be viewed as an extension of the framework of *structural filtering* ([FMN05]), where the high-level idea is to certify exactness of possibly error-prone calculations. It could be interesting to apply this not only to the instance generation step but simultaneously also to the hitting set solution process.

Part III.

Theoretical analysis of Personalized Paths

Chapter 5

Upper Bounding the number of personalized Paths

5.1. Introduction

This chapter is based on joint work with Stefan Funke and Claudius Proissl. It was presented in the 38th European Workshop on Computational Geometry and is pending publication. I made secondary contributions to Section 5.3.3 and contributed to the rest of the chapter.

In this chapter, we consider the question how many optimal paths can exist in the personalized route planning model. As we have shown in the previous chapters, our implementations based on the model perform well in practice. Here, we want to focus on better understanding the theory behind the model. Having an upper bound on the number of unique optimal paths would make it possible to better analyze algorithms that work with the model and differentiate it from other settings like Pareto-optimal paths.

Complexity-wise there can be quite a huge gap between optimizing over all Pareto-optimal paths or only over unique shortest paths. For example, in the resource-constrained shortest path problem in 2-metric graphs, one is interested in finding the st-path minimizing the first metric but having the second metric (the resource) below some limit R. In general – considering all Pareto-optimal solutions – this problem is NP-hard via reduction from knapsack, see [HZ80], while restricting to unique shortest paths, polynomial running-time has been achieved, e.g., see [MZ00]. While the number of paths in both models is not the only factor for this complexity difference, it seems somewhat expectable that a lot more Pareto-optimal paths than unique shortest paths exist. It remains to prove a substantial gap between the number of Pareto-optimal and unique shortest paths. This chapter sheds some light on this question and proves that there is indeed a gap.

For fixed dimension d it is possible to enumerate all unique shortest st-paths in polynomial time (polynomial in the output size) using the exploration technique shown in Chapter 2. Thus, the mentioned gap between the number of Pareto-optimal and unique shortest paths is not only of theoretical interests but can also be leveraged in practice.

5.1.1. Related Work

The fact that the number of Pareto-optimal paths can be exponential in the graph size can be considered folklore. Both, Pareto-optimal paths as well as unique shortest paths have been instrumented to create alternative route recommendations. The former approach, pursued e.g., in [DW09; KRS10; MW06], unfortunately only seems to be viable on rather small graphs due to the too rapidly growing number of Pareto-optimal paths. Restricting to unique shortest paths, though, as in [FS15], has been shown to be feasible in different practical applications [BFP21; BFS19].

Very much related to Pareto-optimal paths are the *maxima of a point set*. In a set of n vectors (points) in d-space, a vector is called maximal if there is no other vector which has a strictly greater value in every component. Maxima (like the convex hull) can be understood as a characterization of the boundary of a point set and have also been instrumental in the analysis of dynamic programming algorithms, see [BKST78]. In the same paper, the authors show that the expected number of maxima of an n-vector set is $O((\ln n)^{d-1})$ if the vector set is drawn from a distribution with component-wise independence, e.g., uniformly at random from a hypercube. On the other hand, the expected number of vertices of the convex hull from such a vector set is also $O((\ln n)^{d-1})$. In [Dwy88] it is shown that this upper bound is tight in many cases. So in this random setting, there is in fact no substantial gap between the number of maximal points and the number of extreme points of the convex hull.

Gusfield [Gus80] has shown an upper bound of $n^{\log n+O(1)}$ on the number of optimal paths for $d = 2$. We adapted his proof for arbitrary d and describe it in more detail in Section 5.3.3. This $d = 2$ bound can also be derived from the work of Gajjar and Radhakrishnan [GR19] on a bound of $n^{(\log n)^2+O(\log n)}$ for $d = 3$. Their proof relies on the relation between the representation of personalized paths in the cost space with Minkowski sums as well as Euler's polyhedral formula. Specifically, given vertices $s, v, t \in V$ the convex hull (in cost space) of all st-paths bisected by v can be obtained by computing the Minkowski sum of the convex hulls of all sv-paths and all vt-paths. Euler's formula allows to put the number of different dimensional faces created by the Minkowski sum into relation and give a strong bound on the 0-dimensional faces in cost space which correspond to unique optimal paths in the graph. Unfortunately, there is no equivalent for Euler's formula in $d > 3$.

5.1.2. Our contribution

In this chapter we show that while there are graphs with an exponential number of Pareto-optimal (and shortest) paths (even for $d = 2$), the number of *unique shortest paths* is in $O(n^{2d\sqrt{n}+d+1})$, which is subexponential for fixed d. On the other hand we construct d-metric graphs with $\Omega(n^{d-1})$ unique shortest paths.

5.2. Preliminaries

In this section we introduce the notions used in Section 5.3 and show some basic properties.

Definition 5.1
The set of all possible preferences

$$\mathcal{P}_d := \{(\alpha_1, \alpha_2, \ldots, \alpha_d) \in \mathbb{R}^d_+ \mid \sum_{i=1}^{d} \alpha_i = 1\}$$

is called d-metric preference space.

We simply write $\mathcal{P}$ instead of $\mathcal{P}_d$ if the dimension is unimportant. Note that a d-metric preference space is a $(d-1)$-dimensional simplex. We say that a preference $\alpha \in \mathcal{P}_d$ is degenerate if it contains zeros (recall that all preference entries are non-negative). The following simple lemma draws a connection between shortest paths and Pareto-optimal paths.

Lemma 5.2
A shortest path is Pareto-optimal if it is optimal for a non-degenerate preference α.

Proof Assume we find a shortest path π that is optimal for a non-degenerate α but that is not a Pareto-optimal path. By definition of Pareto-optimality one then finds another path π' with the cost vector $c(\pi')$ being less or equal to $c(\pi)$ in every dimension and being strictly less in at least one dimension. Hence, $c(\pi, \alpha) > c(\pi', \alpha)$, which is a contradiction to π being optimal for α. □

Definition 5.3
Given a graph $G(V, E)$ and a path π in G. The set $A(\pi) := \{\alpha \in \mathcal{P} \mid \pi \text{ is optimal for } \alpha\}$ is called the preference polyhedron *of π.*

The following lemma justifies the chosen term preference polyhedron.

Lemma 5.4
Given a graph $G(V, E)$. For any path π in G, the preference polyhedron $A(\pi)$ is convex and closed.

Proof Closure follows from the continuity of the aggregated cost. Regarding convexity, consider a path π that is optimal for $k \in \mathbb{N}$ preferences $\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(k)}$ and consider any convex combination $\beta := \sum_{i=1}^{k} \gamma_i \alpha^{(i)}$ of the preferences induced by a vector $\gamma \in [0,1]^k$ with $\sum \gamma_i = 1$. We claim that π is also optimal for preference choice β. The cost of π with preference β can be written as

$$c(\pi, \beta) = \gamma_1 c(\pi, \alpha^{(1)}) + \cdots + \gamma_k c(\pi, \alpha^{(k)})$$

Assume there is a path π' with $c(\pi', \beta) < c(\pi, \beta)$. But then, since π is optimal for each preference $\alpha^{(i)}$, we know that for every summand we have $\gamma_i c(\pi, \alpha^{(i)}) \leq \gamma_i c(\pi', \alpha^{(i)})$, which is a contradiction to our assumption $c(\pi', \beta) < c(\pi, \beta)$. Thus, π is also optimal for any such β and $A(\pi)$ is convex. □

Lemma 5.5
Given a graph $G(V, E)$. A path π in G is a unique shortest path if and only if $A(\pi)$ has non-zero volume.

Proof Given a shortest path $\pi(s, t)$. If π is a unique shortest path we find a preference α such that $c(\pi, \alpha)$ is strictly less than the aggregated cost of all other st-paths. Since the aggregated cost is continuous regarding the preference α we can find an $\epsilon > 0$ such that the sphere with radius ϵ and midpoint α is contained in $A(\pi)$.

On the other hand, if $A(\pi)$ has non zero volume, one can find a sphere within $A(\pi)$ that does not touch the boundary of $A(\pi)$. Obviously, for any preference in this sphere π is the unique shortest st-path. □

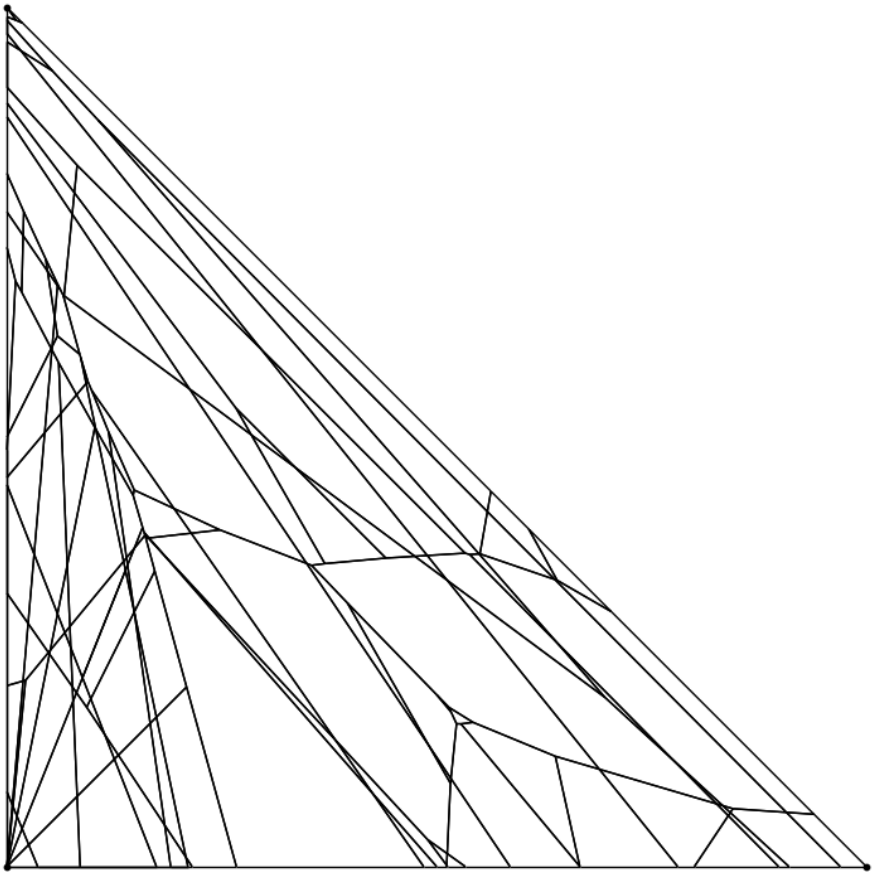


Figure 5.1.: Example preference space subdivision from a real world street network for bicyclists in Baden-Württemberg, Germany. The graph has the cost types unsuitability for bikes, distance and ascent. Each cell in the subdivision is convex and corresponds to a unique shortest path π. We call such a cell preference polyhedron $A(\pi)$. The framing triangle is the preference space $\mathcal{P}_3$. Each corner of the triangle corresponds to a preference where only one metric has a non-zero weight (and therefore weight 1).

Definition 5.6
Given a graph $G(V, E)$ and two nodes s and t in V. The set

$$\mathcal{A}(s, t) := \{A(\pi(s, t)) \mid \pi \text{ is a unique shortest path}\}$$

is called the preference space subdivision from s to t.

From Lemma 5.5 and the fact that preference polyhedra are closed it is clear that $\cup_{A \in \mathcal{A}(s,t)} A = \mathcal{P}$. It follows that for any preference α one can find a unique shortest path that is optimal for α. This cover property makes it especially interesting to find bounds on the number of unique shortest paths. Figure 5.1 shows an example of a preference space subdivision. It was created using the Corner Cutting Approach in Chapter 4 with a real-world street network for bicyclists. The source and target node were randomly chosen.

5.3. Bounds on the Number of Shortest Paths

In this section we investigate upper and lower bounds on the number of shortest paths and unique shortest paths. Since each shortest path is also Pareto-optimal, we implicitly obtain results for Pareto-optimal paths as well. We start with the definition of a graph family we call *multi-edge path graph* or simply *path graph*.

5.3.1. Multi-edge Path Graphs

Definition 5.7
A multi-edge path graph is a directed graph $G(V, E)$ with $V := \{v_1, v_2, \ldots, v_n\}$ and the property $e := (v_i, v_j) \in E \Rightarrow j = i + 1$. Multiple edges between the same pair of nodes are allowed.

Figure 5.2 shows an example path graph. The nodes of a path graph G form a line and each node can only have outgoing edges to its immediate right neighbor. This property makes the path graph especially easy to analyze. Note that if the number of edges between two nodes is $O(1)$, it is easy to transform this graph into an equivalent (non-path) graph without parallel edges but still $O(|V|)$ nodes and $O(|E|)$ edges.

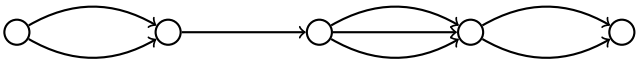


Figure 5.2.: Example path graph

It is a well known result that there can be exponentially many shortest paths with respect to the graph size (see, for instance, [Han79]). Nonetheless, for the sake of completeness we give a proof of the following lemma.

Lemma 5.8
There are 2-metric path graphs with n nodes and $2n-2$ edges that have $\Theta(2^n)$ shortest paths with different cost vectors between v_1 and v_n. The same bound holds for Pareto-optimal paths.

Proof We consider a path graph G with n nodes and two outgoing edges at each node except the last one v_n (which, by definition, cannot have outgoing edges) and some arbitrary $M > 0$. For each of the $2n-2$ edges we choose some rational value $z \in [0, M]$ uniformly at random and set its 2-dimensional cost vector to $(z, M-z)$. Clearly, for any path π of the 2^{n-1} st-paths in this graph, the sum of the aggregated costs is $c(\pi)_1 + c(\pi)_2 = M(n-1)$. Yet, with probability zero two of them have the same cost vector. For preference $\alpha = (0.5, 0.5)$ each one of these paths is optimal and, thus, Pareto-optimal (see Lemma 5.2). The statement follows. □

See Figure 5.3 for a path graph illustrating the previous Lemma together with the respective configuration in the cost space.

5.3.2. Preference Spaces of multi-edge Path Graphs

In the following, we will show that unique shortest paths behave quite differently. While there can be exponentially many Pareto-optimal or shortest paths in path graphs with only 2-metrics according to Lemma 5.8, we show that in a d-metric path graph with n nodes and maximum node (out)degree Δ

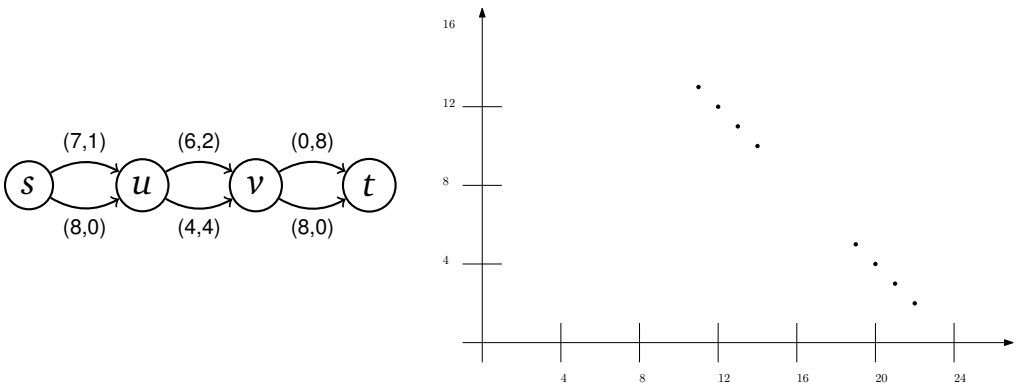


Figure 5.3.: Illustration of Lemma 5.8 for $M = 8$. Each path from s to t in the path graph corresponds to a distinct point on the line $y = 24 - x$ in the cost space on the right. They are all optimal for $\alpha^T = (0.5, 0.5)$. Only $(11, 13)$ (e.g., for $\alpha^T = (1, 0)$) and $(22, 2)$ (e.g., for $\alpha^T = (0, 1)$) are unique shortest.

there are $O(\left(n\Delta^2\right)^{d-1})$ unique shortest paths between v_1 and v_n. This upper bound is exponential in the dimension but polynomial if the dimension is fixed.

The upper bound proof will essentially bound the complexity of the preference space subdivision for a path graph. In the following we will use our definition of a preference polyhedron (Definition 5.3) also for the base case of a path formed by a single edge $e = (v, w)$. Here $A(e)$ describes the choices of α for which e is optimal when compared to other potential paths/edges between v and w.

Lemma 5.9
Given a path graph $G(V, E)$ and a path π in G. Then for the preference polyhedron $A(\pi)$ we have

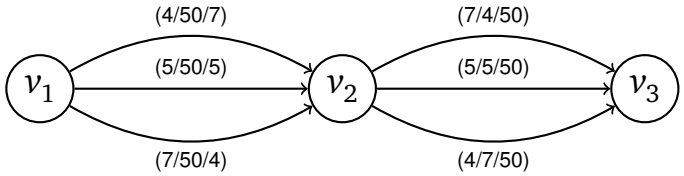
$$A(\pi) = \bigcap_{e \in \pi} A(e).$$

Proof Let α be any preference in $A(\pi(s, t))$. This always implies that each subpath of π is optimal for α as well. Hence, $\alpha \in \bigcap_{e \in \pi} A(e)$. We now assume that $\left(\bigcap_{e \in \pi} A(e)\right) \setminus A(\pi)$ is not empty and let $\tilde{\alpha}$ be a preference in it. Hence, we have that each edge in π is optimal for $\tilde{\alpha}$ but not the complete path π. Let $\tilde{\pi}$ be the shortest path from s to t with respect to $\tilde{\alpha}$. As G is a path graph, the edges in π and $\tilde{\pi}$ connect the same nodes. Therefore, since all edges in π and $\tilde{\pi}$ are optimal for $\tilde{\alpha}$, it holds

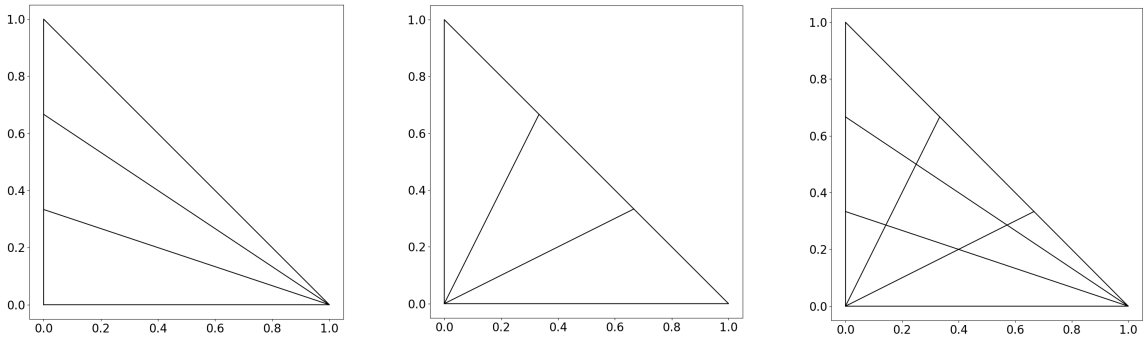
$$c(\pi, \tilde{\alpha}) - c(\tilde{\pi}, \tilde{\alpha}) = 0,$$

which is a contradiction. □

Lemma 5.9 implies that in a path graph G we obtain the preference space subdivision $\mathcal{A}(v_1, v_n)$ by computing the overlay of $\mathcal{A}(v_1, v_2)$, $\mathcal{A}(v_2, v_3)$ and so forth until $\mathcal{A}(v_{n-1}, v_n)$ as illustrated in Figure 5.4. It is easy to see that this property does not hold for general graphs. If we add an edge from v_1 to v_3 that forms an unique optimal path π^*, it will not affect $\mathcal{A}(v_1, v_2)$ and $\mathcal{A}(v_2, v_3)$, but $A(\pi^*)$ will be part of $\mathcal{A}(v_1, v_3)$. This property of path graphs is the main ingredient to the proof of Lemma 5.10, which bounds the number of unique shortest paths by bounding the complexity of the preference space subdivision.



(a) Example path graph with 9 unique shortest paths



(b) $\mathcal{A}(v_1, v_2)$ (c) $\mathcal{A}(v_2, v_3)$ (d) $\mathcal{A}(v_1, v_3)$

Figure 5.4.: Example of a path graph preference subdivision. The preference space subdivision $\mathcal{A}(v_1, v_3)$ is the intersection of $\mathcal{A}(v_1, v_2)$ and $\mathcal{A}(v_2, v_3)$.

Lemma 5.10
Given a d-metric path graph $G(V, E)$. There are $O\left(\left(n\Delta^2\right)^{d-1}\right)$ unique shortest paths between any two nodes in G, where Δ is the maximum node (out)degree in G.

Proof We first consider two consecutive nodes v_i and v_{i+1} and their preference space subdivision $\mathcal{A}(v_i, v_{i+1})$. For each pair of edges e_1, e_2 from v_i to v_{i+1} there is a hyperplane

$$H(e_1, e_2) := \{\alpha \in \mathcal{P}_d \mid c(e_1, \alpha) - c(e_2, \alpha) = 0\}.$$

All boundaries of preference polyhedra in $\mathcal{A}(v_i, v_{i+1})$ are supported by such hyperplanes. Since there are $O(\Delta^2)$ edge pairs from v_i to v_{i+1}, the preference polyhedra in $\mathcal{A}(v_i, v_{i+1})$ are separated by $O(\Delta^2)$ hyperplanes. From Lemma 5.9 it follows that the preference space subdivision $\mathcal{A}(v_1, v_n)$ is the overlay of $\mathcal{A}(v_1, v_2)$, $\mathcal{A}(v_2, v_3), \ldots, \mathcal{A}(v_{n-1}, v_n)$. Therefore, the preference polyhedra in $\mathcal{A}(v_1, v_n)$ are separated by $O(n\Delta^2)$ hyperplanes. Considering $\mathcal{A}(v_1, v_n)$ as an arrangement with $O(n\Delta^2)$ hyperplanes it follows that there are $O(\left(n\Delta^2\right)^{d-1})$ cells or preference polyhedra in $\mathcal{A}(v_1, v_n)$ (recall that the preference space $\mathcal{P}_d$ is $(d-1)$-dimensional). □

Interestingly, a similar result can be found in [GS93] regarding the complexity of the Minkowski sum of convex polytopes. Given a path graph $G(V, E)$ with n nodes, the Minkowski sum $C := \oplus_{i=1}^{n} E_i$, where E_i is the set of cost vectors of the edges from node v_i to node v_{i+1}, is equal to the set of cost vectors of all possible paths from v_1 to v_n. Chapter 2 shows that the cost vector of a unique shortest path π is a vertex of the convex hull of C. Thus, the complexity of the convex hull of C is an upper bound of $|\mathcal{A}(v_1, v_n)|$. In [GS93] it is shown that having n d-dimensional convex polytopes $P_1, P_2, \ldots, P_n$ with at most Δ vertices each, then the convex hull of $\oplus_i P_i$ has $O(n^{d-1}\Delta^{2d-2})$ vertices, which is the same upper bound as in Lemma 5.10. On the other hand, in Lemma 5.8 we show that there can be

exponentially many shortest paths between v_1 and v_n all with distinct cost vectors. Hence, while $|C|$ can be exponential in n, the convex hull of C is polynomial in n for fixed d.

We show that the upper bound in Lemma 5.10 is tight up to the factor Δ^{2d-2}. We show this by essentially constructing an arbitrary arrangement of n hyperplanes within the preference space.

Let us consider a path graph $G(V, E)$ with n nodes and two outgoing edges at each node except the end node. Thus, we have exactly one edge pair per consecutive pair of nodes and therefore also one hyperplane. The task is now to choose the edge costs in a way to maximize the number of preference polyhedra in $\mathcal{A}(v_1, v_n)$. It is well known that in general position n hyperplanes in $d-1$ dimensions induce an arrangement with $\Theta(n^{d-1})$ cells. Our first step to use this result is to show that with two edges we can create any hyperplane. We do that via the halfspaces that are induced by such hyperplanes.

Lemma 5.11
For any halfspace $h \subset \mathbb{R}^{d-1}$ with $h \cap \mathcal{P}_d \neq \emptyset$, there exist a path graph with two nodes, two edges and respective edge cost vectors such that $A(e_1) \subset h$ and $A(e_2) \cap h = \emptyset$.

Proof The inequality $c(e_1, \alpha) \leq c(e_2, \alpha)$ holds for all $\alpha \in A(e_1)$. We rewrite it as $\sum_i^d \alpha_i \cdot (c(e_1)_i - c(e_2)_i) \leq 0$. By setting $\alpha_d = 1 - \sum_i^{d-1} \alpha_i$, we get the halfspace in P_d. It describes a $(d-1)$-dimensional halfspace of the form $\sum_i \alpha_i \cdot a_i \leq b$, where the factors are $a_i = c(e_1)_i - c(e_2)_i - c(e_1)_d + c(e_2)_d$ and $b = c(e_2)_d - c(e_1)_d$. Clearly, we can choose cost vectors for e_1 and e_2 that form any a_i and b. □

All that remains is to show that having n hyperplanes there is an arrangement with $\Omega(n^{d-1})$ cells within the preference space. Recall that the preference space is a $(d-1)$-dimensional simplex.

Theorem 5.12
There are d-metric path graphs with n nodes, $2n-2$ edges and $\Theta\left(n^{d-1}\right)$ unique shortest paths between two nodes.

Proof We know that in general position $(d-1)$-dimensional arrangements with n hyperplanes have $\Theta\left(n^{d-1}\right)$ cells. From Lemma 5.11 we know that we can translate any arrangement within the preference space into a path graph with two outgoing edges per node. We need one node and two edges per hyperplane (plus one end node). All that remains is to show that there are arrangements with $\Theta\left(n^{d-1}\right)$ cells *within* the preference space. Since the preference space $\mathcal{P}_d$ is a $(d-1)$-dimensional simplex we can place a $(d-1)$-dimensional sphere in it with positive volume. Having any arrangement with n hyperplanes and $\Theta\left(n^{d-1}\right)$ vertices we can scale it down such that all vertices of the arrangement fit into this sphere. Clearly, with this new arrangement we still have n hyperplanes and $\Theta\left(n^{d-1}\right)$ cells in the preference space. □

So path graphs seem to be somewhat well understood in terms of the number of Pareto-optimal and unique shortest paths. While for the former, we easily obtain an exponential (in n) number, for the latter we have essentially tight upper and lower bounds for constant outdegree. The number of unique shortest paths is polynomial for fixed d with an exponential dependency on d.

5.3.3. Bounds for General Graphs

Let us now consider general graphs and aim for upper bounds. In this, we follow the approach of [Gus80], but do not restrict our proof to $d = 2$. Since most graphs do not contain multi-edges, we

restrict our considerations to this case. First, we introduce the family of layered graphs. We then show that using the bounds on the path graph we get subexponential bounds in the graph size for this family as well and extend those bounds to general multi-metric graphs.

5.3.3.1. Layered Graphs

Definition 5.13
A layered graph $L(V, E, r, c)$ is a graph without multi-edges that is partitioned into r disjoint node sets $V_1, V_2, \ldots, V_r$ with c nodes each such that any edge $(v, u) \in E$ connects nodes of consecutive node sets (also called layers *of L), i.e., there is an index i with $v \in V_i$ and $u \in V_{i+1}$.*

See Figure 5.5 for an example layered graph. The layered graph is similar to the path graph in the sense that the edges point only in one direction and connect two consecutive rows. In fact, a path graph without multi-edges is a layered graph with one column. However as we show in this section, the layered graph is more powerful in terms of generalization.

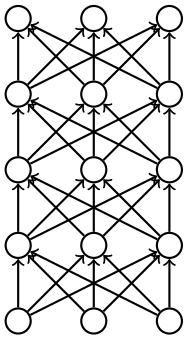


Figure 5.5.: Similar to the path graph, in the layered graph only consecutive rows are connected. Example layered graph with $c = 3$ and $r = 5$.

Lemma 5.14
Given a d-metric layered graph $L(V, E, r, c)$. There are $O(r^d c^{2d\sqrt{r}})$ unique shortest paths between a node in the first and a node in the last layer of L.

Proof Let $T(r, c)$ be the maximum possible number of unique shortest paths between a node v_1 of the first layer and a node v_r of the last layer of any layered graph $L(V, E, r, c)$. Our approach to find an upper bound for T is to decompose L into path graphs and to use Lemma 5.10 for each of them. It is clear that for two numbers $r_1 < r_2$ and a fixed number c it holds $T(r_1, c) \leq T(r_2, c)$. Let $r_0 = 1 < r_1 < \cdots < r_k = r$ be $k+1$ layers of the graph L with $r_i - r_{i-1} \leq \lceil \frac{r}{k} \rceil$ for each $1 \leq i \leq k$. Thus, from a node $v_{r_{i-1}}$ in layer r_{i-1} to a node v_{r_i} in layer r_i there are at most $T\left(\lceil \frac{r}{k} \rceil + 1, c\right)$ unique shortest paths. We remove all layers that do not belong to the $k+1$ chosen layers and connect the $k+1$ layers with the unique shortest paths between consecutive layers (see Figure 5.6 for an illustration). In that way, all unique shortest paths between the first and the last layer are preserved. In the new graph there are c^{k-1} node sequences to reach v_r from v_1. Each node sequence can be considered as a path graph with $k+1$ nodes and maximum node degree $\Delta \leq T\left(\lceil \frac{r}{k} \rceil + 1, c\right)$. Hence, with Lemma 5.10 we get

$$T(r, c) \in O\left(c^{k-1}\left(k \cdot T\left(\left\lceil \frac{r}{k} \right\rceil + 1, c\right)^2\right)^{d-1}\right).$$

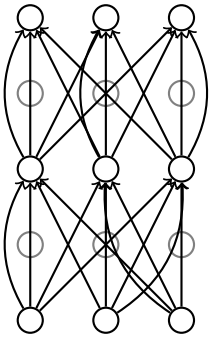


Figure 5.6.: Example layered graph with two skipped layers. Consecutive remaining layers are connected with one edge per unique shortest path in the graph without skipped layers in order to preserve all unique shortest paths from the first to the last layer.

Setting $k = \lceil \sqrt{r} \rceil$ we get

$$\begin{aligned} & T(r,c) \in O\left(c^{\lceil \sqrt{r} \rceil - 1}\left(\lceil \sqrt{r} \rceil \cdot T\left(\lceil \sqrt{r} \rceil + 1, c\right)^2\right)^{d-1}\right) \\ \Rightarrow\ & T(r,c) \in O\left(c^{\sqrt{r}} \cdot r^{\frac{d-1}{2}} \cdot T\left(\lceil \sqrt{r} \rceil + 1, c\right)^{2(d-1)}\right). \end{aligned}$$

A trivial upper bound for $T(r,c)$ is c^{r-2} as this is the number of all possible paths. Thus, with $T(\lceil \sqrt{r} \rceil + 1, c) \leq c^{\sqrt{r}}$ it follows

$$T(r,c) \in O\left(c^{\sqrt{r}} \cdot r^{\frac{d-1}{2}} \cdot c^{2(d-1)\sqrt{r}}\right) \Rightarrow T(r,c) \in O\left(r^d c^{2d\sqrt{r}}\right). \qquad \square$$

5.3.3.2. Extending Upper Bounds to general Graphs

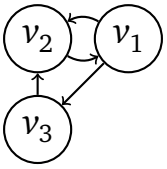
To be able to generalize our results from the previous section, we first show how to represent any graph without multi-edges as a layered graph without loosing unique shortest paths in the process. Then we use this representation to derive general bounds.

Definition 5.15
Given any graph $G(V, E)$ with n nodes, m edges and without multi-edges. The layered graph L of G has n rows and n columns. Each row of L consists of one copy of V and there is one copy of E between each two consecutive rows. Each edge points from one row to the next while the end nodes are the same as in G. Hence, L has n^2 nodes and $(n-1) \cdot m$ edges. For a node $v_i \in V$ we write $v_{i,r}$ for its copy in the r-th row of L.

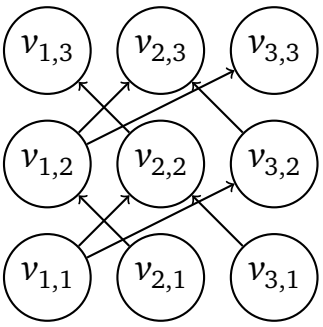
See Figure 5.7a and Figure 5.7b for an example. The following Lemma allows us to focus on layered graphs regarding general upper bounds.

Lemma 5.16
Given a graph $G(V, E)$ without multi-edges and its layered graph L. Then for any two nodes $v_i, v_j \in V$ it holds

$$|\mathcal{A}(v_i, v_j)| \leq \sum_{r=1}^{n} |\mathcal{A}(v_{i,1}, v_{j,r})|.$$



(a) Example graph $G(V, E)$ with $n = 3$ nodes and 4 edges



(b) The layered graph L of G has n rows, one copy of V per row and one copy of E between each two consecutive rows.

Figure 5.7.: Figures 5.7a and 5.7b show an example how a general graph G is translated into a layered graph L.

Proof Any unique shortest path $\pi(v_i, v_j)$ in G consists of at most $n-1$ edges. Hence, we can map all unique shortest paths π of G injectively to L by starting in the first row and by following the copies of π's edges in L. Thus, if π has $k-1$ edges, we stop in row k. Let $\pi'(v_{i,1}, v_{j,k})$ be the path in L obtained in this way from π. We now show that for any preference $\alpha \in A(\pi)$ it holds $\alpha \in A(\pi')$ and, thus, that π' is a unique shortest path in L. Let us assume that there is a preference $\alpha \in A(\pi)$ that is not in $A(\pi')$. Then we can find another path $\pi''(v_{i,1}, v_{j,k})$ in L that is optimal for α. However, since there is a path in G from v_i to v_j with the same cost vector as π'' this is a contradiction to π being optimal for α. □

Theorem 5.17
In any d-metric graph $G(V, E)$ with n nodes there are $O(n^{2d\sqrt{n}+d+1})$ unique shortest paths between any two nodes.

Proof Let L be the layered graph of G. Recall that L has n rows and columns. Let $v_i, v_j \in V$. From Lemma 5.14 we know that for any row r there are $O\left(n^{2d\sqrt{n}+d}\right)$ unique shortest paths between $v_{i,1}$ and $v_{j,r}$. With Lemma 5.16 it follows that there are $O\left(n^{2d\sqrt{n}+d+1}\right)$ unique shortest paths between v_i and v_j in G. □

So we obtain an upper bound for unique shortest paths significantly below the lower bound for shortest paths in general (see Lemma 5.8), hence establishing a significant gap.

5.4. Conclusion

In this chapter we gave upper and lower bounds on the maximum number of 'optimal' paths in multicriteria networks. In case of Pareto-optimality, it is well known that exponentially (in the number of nodes) many optimal paths exist. We show that for a commonly used model of linear aggregation (very popular, e.g., for personalized route planning) the number of optimal paths is subexponential in the graph size, yet possibly exponential in the number of criteria. To this end, we utilized two graph families, the path and layered graphs, first proving bounds for these graph families and then extending the bounds to general graphs. Our arguments are highly dependend on a geometric view on the set of all possible preferences, in particular the complexity of a respective decomposition.

We suspect, though, that for general graphs, our upper bound is still too pessimistic, in particular since in practice, it is often possible to actually enumerate all 'optimal' paths in personalized route planning contexts. One might also try to identify additional properties of the network that allow for stronger upper bounds. On a technical level, we have a feeling that the structure of the preference space for path graphs is rather well understood, yet for general graphs, it is still somewhat unclear, whether it can also be modeled in a compositional way.

Chapter 6
Discussion

6.1. Conclusion

Throughout this work, we have showcased three different practical application of the personalized route planning model. Namely, finding alternative routes, identifying intermediate destinations and clustering trajectories by preference. In Chapter 2 we presented an algorithm that enumerates personalized routes. Even after filtering to avoid too much overlap between the routes, our algorithm can produce a large number of alternative routes that also fulfilled our expectation in a qualitative assessment. In contrast to most previous approaches, these routes do not rely on heuristics but are still optimal routes as defined by our model. Also, we refined our algorithm to efficiently skip many routes that are to expensive in important metrics as defined by the user achieving significant speed up depending on the parameters and improving our result quality. The final contribution of the chapter is a general 'invented' metric that serves to diversify alternative routes in case the natural metrics for the use case are too similar.

Chapter 3 contains a trajectory segmentation approach which aims to discover intermediate destinations in trajectories. We assumed that drivers follow personalized paths and segmented the trajectories into a minimal number of optimal paths. In a real world data set, we were able to recover $\approx 60\%$ of known intermediate destinations exactly with this approach. Most intermediate destinations ($\approx 90\%$) were within a hop distance of two to our predicted intermediate destination. With our experiments we showed that having multiple cost types not only increases segmentability in comparison to relying only on travel time but also improves overall segmentation quality with regard to recovering intermediate destinations.

In Chapter 4 we demonstrated how to cluster trajectories by preferences. By first (approximately) computing the preference spaces for which the trajectories are optimal and then solving the geometric hitting set instance consisting of these spaces, we achieve fast running times in practice although the underlying problems are of high theoretical worst-case complexity. Especially, for the preference space we showed that it can be arbitrary complex in theory but is of constant complexity in our experiments so our approach to exactly construct them is more efficient than our approximation approach.

Furthermore in Chapter 5, we presented a deeper insight in the theoretical working of the model and found that a bound for the number of optimal paths that is subexponential in the graph size

exists.

Our applications showed that given suitable cost types, the model works well in a bicycle and in car scenarios. It gives good results in reasonable running times but is not quite fast enough for large scale online usage. The biggest drawback for using the model in the way we used it lies in the scalability of the PCH speed-up technique. Running times increase drastically with the dissimilarity of the used cost types as well as the size of the graph.

6.2. Future Work

There are multiple interesting direction for future work regarding this route planning model. The most important direction for the practical applicability is to find speed-up techniques that scale better with dissimilar cost types and graph size or improve PCH. An interesting starting point for this can be the *personalizable route planning* based on the customizable route planning speed-up technique presented in [FS15]. As it uses a partitioning of the graph, it might be less susceptible to the combinatorial increase that comes with larger graphs.

Another opportunity for future work are the theoretical foundations of the model and bounds for the number of optimal paths. We expect our current upper bound to be far from tight. Better insights here might be an important stepping stone to give good lower bounds for algorithms and improving the speed-up techniques mentioned before.

Furthermore, we expect there to be more practical applications for which the model is suitable.

Bibliography

[18] *OpenStreetMap*. en-US. Mar. 2018. URL: https://www.openstreetmap.org/ (visited on 03/29/2018) (cit. on pp. 33, 46).

[ABB+14] S. P. Alewijnse, K. Buchin, M. Buchin, A. Kölzsch, H. Kruckenberg, M. A. Westenberg. 'A framework for trajectory segmentation by stable criteria'. In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 2014, pp. 351–360 (cit. on pp. 58, 59).

[ADGW13] I. Abraham, D. Delling, A. V. Goldberg, R. F. Werneck. 'Alternative routes in road networks'. In: *ACM Journal of Experimental Algorithmics* 18 (2013). URL: http://doi.acm.org/10.1145/2444016.2444019 (cit. on pp. 25, 28).

[AHV+05] P. K. Agarwal, S. Har-Peled, K. R. Varadarajan, et al. 'Geometric approximation via coresets'. In: *Combinatorial and computational geometry* 52 (2005), pp. 1–30 (cit. on p. 80).

[AKT13] O. Andersen, B. B. Krogh, K. Torp. 'An Open-source Based ITS Platform'. In: *Proc. of MDM*. Vol. 2. 2013, pp. 27–32 (cit. on p. 61).

[AMS06] N. Alon, D. Moshkovitz, S. Safra. 'Algorithmic construction of sets for k-restrictions'. In: *ACM Trans. Algorithms* 2.2 (2006), pp. 153–177 (cit. on p. 75).

[And12] Andrew Makhorin. *GLPK - GNU Project - Free Software Foundation (FSF)*. 2012. URL: https://www.gnu.org/software/glpk/glpk.html (visited on 06/11/2018) (cit. on pp. 37, 83).

[Bar18] F. Barth. 'Multi-criteria Bicycle Routing'. MA thesis. University of Stuttgart, 2018. URL: http://dx.doi.org/10.18419/opus-10243 (cit. on pp. 27, 28).

[Bat14] G. V. E. Batz. 'Time-Dependent Route Planning with Contraction Hierarchies'. PhD thesis. Karlsruhe Institute of Technology, 2014 (cit. on p. 15).

[BCK+10] R. Bauer, T. Columbus, B. Katz, M. Krug, D. Wagner. 'Preprocessing speed-up techniques is hard'. In: *International Conference on Algorithms and Complexity*. Springer. 2010, pp. 359–370 (cit. on p. 15).

[BDG+15] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, R. F. Werneck. 'Route Planning in Transportation Networks'. In: *CoRR* abs/1504.05140 (2015). arXiv: 1504.05140. URL: http://arxiv.org/abs/1504.05140 (cit. on p. 23).

[BDGS11] R. Bader, J. Dees, R. Geisberger, P. Sanders. 'Alternative Route Graphs in Road Networks'. In: *Proceedings of the First International ICST Conference on Theory and Practice of Algorithms in (Computer) Systems*. TAPAS'11. Rome, Italy: Springer-Verlag, 2011, pp. 21–32. URL: http://dl.acm.org/citation.cfm?id=1987334.1987339 (cit. on pp. 24, 27, 28).

[BDVS11] M. Buchin, A. Driemel, M. Van Kreveld, V. Sacristán. 'Segmenting trajectories: A framework and algorithms using spatiotemporal criteria'. In: *Journal of Spatial Information Science* 2011.3 (2011), pp. 33–63 (cit. on pp. 58, 59, 63, 64).

[BF19] F. Barth, S. Funke. ‘Alternative Routes for Next Generation Traffic Shaping’. In: *Proceedings of the 12th ACM SIGSPATIAL International Workshop on Computational Transportation Science*. ACM, Nov. 2019. URL: https://doi.org/10.1145%2F3357000.3366141 (cit. on p. 23).

[BFJP20] F. Barth, S. Funke, T. S. Jepsen, C. Proissl. ‘Scalable unsupervised multi-criteria trajectory segmentation and driving preference mining’. In: *BigSpatial@SIGSPATIAL*. ACM, 2020, 6:1–6:10 (cit. on p. 57).

[BFP21] F. Barth, S. Funke, C. Proissl. ‘Preference-based Trajectory Clustering: An Application of Geometric Hitting Sets’. In: *32nd International Symposium on Algorithms and Computation (ISAAC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2021 (cit. on pp. 73, 90).

[BFS19] F. Barth, S. Funke, S. Storandt. ‘Alternative Multicriteria Routes’. In: *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, Jan. 2019, pp. 66–80. URL: https://doi.org/10.1137%2F1.9781611975499.6 (cit. on pp. 23, 90).

[BG95] H. Brönnimann, M. T. Goodrich. ‘Almost optimal set covers in finite VC-dimension’. In: *Discrete & Computational Geometry* 14.4 (1995), pp. 463–479 (cit. on p. 75).

[BKST78] J. L. Bentley, H. T. Kung, M. Schkolnick, C. D. Thompson. ‘On the Average Number of Maxima in a Set of Vectors and Applications’. In: *J. ACM* 25.4 (Oct. 1978), pp. 536–543. URL: https://doi.org/10.1145/322092.322095 (cit. on p. 90).

[CBGL15] T. Chondrogiannis, P. Bouros, J. Gamper, U. Leser. ‘Alternative Routing: K-shortest Paths with Limited Overlap’. In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPATIAL ’15. Seattle, Washington: ACM, 2015, 68:1–68:4. URL: http://doi.acm.org/10.1145/2820783.2820858 (cit. on pp. 24, 28).

[Dan51] G. B. Dantzig. ‘Maximization of a Linear Function of Variables Subject To Linear Inequalities’. In: *Activity analysis of production and allocation* 13 (1951), pp. 339–347 (cit. on p. 16).

[DDP19] D. Delling, J. Dibbelt, T. Pajor. ‘Fast and Exact Public Transit Routing with Restricted Pareto Sets’. In: *2019 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*. 2019, pp. 54–65. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611975499.5. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611975499.5 (cit. on p. 43).

[DGG+15] D. Delling, A. V. Goldberg, M. Goldszmidt, J. Krumm, K. Talwar, R. F. Werneck. ‘Navigation Made Personal: Inferring Driving Preferences from GPS Traces’. In: *Proc. of SIGSPATIAL’15*. Seattle, Washington: Association for Computing Machinery, 2015 (cit. on pp. 57, 61).

[DHJ17] O. Devillers, S. Hornus, C. Jamin. ‘dD Triangulations’. In: *CGAL User and Reference Manual*. 4.11. CGAL Editorial Board, 2017. URL: http://doc.cgal.org/4.11/Manual/packages.html#PkgTriangulationsSummary (cit. on pp. 33, 45).

[Dij59] E. W. Dijkstra. ‘A Note on Two Problems in Connexion with Graphs’. In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271 (cit. on p. 14).

[DW09] D. Delling, D. Wagner. ‘Pareto Paths with SHARC’. In: *Proc. 8th International Symposium on Experimental Algorithms (SEA)*. Vol. 5526. Lecture Notes in Computer Science. Springer, 2009, pp. 125–136 (cit. on pp. 26, 28, 29, 90).

[Dwy88] R. A. Dwyer. ‘On the Convex Hull of Random Points in a Polytope’. In: *Journal of Applied Probability* 25.4 (1988), pp. 688–699. URL: http://www.jstor.org/stable/3214289 (cit. on p. 90).

[DYGD15] J. Dai, B. Yang, C. Guo, Z. Ding. ‘Personalized route recommendation using big trajectory data’. In: *2015 IEEE 31st international conference on data engineering*. IEEE. 2015, pp. 543–554 (cit. on p. 60).

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Vol. 10. EATCS Monographs on Theoretical Computer Science. Springer, 1987 (cit. on p. 78).

[EJH+19] M. Etemad, A. S. Júnior, A. Hoseyni, J. Rose, S. Matwin. 'A Trajectory Segmentation Algorithm Based on Interpolation-based Change Detection Strategies.' In: *EDBT/ICDT Workshops*. 2019 (cit. on pp. 58, 59).

[Epp94] D. Eppstein. 'Finding the k Shortest Paths'. In: *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 1994, pp. 154–165. URL: https://doi.org/10.1109/SFCS.1994.365697 (cit. on p. 24).

[FJ17] M. Fruensgaard, T. S. Jepsen. 'Improving Cost Estimation Models with Estimation Updates and road2vec: a Feature Learning Framework for Road Networks'. MA thesis. Aalborg University, 2017 (cit. on p. 62).

[FLS16] S. Funke, S. Laue, S. Storandt. 'Deducing individual driving preferences for user-aware navigation'. In: *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*. Ed. by S. Ravada, M. E. Ali, S. D. Newsam, M. Renz, G. Trajcevski. ACM, 2016, 14:1–14:9 (cit. on p. 74).

[FLS17] S. Funke, S. Laue, S. Storandt. 'Personal Routes with High-Dimensional Costs and Dynamic Approximation Guarantees'. In: *16th Int. Symp. on Experimental Algorithms (SEA 2017)*. Vol. 75. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany, 2017, 18:1–18:13 (cit. on pp. 17, 18, 33, 35, 46, 52, 65, 74).

[FMN05] S. Funke, K. Mehlhorn, S. Näher. 'Structural filtering: a paradigm for efficient and exact geometric programs'. In: *Comput. Geom.* 31.3 (2005), pp. 179–194 (cit. on p. 86).

[FNS16] S. Funke, A. Nusser, S. Storandt. 'On k-Path Covers and their applications'. In: *VLDB J.* 25.1 (2016), pp. 103–123 (cit. on p. 74).

[FPT81] R. J. Fowler, M. Paterson, S. L. Tanimoto. 'Optimal Packing and Covering in the Plane are NP-Complete'. In: *Inf. Process. Lett.* 12.3 (1981), pp. 133–137 (cit. on p. 75).

[FRC+07] Farr Tom G., Rosen Paul A., Caro Edward, Crippen Robert, Duren Riley, Hensley Scott, Kobrick Michael, Paller Mimi, Rodriguez Ernesto, Roth Ladislav, Seal David, Shaffer Scott, Shimada Joanne, Umland Jeffrey, Werner Marian, Oskin Michael, Burbank Douglas, Alsdorf Douglas. 'The Shuttle Radar Topography Mission'. In: *Reviews of Geophysics* 45.2 (May 2007). URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2005RG000183 (visited on 03/29/2018) (cit. on pp. 34, 46).

[FS15] S. Funke, S. Storandt. 'Personalized route planning in road networks'. In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA*. Ed. by J. Bao, C. Sengstock, M. E. Ali, Y. Huang, M. Gertz, M. Renz, J. Sankaranarayanan. ACM, 2015, 45:1–45:10 (cit. on pp. 17, 74, 90, 102).

[GJ+10] G. Guennebaud, B. Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010 (cit. on p. 45).

[GKS10] R. Geisberger, M. Kobitzsch, P. Sanders. 'Route Planning with Flexible Objective Functions.' In: *Proc. 12th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2010, pp. 124–137 (cit. on p. 74).

[GKW09] A. Goldberg, H. Kaplan, R. Werneck. 'Reach for A$*$: shortest path algorithms with preprocessing'. In: *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, July 2009, pp. 93–139. URL: https://doi.org/10.1090/dimacs/074/05 (cit. on p. 25).

[GLS81] M. Grötschel, L. Lovász, A. Schrijver. 'The ellipsoid method and its consequences in combinatorial optimization'. In: *Combinatorica* 1.2 (1981), pp. 169–197 (cit. on p. 16).

[GR19] K. Gajjar, J. Radhakrishnan. 'Parametric Shortest Paths in Planar Graphs'. In: *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, Nov. 2019. URL: https://doi.org/10.1109%2Ffocs.2019.00057 (cit. on p. 90).

[GS93] P. Gritzmann, B. Sturmfels. 'Minkowski addition of polytopes: computational complexity and applications to Gröbner bases'. In: *SIAM Journal on Discrete Mathematics* 6.2 (1993), pp. 246–269 (cit. on p. 95).

[GSSV12] R. Geisberger, P. Sanders, D. Schultes, C. Vetter. 'Exact routing in large road networks using contraction hierarchies'. In: *Transportation Science* 46.3 (2012), pp. 388–404 (cit. on pp. 14, 15, 25).

[Gus80] D. M. Gusfield. 'Sensitivity analysis for combinatorial optimization'. PhD thesis. University of California, Berkeley, 1980 (cit. on pp. 17, 90, 96).

[Han79] P. Hansen. 'Bicriterion path problems'. In: *Lecture Notes in Economics and Mathematical Systems* 177 (1979) (cit. on p. 93).

[HZ80] G. Y. Handler, I. Zang. 'A dual algorithm for the constrained shortest path problem'. In: *Networks* 10.4 (1980), pp. 293–309 (cit. on p. 89).

[JJN20] T. S. Jepsen, C. S. Jensen, T. D. Nielsen. 'Relational Fusion Networks: Graph Convolutional Networks for Road Networks'. In: *IEEE Transactions on Intelligent Transportation Systems* (2020), pp. 1–12 (cit. on pp. 61, 62).

[JJNT18] T. S. Jepsen, C. S. Jensen, T. D. Nielsen, K. Torp. 'On Network Embedding for Machine Learning on Road Networks: A Case Study on the Danish Road Network'. In: *Proc. of Big Data 2018*. 2018, pp. 3422–3431 (cit. on pp. 61, 63).

[Joh74] D. S. Johnson. 'Approximation algorithms for combinatorial problems'. In: *Journal of computer and system sciences* 9.3 (1974), pp. 256–278 (cit. on p. 82).

[JTR+18] A. S. Junior, V. C. Times, C. Renso, S. Matwin, L. A. Cabral. 'A semi-supervised approach for the semantic segmentation of trajectories'. In: *2018 19th IEEE International Conference on Mobile Data Management (MDM)*. IEEE. 2018, pp. 145–154 (cit. on p. 58).

[Kar84] N. Karmarkar. 'A New Polynomial-Time Algorithm for Linear Programming'. In: *Combinatorica* 4.4 (Dec. 1984), pp. 373–395. URL: https://doi.org/10.1007%2Fbf02579150 (cit. on p. 16).

[KJT16] B. Krogh, C. S. Jensen, K. Torp. 'Efficient in-memory indexing of network-constrained trajectories'. In: *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 2016, pp. 1–10 (cit. on p. 58).

[KM72] V. Klee, G. J. Minty. 'How Good Is the Simplex Algorithm'. In: *Inequalities* 3.3 (1972), pp. 159–175 (cit. on p. 16).

[KRS10] H.-P. Kriegel, M. Renz, M. Schubert. 'Route skyline queries: A multi-preference path planning approach'. In: *26th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 2010, pp. 261–272 (cit. on pp. 26, 28, 90).

[MR09] N. H. Mustafa, S. Ray. 'PTAS for geometric hitting set problems via local search'. In: *Symposium on Computational Geometry*. ACM, 2009, pp. 17–22 (cit. on p. 75).

[MS08] K. Mehlhorn, P. Sanders. *Algorithms and data structures: The basic toolbox*. Vol. 55. Springer, 2008 (cit. on p. 14).

[MW06] M. Müller-Hannemann, K. Weihe. 'On the cardinality of the Pareto set in bicriteria shortest path problems'. In: *Annals of Operations Research* 147.1 (2006), pp. 269–286 (cit. on p. 90).

[MZ00] K. Mehlhorn, M. Ziegelmann. 'Resource constrained shortest paths'. In: *European Symposium on Algorithms*. Springer. 2000, pp. 326–337 (cit. on p. 89).

[ONH17] J. Oehrlein, B. Niedermann, J. Haunert. 'Inferring the Parametric Weight of a Bicriteria Routing Model from Trajectories'. In: *SIGSPATIAL/GIS*. ACM, 2017, 59:1–59:4 (cit. on p. 74).

[PW00] F. A. Potra, S. J. Wright. 'Interior-Point Methods'. In: *Journal of computational and applied mathematics* 124.1-2 (2000), pp. 281–302 (cit. on p. 16).

[PYJ20] S. A. Pedersen, B. Yang, C. S. Jensen. 'Fast stochastic routing under time-varying uncertainty'. In: *The VLDB Journal* 29.4 (2020), pp. 819–839 (cit. on p. 57).

[SMT+15] A. Soares Júnior, B. N. Moreno, V. C. Times, S. Matwin, L. d. A. F. Cabral. 'GRASP-UTS: an algorithm for unsupervised trajectory segmentation'. In: *International Journal of Geographical Information Science* 29.1 (2015), pp. 46–68 (cit. on pp. 58, 59).

[SSZZ14] R. Song, W. Sun, B. Zheng, Y. Zheng. 'PRESS: A Novel Framework of Trajectory Compression in Road Networks'. In: *Proceedings of the VLDB Endowment*. Vol. 7. May 2014, pp. 661–672 (cit. on p. 58).

[The17] The CGAL Project. *CGAL User and Reference Manual*. 4.11. CGAL Editorial Board, 2017. URL: http://doc.cgal.org/4.11/Manual/packages.html (cit. on pp. 33, 45).

[The20] The CGAL Project. *CGAL User and Reference Manual*. 5.0.3. CGAL Editorial Board, 2020. URL: https://doc.cgal.org/5.0.3/Manual/packages.html (cit. on p. 83).

[Van20] R. J. Vanderbei. *Linear programming*. Springer, 2020 (cit. on p. 16).

[WBF+20] R. Wein, E. Berberich, E. Fogel, D. Halperin, M. Hemmer, O. Salzman, B. Zukerman. '2D Arrangements'. In: *CGAL User and Reference Manual*. 5.0.3. CGAL Editorial Board, 2020. URL: https://doc.cgal.org/5.0.3/Manual/packages.html#PkgArrangementOnSurface2 (cit. on p. 83).

[YGMJ15] B. Yang, C. Guo, Y. Ma, C. S. Jensen. 'Toward Personalized, Context-aware Routing'. In: *The VLDB Journal* 24.2 (Apr. 2015), pp. 297–318 (cit. on pp. 60, 71).

[Zhe15] Y. Zheng. 'Trajectory Data Mining: An Overview'. In: *ACM Trans. Intell. Syst. Technol.* 6.3 (May 2015), 29:1–29:41 (cit. on p. 57).

All URLs were last checked on 11.04.2022.

List of Figures

List of Tables