Universität Stuttgart

On Hierarchical Search and Preference-Based Routing in Transportation Networks

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
Claudius Daniel Proissl
aus Bietigheim-Bissingen

Hauptberichter:	Prof. Dr. Stefan Funke
Mitberichterin:	Prof. Dr. Hannah Bast
Tag der mündlichen Prüfung:	25. Mai 2023

Institut für Formale Methoden der Informatik

2023

Contents

Zusammenfassung

In dieser Arbeit werden verschiedene Problemstellungen der Routenplanung in Transportnetzwerken behandelt.

Im ersten Teil (Kapitel 2) liegt der Fokus auf Algorithmen zur Berechnung kürzester Pfade, die sich die typische hierarchische Struktur von Transportnetzwerken zunutze machen, um deutlich schneller Ergebnisse berechnen zu können als allgemeine Ansätze wie beispielsweise Dijkstras Algorithmus. Wir nennen diese Art von Algorithmen *hierarchische Suche*.

Wir beginnen in Abschnitt 2.3 mit der Vorstellung eines neuen Algorithmus, der Ideen aus zwei unterschiedlichen hierarchischen Ansätzen kombiniert, um die Effektivität in der Berechnung kürzester Wege deutlich zu erhöhen. Diese zusätzliche Effektivität erlaubt die Verwendung deutlich größerer Graphen als bisher. Wir sind beispielsweise dadurch in der Lage, Distanzen zwischen Knoten in einem Straßengraphen mit 1,2 Milliarden Kanten innerhalb von 15,2 Mikrosekunden zu berechnen.

In Abschnitt 2.4 stellen wir eine möglicherweise etwas überraschende Modifikation eines sehr erfolgreichen hierarchischen Ansatzes vor. Kontraktionshierarchien (engl. Contraction Hierarchies) ordnen die Knoten im Netzwerk nach Wichtigkeit und fügen zusätzliche Kanten hinzu, sogenannte Abkürzungen, um schneller kürzeste Wege berechnen zu können. Wir zeigen, wie hierarchische Suche auch ohne Abkürzungen gelingt. Stattdessen merken wir uns für jeden Knoten, ob er in der Kontraktionshierarchie Bestandteil wichtiger Abkürzungen ist oder nicht. Dadurch reduziert sich der Speicherbedarf auf ein Minimum und trotzdem kann eine erhebliche Verbesserung der Laufzeiten gegenüber Dijkstras Algorithmus beobachtet werden.

Wir schließen den ersten Teil dieser Arbeit ab mit einer theoretischen Analyse zur Komplexität der Berechnung kürzester Wege in Kontraktionshierarchien. Wir zeigen, dass die Anzahl besuchter Kanten die Anzahl besuchter Knoten deutlich übersteigen kann, selbst wenn das zugrundeliegende Transportnetzwerk günstige Eigenschaften besitzt. Diese Erkenntnis ist ein Beitrag in der Suche unterer Schranken für die Berechnung kürzester Pfade in Kontraktionshierarchien.

In Kapitel 3, dem zweiten Teil dieser Arbeit, wenden wir uns der präferenzbasierten Routenplanung zu. Als Basis unserer Überlegungen wählen wir ein lineares Präferenzmodell, das in [FS15a] vorgestellt wurde und das sowohl intuitiv als auch vielseitig einsetzbar ist. Dieses Modell erlaubt es uns, mehrere Kostentypen für die Kanten zu definieren und diese je nach Präferenz linear miteinander zu kombinieren.

Wir beginnen das Kapitel in Abschnitt 3.2 mit Überlegungen zur maximalen Anzahl kürzester Wege in dem genannten mehrdimensionalen Kostenraum. Es gelingt uns zu zeigen, dass diese Anzahl deutlich unter der Anzahl Pareto-optimaler Pfade liegt. Genauer zeigen wir, dass in einem Graph mit

n Knoten und d Kostentypen die Anzahl kürzester Pfade zwischen zwei Knoten in $n^{O\left(\log^{d-1} n\right)}$ liegt. Dieses Ergebnis ist eine Verallgemeinerung bekannter oberer Schranken für $d = 2$ und $d = 3$.

In den Abschnitten 3.3 und 3.4 stellen wir praxisnahe Problemstellungen und effektive Lösungsansätze im Bereich präferenzbasierter Routenplanung vor.

Zunächst präsentieren wir in Abschnitt 3.3 eine Methode zur Ermittlung von Routenpräferenzen von Verkehrsteilnehmern, wenn Daten zu bereits unternommenen Routen vorliegen. Die ermittelten Präferenzen können dann beispielsweise dafür verwendet werden, um neue Routenvorschläge zu generieren. Dieser Ansatz ist eine Erweiterung bestehender Verfahren, die nur mit Routen arbeiten können, die bestimmte Optimalitätskriterien erfüllen.

Wir schließen das Kapitel in Abschnitt 3.4 mit einem praktischen Ansatz zur Ermittlung von Präferenzmustern vieler Verkehrsteilnehmerinnen. Wir beschreiben Algorithmen, die für gegebene optimale Pfade eine Menge Präferenzen minimaler Größe berechnen, sodass jeder Pfad in der Menge optimal zu mindestens einer dieser Präferenzen ist. Es gelingt uns, dieses Problem selbst für relativ große Probleminstanzen optimal zu lösen.

Abstract

In this thesis we address problems in the context of optimizing and personalizing routing algorithms for transportation networks.

In the first part, Chapter 2, we discuss shortest path speed-up techniques that use the typical hierarchical structure of transportation networks to achieve better run times than standard approaches such as Dijkstra's algorithm. We call this branch of speed-up techniques *hierarchical search*.

In Section 2.3 we introduce a new approach of hierarchical search based on ideas of the two popular methods Contraction Hierarchies (CH) and Hierarchical Hub Labeling (HHL). Our algorithm achieves significantly better tradeoffs with respect to space consumption and query times than CH and HHL.

Algorithms in the context of hierarchical search typically create an overlay graph of shortcuts in order to improve the query times. The downside of this approach is that the overlay graph increases the space consumption. We look in Section 2.4 for a possibility to conduct hierarchical search without shortcuts. Our approach is based on CH and improves the query times by an order of magnitude compared to Dijkstra's algorithm while hardly causing any additional space consumption.

We conclude Chapter 2 with considerations regarding the query complexity of CH. We show that in a CH query the number of traversed edges can be almost the square of the number of visited nodes, even if the network shows a favorable property called bounded growth. This insight is a contribution to the effort of finding good lower bounds for the CH query complexity.

The second part of this thesis, Chapter 3, is dedicated to preference-based routing. We take an intuitive and versatile linear preference model introduced in [FS15a] as the basis of our considerations. The idea behind this model is to allow multiple edge cost types in the same network and to linearly combine them based on a preference vector.

We start in Section 3.2 with a theoretical analysis of the number of shortest paths in this multi-dimensional cost setting. Our calculations show that there are at most $n^{O\left(\log^{d-1} n\right)}$ shortest paths between any two nodes, where n is the number of nodes in the graph and d the number of cost types. This result generalizes previous upper bounds for $d = 2$ [Gus80] and $d = 3$ [GR19].

We continue with efficient solutions for practical applications regarding preference-based routing. We first show in Section 3.3 an approach to elicit the preference of a driver if past trajectories of this driver are available. This method is a generalization of an existing algorithm [FLS16] that specifically improves its robustness in terms of the input trajectory.

In the multi-dimensional setting, a shortest path typically is only optimal for a subset of all preferences. In Section 3.4 we address the problem of finding a set of preferences of minimum size such that each shortest path of a given set of trajectories is optimal for at least one of these preferences. Although it is unclear if this problem can be solved in polynomial time, we show that it can be solved efficiently in practice even for rather large problem instances.

Acknowledgments

I would like to thank my supervisor Stefan Funke for his great support and for providing me the perfect environment to concentrate on my ideas.

A special thanks goes to Daniel Bahrdt, Florian Barth, Tobias S. Jepsen, Sokol Makolli, Tobias Rupp, and Felix Weitbrecht for all the inspiring conversations and for having a good time. It's really been a pleasure working with you.

Furthermore, I would like to thank Kshitij Gajjar and Jaikumar Radhakrishnan for their very helpful input regarding Section 3.2.

Last but not least, I would like to thank my family and friends for reminding me that there is a world apart from transportation networks. Most of all, thank you, Theresa, for your love and patience.

The work presented in this thesis was in part supported by the Deutsche Forschungsgemeinschaft (DFG) within the priority program 1894: Volunteered Geographic Information: Interpretation, Visualization and Social Computing.

Chapter 1

Introduction

1.1 Motivation and Outline

Finding the best path in a transportation network is a fundamental optimization problem, which we call the *shortest path problem* (SPP). It has numerous applications that affect our daily lives. Perhaps the most prominent examples are route planning systems for street networks (think of applications like Google Maps), which are also the focus of this work. Dijkstra's algorithm, published in 1959 by Edsger W. Dijkstra [Dij+59], is still *the* standard approach to solve the SPP as it is intuitive, simple to implement and requires no preprocessing of the street network. We call algorithms that solve the SPP shortest path algorithms or, short, SPAs.

For larger networks, though, Dijkstra's algorithm tends to be too slow, which motivated the development of speed-up techniques. Most of them are based on Dijkstra's algorithm, among which are also hierarchical approaches that try to reduce the computational effort by dividing the network into more and less important elements. The perhaps most popular example of the hierarchical branch of shortest path algorithms are Contraction Hierarchies (CH) [GSSV12]. CH can be considered as a remarkable breakthrough in the scientific community of SPAs as the authors of [GSSV12] managed to simplify *and* improve existing hierarchical approaches significantly. This combination of effectiveness is what makes CH special among modern SPAs.

Ever since their introduction in 2008 [GSSD08], CH have been subject of scientific work that either tried to improve CH or to understand and characterize its effectiveness. Chapter 2 is dedicated to both of these aspects.

The users of routing services often have heterogeneous needs and preferences. Thus, computing the same shortest path for all of them is not always a good option. An intuitive way to allow personalized solutions to the SPP was introduced in [FS15a] in the form of a linear preference model. In Chapter 3 we give new theoretical and practical insights to this model.

The remainder of this chapter provides the common preliminaries of Chapter 2 and 3. We first introduce the basics of route planning, which include the notion of graphs and shortest paths. Afterwards, we discuss the two most relevant SPAs for our work, which are Dijkstra's algorithm and Contraction Hierarchies. At the end of this chapter we summarize the author's main scientific contributions.

1.2 Basics of Route Planning and Definitions

The starting point for most route planning algorithms is a graph $G(V, E)$ with n nodes (or vertices) and m edges that represents the transportation network of interest. In this work the graphs have directed edges, if not noted otherwise. We are especially interested in street networks, which is why nodes typically represent intersections and edges streets between them. The task to find the best route in G from a source node $s \in V$ to a target node $t \in V$ is usually modeled as a cost minimization problem. Typically, one is looking for the shortest route in terms of travel time. In some applications other cost types such as travel distance or energy consumption may be of interest as well. Whatever cost type or combination of cost types is chosen, the result is a cost function $c : E \to \mathbb{R}$ that expresses how much it costs (for instance, how long it takes) to traverse an edge $e \in E$. We only consider non-negative cost types as this is a common requirement of many shortest path algorithms covered in this work. All considered graphs in this work do have such a cost function even if we do not explicitly introduce it.

A path $p = e_1 e_2 \dots e_{|p|}$ is a concatenation of edges in G such that for any two consecutive edges $e_i, e_{i+1} \in p$ with $e_i = (v_i, u_i)$, $e_{i+1} = (v_{i+1}, u_{i+1})$ it holds $u_i = v_{i+1}$. We use the terms *route* and *trajectory* as synonyms for path. The number of edges $|p|$ of path p is its hop length. We write $v \in p$ for a node $v \in V$ if there is an edge $e \in p$ that is adjacent to v. We say that a path p goes from a nodes s to a node t or is an st-path if $e_1 = (s, \cdot)$ and $e_{|p|} = (\cdot, t)$. The nodes s and t are also called the end nodes of p. A path p is simple if and only if each node in V is adjacent to at most two edges in p. The cost $c(p)$ of a path p is defined as

$$c(p) := \sum_{e \in p} c(e).$$

Let Π_{st} be the set of all simple paths from node s to node t.

Definition 1.1
The distance $d(s, t)$ from node $s \in V$ to node $t \in V$ is then defined as follows.

$$d(s, t) := \min_{p \in \Pi_{st}} c(p)$$

Accordingly, the shortest path from s to t is defined as

$$\pi(s, t) := \operatorname*{argmin}_{p \in \Pi_{st}} c(p).$$

We call $d : V \times V \to \mathbb{R}$ the *distance function* of G. Note, though, that the word *distance* doesn't mean that the edge weights represent actual euclidean distances. Another cost type that is at least as common in practice is travel time. In principle, there are no restrictions to the cost function c. Sometimes it makes sense to examine distance functions independently from graphs. We therefore define what a distance function is.

Definition 1.2
Given a node set V, a function $d : V \times V \to \mathbb{R}_{\geq 0}$ is a distance function *if and only if for any node $v \in V$ we have $d(v, v) = 0$ and for any three nodes $v, w, u \in V$ it holds*

$$d(v, w) + d(w, u) \geq d(v, u) \text{ (triangle inequality).}$$

One can show that, given a distance function d, it is always possible to construct a graph $G(V, E)$ and a cost function $c: E \rightarrow \mathbb{R}_{\geq 0}$ such that d is the distance function of G.

We refer to the task of finding the shortest path between two nodes as *shortest path computation* and to the task of computing the distance as *distance computation*. Note that having the shortest path it is trivial to compute the distance. In fact, the distance is typically a side product of the shortest path computation. However, it may be difficult to construct the shortest path based on the results of a distance computation. Thus, all algorithms that perform shortest path computations can also perform distance computations. Typically, such algorithms can be modified to distance-only algorithms, which can lead to improved runtimes.

A query is a problem instance and the query time is the time it takes to compute its solution.

1.3 Dijkstra's Algorithm and Contraction Hierarchies

In this section we describe and discuss Dijkstra's algorithm and Contraction Hierarchies as they are the most relevant SPAs for the remainder of this work. CH is based on Dijkstra's algorithm, which is why we start with the latter.

1.3.1 Dijkstra's Algorithm

Dijkstra's algorithm [Dij+59] has a simple structure, requires no preprocessing and is efficient compared to other approaches with similar low requirements. There are multiple versions of Dijkstra's algorithm. We start with the most basic one, which computes, given a graph $G(V, E)$, the distances from a source node s to all nodes in G.

The idea of the algorithm is to visit (or settle) the nodes one by one. The next (yet unvisited) node is chosen greedily by taking the one with the lowest known distance from source node s. The algorithm starts by initializing a distance array D that keeps track of the best distances from node s to all other nodes it has found so far. Each entry gets the initial value ∞, except the one for s, which is set to 0. Furthermore, it keeps track of the nodes it has reached but not visited (or settled) yet. Let us call this set V', which is initially set to $\{s\}$. The algorithm then iteratively removes the node v from V' with the lowest value in D. This task is typically done by using a priority queue. The algorithm inspects all outgoing edges from v and updates the distances in D. If a better distance for a node w is found, it is added to V' (if it is not already part of it). The algorithm terminates as soon as the set V' is empty and returns the distance array D. See Algorithm 1.1 for the pseudocode.

One can show that Dijkstra's algorithm computes the correct distances if the edge weights are non-negative. This is the reason why in the rest of this work we only consider non-negative distance functions.

Dijkstra's algorithm runs in $\Theta(m + n \log n)$ time when using a Fibonacci heap [FT87].

In its basic form, Dijkstra's algorithm performs distance computations only. To expand it to shortest path computations, for each node the algorithm needs to keep track on its predecessor in the shortest path from s. Thus, a second array beside D is necessary.

One can show that, as soon as a node is settled, its value in D is the correct distance from node s. Hence, if one is only interested in a specific distance, say from node s to node t, Dijkstra's algorithm can abort the computation as soon as it settles node t. This is the version we refer to as Dijkstra's algorithm in the rest of the work.

Algorithmus 1.1 Dijkstra's algorithm

```
Input: Graph $G(V, E)$, source node $s$
  Initialize distance array $D \leftarrow \infty \ \forall v \in V$
  $D(s) \leftarrow 0$
  Initialize reached nodes $V' \leftarrow \{s\}$
  while $V'$ is not empty do
      $v \leftarrow$ node in $V'$ with smallest value in $D$        ▷ Settle node $v$
      Remove $v$ from $V'$
      for all $e = (v, w) \in E$ do                                ▷ Update distances
          if $D(w) > D(v) + c(e)$ then
              $D(w) \leftarrow D(v) + c(e)$
              Add $w$ to $V'$
          end if
      end for
  end while
Return: $D$
```

A popular speed-up technique of Dijkstra's algorithm is its bidirectional version. It starts two searches, one from the source node s and one from the target node t. The search from the source node is done as in the basic version, while the search from t is performed backwards, meaning that the edges (if they are directed) are traversed in the opposite direction. We refer to these two types of searches as forward and backward searches. The searches maintain separate distance arrays and the algorithm keeps track of the best distance between s and t found so far by taking the sum of the corresponding entries in the arrays. The algorithm can stop as soon as some node w is visited from both sides. Note, though, that this node w is not necessarily part of the shortest st-path. Typically, the two searches are conducted alternately or by using one priority queue for both directions.

The speed-up of the bidirectional version compared to the basic one depends on the distribution of the nodes in our graph $G(V, E)$. If G is, for instance, a path graph with unit edge costs, there won't be a speed-up at all. On the other hand, if G is a grid with unit edge costs, the number of nodes within a circle of radius r grows quadratically with r. Two circles with radius $\frac{R}{2}$ contain half as many nodes compared to a circle with radius R and, thus, we can hope for a speed-up of around two.

1.3.2 Contraction Hierarchies

Contraction Hierarchies (CH) [GSSD08] are probably the most popular hierarchical search approach for transportation networks. They have shown to be very effective in terms of query times, space consumption and preprocessing time [GSSV12]. It is common to introduce CH by first explaining how CH are usually constructed. We prefer another approach that highlights the properties of CH before going into the implementation details. The lemmas of this sections are all well-known and we only give proofs for the sake of completeness.

The idea behind CH is motivated by the observation that a long-distance journey in a street network typically leads from unimportant roads to more important highways and back to unimportant roads. If we could be sure that this up-down pattern always holds, we could prune a lot of paths during our search. Unfortunately, this pattern may be a good rule of thumb of how a typical journey looks like but not all shortest paths comply with it. The solution is to add more edges (called shortcuts) to the graph until there is an up-down shortest path between any two nodes.

Formally, the preprocessing step of CH takes a graph $G(V, E)$ and a bijective mapping $r : V \rightarrow$

$\{1, \ldots, n\}$ and outputs a set of shortcuts E'. The graph $G'(V, E \cup E')$ is then called the CH of G (with respect to the mapping r). The mapping r is called node ordering and defines the importance of the nodes. The value $r(v)$ is called the rank of node v. The greater the rank, the more important the node. Each edge (u, v) in G' is either an upward edge if $r(u) < r(v)$ or a downward edge otherwise. Analogously, a path p is an upward/downward path if it solely consists of upward/downward edges. An up-down path is a path $p(s, t)$ that can be split into an upward path $p^{\wedge}(s, w)$ and a downward path $p^{\vee}(w, t)$ (for some node $w \in p$). The special cases $s = w$ and $w = t$ are both allowed. We say that an upward path $\pi(s, t)$ is a *shortest* upward path if $c(\pi) = d(s, t)$. The same holds for downward and up-down paths. Then the CH G' is defined as follows.

Definition 1.3
Given a graph $G(V, E)$ and a node ordering r. The graph $G'(V, E \cup E')$ is a CH of G with respect to r if and only if the following holds (with d' being the distance function of G').

1. ***Distance Preservation:*** *For any two nodes $u, v \in V$ it holds $d(u, v) = d'(u, v)$.*
2. ***Unimodality:*** *For any two nodes $u, v \in V$ with $d(u, v) < \infty$ one can find a shortest path $\pi(u, v)$ in G' that is an up-down path with respect to r.*

Before we discuss how to compute E' we first look at how to compute shortest paths in G' efficiently. Suppose we have two nodes $s, t \in V$ for which we would like to compute a shortest path $\pi(s, t)$ in G'. As G' is a CH we can be sure that there is a node $w \in \pi$ such that π is a concatenation of an upward path from s to w and a downward path from w to t. If we had an oracle that revealed w to us we could do the following. First, we perform a forward search from s as in Dijkstra's algorithm. But since we are only interested in upward paths we only expand upward edges. We call such a search an upward search. We stop as soon as we visit w. Afterwards, we perform a backward search from t. This time we only expand downward edges but as we do it backwards, the edges lead upwards as well. As soon as we settle w we can stop and construct the path π. Since we do not have such an oracle we have to do a bit more work. Naively, one could perform the upward searches until no more nodes can be reached. In this case, we can be sure that w is among the nodes that are reached from both sides. Let $d^{\wedge}$ and $d^{\vee}$ be the distances computed in the forward and backward search. Then all we have to do is to check which nodes are settled from both sides and take the one w with minimum distance $d^{\wedge}(w) + d^{\vee}(w)$. If the upward searches from s and from t are performed simultaneously, one can also keep track of the best distance $d^{\wedge}(w) + d^{\vee}(w)$ from s to t found so far. Then the forward search can be stopped as soon as a node u is settled with $d^{\wedge}(u) \geq d^{\wedge}(w) + d^{\vee}(w)$ (and the backward search analogously).

In practice it turns out that upward searches in typical transportation networks visit much less nodes and expand less edges than normal searches. As a result, shortest path queries can be answered much faster using a CH. The factor of improvement compared to Dijkstra's bidirectional algorithm lies around three to four orders of magnitude for large street networks with travel time as edge weights. If one is only interested in distances, the improvement is even greater [GSSV12].

A common and very effective speed-up technique of the CH query is called *stall-on-demand*, which was first introduced in [SS07] in the context of Highway-Node Routing. We explain its simplest form for the forward search from node s. Before expanding the edges of a settled node v, we check whether we can find a shorter path from s to v via incoming downward edges $(\cdot, v)$. Note that our upward search did not take these edges into account such that we might indeed find a shorter path. If this is the case, the discovered upward path from s to v is not a shortest path and, therefore, this path does not have to be extended any further. We say that node v has been stalled. In more advanced forms of *stall-on-demand* the stalling is propagated to neighbors of v as explained in [GSSD08]. However, [FM19] show that such a propagation does not lead to a very significant improvement.

Now we come back to the question of how to compute the shortcuts E' when the graph G and the node ordering r are given. This is done by performing the so called *contraction operation* once for each node, ordered by ascending rank. We start with $E' = \emptyset$. Hence, at the beginning the graph $G'(V, E \cup E')$ is identical to G. Let G'_R be the subgraph of G' induced by the nodes with rank greater than or equal to R (for some number R). When a node v is contracted, we look at all paths of the form $p := uvw$ in $G'_{r(v)}$. We add a shortcut $e' = (u, w)$ with $c(e') = c(p)$ to E' if and only if p is a shortest path in $G'_{r(v)}$. We show that the resulting graph G' is indeed a CH.

Lemma 1.1
Let $G'(V, E \cup E')$ be the result of the contraction operations described above. Then for any two nodes $s, t \in V$ it holds $d(s, t) = d'(s, t)$.

Proof. At the beginning of the contraction operations G and G' are identical. Furthermore, whenever we add a shortcut $e' = (u, v)$ to E' we know that a path from u to v with the same weight already exists in G'. Hence, we do not alter any distances in G' by adding shortcuts to E'. □

For the second CH condition we first prove the following auxiliary lemma.

Lemma 1.2
Let G'_R be the subgraph of G' induced by the nodes of rank greater than or equal to R and let d'_R be the distance function of G'_R. Then for any value R and any two nodes s, t in G'_R it holds $d'_R(s, t) = d(s, t)$.

Proof. The statement is easy to show for graphs with less than three nodes. Thus, we may assume that $n > 2$. From Lemma 1.1 it follows that $d'_R(s, t) \geq d(s, t)$. We show that $d'_2(s, t) \leq d(s, t)$ for any two nodes s, t with ranks greater than one. Let $\pi(s, t)$ be the shortest path from s to t in G. In G'_2 only the node v with $r(v) = 1$ is missing. Hence, if $v \notin \pi$, we are done. Otherwise, path π is of the form $s \dots uvw \dots t$ and uvw must be a shortest path in G. Thus, the shortcut $e' = (u, w)$ with $c(e') = c(uvw)$ is added to E' when v is contracted. With the same argument we can show that for two nodes s, t with ranks greater than two it holds $d'_3(s, t) \leq d'_2(s, t)$. Therefore, by induction we get $d'_R(s, t) \leq d(s, t)$ for any two nodes s, t with $r(s)$ and $r(t)$ being greater than R. □

Lemma 1.3
For any two nodes $s, t \in V$ there is a shortest up-down path $\pi'(s, t)$ in G'.

Proof. Let $\pi(s, t)$ be a shortest path in G from s to t and let w be the node with the maximum rank in π. We show that there is a shortest upward path $\pi^\wedge(s, w)$ in G'. If $s = w$, we are done. We know from Lemma 1.2 that $d'_{r(s)}(s, w) = d(s, w)$. Thus, there is a shortest path from s to w that starts with an upward edge. It follows by induction that the upward path $\pi^\wedge$ exists. The existence of a shortest downward path $\pi^\vee(w, t)$ can be shown analogously. □

An explicit node ordering is not necessary at the beginning of the preprocessing phase. It suffices to have an oracle that returns the next node to be contracted. We refer to [GSSD08] for an overview of possible techniques to implement such an oracle. In practice, it is also common that the oracle returns not a single node but a set of independent nodes that are then contracted simultaneously [GSSV12]. This technique has two benefits. First, it is easy to parallelize the simultaneous contractions, which improves the preprocessing time. Second, independent node sets tend to be well distributed within the graph, which improves the quality of the resulting CH. It is common to speak of the *contraction level* $l(v)$ of a node v in this context. If node v is contained in the i-th independent set returned by the oracle, then $l(v) = i$. The node ordering r is chosen in an arbitrary way such that for each two nodes $u, v \in V$ it holds $l(u) < l(v) \Rightarrow r(u) < r(v)$.

1.4 Main Contributions

We present new insights and algorithms in the domains of hierarchical search and preference-based routing. Our algorithms significantly improve existing approaches either in terms of efficiency or in terms of applicability. This work is based on six publications [BFJP20; BFMP22; BFP21; BFP22; PR21; Pro22]. We present only those parts of the publications that contain a significant contribution of the author.

1.4.1 Hierarchical Search

In Chapter 2 we present two new algorithmic approaches and a theoretical work related to hierarchical search in SPAs.

We start in Section 2.3 with a new speed-up technique for shortest path computation that combines the ideas of two seemingly different routing algorithms, namely Contraction Hierarchies and Hierarchical Hub Labeling. With this method we are able to achieve a new level of efficiency with respect to space requirements and query time. For instance, for a graph with 1.2 billion edges and 597 million nodes we are able to achieve average query times for distance computations of 25 microseconds using only 50 GB of space. We consider this section, which is based on [BFMP22], the most significant contribution of the thesis from a practical point of view.

We continue in Section 2.4 with a rather surprising modification of Contraction Hierarchies: we remove the shortcuts. Instead of the shortcuts, we store information at the nodes using two bytes of space per node. We show that with these bits of information we are able to perform a modified version of hierarchical search that improves the query times of Dijkstra's bidirectional algorithm by around one order of magnitude. The corresponding article was published in the proceedings of the Annual Symposium on Combinatorial Search [Pro22].

Finally, in Section 2.5, which is based on [PR21], we address a theoretical question concerning Contraction Hierarchies that was raised in [BFS21]. If a graph with a certain property called *bounded growth* is contracted in random order, what is the expected result in terms of query times? The authors of [BFS21] show that the expected size of the node search spaces is in $O\left(\sqrt{n}\log n\right)$ and refer to this size as the query complexity. We show that the actual expected query complexity is much worse by proving that the expected number of visited edges during a query is in $\Omega(n)$.

1.4.2 Preference-Based Routing

The second part of this thesis, Chapter 3, is dedicated to preference-based routing. We present a theoretical analysis of the algorithmic complexity related to preference-based routing and give two new applications and corresponding efficient algorithms.

We begin with a theoretical analysis in Section 3.2, where we show that the number of a special class of shortest paths, called *extreme shortest paths*, is in $n^{O\left(\log^{d-1} n\right)}$, where n is the number of nodes in the graph and d is the number of edge cost types. This is a generalization of known upper bounds for $d = 2$ and $d = 3$ and has significant consequences for the complexity analysis of many algorithms related to preference-based routing. Section 3.4 gives examples of such algorithms. We consider this result to be the most relevant theoretical contribution of the thesis. The corresponding article was part of the proceedings of the 30th Annual European Symposium on Algorithms [BFP22].

In Section 3.3 we present a simple, yet versatile approach to elicit a driver's routing preference if trajectories of the diver are available. Our approach is LP-based and derived from previous work on that subject [FLS16]. However, in contrast to [FLS16], we do not require any special properties of the input trajectories. Our approach works with any input trajectories and computes a result that minimizes an intuitive cost function. Thus, in contrast to black-box learning approaches, it is simple to interpret the results of our method. We first introduced this approach in [BFJP20].

We present our third contribution to preference-based routing in Section 3.4, an algorithm to cluster input trajectories based on driving preferences. The output of our algorithm is a set of preferences of minimum size such that each input trajectory is optimal for at least one of these preferences. The preferences can therefore be used to analyze mobility patterns. We show that the mentioned minimization problem is equivalent to a geometric hitting set problem. We give exact and approximate approaches to solve the problem and show that, in practice, the exact approach is feasible even for large input sets. We first presented this method in [BFP21].

Chapter 2
Hierarchical Search

With *hierarchical search* we refer to speed-up techniques for shortest path computation that specifically aim to exploit the hierarchical structure of the shortest paths in a given graph. All these techniques have in common that they define an importance measure for the nodes or edges. This measure could be realized by an bijective map as in CH, which is a fine-grained measure. Or it could divide the nodes into two groups of less important and important nodes as in (single-level) Transit Node Routing [BFM06]. We are specifically interested in techniques with a strong focus on hierarchy such as CH and Hierarchical Hub Labeling (or HHL, described in Section 2.1.7).

The rest of this chapter is organized as follows. First, we give some examples of hierarchical searches which we find to be especially interesting or which are relevant for the remainder of the chapter. Afterwards, we introduce a graph property called *unimodality* which is a common property of CH and HHL and show how these two approaches can be generalized (Section 2.3). In Section 2.4 we show how CH can be adapted to work without shortcuts. The chapter concludes with theoretical considerations about the complexity of CH when the nodes are contracted in random order (Section 2.5).

2.1 Examples of Hierarchical Search

In this section we present a selection of hierarchical SPAs. Most of them are not particularly relevant for the remainder of this chapter. An exception is Hierachical Hub Labeling, presented in Section 2.1.7.

2.1.1 REACH

The algorithm REACH [Gut04] is an early contribution to hierarchical search. It stores one number per node that describes the importance of the node regarding long-distance journeys. This number is called *reach*. In the following, we describe how this number is defined given a graph $G(V, E)$. Let $\pi(s, t)$ be the shortest st-graph in G and let u be any node in π. The reach of u regarding π is then defined as

$$re(u, \pi) := \min\{d(s, u), d(u, t)\}.$$

The (global) reach of u, written as $re(u)$, is then simply the maximum reach over all shortest paths that contain u. This number is computed and stored for each node in V during the preprocessing step. In the query phase a modified version of Dijkstra's bidirectional algorithm is performed that prunes all nodes w for which it is already known from the preliminary search results that $re(w) < \min\{d(s, w), d(w, t)\}$. The results stay correct because if $re(w) < \min\{d(s, w), d(w, t)\}$ is true, then w is certainly not part of the shortest st-path. The main drawback of REACH is the complexity of its preprocessing step, which requires an all-to-all shortest path computation. This is not feasible for many applications. It is possible to reduce the computational effort in the preprocessing step by computing upper bounds of the reach values as shown in [Gut04].

2.1.2 Transit Node Routing

The introduction of Transit Node Routing (TNR) [BFM06] is an important milestone in the development of SPAs. The conceptual idea behind TNR is simple and the query times of TNR are still among the fastest of all known SPAs.

Let $G(V, E)$ be our graph with an embedding in the plane and let $\Pi(l)$ be the set of all shortest paths π in G where the straight line between the endpoints in the embedding is longer than the parameter $l \in \mathbb{R}$. In TNR a subset of nodes $T \subseteq V$ is computed that should be as small as possible such that every path $\pi \in \Pi(l)$ contains at least one node in T. The nodes in T are called the transit nodes.

The distances between every pair of nodes in T are precomputed and stored as well as the distances between every node in $V \setminus T$ and its nearest transit nodes.

In the query phase the algorithm first checks whether the straight line between s and t is longer than l. If not, the query is treated as a local query with some fallback strategy such as Dijkstra's bidirectional algorithm. Otherwise, the precomputed distances from s to its nearest transit nodes $T(s)$, from $T(s)$ to $T(t)$ and from $T(t)$ to t can be used to compute the distance $d(s, t)$.

One challenge in TNR is to compute the transit nodes T. The authors of [BFM06] propose a sweep-line procedure that triggers many local queries to Dijkstra's algorithm.

TNR can be considered as a hierarchical SPA as it distinguishes between important nodes T and less important nodes $V \setminus T$. Furthermore, the transit nodes could also be organized in a multi-level way, as suggested in [BFM06], by introducing local and global transit nodes.

2.1.3 Highway Hierarchies

Highway Hierarchies (HWH) [SS05; SS06] is a hierarchical SPA with many similarities to CH. Given a graph $G(V, E)$, the result of the preprocessing phase of HWH are subgraphs $G_0(V_0, E_0)$, $G_1(V_1, E_1), \ldots, G_L(V_L, E_L)$ called *cores* with $G_0 = G$ and $V_{i+1} \subseteq V_i$ together with shortcuts $S := \bigcup_{i=1}^{L} S_i \subseteq V_i \times V_i$. The resulting HWH $G'(V, E \cup S)$, thus, consists of the original graph G together with a set of shortcuts S, as it is in CH.

In HWH, however, not only the nodes have ranks but also the edges. The rank of a node v is given by the maximum number i such that $v \in V_i$ holds. The ranks of the edges and shortcuts are defined analogously.

The core G_{i+1} is constructed based on G_i. First, less important edges are removed using a heuristic that marks edges as important if they are part of long shortest paths and if they are not too close to the end points of these paths. Additionally, a node contraction similar to the one in CH takes place. In HWH, the contracted nodes are called *bypassable* nodes. In this step the shortcuts are created.

Also the query phase of HWH shows similarities with CH. As in CH, a bidirectional upward search is performed. In the context of HWH this means that, once a core G_i is reached by a search branch, this branch won't visit any further nodes outside G_i. In this way it is possible to reduce the search space compared to the normal bidirectional search.

CH considerably simplify the preprocessing phase as well as the query phase of HWH without losing efficiency [GSSV12].

2.1.4 Highway Node Routing

Highway Node Routing (HNR) [SS07] can be considered as the direct predecessor of CH. One could even argue that CH was a special case of HNR. However, while the description of the preprocessong is rather vague and general in HNR, CH offers specific instructions on how to generate the overlay graph efficiently that HNR and CH have in common.

In HNR the preprocessing step takes a graph $G(V, E)$ and a series of node sets $V_0, V_1, \dots, V_L$ as input with $V_0 = V$ and $V_{i+1} \subseteq V_i$ for $0 \leq i \leq L-1$. The output is a set of shortcuts S such that the graph $G'(V, E \cup S)$ is unimodal for the node sets $V_0, V_1, \dots, V_L$. This means that for any node set V_i and any node pair $s, t \in V_i$ the distance from s to t in the subgraph of G' induced by V_i is identical to the distance in G' (or G).

We obtain the definition of CH if we specify that $|V_i| - |V_{i+1}| = 1$ and that we obtain V_{i+1} from V_i by contracting a node as described in Section 1.3.2.

The authors of [SS07] also introduce the speed-up technique stall-on-demand, which is a common element of the CH query algorithm.

2.1.5 Contraction Hierarchies

Contraction Hierarchies are probably the most popular hierarchical speed-up technique. A description can be found in Section 1.3.2.

2.1.6 Customizable Contraction Hierarchies

In the preprocessing step of CH we implicitly assume that the edge weights are static. This assumption is not satisfied in many real-world applications, including routing in street networks. Typically, the actual travel times strongly depend on the current amount of traffic and, thus, are in constant change. The implication of this uncertainty is that during the preprocessing phase we cannot decide if a shortcut is needed or not.

The authors of [DSW14] address this issue by introducing a new preprocessing scheme for CH, which they call Customizable Contraction Hierarchies (CCH). In CCH the preprocessing phase is divided into two parts, the construction part and the customization part. In the construction part CCH sets the ranks of the nodes and creates the shortcuts. This is done metric-independently. In the customization part the shortcuts and edges receive their weights. Therefore, if the weights change, only this part needs to be repeated.

As the construction part runs metric-independently, it has to assume that every shortcut candidate needs to be added to the graph. This rather simple change in the preprocessing phase then allows to set the edge weights in phase two in an arbitrary (non-negative) way without violating the unimodality property described in Section 1.3.2. The problem, of course, is that phase one typically creates many more shortcuts than a normal CH. And, indeed, if we employed the common heuristics of CH to determine the node ordering the space requirements of CCH would be prohibitive in practice.

Thus, the authors of [DSW14] propose a new way to determine node orderings based on nested dissection [Geo73]. Instead of the typical bottom-up scheme, their algorithm determines the node ordering top-down by recursively computing balanced node separators. Given a graph $G(V, E)$, a (preferably small) node separator $S \subseteq V$ is computed that divides V into two parts A and B of roughly the same size. The nodes in S then receive the highest ranks in arbitrary order. This procedure is recursively repeated for the sets A and B until all nodes have received their rank. The results shown in [DSW14] indicate that node orderings computed in this way lead to feasible space requirements for CCH.

The customization phase takes edge weights as input and computes the corresponding shortcut weights. This can be done in a bottom-up way. The shortcuts are sorted by ascending rank taking the minimum rank of the adjacent nodes. Each shortcut has two child shortcuts/edges which can be used to compute the weight of the shortcut. The sorting ensures that the child shortcuts receive their weight before their parent shortcuts. Note, though, that the child shortcuts depend on the edge weights and, thus, need to be determined as well.

While the construction part may be computationally expensive the customization part is a lightweight procedure that can be frequently repeated in practice. The customization is also much cheaper than constructing a normal CH.

2.1.7 Hierarchical Hub Labeling

Hierarchical Hub Labeling (HHL) is a special version of Hub Labeling (HL). We, thus, first discuss HL.

2.1.7.1 Hub Labeling

The idea behind HL was first introduced in [CHKZ03] and reconsidered in [ADGW11]. Given an undirected, connected graph $G(V, E)$ with distance function $d: V \times V \rightarrow \mathbb{R}$ each node $v \in V$ obtains a label $L(v)$, which is a set of node-distance pairs. Such an assignment of a label to each node is called a labeling. For each pair $(u \in V, l \in \mathbb{R}) \in L(v)$ it holds $d(v, u) = l$. We say that a node u is a hub of node v if u is part of a pair in $L(v)$, which we express with $u \in L(v)$. With two labels $L(v)$, $L(w)$ it follows from the triangle inequality that

$$d(v, w) \leq \min_{u \in L(v) \wedge u \in L(w)} d(v, u) + d(u, w). \tag{2.1}$$

One can show that the inequality of (2.1) becomes an equality if and only if among the common hubs of $L(v)$ and $L(w)$ there is a node of a shortest path $\pi(v, w)$. The labels of an HL must achieve exactly this coverage of the shortest paths for each node pair $v, w \in V$, which is called the *cover property*. Thus, a labeling is an HL if and only if (2.1) can be written as an equality for each node pair $v, w \in V$.

HL is also called a 2-hop cover. If we interpret the pairs $(u, d(v, u))$ within a label $L(v)$ as (undirected) edges (v, u), we can find a shortest path of two edges between any two nodes. This idea of bounding the hop length of the shortest paths is discussed further in Section 2.3.

HL is also applicable to directed graphs. The difference is that each node receives two labels, one for forward distances and one for backward distances. Furthermore, HL can handle not fully connected graphs by interpreting an empty intersection of the hubs of two labels as infinite distance.

Having an HL we can compute the distance between two nodes v, w as implicitly shown in (2.1). To obtain the set of common hubs efficiently we sort the label pairs by node ID. Therefore, a linear sweep over $L(v)$ and $L(w)$ suffices to compute $d(v, w)$.

The great advantage of HL compared to most other speed-up techniques is its cheap and cache-efficient method to compute distances. The main disadvantage is that the space requirement to store the labels turns out to be infeasible for many applications. Furthermore, an efficient way to compute small labels had been an open issue until the introduction of HHL.

2.1.7.2 Hierarchical Hub Labeling

Hierarchical Hub Labeling [ADGW11; ADGW12] is a valid HL with a bijective node ordering $r: V \to \{1, \dots, n\}$ as in CH. Furthermore, the labels $L(v)$ satisfy the following property for all node pairs $v, u \in V$.

$$r(u) < r(v) \Rightarrow u \notin L(v) \tag{2.2}$$

We call (2.2) in the following *hierarchy constraint*. The hierarchy constraint is a side product of the typical way to construct an HHL. Given a CH $G(V, E)$ with node ordering r we define the forward hubs of node $v \in V$ to be the nodes that are reachable from v via shortest upward paths, which we call the direct search space of v (see Definition 2.4). Due to the correctness of the CH query we can be certain that this definition leads to a valid HL. Furthermore, the hierarchy constraint is clearly satisfied.

In other words, the construction algorithm of HHL takes our CH G and adds more shortcuts until each node is a neighbor of its direct search spaces. The output is another CH $G'(V, E')$ with the property that for any two nodes $v, w \in V$ with $d(v, w) < \infty$ there is a shortest up-down path with at most one upward and at most one downward edge.

An HHL can be constructed rather efficiently in an up-down sweep, starting with the highest rank and descending, as described in [ADGW12]. The query time speed-up of HHL compared to CH for distance computations, as reported in [ADGW12], lies around two to three orders of magnitude.

2.2 Unimodality

Unimodality is a property of graphs with a hierarchical structure of the shortest paths. We first mentioned the term unimodality within Definition 1.3 in the context of CH and now define it in general.

Definition 2.1
Given a node set V and a bijective node ordering $r: V \to \{1, 2, \dots, n\}$. A graph $G(V, E)$ with distance function d is unimodal with respect to r if and only if for any two nodes $u, v \in V$ with $d(u, v) \neq \infty$ there is a shortest up-down path in G.

An alternative, equivalent definition of unimodality looks as follows.

Definition 2.2
Given a node set V and a bijective node ordering $r: V \to \{1, 2, \dots, n\}$. A graph $G(V, E)$ with distance function d is unimodal with respect to r if and only if for any $R > 0$ and node set $V_R := \{v \in V \mid r(v) \geq R\}$

the distance from v to u for any two nodes $v, u \in V_R$ in the subgraph G_R of G induced by V_R is equal to $d(v, u)$.

Definition 2.2 says that in a unimodal graph one can remove any number of least important nodes without changing the distances between the remaining nodes.

Lemma 2.1
Definition 2.1 and Definition 2.2 are equivalent.

Proof. Let the graph $G(V, E)$ be unimodal according to Definition 2.1 and with respect to the node ordering r. As there is a shortest up-down path between any two nodes $v, u \in V$ with $d(v, u) < \infty$ it is clear that we can remove all nodes with lower rank than v and u from G without changing the distance between v and u.

Let now G be unimodal according to Definition 2.2. By definition, there is a shortest path $\pi(v, u)$ in G that solely contains nodes of $V_{\min\{r(v), r(u)\}}$. There are now two cases to consider. First, if $r(v) < r(u)$, the first edge of π must be an upward edge. Second, if $r(v) > r(u)$, the last edge of π must be a downward edge. In both cases we are able to recursively construct the up-down path from v to u in G. □

Given two node orderings r_1 and r_2 it can happen that a graph G is unimodal with respect to exactly one of them. The following definition describes what we mean when we say that a graph is unimodal.

Definition 2.3
A graph $G(V, E)$ is unimodal if and only if there is a bijective node ordering $r : V \to \{1, 2, \dots, n\}$ for which G is unimodal.

Every CH is by definition unimodal. As HHL can be considered as a special kind of CH (according to Definition 1.3), they are unimodal as well. Thus, the term unimodality embraces at least these two popular speed-up techniques that usually are not considered to be very similar.

The term CH is strongly linked to the typical way to construct a CH. Therefore, in our opinion saying HHL is a special kind of CH leads to confusion. We prefer to say that HHL and CH are unimodal.

There are other unimodal graph families. One example are trees. However, many (if not most) graphs are not unimodal, especially real-world transportation networks. Given a graph $G(V, E)$, the task of finding a set of distance-preserving edges E' of minimum cardinality such that $G'(V, E \cup E')$ is unimodal has been shown to be NP-complete (even APX-hard) [BCK+10; Mil12].

With these results in mind the following lemma may be a bit surprising.

Lemma 2.2
Given a graph $G(V, E)$, deciding whether G is unimodal is in P.

Proof. If G is unimodal, there is a corresponding node ordering r and a least important node $v \in V$ (i.e., $r(v) = 1$). It follows that we do not find any two other nodes $u, w \in V$ such that all shortest paths from u to w go via node v (because there must be a shortest up-down path from u to w). Thus, we can remove node v and its adjacent edges without changing the distances between the other nodes. Furthermore, the remaining graph is still unimodal with the same node ordering r. Clearly, checking whether node v exists is possible in polynomial time (with respect to the graph size). □

In the following we define search spaces (analogously to [BFS21]). They are especially important to analyze the query time of CH and the space consumption of HHL.

Definition 2.4
Given a unimodal graph $G(V, E)$ with node ordering r. The forward search space $S^{\wedge}(v)$ for a node $v \in V$ is defined as follows.

$$S^{\wedge}(v) := \{u \in V \mid \exists \text{ upward path } p(v, u) \text{ in } G\}$$

The backward search space $S^{\vee}(v)$ is defined as

$$S^{\vee}(v) := \{u \in V \mid \exists \text{ downward path } p(u, v) \text{ in } G\}.$$

The forward and backward direct *search spaces $D^{\wedge}(v)$ and $D^{\vee}(v)$ are defined analogously with the difference that the paths p must be* shortest *upward/downward paths.*

The search space $S^{\wedge}(v)$ is an upper bound of nodes settled during the CH upward search from node v. The direct search space $D^{\wedge}(v)$ is a subset of $S^{\wedge}(v)$ and contains all nodes that are settled with the correct distance in an upward search from node v. Thus, the size of the direct search space is a (worst case) lower bound of nodes that have to be settled in a CH query. Furthermore, $D^{\wedge}(v)$ is the set of hubs that need to be contained in the forward label of v. Therefore, the size of the direct search spaces is equivalent to the space requirement of HHL.

The new terminology of this section allows us to rephrase the definitions of CH and HHL in order to clearly distinguish between them. Definition 1.3 describes CH in a broad sense such that any distance-preserving function that takes a graph $G(V, E)$ and outputs a unimodal graph $G'(V, E \cup E')$ would be a valid CH constructor. This is a good definition to start with because it describes the minimum requirements to ensure the correctness of the CH query. However, the original idea behind CH, as described in [GSSD08], is more concrete and tries to be as space conservative as possible. We now give a definition of CH that comes much closer to this idea.

Definition 2.5
Given a set of nodes V, a distance function $d : V \times V \to \mathbb{R}_{>0}$ and a bijective node ordering $r : V \to \{1, 2, \ldots, n\}$. Furthermore, let $\mathcal{G}$ be the set of unimodal (with respect to r) graphs $G(V, E)$ with distance function d. Then $G'(V, E') \in \mathcal{G}$ is a CH with respect to r if and only if

$$|E'| = \min_{G(V,E) \in \mathcal{G}} |E|.$$

Note that the distances must be positive. The reason is that we would like the set E' to be unique, which is not always the case if edges with weight zero are allowed.

We said that it is NP-complete to find the set of (distance-preserving) edges E' of minimum cardinality such that $G'(V, E \cup E')$ is unimodal. This is why in Definition 2.5 the node ordering r for which G' should be unimodal is fixed. In this way, the task of finding E' is in P. As mentioned in 1.3.2, the node ordering r is typically not an input to the CH construction algorithm. The construction algorithm itself defines r on the fly using heuristics to keep $|E'|$ and the average search space size small (see [GSSD08] for details).

As Definition 2.5 is not constructive we give another, constructive definition of CH and show that it is equivalent to the first one.

Definition 2.6
Given a bijective node ordering $r: V \to \{1, 2, \dots, n\}$ and a graph $G(V, E)$ with distance function $d: V \to \mathbb{R}_{>0}$ such that each edge $e := (v, u) \in E$ is the only shortest path from node v to node u. Furthermore, let $\mathcal{P}(v, u) \subseteq V$ be the subset of nodes that are contained in a shortest path from node v to node u and let

$$\begin{aligned} E' :=& \{(v, u) \in V \times V \mid \{w \in \mathcal{P}(v, u) \mid r(w) > r(v)\} = \{u\}\} \cup \\ & \{(u, v) \in V \times V \mid \{w \in \mathcal{P}(u, v) \mid r(w) > r(v)\} = \{u\}\}, \end{aligned}$$

where for each edge $e := (v, u) \in E'$ the cost $c(e) = d(v, u)$. Then $G'(V, E \cup E')$ is the Contraction Hierarchy of G with respect to r.

Note that the two sets in the definition of E' denote the upward and downward edges.

Lemma 2.3
Definition 2.5 and Definition 2.6 are equivalent.

Proof. First, observe that, given a node set V and a distance function $d: V \times V \to \mathbb{R}_{>0}$ as in Definition 2.5, one can construct the graph $G(V, E)$ of Definition 2.6 and vice versa.

Given a graph $G(V, E)$, a node ordering r and a set of edges E' as defined in Definition 2.6 and let $e' := (v, u) \in E'$. Furthermore, let $G^*(V, E^*)$ be any unimodal graph with the same distances as in G. We will show that $e' \in E^*$.

By definition of unimodality there must be a shortest up-down path from v to u in G^*. We know that in G all intermediate nodes in the shortest paths from v to u have a lower rank than v and u. This must also be the case in G^* because the distances in G^* are identical to the distances in G. Thus, e' must be in E^* in order to have a shortest up-down path from v to u.

What remains is to show that $G'(V, E \cup E')$ is a unimodal graph. Let $R > 0$ and $V_R := \{v \in V \mid r(v) \geq R\}$ as in Definition 2.2. Furthermore, let G_R be the subgraph of G' induced by the node subset V_R and let d_R be its distance function. We will show that for any two nodes $v, u \in V_R$ it holds $d_R(v, u) = d(v, u)$, which finishes the proof. Suppose that this is not true. We then choose $v, u \in V_R$ such that

$$d(v, u) = \min\{d(v', u') \mid v', u' \in V_R \text{ and } d_R(v', u') \neq d(v', u')\}.$$

The set $\mathcal{P}(v, u)$ must contain another node of V_R as otherwise $(v, u) \in E'$. Let $w \in \mathcal{P}(v, u)$ be this node. It follows that $d_R(v, w) \neq d(v, w)$ or $d_R(w, u) \neq d(w, u)$, which contradicts the choice of v and u. □

We now come to the definition of HHL.

Definition 2.7
Let $G(V, E)$ be a CH with distance function d, node ordering r and corresponding direct search spaces $D^\wedge$ and $D^\vee$. Then $G'(V, E')$ is the Hierarchical Hub Labeling of G if and only if

1. *d is the distance function of G',*
2. *for every $v \in V$ and $u \in D^\wedge(v)$ there is an edge $e := (v, u)$ in E' with $d(v, u) = c(e)$ and*
3. *for every $v \in V$ and $u \in D^\vee(v)$ there is an edge $e := (u, v)$ in E' with $d(u, v) = c(e)$.*

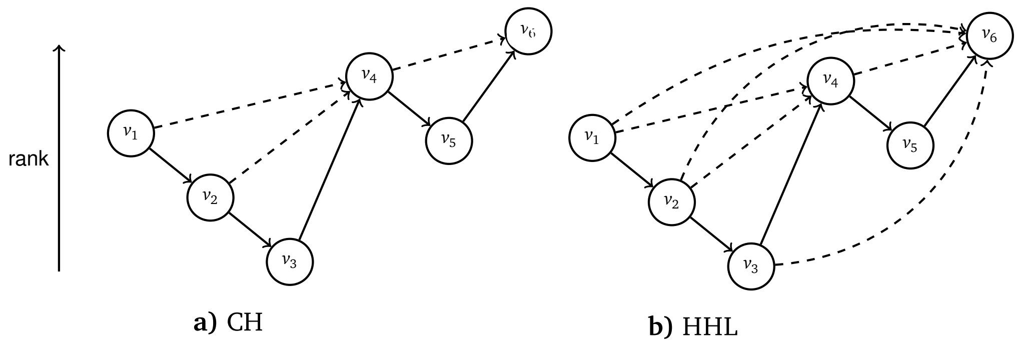


Figure 2.1: This example demonstrates the difference between CH and HHL. The dashed edges are shortcuts, the other edges are from the input graph. The nodes are vertically ordered by rank. HHL requires considerably more shortcuts than CH.

If we would like to turn an input graph G into an HHL we first compute the CH of G and then add the edges as listed in Definition 2.7.

However, what if the node ordering is not given and we would like to find one that minimizes the number of required edges? The authors of [BGK+15] show that this task is NP-complete. In the same work they show that, despite the practical success of HHL, there are graph families where the optimal space consumption of HHL is by a factor of $\Omega\left(\sqrt{n}\right)$ worse than the optimal space consumption of HL. A similar result was already shown in [GRS13]. Thus, the hierarchy constraint (2.2) is indeed a constraint and not a natural labeling property.

CH and HHL are two extreme examples of unimodal graphs. On the one hand, a CH consists of relatively few edges but may contain long (with respect to the number of edges) shortest upward paths. On the other hand, an HHL typically consists of many edges but for every shortest upward path there is a corresponding edge with the same endpoints and the same cost. See Figure 2.1 for an example.

2.3 Unifying Search- and Lookup-based Approaches

This section is based on [BFMP22], which is joint work with Daniel Bahrdt, Stefan Funke and Sokol Makolli. The author's main contribution to this section is the theoretical part and most of the in-memory experiments.

2.3.1 Introduction

In this section we develop a generalization of HHL that allows us to bound the hop length of shortest upward/downward paths to any number k and not only to $k = 1$. For large values of k we obtain a normal CH while for small k we are close to HHL. Thus, the parameter k allows us to interpolate between a search- and a lookup-based approach. We will see that choosing a value $k > 1$ can lead to a much better tradeoff between space consumption and query time as with HHL or CH. This improvement is especially interesting when dealing with very large, world-scale street networks as for them the space requirements of HHL often become infeasible.

2.3.2 CHHL

Funke introduces in [Fun20] a method, called CHHL, with a similar intention to interpolate between CH and HHL. The general idea of CHHL is to define a threshold level T and to compute the full HHL label for nodes with contraction level $l(u) \geq T$. All other nodes with lower ranks merely obtain their CH edges. In this way, upward searches can stop settling nodes as soon as they reach nodes with contraction level greater than or equal to T. This helps to improve query times considerably compared to CH. However, there is no explicit intention in CHHL to bound the hop length of shortest upward paths as in our method. We will show that CHHL can be considered as a special case of our method and that we are able to improve the results of CHHL.

2.3.3 Distance Closures

In this section we describe a framework that allows very flexible and powerful tradeoffs between space and query time for distance computation. We take CH as our basis with lowest space requirements and would like to find a way to augment it with additional shortcuts in order to improve the query times. However, simply adding shortcuts actually *slows down* CH queries. So we need to identify suitable properties (and matching shortcut addition strategies) such that queries can in fact be accelerated.

Before diving into the rather technical definitions, we first outline our general idea. We divide the nodes into groups that efficiently share their knowledge. Let us say, each node group is a village and each node is an inhabitant of that village. Each inhabitant knows all the shortest paths from his or her house to the other spots in and around the village he or she cares about. Here comes the hierarchy into play. The inhabitants only care about the spots that are more important than their house. A traveler that comes to the village (via his or her upward path) can therefore ask the first inhabitant on the way for the shortest path to the next village without needing to ask any other inhabitant. Technically, this means that along a shortest (upward) path we only need to settle one node per group. The larger a group becomes the more knowledge must be shared, which means more shortcuts are necessary. The most extreme example is when all nodes are in one group. Surprisingly, we will see that this case is identical to HHL.

2.3.4 Unimodality

While both CH and HHL are unimodal graphs, they differ in the maximum lengths of the up-down paths. For a more detailed differentiation, let us define the forward depth $d^\wedge$ and the backward depth $d^\vee$ of a unimodal graph.

Definition 2.8
Given a unimodal graph $G(V, E)$. Let $\Pi(u, v)$ be the set of all shortest up-down paths from node u to node v. Then the forward depth $d^\wedge$ of G is defined as

$$d^\wedge := \max_{(u,v)\in V\times V} \min_{(\pi^\wedge,\pi^\vee)\in\Pi(u,v)} |\pi^\wedge|,$$

where $|\pi^\wedge|$ denotes the number of edges in the upward path $\pi^\wedge$. Analogously, the backward depth $d^\vee$ is defined as

$$d^\vee := \max_{(u,v)\in V\times V} \min_{(\pi^\wedge,\pi^\vee)\in\Pi(u,v)} |\pi^\vee|.$$

HHL, as mentioned above, has forward and backward depth 1 whereas the depths of CH are not bounded. This is the key difference between CH and HHL that causes the significant speed-up and additional space requirement of HHL compared to CH. While the direct search spaces in CH and HHL do not differ, CH needs to compute them during the query by performing upward searches. These searches take time and they are imprecise as they may visit nodes outside the direct search spaces. The query algorithm of HHL is much more time-efficient. It traverses exactly one edge per node in the direct search spaces and visits no nodes outside of them.

The idea is now to increase the depth bound of HHL to some small value greater than one the get the best of both worlds: a space- and time-efficient query algorithm.

2.3.5 Distance Closures and Group Partitions

The following section constitutes our main contribution, namely a scheme to augment a unimodal graph by additional edges in a structured manner such that not only unimodality can be made use of in the query phase, but also bounded forward/backward depth. Depending on the desired bound on the forward/backward depth, query times can be improved at the cost of more additional edges. Note that, obviously, adding edges (with shortest path distances) to an unimodal graph cannot destroy the unimodality property.

We now define the term Distance Closure (DC). Intuitively, a Distance Closure of a node set $V^* \subseteq V$ is a set of edges that connects each node $u \in V^*$ with those neighbors of V^* that are part of the direct search space of u. To come back to the example at the beginning, this is exactly the knowledge inhabitant u needs to have in order to guide any traveler through the village V^*.

Definition 2.9
Given a node set $V^* \subseteq V$ of a unimodal graph $G(V, E)$ and any node pair $(u, v) \in V^* \times V$. The edge (u, v) with $c((u, v)) = d(u, v)$ is in the forward distance closure *$E^\wedge(G, V^*)$ if and only if there is a shortest upward path $\pi(u, v) := (u, \ldots, w, v)$ with $w \in V^*$ in G ($u = w$ is allowed).*

Analogously, the edge (v, u) with $c((v, u)) = d(v, u)$ is in the backward distance closure *$E^\vee(G, V^*)$ if and only if there is a shortest downward path $\pi(v, u) := (v, w, \ldots, u)$ with $w \in V^*$ ($u = w$ is allowed).*

The distance closure *(DC) $E^*(G, V^*)$ is the union of $E^\wedge(G, V^*)$ and $E^\vee(G, V^*)$.*

We call such a node set V^* *distance group* (DG).

Definition 2.10
For a unimodal graph $G(V, E)$ and a node set $V^* \subseteq V$, V^* is a distance group *(DG) of G if and only if the distance closure $E^*(G, V^*)$ is a subset of E.*

Knowing that a unimodal graph $G(V, E)$ contains a distance group V^* greatly helps to improve the efficiency of distance computations. The reason for that is given by the following key lemma. It says that within each shortest upward or downward path there is at most one settled node per DG. Using the terminology of our example, the lemma says that if a traveler passes through a village, he or she has to ask only one inhabitant in order to stay on the shortest path.

Lemma 2.4
Given a unimodal graph $G(V, E)$ with a distance group V^*. Let v be any node in V. For each node $u \in D^\wedge(v)$ there is an upward shortest path $\pi(v, u)$ such that $|\pi \setminus \{u\} \cap V^*| \leq 1$. The analogous statement for the backward search space $D^\vee(v)$ holds as well.

Proof. Let $v \in V$ and $u \in D^{\wedge}(v)$. By definition, there is at least one upward shortest path $\pi(v,u)$ in G. W.l.o.g. we may assume $v \in V^*$. If $|\pi \backslash \{u\} \cap V^*| \leq 1$ there is nothing to show. Hence, we may assume that there are nodes in $\pi \backslash \{u\}$ apart from v that belong to the distance group V^*. Let w be the last one of them (with respect to π). Let w' be the direct successor of w in π. Hence, there is a shortest upward path $\pi'(v,w') := (v, \ldots, w, w')$ and, by definition, $(v,w') \in E^*(G,V^*)$. The backward case can be shown analogously. □

So intuitively, the distance closure of V^* allows us to prune paths with more than one intermediate node from V^* in the bidirectional search for the optimal path.

For improved query times (compared to CH) we divide the graph into distance groups and call them a distance group partition. Nodes with similar ranks are assigned to the same group.

Definition 2.11
Given a unimodal graph $G(V,E)$. A collection of disjoint subsets of V, $\mathcal{U} := \{V_1^*, V_2^*, \ldots, V_k^*\}$, is a distance group partition (DGP) of G if and only if $\cup_{i=1}^{k} V_i^* = V$, each set V_i^* in $\mathcal{U}$ is a distance group of G and consists of exactly one interval of node ranks.

For example, having the nodes v_1 to v_7 (index denotes rank) a possible DGP $\mathcal{U}$ could be

$$\{\{v_1, v_2\}, \{v_3, v_4, v_5\}, \{v_6, v_7\}\}$$

because each set consist of exactly one rank interval. On the other hand, $\{\{v_1, v_3\}, \{v_2, v_4, v_5\}, \{v_6, v_7\}\}$ is not allowed. This restriction does not have any technical reason. We introduce it to keep the framework simple. From now on we sort the distance groups by rank such that V_1 contains the nodes with the lowest ranks.

The following straightforward, yet crucial Lemma yields a bound on the forward/backward depth. It says that if there are in total k villages, a traveler has to ask at most k persons for the shortest way. Or technically, each shortest upward/downward path consists of at most $k+1$ nodes (or k settled nodes).

Lemma 2.5
For a unimodal graph $G(V,E)$ with DGP $\mathcal{U}$ we have:

$$d^{\wedge} \leq |\mathcal{U}|,$$
$$d^{\vee} \leq |\mathcal{U}|.$$

Proof. Given any node pair $(v,u) \in V \times V$ with $u \in D^{\wedge}(v)$. From Lemma 2.4 we know there is an upward shortest path $\pi(v,u)$ such that $\pi(v,u) \backslash \{u\}$ contains at most one node per distance group. Since V is covered by $\mathcal{U}$, each node is part of a distance group. Therefore, $|\pi(v,u) \backslash \{u\}| \leq |\mathcal{U}|$, which means that $\pi(v,u)$ consists of at most $|\mathcal{U}|$ edges. □

So the only remaining question is how to choose the DGP $\mathcal{U}$. Let us first see to what choices of $\mathcal{U}$ the known speed-up techniques CH, HHL, and CHHL actually correspond to.

2.3.5.1 CH, HHL, and CHHL are special DGP strategies

For CH, we could simply have $\mathcal{U}$ as $|V|$ singleton sets containing one node each.

Corollary 2.1
CH is a DGP strategy with $\mathcal{U} := \{\{v\} : v \in V\}$

For HHL the DGP $\mathcal{U}$ only contains a single distance group $V^* = V$.

Lemma 2.6
HHL is a DGP strategy with $\mathcal{U} := \{V\}$

Proof. Given an HHL $G(V, E)$. Per Definition 2.7, a node $u \in V$ has upward edges to all nodes in $D^\wedge(u)$ and backward downward edges to all nodes in $D^\vee(u)$. Thus, the DC $E^*(G, V)$ is a subset of E. Furthermore, to each node $v \in D^\wedge(u)$ there is a shortest upward path $\pi := (u, \ldots, v)$ and hence, $(u, v) \in E^*(G, V)$. The same holds for nodes in $D^\vee(u)$. Hence, E is also contained in $E^*(G, V)$. □

The CHHL approach combines ordinary CH-search up to a threshold level T with regular hub labels for nodes of level $> T$. This also fits in our framework.

Corollary 2.2
CHHL with threshhold level T is a special DGP strategy with $\mathcal{U} := \{\{v\} : l(v) < T\} \cup \{\{v : l(v) \geq T\}\}$

2.3.5.2 Implementation

We now come to the more practical point how to create and use a DGP. We write k-DGP for a DGP $\mathcal{U}$ with $|\mathcal{U}| = k$ and $U(v)$ for the group index of node v. Thus, for two nodes $(v, u) \in V \times V$ with $v < u$ it follows $U(v) \leq U(u)$. Note that 1-DGP is equivalent to HHL. Algorithm 2.1 provides a sketch of an upward search in a DGP. The backward search can be performed analogously. We suggest storing the nodes marked as visited in buckets, one bucket per DG. Note that no priority queue is necessary. The computed distances in the forward search from s and the backward search from t can then be used to compute the distance from s to t similarly to HHL or CH. Note that the algorithm only marks those nodes as visited that belong to a higher node group. Marking a node as visited is the equivalent operation to pushing a node in the PQ in CH.

Algorithmus 2.1 Upward search with DGP

```
Input: $G = (V, E), \mathcal{U} := \{V_1, V_2, \ldots, V_k\}, s \in V$
 1: Mark node $s$ as visited and set distance to 0
 2: for all DGs $V_i$ from $V_1$ to $V_k$ do
 3:     for all nodes $v_i$ in $V_i$ marked as visited do
 4:         for all fwd. neighb. $v_j$ of $v_i$ with $r(v_j) > r(v_i)$ do
 5:             Update distance to $v_j$ (if necessary)
 6:             if $U(v_j) > U(v_i)$ then
 7:                 Mark $v_j$ as visited
 8:             end if
 9:         end for
10:     end for
11: end for
```

Algorithm 2.2 shows the upward part of the construction of a DGP. The algorithm requires a CH and the desired DGP as input. The additional shortcuts of a node v are constructed with the help of the already computed shortcuts of the upward neighbors of v. The most expensive part is the distance check in Line 4. The construction of downward edges works analogously. Note that the edges are only shared among nodes of the same distance group (see expression $U(u) = U(v)$ in Line 2).

Algorithmus 2.2 Construction of DGP (upward case)

```
Input: CH $G = (V, E), \mathcal{U} := \{V_1, V_2, \dots, V_k\}$
 1: for all nodes $v \in V$ sorted by decreasing rank do
 2:     for all fwd. neighb. $r(u) > r(v)$ of $v$ in CH with $U(u) = U(v)$ do
 3:         for all fwd. neighb. $w$ of $u$ with $r(w) > r(u)$ do
 4:             if $d(v, w) = d(v, u) + d(u, w)$ then
 5:                 Set length of $e := (v, w)$ to $d(v, w)$
 6:                 If $e \notin E$, add $e$ to $E$
 7:             end if
 8:         end for
 9:     end for
10: end for
```

A DGP can be defined with the help of a set $S := \{u_1, u_2, \dots, u_{k-1}\} \subseteq V$ of $k-1$ nodes that define the interval boundaries of the distance groups. A node v is in group V_i if and only if the set $S(v) := \{u \in S : u \leq v\}$ has exactly $i-1$ elements. The nodes in S are called *group separators*.

So for example, in a graph with 100 million nodes, we might put the first (according to rank) 50 million nodes into V_1, the next 30 million nodes into V_2, and the last 20 million nodes into V_3. Computing the group closures for V_1, V_2, V_3 and adding them to the edge set would guarantee a forward and backward depth of 3. We would need two group separators in this case, one at 50 million and one at 80 million.

Of course, even for a fixed number of groups (3 in the example above), there is still a lot of freedom on how to choose the separators. Intuitively, the larger a group becomes, the more shortcuts per node are necessary. Moreover, the groups of higher ranks tend to become more complex and should therefore be chosen smaller.

2.3.5.3 Systematic choice of group separators

Given a unimodal graph $G(V, E)$ with DGP $\mathcal{U}$ and a shortest upward path $\pi(v_1, v_l) := (v_1, v_2, \dots, v_l)$ in G. We know from Definition 2.9 that $(v_1, v_i) \in E$ if v_{i-1} and v_1 are in the same DG (assuming that shortest upward paths are unique). The DG of a node v only depends on its rank and the group separators. Therefore, we do not have to compute G explicitly in order to estimate its size. Assuming $\tilde{G}$ is the CH with the same ordering and distances as in G it suffices to sample upward and downward searches in $\tilde{G}$ to obtain a good estimation of the size of G. Other important quantities such as the number of settled nodes and the number of relaxed edges in queries with G can be estimated in a similar way by sampling searches in $\tilde{G}$.

This sampling procedure can be conducted with different group separators. In that way it is possible to evaluate a large selection of group separators relatively efficiently. More sophisticated methods to determine group separators are subject to future research.

2.3.6 Path Retrieval

If not only shortest path distances are required but the actual optimal paths, CH implementations typically maintain shortcut pointers which explicitly associate with each shortcut the two edges spanned by this shortcut. This allows for efficient recursive unfolding of optimal paths in the unimodal graph to optimal paths in the original graph. An HHL representation can be augmented in a similar way, see [Fun20], though in this case the large number of additional pointers required make the space requirement even more prohibitive. Hence we assume that the shortcuts of the underlying CH are equipped with shortcut pointers, whereas the (possibly very many) additional shortcuts introduced via group closures are not.

The result of a query in a DGP-equipped graph is an up-down path which might contain shortcut edges that are not CH shortcuts. The main challenge is first to unfold them into CH edges. So assume we have a shortcut edge (v, w) which corresponds to an upward shortest path $\pi(v, w)$ in the CH. During the search in the DGP-equipped graph, the predecessor w' of w in $\pi(v, w)$ had its shortest path distance assigned because from $w' \in \pi(v, w)$ it follows $w' \in D^\wedge(v)$. So by iterating over all incoming (CH) edges $(., w)$ we can determine w' with $d(s, w') + c(w', w) = d(s, w)$ and recursively continue on (v, w').

The CH representation of a shortest path with k edges typically contains a lot less shortcut edges (also see [FS15b] for a theoretical explanation for that), which then can be unfolded completely or partially depending on the application scenario.

2.3.7 Experiments

In this section we experimentally evaluate concrete instances of our k-DGP strategy with k between 2 and 6. As a baseline we will always compare to regular CH, as this seems to be the most common and widespread acceleration technique in practice. Relative performance of CH compared to other speed-up techniques (both in terms of space as well as query times), can be found, e.g., in [Fun20], Table 4, or in [BDG+16]. Our code for the experiments is available online.[1]

2.3.7.1 Setup

We measured the construction and query time on multiple machines with various additional constraints. All machines use Linux as operating system. See Table 2.1 for a detailed comparison.

We used road networks from the OpenStreetMap (OSM) project [20] to construct our benchmark instances. Our first instance, the road network of South America with about 63 million nodes has a size that still allows the construction of HHL using our hardware. The second graph is the largest connected road network in OSM, which consists of Africa, Asia and Europe. We call it Eurafrasia. With its nearly 600 million nodes it is too large to construct its HHL. In both cases we use travel time as edge weight. See Table 2.2 for more details. We used a standard CH constructor according to [GSSV12] which completed the CH construction for even the largest graph within less than 3 hours, in fact graph extraction from the OSM data was the most time consuming part of the whole process.

For us, the most relevant aspects of speed-up-techniques are *space requirement* and *query time*. In spite of also being of interest, the preprocessing time is often strongly correlated with the space requirement, so while we also state preprocessing times, they are not subject to our optimizations. Still, even with only two objectives at hand, there is a plethora of parameter choices which are pareto-optimal (with respect to these objectives). For sake of a simpler exposition, we will therefore focus on optimizing (that is, minimizing) the *product* of space requirement and query time. In our tables we will always state the quotient $1/(time \times space)$ as overall performance indicator. We call

[1] https://osf.io/7w6t3/?view_only=ba2945aefeb348d5b071f2d58c5c24b6

	Laptop P52s	Desktop	Workstation
CPU	Intel Core i7-8550U	AMD Ryzen 3700x	AMD Threadripper 2950X
	4×1.8 GHz	8×3.6 GHz	16×3.5 GHz
RAM	64 GiB DDR4	16 GiB DDR4	256 GiB DDR4
Storage	Samsung 970 Evo Plus	Corsair Force MP510	SanDisk Extreme Pro
	2 TB	256 GB	500 GB
OS	Arch Linux	Ubuntu 20.04	Ubuntu 20.04

Table 2.1: Various technical specifications of the machines used to conduct our benchmarks.

CH	#Nodes	#Edges	#Edges incl. CH shortcuts
South America	$63 \cdot 10^6$	$129 \cdot 10^6$	$253 \cdot 10^6$
Eurafrasia	$597 \cdot 10^6$	$1,207 \cdot 10^6$	$2,269 \cdot 10^6$

Table 2.2: Benchmark data sets as derived from the OpenStreetMap project.

this quantity the *space-time efficiency* and normalize it such that CH corresponds to a space-time efficiency of 100%. Values larger than 100% indicate a smaller product of space and query time than CH.

2.3.7.2 Construction

In this section we evaluate the DGP strategies CH, HHL, CHHL and k-DGP on our data sets using the *workstation* as our benchmark machine.

For k-DGP we first compute the group separators for different depths k that minimize the product of space consumption and query time using the sampling approach sketched out in Section 2.3.5.3. During sampling we only consider those nodes as group separator candidates that have the lowest rank of their CH level. Table 2.3 shows the results. The ranks of the group separators are given in %, where 100% denotes the maximum rank.

Depth	South America	Eurafrasia
$k = 2$	95.6	98.3
$k = 3$	83.1 99.0	81.0 99.5
$k = 4$	65.8 89.8 98.7	68.7 96.5 99.5
$k = 5$	43.6 77.2 93.9 99.5	45.2 81.0 95.3 99.6
$k = 6$	43.6 77.2 92.3 94.9 99.6	45.2 68.7 81.0 95.3 99.6

Table 2.3: Space-time optimized group separators in % for different depths k of DGP (100% is maximum rank).

Table 2.4 shows the construction times and sizes of HHL and k-DGP. The construction was performed in parallel, construction times are always given as a product of actual run time and number of threads. The construction time as well as the size considerably decrease with increasing depth. For South America the most remarkable improvement is between the construction of HHL and k-DGP with depth $k = 2$. The construction time drops by a factor of 4.8 and the size even by a factor of 12.9. Unfortunately, the space consumption of HHL for Eurafrasia exceeded the RAM of even our largest machine. With $k > 4$ and 16 threads the construction for South America is completed in less than two minutes.

Depth	Time	Size	Time	Size
HHL	05:30:28	216.8	–	–
$k = 2$	01:09:25	16.8	17:23:38	146.8
$k = 3$	00:56:35	8.0	10:36:26	71.2
$k = 4$	00:40:33	7.0	08:22:17	57.2
$k = 5$	00:31:51	5.3	05:25:51	50.0
$k = 6$	00:31:26	4.9	03:43:37	49.2

Table 2.4: Construction times of k-DGP. Shown time is product of actual run time and number of used cores. Time in HH:MM:SS and size in GB. South America (left) and Eurafrasia (right). Workstation.

Type	Size	Query Time	Space-Time Efficiency
CH	2.3 GB	228 μs	100%
HHL	216.8 GB	2.4 μs	101%
$k = 2$	16.8 GB	10.1 μs	309%
$k = 3$	8.0 GB	16.4 μs	400%
$k = 4$	7.0 GB	17.7 μs	423%
$k = 5$	5.3 GB	22.1 μs	448%
$k = 6$	4.9 GB	24.3 μs	440%

Table 2.5: Space consumption, query time, and space-time efficiency of k-DGP. South America. Workstation.

Type	Size	Query Time	Space-Time Efficiency
CH	21.4 GB	529 μs	100%
$k = 2$	146.8 GB	15.2 μs	507%
$k = 3$	71.2 GB	20.4 μs	779%
$k = 4$	57.2 GB	24.4 μs	811%
$k = 5$	50.0 GB	25.2 μs	898%
$k = 6$	49.2 GB	26.5 μs	868%

Table 2.6: Space consumption, query time, and space-time efficiency k-DGP. Eurafrasia. Workstation.

2.3.7.3 RAM-based

We compare the query times and space consumption of the DGP strategies CH, HHL, CHHL and k-DGP on the workstation. Shown run times are the average of one million randomly generated queries.

Table 2.5 shows the run times and sizes of k-DGP with different depths k for South America. The k-DGP instances are those presented in Section 2.3.7.2 that optimize the space-time efficiency. Interestingly, CH and HHL have almost the same space-time efficiency – HHL is about 100 times faster but also uses about 100 times more space – while the space-time efficiency of k-DGP reaches its maximum at $k = 5$ and outperforms CH and HHL by a factor of around 4.5.

The results for Eurafrasia (see Table 2.6) are even better. Our approach again reaches the space-time maximum with depth 5. We improve the space-time efficiency compared to CH now by a factor of 9. As stated before, HHL (1-DGP) could not be evaluated due to excessive space consumption.

We compare these results with CHHL. Table 2.7 shows the results for South America. CHHL also improves the space-time efficiency compared to CH but only by a maximum factor of 3.1. Looking at

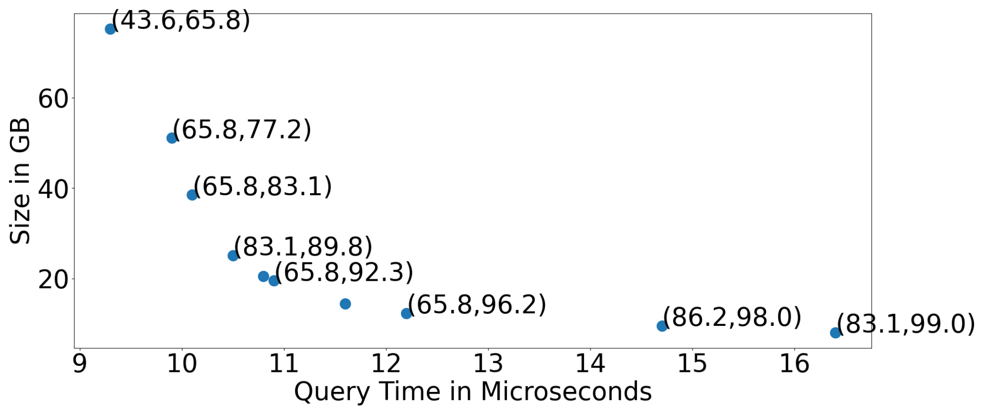


Figure 2.2: Different space-time tradeoffs for 3-DGP. Ranks of group separators in brackets, ranks in %. South America. Workstation.

the case $L = 12$ CHHL consumes 8.4 GB of memory while allowing query times of 22.8 μs. 3-DGP reaches query times of 16.4 μs using 8.0 GB space, which is a considerable improvement. Also for other fixed sizes k-DGP improves the query times of CHHL. Or to put it differently, to achieve a query time of around 22 μs, 5-DGP requires 5.3 GB of space whereas CHHL requires 8.4 GB of space. Again a considerable improvement.

Type	Size	Query Time	Space-Time Efficiency
CH	2.3 GB	228 μs	100%
$L = 4$	37.3 GB	14.9 μ	94.4%
$L = 8$	14.5 GB	18.8 μ	192%
$L = 12$	8.4 GB	22.8 μ	274%
$L = 16$	6.1 GB	30.4 μs	283%
$L = 20$	4.8 GB	37.5 μs	291%
$L = 24$	4.0 GB	42.9 μs	306%
$L = 28$	3.5 GB	49.9 μs	300%
$L = 32$	3.1 GB	56.2 μs	301%

Table 2.7: Space, query time, and space-time efficiency of CHHL [Fun20]. South America. Workstation.

The comparison for Eurafrasia looks similar. CHHL greatly improves the space-time efficiency compared to CH but with a smaller factor (6.4 instead of 9) than k-DGP. For a fixed size k-DGP shows better query times than CHHL, and for a desired query time, k-DGP requires less space than CHHL.

Finally, we look at different space-time tradeoffs for the special case 3-DGP. These tradeoffs are achieved by choosing the two group separators differently. As shown in Figure 2.2, improvements in the query time become significantly more expensive with small query times. At that point it is more efficient to switch to a 2-DGP instead.

2.3.7.4 Path Retrieval

While some applications demand only the computation of shortest path *distances*, others might require an actual route realizing this optimal distance. Very often, though, in particular if the main purpose

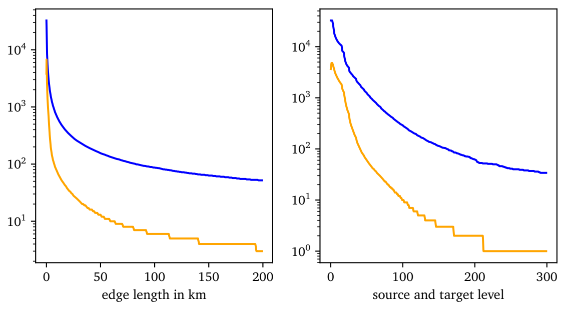


Figure 2.3: Path unfolding until edge length x in km (left) and CH level x (right). Path length in #edges (blue) and query times in μs (orange). 5-DGP. Eurafrasia. Desktop in-RAM.

is the visualization of the shortest path, it is sufficient to provide a 'faithful' approximation of the shortest path. CH representations of paths (with not all shortcuts fully unfolded) have been shown to provide visually appealing, yet space saving representations, see [FSS17]. We experimentally investigate the time and space consumption of path unfolding/retrieval dependent on the desired approximation quality. All experiments of this section are for the 5-DGP instance of Eurafrasia.

We first construct the CH representation of the path from the results of the distance computation following the ideas described in Section 2.3.6. On average, together with the distance computation this step takes 56.0 μs (compared to 25.2 μs for distance only) and the CH representation consists of 27 shortcuts. If we store the index of the last CH edge of each shortcut, this time can be even brought down to 38.7 μs.

In a second step we unfold the CH path using the following two strategies: we replace shortcut edges $e = (u, v)$ as long as (a) e has length more than x km in the current path representation or (b) the CH level (and therefore the ranks) of both u and v are above a certain threshold. Large thresholds lead to coarser approximations of the path with few (shortcut) edges, whereas small threshold values let the path representation converge towards the shortest path in the original graph. In Figure 2.3 (left) we have denoted both the number of edges as well as the time required for path unfolding according to (a) dependent on the threshold value in km (averaged over 10,000 shortest paths in the Eurafrasia graph). For example, for a threshold length of 50km, the time required to unfold is less than 20 μs and results in about 150 edges. Reducing the threshold to 25km roughly doubles the number of edges and increases the required time to about 30 μs. Fully unfolding a path takes about 3.7 ms and results on average in about $32,000$ edges.

2.3.7.5 External Memory

When considering road networks on a continental- or even planet-wide scale, the assumption that graph representation and precomputed auxiliary information can be stored in internal RAM only holds for rather powerful non-commodity hardware. In this section we investigate to what extent k-DGP applies to use cases where a large part of the graph and auxiliary information is stored in external memory.

We consider a powerful, yet standard *Laptop* system with a fast PCIe solid state disk, see Table 2.1 for detailed specs. We used `mmap` to externalize the data structures for graph representation and auxiliary data relying on the operating system to fetch the required data and caching. In this context, two important aspects arose during our benchmarking:

prefetching: By default, the OS uses a prefetching strategy to load data from external memory which has not been asked for by the application, but which the OS suspects to be asked for in the near future. For most applications such data is located 'close' to data previously asked for. As our scheme involves quite arbitrary data access patterns, prefetching can be counterproductive. We differentiate between active and disabled prefetching.

asynchronous I/O: Whenever data has to be loaded from external memory, the OS will suspend the task until the respective data has been fetched. In the meanwhile, the next query might proceed with its processing. To make use of such asynchronous I/O accesses, we process queries in parallel 8 threads, but restricted to one single core (as we do not want to benchmark the computational power of several cores). Waiting for data to be fetched from external memory then only blocked one thread and the other threads could continue processing and possibly issuing other I/O requests. Our evaluation includes variants both with and without asynchronous I/O.

For our measurements we generated 100k queries and evaluated them on the Laptop system, taking measurements every 100 queries and with a total of 5 repetitions. The results are depicted in Figure 2.4, on the left for 5-DGP, on the right for ordinary CH ($|V|$-DGP). For 5-DGP, disabling prefetching of the OS tremendously decreases the initial query times from about 158 ms down to 6 ms (c.f. *prefetch* vs. *no prefetch*). This query time could be improved even further to 1 *ms* per query by allowing for asynchronous I/O as described above. Note that our laptop system has enough memory to hold the entire data set in memory, so over time more and more of the data necessary to answer a query is already present in internal RAM. At some point the variant with active prefetching has an advantage as it has more of the required data available in internal memory.

For CH essentially the same effects occur, even though the advantage of prefetching kicks in earlier (probably due to the smaller memory footprint of the CH). CH also always benefits from asynchronous I/O.

Note, though, that 5-DGP is always faster than CH by around a factor of 10; at the beginning due to the fewer data items to be fetched from external memory, later due to the faster on-CPU computation.

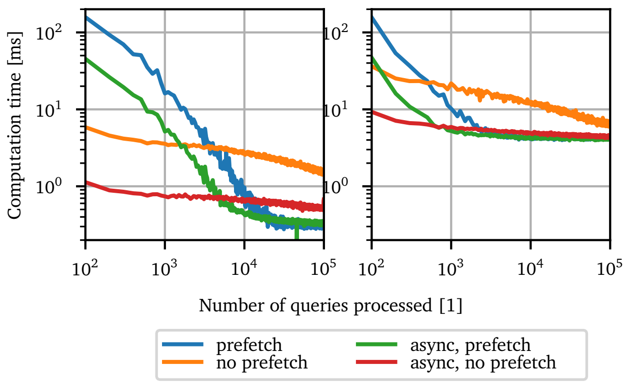


Figure 2.4: Impact of various I/O options on the query time of a single query over the course of 10^5 issued queries. Queries issued at a later stage benefit from memory cache. Figure shows 5-DGP (left) and CH (right). Eurafrasia. Laptop.

2.3.8 Conclusion and Outlook

In this section we presented a framework that unifies the view on very popular lookup- and search-based acceleration schemes for shortest path computations in road networks. Apart from this conceptual contribution, our framework allows for almost arbitrary tradeoffs between space requirements and query times, beating the best known schemes with respect to interesting scenarios in the space-time spectrum. We have evaluated our scheme in the traditional internal memory setting as well as in external memory with very favorable outcomes.

There are several promising avenues of future research. First, we aim at investigating the effect of different node orderings on our scheme. We used the node ordering obtained as a by-product of a standard CH construction which employs prioritization based on edge differences. Studies as in [LUYK17] and [ADGW12] have demonstrated the positive effect of more deliberate node ordering strategies. We hope that they also integrate well with our framework. Second, we focused on choices of $\mathcal{U}$ forming partitions of the node set V. In principle, it might be beneficial to also allow for overlapping sets within $\mathcal{U}$. Furthermore, we plan to find more efficient ways to determine $\mathcal{U}$. And as for most speed-up schemes, a great challenge is the adaptation of our scheme to dynamically changing network topologies.

2.4 Hierarchical Search Without Shortcuts

Hierarchical approaches to compute shortest paths seem to be intrinsically tied to shortcuts. Highway Hierarchies [SS06], Highway Node Routing [SS07], Contraction Hierarchies [GSSD08], Transit Node Routing [BFM06], Hierarchical Hub Labeling [ADGW12], they all heavily rely on the idea of adding shortcuts to the graph (see Section 2.1 for descriptions). Interestingly, the shortcuts are not only the main cause for the speed-up but are also necessary to ensure the correctness of the results. In fact, none of the mentioned algorithms would be correct without shortcuts.

In this section we give an example of hierarchical search without shortcuts. The rough idea is to take a CH $G'(V, E \cup E')$ with node ordering r and to replace the shortcuts E' by another node mapping $r_{max} : V \to \{1, 2, \ldots, n\}$. We show that two bytes per node are sufficient to store the mappings r and r_{max} and that the resulting algorithm is able to improve the query times of Dijkstra's bidirectional algorithm by one order of magnitude. We call our method *Light Contraction Hierarchies*.

2.4.1 Existing Hierarchical Search Without Shortcuts

We are aware of merely one hierarchical approach to compute shortest paths that does not use shortcuts, called REACH [Gut04]. We describe this approach in Section 2.1.1. The auxiliary data consists of one integer per node. The disadvantage of REACH is its costly preprocessing step, which requires an all-to-all shortest path computation. This is too time consuming for large real-world transportation networks.

2.4.2 Light Contraction Hierarchies

In this section, which is based on [Pro22], we describe a new method to compute shortest paths and prove its correctness. As it essentially takes a CH and removes its shortcuts we call the method Light Contraction Hierarchies or LCH. Let $G(V, E)$ be our street network and $G'(V, E \cup E')$, $r : V \to \{1, 2, \ldots, n\}$ the corresponding output of the CH preprocessing step (see Section 1.3.2). Since all distances between nodes in G and G' are identical, for each shortest path $\pi'(s,t)$ in G' we find a shortest path $\pi(s,t)$ in G such that each node $w \in \pi'$ is also contained in π. Let Π be the function that maps shortest paths in G' to their corresponding shortest path in G. We say that Π unpacks the paths in G' to their original form in G. Note that each shortcut $e' := (u,v) \in E'$ is a shortest path itself. Let $\pi(u,v) \in \Pi(e')$ be the unpacked path of e', what ranks do the nodes in π have? Can it happen that there is a node $w \in \pi$ with greater rank than u and v? Lemma 2.7 answers this question with no, which is going to be important for our further considerations.

Lemma 2.7
For every shortcut $e' := (u,v) \in E'$ and every node $w \in \Pi(e')$ with $w \notin \{u,v\}$ it holds $r(w) < \min\{r(u), r(v)\}$.

Proof. We assume that $r(u) < r(v)$. The other case can be shown analogously. Let path $\pi := \Pi(e')$ and suppose that we find a node $w \in \pi$ with $r(w) > r(u)$ and $w \neq v$. The edge set E' is, by definition, of minimum cardinality such that G' is unimodal with respect to the node ordering r. Thus, if we remove edge e' from G', the distance from u to v in the subgraph of G' induced by $V_{r(u)}$ (as defined in Definition 2.2) must be greater than $d(u,v)$. However, we have $d(u,w) + d(w,v) = d(u,v)$, which is a contradiction. Thus, such a node w cannot exist. □

We define the rank $r_E(e)$ of an edge or shortcut e as follows.

$$r_E : E \cup E' \to \{1, 2, \dots, n\},$$
$$r_E((u,v)) = \min\{r(u), r(v)\}$$

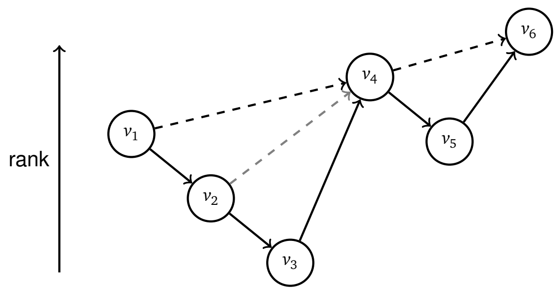
The overall idea of our method is to save for each node u the rank of the most important shortcut e' for which $u \in \Pi(e')$ holds. We call this new mapping r_{max}. Hence, if $r_{max}(u)$ is small for a node u, it means that u is not part of any important shortcut and we may be able to ignore u during our bidirectional search. We formally define r_{max} as follows.

$$r_{max} : V \to \{1, 2, \dots, n\},$$
$$r_{max}(u) = \max\{r(u), \max_{\{e' \in E' : u \in \Pi(e')\}} r_E(e')\}$$

Furthermore, let $r_P(\pi) := \max_{u \in \pi} r(u)$ for a path π. Our method tries to mimic the CH query algorithm in G' without having access to the shortcuts E'. That means, if the upward search G' expands an edge $e \in E$, our method expands this edge as well. And if the upward search in G' expands a shortcut $e' \in E'$, our method follows along the unpacked path $\Pi(e')$. See Figure 2.5 for an example. However, since E' is unavailable, it is unclear how to find the path $\Pi(e')$. In the following lemma we show how the mappings r and r_{max} can help in this task.

Lemma 2.8
Given a shortest upward (or downward) path $\pi'(s,t)$ in CH G'. For every node $w \in \Pi(\pi')$ it holds $r_{max}(w) \geq r_P(\pi(s,w))$, where π is the subpath of $\Pi(\pi')$ in G that starts in s and ends in w.

Proof. Let $\pi'(s,t)$ be a shortest upward path in G' and let node w and path $\pi(s,w)$ be defined as above. Clearly, there is an edge $e := (u,v)$ in π' with $w \in \Pi(e)$. Thus, $r_{max}(w) \geq r_E(e) = r(u)$. As π' is an upward path, it follows from Lemma 2.7 that $r_P(\pi) = r(u)$. The case when π' is a shortest downward path can be shown analogously. □



a) Example CH G' with dashed shortcuts. The path $\pi' = v_1 v_4 v_6$ in G' consisting of two shortcuts (dashed black arrows). The corresponding mimicked path in the original graph G (without shortcuts) is $\pi = v_1 v_2 v_3 v_4 v_5 v_6$.

	v_1	v_2	v_3	v_4	v_5	v_6
$r(v)$	4	2	1	5	3	6
$r_{max}(v)$	4	4	4	5	5	6

b) The mappings r and r_{max} of the example CH G'. The node v_3, for instance, is part of two shortcuts (v_1, v_4) and (v_2, v_4). Therefore, $r_{max}(v_3) = \max\{r(v_3), \min\{r(v_2), r(v_4)\}, \min\{r(v_1), r(v_4)\}\} = 4$.

Figure 2.5: An example CH to demonstrate the mappings r and r_{max} and what it means to mimic a path.

Lemma 2.8 says that whenever we find a shortest path $\pi(s, w)$ during our search with $r_P(\pi) > r_{max}(w)$ we know that π cannot be a subpath of an unpacked shortest upward (or downward) path in G'. Therefore and because we aim to mimic the upward search in G' we do not need to continue our search at w.

We are now ready to describe our hierarchical search, which is a modification of Dijkstra's bidirectional algorithm similar to the query algorithm of CH. We only describe the forward search as the backward search is analogous with flipped edges. Our search performs the same operations as Dijkstra's algorithm but additionally keeps track of $r_P(\pi)$ for each search branch π. Furthermore, the method only expands an edge $e = (u, v)$ if the condition $r_P(\pi(s, u)) \leq r_{max}(v)$ is satisfied. If this is not the case, it follows from Lemma 2.8 that we do not need to settle node v.

The mappings r and r_{max} cause the only space overhead of LCH. In theory, a rank is a number between one and n, where n is the number of nodes in G. Fortunately, the rank can be replaced by the contraction round in which the node was contracted. In this way, many nodes share the same rank and it suffices to spend one byte per rank even for large graphs without any negative impact on the correctness and the query performance. If the CH happens to have more than 256 contraction rounds, we map all nodes contracted after round 255 to rank 255. This does not affect correctness but may slightly slow down the queries. Similar rank plateaus are, for instance, proposed in [FLS17]. Hence, in summary we spend two additional bytes per node. This space overhead is compensated by an efficient way to store the edges as we show and discuss in the following section.

2.4.3 Experiments

We tested our method on the road network of Germany and of South America, which we both obtained from OpenStreetMap [Ope18]. We conducted all experiments on an AMD Threadripper 2950X with 256 RAM and Ubuntu 20.04 as operating system. Our implementations of CH, Dijkstra's

	Germany	South America
Nodes	25,115,477	62,562,908
Edges	50,774,067	129,634,220
Shortcuts of CH	40,956,462	122,964,150

Table 2.8: Details about the size of the street networks that were subject of our experiments.

	Germany	South America
BiDijk	3,024	5,693
LCH	347	349
CH	2	9

Table 2.9: Average shortest path query times in milliseconds.

bidirectional algorithm and LCH are written in C++ (using g++, version 9.3.0) and are publicly available[1]. An experiment consisted of one thousand source-target shortest path queries, selected uniformly at random. We built our CH using a standard approach described in [GSSV12], which always contracts independent sets of nodes with low edge difference. Our CH queries conduct the common speed-up technique *stall-on-demand* [GSSD08].

The construction of the LCH that takes a CH as input, computes the mapping r_{max} and removes the shortcuts took 68 seconds for Germany and 166 seconds for South America (both conducted on a single core). The input CH can be constructed within a couple of minutes in both cases. The CH of Germany was constructed within 303 and the CH of South America within 332 contraction rounds. In both cases less than 120 nodes were contracted after round 255.

Table 2.9 shows the average query times of Dijkstra's bidirectional algorithm (BiDijk), LCH and CH. Note that we do not compare LCH to REACH as the preprocessing of REACH would be too time consuming for the chosen graphs. First, we can observe that LCH is able to improve the query times compared to BiDijk considerably for both street networks. However, the factor of improvement is smaller for Germany (8.71) than for South America (16.31). The average query time of LCH for South America is almost equal to that of Germany despite South America being the larger street network (see Table 2.8). We see the opposite when comparing the query times of BiDijk and CH. Here, the factor of improvement is 1,414 for Germany and 622 for South America. This result indicates that optimizing the query times of CH and of LCH (by choosing appropriate node orderings) are two different objectives. Note that if it is not necessary to have a complete representation of the shortest path, the CH queries can be considerably faster.

We now address the question whether our method is able to mimic the CH upward search efficiently. Let V_{BiDijk} and V_{LCH} be the set of nodes visited during the BiDijk query and LCH query, respectively. Furthermore, let $V_G^\wedge$ be the set of nodes that are adjacent to edges or part of unpacked shortcuts expanded during the CH query. If our method was able to mimic the CH search perfectly, the two sets $V_G^\wedge$ and V_{LCH} would be equal. Thus, the size of $V_G^\wedge$ is a lower bound of the size of V_{LCH}. We examine the efficiency of our method by comparing these two numbers. Table 2.10 shows the results. In both cases, our method is surprisingly close to the lower bound and visits significantly less nodes than BiDijk. The factor to the lower bound is 2.4 for Germany and 2.1 for South America. This shows that the mapping r_{max} is a good indicator whether a node is part of $V_G^\wedge$.

Finally, we look at the actual space requirements of BiDijk, LCH and CH. The edges and shortcuts

[1]https://github.com/proisscs/chlight

	Germany	South America
$\lvert V_{BiDijk}\rvert$	7,578,599	21,384,160
$\lvert V_{LCH}\rvert$	973,928	1,232,816
$\lvert V_G^{\wedge}\rvert$	405,612	598,501
$\lvert V^{\wedge}\rvert$	898	984

Table 2.10: Comparison of the average number of visited nodes in the forward and backward search of BiDijk ($\lvert V_{BiDijk}\rvert$), LCH ($\lvert V_{LCH}\rvert$), CH ($\lvert V^{\wedge}\rvert$) and the average number of nodes that are part of edges/shortcuts expanded during CH queries ($\lvert V_G^{\wedge}\rvert$). The size of $V_G^{\wedge}$ is a lower bound of the size of V_{LCH}.

	Germany	South America
BiDijk	0.94 GB	2.40 GB
LCH	0.91 GB	2.41 GB
CH	1.21 GB	3.27 GB

Table 2.11: Comparison of space requirements.

of our CH are stored in separate arrays to save space as the shortcuts additionally need to contain information about how to unpack them. This could either be two pointers to their child shortcuts/edges or one pointer to the midnode of the shortcut (see [GSSD08] and [GSSV12] for details). The first option allows to unpack the shortcut faster while the second is space conservative. We decided to implement the second option to have a fair comparison. Table 2.11 shows the results. We observe that the space overhead of LCH compared to BiDijk is almost non-existent (South America) or even negative (Germany). LCH does save a considerable amount of space compared to CH but probably not as much as one might have expected. The reason is that for BiDijk it is necessary to store each edge (u, v) twice, once as forward edge for u and once as backward edge for v, to allow an efficient bidirectional search. CH have the great advantage that it suffices to store each edge only once despite conducting bidirectional search. This is because in CH each edge is either an upward edge or a downward edge. The forward search only expands upward edges and the backward search only expands downward edges. In the case of LCH, an edge $e = (u, v)$ is an upward edge if $r_{max}(u) < r(v)$. Analogously, the edge e is a downward edge if $r(u) > r_{max}(v)$. These edges need to be stored only once. This is the reason why LCH can achieve a negative space overhead compared to BiDijk. Most edges are neither an upward nor a downward edge, though, and must be stored for both directions.

2.4.4 Conclusion

We presented a method to perform hierarchical search without shortcuts. We call it Light Contraction Hierarchies or LCH as it is based on Contraction Herarchies, one of the most popular hierarchical search techniques. Our method is very simple to implement (provided a way to construct CH is available). We showed that our method, despite its very modest space overhead, is able to improve the average query time of shortest path computations by one order of magnitude compared to Dijkstra's bidirectional algorithm. Furthermore, we were able to show that our method is also efficient with respect to the number of visited nodes.

An interesting open problem is the relationship between the query times of CH and LCH. Our results indicate that finding a node ordering that achieves good CH query times is a different objective than finding such an ordering for LCH. Furthermore, a drawback of our method is that the preprocessing step needs a CH as input. We plan to investigate if it is possible to construct the LCH efficiently without having the complete CH.

2.5 On the Query Complexity of Contraction Hierarchies

Any graph $G(V, E)$ can be transformed into a unimodal graph by adding shortcuts to E. A space-conservative way is the CH construction that adds for a given node ordering r as few edges E' as possible such that $G'(V, E \cup E')$ is unimodal with respect to r. The most challenging part, however, is finding a good node ordering r.

There are typically two questions about G we would like to answer. First, how many shortcuts E' do we asymptotically need to turn G into a CH? And second, what time complexity of the CH query in $G'(V, E \cup E')$ can we expect? In the following subsection we give an overview of existing upper and lower bounds to these questions. Afterwards we elaborate an example where the number of edges expanded during a CH query greatly exceeds the number of visited nodes. Thus, it is mandatory to take the number of edges into account when we study the complexity of the CH query.

2.5.1 Existing Bounds

The *highway dimension* (HD) h [AFGW10] is among the most successful approaches to upper bound the query complexity of CH. It is primarily used to upper bound the space requirements and time complexity of various hierarchical speed-up techniques such as Transit Nodes, Hub Labeling and, specifically, CH. We are aware of multiple slightly different definitions of HD [ADF+11; ADF+16; AFGW10] but with respect to the upper bounds for CH these versions are equivalent (according to [ADF+16]). We stick to the version of [AFGW10].

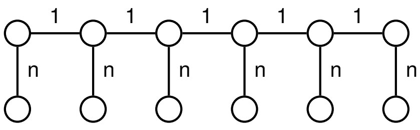
Before we introduce HD, we need to define the distance $d(\pi, v)$ between a path π and a node v. We set

$$d(\pi, v) := \max_{w \in \pi} d(w, v).$$

Definition 2.12
Given an undirected graph $G(V, E)$ with distance function d, the highway dimension of G is the smallest number h such that for any $v \in V$ and $R > 0$ one can find a subset $S \subseteq V$ with at most h elements such that for each shortest path π with $c(\pi) > R$ and $d(\pi, v) < 4R$ one can find a node w in π with $w \in S$.

Thus, the HD upper bounds the size of hitting sets S of shortest paths with specific length and location.

The authors of [AFGW10] show that for an undirected graph $G(V, E)$ with HD h, diameter D and with integral, positive edge weights it is possible to find a set of edges E' such that $G'(V, E \cup E')$ is a CH of G and each node in G' is adjacent to $O(h \log D)$ edges. This upper bound is tight [Mil12]. Furthermore, for any node $v \in V$ the size of the direct search space $D(v)$ is in $O(h \log D)$ as well. The upper bound on the size of the direct search space cannot directly be used to upper bound the query time because, as mentioned in Section 2.2, it is a lower bound of the query time. The authors of [AFGW10] solve this problem by introducing a slight variation of the CH query, which requires to store an additional constant amount of information per node and which ensures that no more than $O(h \log D)$ nodes are settled during the CH query. If we combine these two upper bounds, we get that a CH query expands $O\left((h \log D)^2\right)$ edges, which is tight [Whi15].



a) The comb graph has HD $h \in \Theta(n)$.

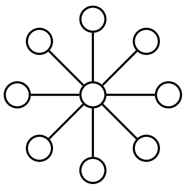

b) The star graph has SD $k \in \Theta(n)$.

Figure 2.6: Example graph families with poor upper bounds.

The main purpose of HD is to explain the success of hierarchical speed-up techniques on road networks. Concerning graphs in general, it is easy to construct examples where the bounds above are rather weak. Consider, for instance, the graph $G(V, E)$ shown in Figure 2.6a, which we call the comb graph. With $R = n - 1$ each path has distance less than $4R$ irrespective of the choice of node v. Furthermore, there are $\frac{n}{2}$ node disjoint shortest paths of length n that need to be hit, which leads to an HD $h \in \Theta(n)$. The resulting upper bounds for CH are trivial and far from being tight. In fact, the comb graph *is* a CH for some node orderings. These node orderings lead to large search spaces, though. There are other node orderings that lead to a size of E' in $\Theta(n)$ and direct search space sizes of $O(\log n)$.

The *skeleton dimension* (SD) k, introduced in [KV17], is another important property to upper bound the query complexity of hierarchical speed-up techniques. Roughly speaking, the skeleton dimension is an upper bound of the width of shortest path trees. The details, however, are more complicated and require some more notation, which is why we do not define it formally here. The upper bounds shown in [KV17] are similar to the ones with HD h. The authors of [KV17] show that for a graph G with constant maximum node degree, HD h and SD k it holds $k \in O(h)$ and that there is a CH G' with average direct search space size in $O(k \log D)$.

Also for the skeleton dimension it is simple to find graph families where the bounds are far from being tight. The comb graph is one example with SD $k \in \Theta(n)$. Another example is the star graph, shown in Figure 2.6b, with $k \in \Theta(n)$. The star graph can be considered as the perfect match for CH because the star graph itself is a CH as long as the center node receives the greatest rank. Furthermore, the search space sizes are in $O(1)$.

Thus, HD and SD, despite their rather sophisticated definitions, are not able to fully capture the query complexity of CH. Note that both properties depend on the cost function of the graph. Thus, two graphs of the same topology but with different cost functions could have different HDs and SDs.

The authors of [BCRW16] pursue a different approach based solely on the topology of the graph. They show, for instance, that for a graph $G(V, E)$ with treewidth t there is a node ordering r and a corresponding CH $G'(V, E \cup E')$ such that $|E'| \in O(tn \log n)$. Furthermore, the search space sizes of G' are in $O(t \log n)$. As both the comb graph and the star graph are trees, their treewidth $t = 1$ and, thus, for these graphs the bounds of [BCRW16] clearly outperform the bounds obtained with HD or SD.

Another approach to upper bound the CH query complexity can be found in [FS15b] and [BFS21]. They use the notion of *bounded growth*, which we define in Section 2.5.2, to show sublinear search space sizes, even if the node ordering is chosen uniformly at random. The remarkable about this result is that the underlying property is much simpler than HD or SD. They show that, if a graph family $G(V, E)$ has bounded growth, one can find a node ordering r such that the corresponding

CH has search spaces of average size $O\left(\sqrt{n}\log n\right)$. However, this upper bound only applies to the number of visited nodes and says nothing about the number of expanded edges. A conservative upper bound for the edges would be the square of the visited nodes, which leads to an upper bound that is worse than the run time of Dijkstra's algorithm in G. And indeed, in Section 2.5.3.1 we elaborate an example where the expected number of expanded edges is in $\Omega(n)$, which is almost as bad.

Lower bounds for the complexity of CH are another interesting subject that has received considerably less attention so far. As the comb graph and the star graph show, HD and SD are not suitable for general lower bounds. In [Whi15] the author shows that for any HD h and diameter D there is a corresponding CH G with a query complexity in $\Omega\left((h\log D)^2\right)$, which shows that the upper bound of [AFGW10] is tight. However, the result of [Whi15] says nothing about the size of G.

In [RF20] and [PR21] a grid graph with carefully chosen edge weights is presented that leads to search space sizes in $\Omega\left(\sqrt{n}\right)$ for all possible node orderings. Furthermore, a separation approach to determine the node ordering as suggested in [BCRW16] leads to a query time in $\Omega(n)$ in this grid graph.

The authors of [BS20] elaborate lower bounds for so called *weak* CH. Weak CH are graphs that are unimodal for every cost function. Thus, they typically have many more edges than normal CH. In [BS20] they show that the average search space size in a weak CH $G(V,E)$ is in $\Omega(\beta_\alpha)$, where $\alpha \geq \frac{2}{3}$ and β_α is the size of a smallest α-balanced node separator in G. An α-balanced separator is a node separator $V' \subseteq V$ such that all connected components in the graph G without V' have size at most αn.

In the remainder of this section we define the graph property bounded growth, define a graph family with bounded growth and show that the expected query time in a CH of this graph is in $\Omega(n)$ if the node ordering is chosen uniformly at random. This result shows that the sublinear upper bounds for the search space sizes presented in [BFS21] do not imply sublinear query complexity. The following work is based on the author's contribution to [PR21], which is a joint publication with Tobias Rupp.

2.5.2 Bounded Growth

We define bounded growth analogously to [FS15b] and [BFS21]. A graph family $G(V,E)$ has bounded growth if there is a constant C such that for any radius R and any node u

$$|\{v \in V:\ d(u,v) = R\}| \leq C \cdot R.$$

Typically, graphs considered in this context have unit edge costs.

2.5.3 Torus Graph

We introduce a graph family which we call *torus graph*. This graph family is subject to the analysis in the following section. A torus graph can be considered as a grid without boundaries. For the sake of simplicity, the edges of this graph are undirected.

Definition 2.13
Let $G = (V,E)$ be an undirected graph with $n = k^2$ nodes, $k \in \mathbb{N}$. Let the nodes be bijectively labeled with the numbers from 1 to n. G is called a torus graph if there exists a labeling such that

$$E = \{(u,v) : |u-v| \equiv 1 \mod k\} \ \cup\ \{(u,v) : |u-v| \equiv k \mod n\}.$$

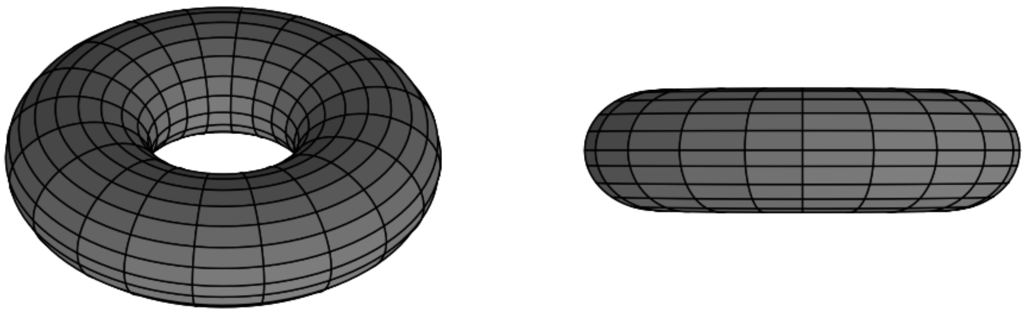


Figure 2.7: Typical visualizations of a torus graph

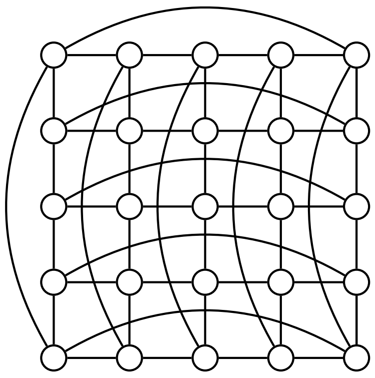


Figure 2.8: Example torus graph with $k = 5$

We can construct a torus graph of k^2 nodes by starting with a grid graph of that size. Then, we add edges connecting opposite boundaries of the grid. If the boundary is a column, then each node in that column is connected to the node in the same row of the opposite boundary. We proceed accordingly with boundaries that are rows. Figure 2.8 shows an example. A possible labeling in this example would be to count the nodes from left to right, top to bottom. In that case the set $\{(u, v) : |u - v| \equiv 1 \mod k\}$ contains the horizontal edges and the set $\{(u, v) : |u - v| \equiv k \mod n\}$ contains the vertical edges.

The torus graph is similar to grid graphs, which appear in real world networks. However, the torus graph shows symmetries grids do not have. In fact, from each node the adjacency structure of the torus graph looks identical.

To make the following analysis as easy as possible we would like to have unit edge costs and unique shortest paths. However, due to the symmetries of the torus graph we cannot have both. A simple solution to this problem is to introduce an ϵ-perturbation of the edge costs such that the cost of each edge is still arbitrarily close to one and the shortest paths are always unique. If we then simply round the path costs to the next integer before applying the definition of bounded growth the perturbation does not impact our proofs. We, thus, decided to introduce the perturbation only implicitly to keep the notation as simple as possible. Therefore, think of the torus graph as a graph with (almost) unit edge costs and unique shortest paths.

The torus graph is a graph family with bounded growth. The bounded growth constant C is 4.

Furthermore, for distances $R < \frac{k}{2}$ and unit edge length there are exactly $4R$ neighbors with that distance. This is true for each node due to the symmetries of the torus graph.

2.5.3.1 Analysis of Edge Search Spaces in Torus Graphs

Based on the work of [BFS21] we study CH with random node ordering. Given a graph G with n nodes. First, a permutation of the numbers from 1 to n is chosen uniformly at random. Second, G is contracted in this order. We call a CH of this kind Random Contraction Hierarchy (RCH).

In this section we show that for the torus graph the expected number of edges that need to be considered during a query of an RCH is in $\Omega(n)$. The graphs in this section always have a random node ordering r.

As we only consider undirected graphs in this section the forward and backward direct search spaces are identical. We denote the direct search space of node v with $D(v)$. Recall that, given a CH $G(V, E)$, the size of the direct search space $D(v)$ of a node $v \in V$ is a worst-case lower bound for the number of nodes one has to settle in a query with v as source. With worst-case lower bound we mean that one can find a target $u \in V$ such that all nodes in $D(v)$ must be settled when computing the distance between v and u. This concept can be expanded to edges. The set of all edges that are adjacent to nodes in $D(v)$ is a worst-case lower bound of the number of edges that need to be processed. Our plan is to prove that there are on average $\Omega(n)$ of them in an RCH of a torus graph.

Thus, direct search spaces play an important role in our proofs. Given a torus graph $G(V, E)$ and a node ordering r, one typical question will be whether a node $u \in V$ is in the $D(v)$ of another node $v \in V$ in the CH of G. The following lemma helps us answering such a question.

Lemma 2.9
Given a graph with unique shortest paths $G(V, E)$ and its CH $G'(V, E \cup E')$ with respect to a node ordering r. Then for two nodes $v, u \in V$ it holds $u \in D(v)$ in G' if and only if for all nodes w in the shortest path $\pi(v, u)$ in G it holds $r(w) \leq r(u)$.

Proof. Let $\pi(v, u)$ be the shortest path from v to u in G and let $\pi'(v, u)$ be a shortest up-down path in the CH G', which exists by definition. Since the distances in G and G' are equal, it follows that each node in π' is also contained in π. Thus, if $r(w) \leq r(u)$ is true for all nodes $w \in \pi$, then π' must be an upward path and, by definition, $u \in D(v)$.

If $u \in D(v)$, then there is a shortest upward path $\pi'(v, u)$ in G'. For each edge $(v', u') \in \pi'$ we know with Definition 2.6 and Lemma 2.7 that u' has the greatest rank among the nodes in the shortest path $\pi(v', u')$ in G. As π' is an upward path, it follows that u has the greatest rank in the shortest path $\pi(v, u)$ in G. □

Thus, if $r(u) > r(v)$ and all intermediate nodes in the shortest path $\pi(v, u)$ have a lower rank than u, then $u \in D(v)$. What if we set the rank of v to be greater than $r(u)$? Then v would be the only node in π with a rank greater than $r(u)$ and, by Definition 2.6, the CH of G contained the edge (v, u). Therefore, depending on $r(u)$ being greater than $r(v)$ we either get a node in $D(v)$ or an edge in the CH. This is going to be very useful in the following considerations.

Definition 2.14
Let $G = (V, E)$ be an undirected graph with node ordering r and unique shortest paths. We define

$$X_u := \{v \in V : \forall w \in \pi(u, v) \setminus \{u, v\} \; r(w) < r(u)\},$$

where $\pi(u,v)$ denotes the shortest path from u to v.

See Figure 2.9 for an example. In words, a node v is in X_u if the shortest path between u and v (excluding u and v) does exclusively consist of nodes with smaller rank than $r(u)$. Hence, if $v \in X_u$, then all nodes in the shortest path $\pi(u,v)$ are in X_u. The following corollary summarizes the meaning of X_u.

Corollary 2.3
Given an undirected CH $G = (V, E \cup E')$ of a graph $G(V,E)$ and two nodes $u, v \in V$ with $v \in X_u$.

1. *If $r(v) < r(u)$, then $u \in D(v)$.*
2. *If $r(v) > r(u)$, then $(u,v) \in E \cup E'$.*

Proof. Follows from Lemma 2.9 and Definition 2.6. □

Our goal is to count edges that need to be processed in a CH query. Surprisingly, with the help of the sets X_u we can do so by considering triples of nodes. Say we have three nodes u_1, u_2 and u_3 with the ranks $r(u_1) < r(u_2) < r(u_3)$ and $u_1, u_3 \in X_{u_2}$. From Corollary 2.3 we know that u_2 is in $DSS(u_1)$. Hence, u_2 is settled during queries from u_1. Moreover, there is an edge $(u_2,u_3) \in E \cup E'$ (Corollary 2.3). Hence, even though it may happen that u_3 is not further processed, the edge (u_2,u_3) definitely needs to be considered during queries from u_1 (because it is an upward edge and u_2 is in $DSS(u_1)$).

Before we continue, we give a short sketch of the remainder of this section. First, we define the TDES to be the set of all triples that have the form of the triple mentioned above. Afterwards, we prove that the cardinality of this set divided by the number of nodes is a lower bound of the average query complexity. Finally, we show that the expected cardinality of the TDES is in $\Omega(n^2)$ in RCH of torus graphs, which completes the proof of our claim.

Definition 2.15
Let $G = (V,E)$ be an undirected graph with node ordering r. The Total Direct Edge Search Space (TDES) *of G is given by*

$$T := \{(u_1,u_2,u_3) \in V^3 \mid r(u_1) < r(u_2) < r(u_3) \text{ and } u_1, u_3 \in X_{u_2}\}.$$

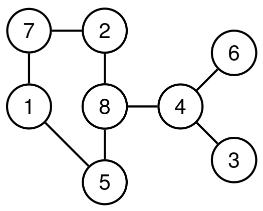
For the graph shown in Figure 2.9

$$T = \{(1,5,7), (1,5,8), (1,7,8), (2,7,8), (3,4,6), (3,4,8), (3,6,8), (4,6,8), (5,7,8)\}.$$

Think of an element $(u_1,u_2,u_3) \in T$ as an edge (u_2,u_3) in the direct search space of u_1 (as motivated above).

Lemma 2.10
Let $G = (V,E)$ be an undirected CH with node ordering r. The value $\frac{1}{n}|T|$ is a lower bound of the average query complexity in G, where T is the TDES of G.

Proof. Let T be the TDES of G. We fix a node $u_1 \in V$ and take all elements of the form $(u_1,u_2,u_3) \in T$ (remember that $r(u_1) < r(u_2) < r(u_3)$). By definition of T, node u_2 is in $D(u_1)$. Moreover, as G is a CH and by definition of T, there must be an edge $(u_2,u_3) \in E$. Since $r(u_2) < r(u_3)$, this edge must be considered during queries from node u_1. Hence, the number of elements of the form $(u_1,u_2,u_3) \in T$



X_1	$\{5,7\}$
X_2	$\{7,8\}$
X_3	$\{4\}$
X_4	$\{3,6,8\}$
X_5	$\{1,7,8\}$
X_6	$\{3,4,8\}$
X_7	$\{1,2,5,8\}$
X_8	$\{1,2,3,4,5,6,7\}$

Figure 2.9: Example graph with node ordering and unit edge length

is a lower bound of the query complexity if the source node is u_1. It directly follows that $\frac{1}{n}|T|$ is a lower bound of the average query complexity. □

In the following we show that the expected size of T is in $\Omega(n^2)$ in RCH of torus graphs. This result in combination with Lemma 2.10 proves our claim that the expected query complexity is in $\Omega(n)$.

In fact, it turns out that for our purposes it is sufficient to consider a subset of T. We divide the nodes into three disjoint sets L, M and H according to their ranks (low, medium and high ranks). The sets are defined as follows.

$$\begin{aligned} L &:= \{u \in V \mid r(u) \leq n - \sqrt{n}\} \\ M &:= \{u \in V \mid n - \sqrt{n} < r(u) \leq n - \frac{\sqrt{n}}{2}\} \\ H &:= \{u \in V \mid n - \frac{\sqrt{n}}{2} < r(u)\} \end{aligned}$$

It is easy to see that there are $\Omega(n^2)$ different triples $(u_1,\ u_2,\ u_3)$ of the form $u_1 \in L$, $u_2 \in M$ and $u_3 \in H$. Let $\tilde{T}$ be the set of these triples:

$$\tilde{T} := \{(u_1,\ u_2,\ u_3) \mid u_1 \in L, u_2 \in M, u_3 \in H\}$$

Note that for each of these triples $(u_1,\ u_2,\ u_3) \in \tilde{T}$ it holds $r(u_1) < r(u_2) < r(u_3)$. However, not every triple in $\tilde{T}$ is also contained in the TDES T. Recall that each triple $(u_1,\ u_2,\ u_3) \in T$ satisfies two constraints. First, $r(u_1) < r(u_2) < r(u_3)$ and second $u_1,\ u_3 \in X_{u_2}$. Some of the triples in $\tilde{T}$ may violate the second constraint. Hence, we can not directly lower bound the size of the TDES T with the size of the set $\tilde{T}$.

We address this issue with the following lemma.

Lemma 2.11
One can find a constant $p > 0$ such that for any torus graph $G = (V, E)$ with random node ordering the following holds. For any $(u_1, u_2, u_3) \in \tilde{T}$ the probability $P((u_1, u_2, u_3) \in T) \geq p$, where T is the TDES of G.

Since the proof of Lemma 2.11 is rather lengthy, we give it at the end of this section. The lemma states that for any torus graph with random node ordering the expected size of the cut $T \cap \tilde{T}$ is lower bounded by $p|\tilde{T}|$, where p is a constant greater zero.

With the help of Lemma 2.11 we can now easily prove the following theorem.

Theorem 2.1
Let $G = (V, E)$ be a torus graph with random node ordering. The expected size of the TDES $E[|T|] \in \Omega(n^2)$.

Proof.

$$
\begin{aligned}
E[|T|] &= \sum_{r(u_2)=1}^{n} \sum_{r(u_1)=1}^{r(u_2)-1} \sum_{r(u_3)=r(u_2)+1}^{n} P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2}) \\
&\geq \sum_{(u_1,\, u_2,\, u_3)\in \tilde{T}} P((u_1,\, u_2,\, u_3) \in T)
\end{aligned}
$$

We can lower bound the probability $P((u_1, u_2, u_3) \in T)$ with the constant $p > 0$ from Lemma 2.11.

$$
\begin{aligned}
E[|T|] &\geq \sum_{(u_1,\, u_2,\, u_3)\in \tilde{T}} p \\
&= \sum_{r(u_1)=1}^{n-\sqrt{n}} \sum_{r(u_2)=n-\sqrt{n}+1}^{n-\frac{\sqrt{n}}{2}} \sum_{r(u_3)=n-\frac{\sqrt{n}}{2}+1}^{n} p \\
&= p(n-\sqrt{n})\frac{\sqrt{n}}{2}\frac{\sqrt{n}}{2} \in \Omega(n^2)
\end{aligned}
$$
□

The following corollary combines Theorem 2.1 and Lemma 2.10 to prove our previously stated claim.

Corollary 2.4
The expected average query complexity of an RCH of a torus graph takes $\Omega(n)$ time.

All that remains is to prove Lemma 2.11.

Proof. If the set $\tilde{T}$ is empty, there is nothing to show. Let us fix three ranks r_1, r_2, r_3 such that we find three nodes $u_1 \in L$, $u_2 \in M$, $u_3 \in H$ with $r(u_1) = r_1$, $r(u_2) = r_2$, $r(u_3) = r_3$ and, thus, $(u_1, u_2, u_3) \in \tilde{T}$.

The probability that $(u_1, u_2, u_3) \in T$ (with T being the TDES) depends on the distances the nodes u_1, u_2 and u_3 have to each other. Unfortunately, as the ranks r_1, r_2 and r_3 are assigned to the nodes uniformly at random, these distances are random variables.

We therefore introduce the random variables d_1 and d_3, where d_1 denotes the distance between the nodes u_1 and u_2. Analogously, the random variable d_3 denotes the distance between the nodes u_3 and u_2. The term $P(d_1 = R)$, thus, denotes the probability that the distance between u_1 and u_2 is R.

Furthermore, let $\delta_\pi \in \{0, 1\}$ be the random variable that is one if $u_3 \in \pi(u_1, u_2)$ and zero otherwise. With the law of total probability we get

$$
\begin{aligned}
P((u_1,\, u_2,\, u_3) \in T) &= P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2}) \\
&= \sum_{i=0}^{1} P(\delta_\pi = i) P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2} \mid \delta_\pi = i).
\end{aligned}
$$

We know that u_3 cannot be part of $\pi(u_1, u_2)$ if $u_1 \in X_{u_2}$ (because $r_3 > r_2$). Therefore, the term simplifies to

$$
P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2}) = P(\delta_\pi = 0) P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2} \mid \delta_\pi = 0).
$$

We further transform the equation by applying the law of total probability two more times.

$$\begin{aligned}
&P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2}) \\
&= P(\delta_\pi = 0)P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2} \mid \delta_\pi = 0) \\
&= \sum_{R_3=1}^{D} \Big(P(d_3 = R_3)P(\delta_\pi = 0 \mid d_3 = R_3)P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2} \mid \delta_\pi = 0 \cap d_3 = R_3)\Big) \\
&= \sum_{R_1=1}^{D} P(d_1 = R_1) \sum_{R_3=1}^{D} \Big(P(d_3 = R_3 \mid d_1 = R_1)P(\delta_\pi = 0 \mid d_1 = R_1 \cap d_3 = R_3) \\
&\qquad\qquad P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2} \mid \delta_\pi = 0 \cap d_1 = R_1 \cap d_3 = R_3)\Big),
\end{aligned} \tag{2.3}$$

where D is the diameter of G. In the following, we analyze this lengthy equation term by term.

We start with $P(d_1 = R_1)$. Let $N(v, R)$ be the number of nodes from node v with distance R. Due to the symmetric structure of the torus graph the function N does not depend on the node v. Hence, we can write $N(R)$ instead. Since u_1 can be any node other than u_2 with the same likelihood we have

$$P(d_1 = R_1) = \frac{N(R_1)}{n-1}.$$

Our following considerations only require a rough estimate of this term. It is easy to see that for any torus graph we have

$$N(R) \geq R \text{ if } R \leq \frac{\sqrt{n}}{2}.$$

Note that for $R < \frac{\sqrt{n}}{2}$ we have $N(R) = 4R$. Hence, we obtain the lower bound

$$P(d_1 = R_1) \geq \frac{R_1}{n} \text{ if } R_1 \leq \frac{\sqrt{n}}{2}.$$

Next, we estimate the term $P(d_3 = R_3 \mid d_1 = R_1)P(\delta_\pi = 0 \mid d_1 = R_1 \cap d_3 = R_3)$ in Equation 2.3. We distinguish between the two cases $R_1 = R_3$ and $R_1 \neq R_3$.

If $R_1 = R_3$, we know that $P(d_3 = R_3 \mid d_1 = R_1) = \frac{N(R_3)-1}{n-2}$ because there is one "free seat" less with distance R_3 to node u_2 and there are $n-2$ possible candidates (any node except u_1 and u_2). Moreover, u_3 cannot be part of $\pi(u_1, u_2)$ if it has the same distance to u_2 as u_1. Hence, $P(\delta_\pi = 0 \mid d_1 = R_1 \cap d_3 = R_3) = 1$.

If $R_1 \neq R_3$, $P(d_3 = R_3 \mid d_1 = R_1) = \frac{N(R_3)}{n-2}$ and $P(\delta_\pi = 0 \mid d_1 = R_1 \cap d_3 = R_3) \geq \frac{N(R_3)-1}{N(R_3)}$ because at each distance R_3 there is at most one node that is part of $\pi(u_1, u_2)$ (there could be none if $R_3 > R_1$).

In both cases $P(d_3 = R_3 \mid d_1 = R_1)P(\delta_\pi = 0 \mid d_1 = R_1 \cap d_3 = R_3) \geq \frac{N(R_3)-1}{n-2}$ holds. For distances $R \leq \frac{\sqrt{n}}{2}$ the rough lower bound $N(R) \geq R+1$ holds for any torus graph. Hence, we obtain the lower bound

$$P(d_3 = R_3 \mid d_1 = R_1)P(\delta_\pi = 0 \mid d_1 = R_1 \cap d_3 = R_3) \geq \frac{R_3}{n} \text{ for } R_3 \leq \frac{\sqrt{n}}{2}.$$

We address the remaining term $P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2} \mid \delta_\pi = 0 \cap d_1 = R_1 \cap d_3 = R_3)$ in Equation 2.3 from a combinatorial point of view. But first, let us put in words what the term means. It is the conditional probability that u_1 and u_3 are in X_{u_2} if the distances of these nodes to u_2 are known and if u_3 is not part of the shortest path $\pi(u_1, u_2)$.

Let $\Pi := \pi(u_1, u_2) \cup \pi(u_2, u_3)$ be the union of the two shortest paths from u_1 and u_3 to u_2. The only thing we know about the set Π for sure is that it contains u_1, u_2 and u_3. The remaining $|\Pi| - 3$ nodes are unknown to us. However, in order to make the term $u_1 \in X_{u_2} \cap u_3 \in X_{u_2}$ true, these nodes must have a smaller rank than u_2. Hence, there are $r(u_2) - 2$ nodes we can choose from (u_1 is already taken). On the other hand, there are $n - 3$ possible candidates that could be part of Π. Due to the randomness of the node order, each node among these $n - 3$ candidates is part of Π with the same likelihood. That leads to

$$P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2} \mid \delta_\pi = 0 \cap d_1 = R_1 \cap d_3 = R_3) = \frac{\binom{r(u_2)-2}{|\Pi|-3}}{\binom{n-3}{|\Pi|-3}}.$$

The problem remains that we do not know the size of Π. However, we can easily see that the probability decreases with increasing size of Π. That means, an upper bound of the size of Π leads to a lower bound of the considered probability. Taking the distances $d_1 = R_1$ and $d_3 = R_3$ into account it is clear that

$$|\Pi| \le R_1 + R_3 + 1.$$

Hence, we get

$$P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2} \mid \delta_\pi = 0 \cap d_1 = R_1 \cap d_3 = R_3) \ge \frac{\binom{r(u_2)-2}{R_1+R_3-2}}{\binom{n-3}{R_1+R_3-2}}.$$

We are able to get rid of the binomial coefficients with the following considerations.

$$\begin{aligned} \frac{\binom{r(u_2)-2}{R_1+R_3-2}}{\binom{n-3}{R_1+R_3-2}} &= \frac{\prod_{i=1}^{R_1+R_3-2} r(u_2) - i - 1}{\prod_{i=1}^{R_1+R_3-2} n - i - 2} \\ &\ge \left(\frac{r(u_2) - R_1 - R_3}{n}\right)^{R_1+R_3-2} \\ &\ge \left(\frac{r(u_2) - \sqrt{n}}{n}\right)^{R_1+R_3-2} \quad \text{if } R_1, R_3 \le \frac{\sqrt{n}}{2}. \end{aligned} \tag{2.4}$$

We obtain (2.4) by replacing each factor $r(u_2) - i - 1$ of the numerator with the lower bound $r(u_2) - R_1 - R_3$ and each factor $n - i - 2$ of the denominator with the upper bound n.

We now come back to (2.3). We substitute the diameter D with $\frac{\sqrt{n}}{2}$. In this way, we only consider distances d_1 and d_2 that are less or equal to $\frac{\sqrt{n}}{2}$. Hence, we can apply the previously shown lower bounds.

$$\begin{aligned} P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2}) &\ge \sum_{R_1=1}^{\frac{\sqrt{n}}{2}} \frac{R_1}{n} \sum_{R_3=1}^{\frac{\sqrt{n}}{2}} \frac{R_3}{n} \left(\frac{r(u_2) - \sqrt{n}}{n}\right)^{R_1+R_3-2} \\ &\ge \frac{1}{n^2} \sum_{R_1=1}^{\frac{\sqrt{n}}{2}} R_1 \sum_{R_3=1}^{\frac{\sqrt{n}}{2}} R_3 \left(1 - \frac{2}{\sqrt{n}}\right)^{R_1+R_3-2} \end{aligned} \tag{2.5}$$

In Equation (2.5) we move the variable n to the front and replace $r(u_2)$ by its lower bound (as stated in the lemma) $n-\sqrt{n}$.

We continue with $x := 1-\frac{2}{\sqrt{n}}$.

$$\begin{aligned} P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2}) &\geq \frac{1}{n^2} \sum_{R_1=1}^{\frac{\sqrt{n}}{2}} R_1 \sum_{R_3=1}^{\frac{\sqrt{n}}{2}} R_3 x^{R_1+R_3-2} \\ &= \frac{1}{n^2} \sum_{R_1=1}^{\frac{\sqrt{n}}{2}} R_1 x^{R_1-1} \sum_{R_3=1}^{\frac{\sqrt{n}}{2}} R_3 x^{R_3-1} \end{aligned}$$

Writing the lower bound in this way makes it clear that we are facing two nested arithmetico-geometric sums. Remember that for any $x \geq 0$ and $m \in \mathbb{N}$ it holds

$$\begin{aligned} \sum_{i=1}^{m} i x^{i-1} &= \frac{1-(m+1)x^m + m x^{m+1}}{(1-x)^2} \\ &= \frac{1-(m+1-mx)x^m}{(1-x)^2}. \end{aligned}$$

With $m = \frac{\sqrt{n}}{2}$ we get

$$\begin{aligned} P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2}) &\geq \frac{1}{n^2} \sum_{R_1=1}^{\frac{\sqrt{n}}{2}} R_1 x^{R_1-1} \frac{1-\left(\frac{\sqrt{n}}{2}+1-\frac{\sqrt{n}}{2}x\right)x^{\frac{\sqrt{n}}{2}}}{(1-x)^2} \\ &= \frac{1}{n^2} \sum_{R_1=1}^{\frac{\sqrt{n}}{2}} R_1 x^{R_1-1} \frac{1-\left(\frac{\sqrt{n}}{2}+1-\frac{\sqrt{n}}{2}\left(1-\frac{2}{\sqrt{n}}\right)\right)x^{\frac{\sqrt{n}}{2}}}{(1-x)^2} \\ &= \frac{1-2x^{\frac{\sqrt{n}}{2}}}{n^2(1-x)^2} \sum_{R_1=1}^{\frac{\sqrt{n}}{2}} R_1 x^{R_1-1}. \end{aligned}$$

Again, we apply the formula for arithmetico-geometric sums. With a similar calculation as above we get

$$\begin{aligned} P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2}) &\geq \frac{\left(1-2x^{\frac{\sqrt{n}}{2}}\right)^2}{n^2(1-x)^4} \\ &= \frac{1}{16}\left(1-2x^{\frac{\sqrt{n}}{2}}\right)^2 \\ &= \frac{1}{16}\left(1-2\left(1-\frac{2}{\sqrt{n}}\right)^{\frac{\sqrt{n}}{2}}\right)^2. \end{aligned}$$

Let

$$f(n) := \left(1-\frac{2}{\sqrt{n}}\right)^{\frac{\sqrt{n}}{2}}.$$

One can show that the function f is monotonically increasing for $n \geq 4$ and that

$$\lim_{n\to\infty} f(n) = \frac{1}{e},$$

where e is Euler's number. In particular, $f(n) < \frac{1}{2}$ for $n \geq 4$. That means, for $n \geq 4$

$$\begin{aligned} P(u_1 \in X_{u_2} \cap u_3 \in X_{u_2}) &\geq \frac{1}{16}\left(1 - \frac{2}{e}\right)^2 \\ &> 0. \end{aligned}$$

As $n = 4$ is the smallest possible size of a torus graph with edges, the lemma is true for all torus graphs. □

2.5.4 Conclusion

We discussed existing bounds on the query complexity of CH. Especially with respect to lower bounds not much progress has been made so far. With respect to upper bounds, the most successful approach is in our humble opinion [BCRW16], which is solely based on the topology of the graphs. The results of [BFS21], despite their elegant derivation, do not imply sublinear query complexity, which we showed in Section 2.5.3.1.

In summary, we have to note that the complexity of the CH query is far from being fully understood. Our results indicate that the number of expanded edges during a CH query can greatly exceed the number of visited nodes, even when the CH was constructed from a graph with bounded growth.

2.6 Concluding Remarks

In this chapter we addressed the subject of hierarchical search on the practical as well as on the theoretical level. The focus of our considerations lied on Contraction Hierarchies, as they are among the most popular and influential approaches of hierarchical SPAs.

We presented in Section 2.3 a method called Distance Closures that combines the ideas behind CH and HHL to a new speed-up technique. Distance Closures outperform state-of-the-art SPAs in terms of space-time efficiency. In our opinion this is our most significant contribution of this chapter. We also think that there is still a lot to learn and to improve about this method. One open aspect is to find good distance group separators in an efficient way, which is one of the first steps in the preprocessing phase and which greatly influences the performance of the method. Another interesting open research question is whether optimizing the node ordering for CH is equal to optimizing it for distance groups.

In Section 2.4 we showed that it is possible to perform hierarchical search without employing shortcuts. In this way we are able to reduce the space requirements compared to CH. Furthermore, the shortest path computation is conducted in the original network and not in an overlay graph, which can be beneficial in certain applications. The speed-up of our approach, though, cannot compete with methods that do rely on shortcuts.

Finally, we discussed the current state of research with respect to complexity bounds for the CH query phase. We outlined that, while there are interesting results regarding upper bounds, there is not as much progress in finding lower bounds. We concluded the section with a detailed analysis of specific CH defined in [BFS21] showing that there is a surprisingly large gap between the number of visited nodes and the number of traversed edges in the query phase.

Chapter 3 Preference-Based Routing

The optimal route from A to B often depends on individual preferences. For instance, a bicyclist may decide to take a detour in order to avoid a steep passage while another prefers to take the direct route. Thus, the cost of a path depends on the personal preferences and is influenced by multiple cost types such as travel time, euclidean distance and energy consumption. If we assign multiple cost types to the edges, however, it becomes unclear how to compute shortest paths. We need to define a model that formalizes the customization of edge costs to personal preferences and that allows us to compute shortest paths.

In this chapter we consider a simple, yet versatile model for preference-based routing introduced in [FS15a], which we call *linear preference model*. The idea is to assign weights to the cost types according to their personal importance. For instance, if a bicyclist prefers to take a route with low energy consumption, her weight of this cost type would be high compared to the other weights. The vector of these weights is then called the preference vector or simply the preference. We then compute shortest paths by assigning the scalar product of cost vector and preference vector to the edges, which is why we call the model a linear model.

Note that, in contrast to many other articles in the literature, we treat the terms path, route and trajectory as synonyms.

The remainder of the chapter is organized as follows. We first introduce the common preliminaries of the subsequent sections and then dive into a theoretical analysis of the linear preference model (Section 3.2). We show that the number of a special kind of shortest paths, called extreme shortest paths, can be upper bounded by $n^{O\left(\log^{d-1} n\right)}$, where n is the number of nodes in the graph and d is the number of cost types.

Afterwards, in Section 3.3 and 3.4, we present two applications of the linear preference model. We first show how to efficiently elicit preferences based on past trajectories of a single driver. The second application deals with trajectories of multiple drivers. We present an algorithm that clusters trajectories based on driving preferences, which, for instance, allows to analyze mobility patterns.

3.1 Preliminaries

In this section we discuss common preliminaries of the chapter and show some well-known properties.

3.1.1 Definitions

In the whole chapter, if not explicitly stated otherwise, we assume that a directed graph $G(V, E)$ without multi-edges and self-loops is given. In contrast to the previous chapter, however, we allow multidimensional cost functions $c: E \to \mathbb{R}^d_{\geq 0}$. We write $c \in \mathcal{C}_d$ or say c is d-dimensional/d-metric if c maps to $\mathbb{R}^d_{\geq 0}$. We extend the cost function to paths p as usual:

$$c(p) := \sum_{e \in p} c(e).$$

We call $c(p)$ the cost vector of p.

Definition 3.1
A d-dimensional preference *(also: driving preference) $\alpha \in \mathbb{R}^d$ is a convex weighting of the d-dimensional cost function. Thus, $\sum_{\alpha_i \in \alpha} \alpha_i = 1$ and $\alpha_i \geq 0\ \forall i \leq d$. The d-dimensional preference space $\mathcal{P}_d$ is the set of all d-dimensional preferences.*

For a set $f \subseteq \mathbb{R}^d$, we define its dimension $\dim(f)$ to be the maximum number of affinely independent points in f minus one. For a finite set $P \subset \mathbb{R}^d$ we denote the number of elements in P with $|P|$.

The preference space $\mathcal{P}_d$ lives in d dimensions but is itself a $(d-1)$-dimensional simplex due to the constraint $\sum_{\alpha_i \in \alpha} \alpha_i = 1$. For instance, the preference space $\mathcal{P}_3$ is a triangle.

With the help of a preference α we can map a d-dimensional cost vector to a scalar as described in the following.

Definition 3.2
Given a path p in G, a cost function $c \in \mathcal{C}_d$ and a preference $\alpha \in \mathcal{P}_d$, then

$$c(\alpha, p) := \alpha c(p)^T$$

is the aggregated cost *of path p (with respect to preference α), where $\alpha c(p)^T$ is the dot product of the (row) vectors α and $c(p)$.*

Note that a single edge $e \in E$ is itself a path such that Definition 3.2 also describes the aggregated cost of edges. Furthermore, note that the aggregated cost of any path is non-negative irrespective of the choice of the cost function c and the preference α. The reason is that cost vectors as well as preferences must not have negative entries. Thus, having the aggregated costs of the edges we can apply Dijkstra's algorithm to compute preference-based or α-shortest paths as described in Definition 3.4.

In the context of preference-based routing it is sometimes beneficial to consider paths as cost vectors.

Definition 3.3
Given a set of paths Π and a cost function $c \in \mathcal{C}_d$ for some $d \geq 1$, we call $P(\Pi, c) := \{c(\pi) \mid \pi \in \Pi\}$ the point set induced by the path set Π and the cost function c.

With this definition we can define α-shortest paths in a general way.

Definition 3.4
Given a set of paths $\varPi$, a cost function $c \in \mathcal{C}_d$ and a preference $\alpha \in \mathcal{P}_d$ for some $d \geq 1$, we call a cost vector $v \in P(\varPi, c)$ α-shortest path (with respect to $\varPi$ and c) if and only if

$$\alpha v^T = \min_{v' \in P(\varPi, c)} \alpha v'^T.$$

Furthermore, we say that v is a shortest or optimal path with respect to $P(\varPi, c)$ if and only if there is a preference α for which v is an α-shortest path in $P(\varPi, c)$.

It may seem a bit unusual that we speak of shortest paths v that actually are mere cost vectors. The benefit is that this notation allows a flexible description of shortest paths.

The most common setting we face in the remainder of the chapter, though, is that $\varPi$ is the set of all simple paths from a node $s \in V$ to a node $t \in V$ in our graph G. We write $\varPi_{st}$ in that case and call $\varPi_{st}$ a *complete* path set. The following definition facilitates the notation for this setting.

Definition 3.5
Given a path $\pi(s, t)$ in graph G and a cost function $c \in \mathcal{C}_d$ with $d \geq 1$, we say that path π is an α-shortest path for some preference $\alpha \in \mathcal{P}_d$ if and only if

$$\alpha c(\pi)^T = \min_{v \in P(\varPi_{st}, c)} \alpha v^T.$$

Furthermore, we say that path π is a shortest or optimal path if there is a preference $\alpha \in \mathcal{P}_d$ for which π is an α-shortest path.

We continue with our general notation of point sets to define preference polyhedra as follows.

Definition 3.6
Given a finite set of points $P \subset \mathbb{R}^d$ for some $d \geq 1$ and an element $v \in P$, the set

$$f_P(v) := \{\alpha \in \mathcal{P}_d \mid \forall v' \in P \; \alpha\left(v - v'\right)^T \leq 0\}$$

is called the preference polyhedron *of v with respect to P.*

Preference polyhedra are the intersection of half-spaces and, thus, are convex objects. For a path set $\varPi$ and cost function $c \in \mathcal{C}_d$ the preference polyhedron $f_{P(\varPi,c)}(c(\pi))$ is the set of preferences for which path $\pi \in \varPi$ is optimal with respect to the set $\varPi$ and the cost function c. Preference polyhedra are of interest for practical applications as we demonstrate in Section 3.4.

Clearly, every preference polyhedron $f_P(v)$ with $P \subseteq \mathbb{R}^d$ is by definition a subset of $\mathcal{P}_d$. However, not for every $v \in P$ with $f_P(v) \neq \emptyset$ we necessarily have $\dim(f_P(v)) = \dim(\mathcal{P}_d) = d-1$. Let us consider the point set $P := \{(0, 10), (5, 5), (10, 0)\}$. Every point in P is a $(0.5, 0.5)$-shortest path because the sum of the two cost entries always equals 10. However, the point $v := (5, 5)$ is not optimal for any other $\alpha \in \mathcal{P}_2$. Thus, $f_P(v) = \{(0.5, 0.5)\}$ and $\dim(f_P(v)) = 0$.

Definition 3.7
Given a finite set of points $P \subset \mathbb{R}^d$, we call the subset

$$\mathcal{M}_P := \{v \in P \mid \dim(f_P(v)) = d - 1\}$$

minimum preference cover (MPC) of P.

The reason for calling $\mathcal{M}_P$ a *minimum* preference cover is shown by the following lemma.

Lemma 3.1
Given a finite point set $P \subset \mathbb{R}^d$, let $\mathcal{X}$ be defined as

$$\mathcal{X} := \{X \subseteq P \mid \bigcup_{v \in X} f_P(v) = \mathcal{P}_d\},$$

then $\mathcal{M}_P \in \mathcal{X}$ and $|\mathcal{M}_P| = \min_{X \in \mathcal{X}} |X|$.

Proof. First, note that the preference polyhedra of the points in $\mathcal{M}_P$ cover $\mathcal{P}_d$ since the preference polyhedra are closed objects and since the number of points in P is finite.

We prove the rest of the lemma by contradiction and assume that we have a set $X \in \mathcal{X}$ with $|X| < |\mathcal{M}_P|$. W.l.o.g. we may assume that for every point $v \in X$ we have $\dim(f_P(v)) = d-1$ because otherwise the preference polyhedra of the set $X \setminus \{v\}$ would cover $\mathcal{P}_d$ as well. Let $v' \in \mathcal{M}_P \setminus X$. We know that $\dim(f_P(v')) = d-1$. Thus, there must be a point $v \in X$ with $\dim(f_P(v) \cap f_P(v')) = d-1$. The intersection $f_P(v) \cap f_P(v')$ is contained in the hyperplane $H := \{\alpha \in \mathbb{R}^d \mid \alpha(v-v')^T = 0\}$. We therefore can conclude that $\dim(H \cap \mathcal{P}_d) = d-1$. This can only be true if $\mathcal{P}_d \subset H$. There is only one such hyperplane, namely $H' = \{\alpha \in \mathbb{R}^d \mid \alpha \mathbb{1}_d^T = 1\}$, where $\mathbb{1}_d$ is a row vector with d ones as entries.

At least one entry of $v - v'$ must be negative as otherwise v' would dominate v and $f_P(v) = \emptyset$. Let $v - v'$ be negative in the i-th dimension. Let $\alpha' \in \mathcal{P}_d$ be the corner of $\mathcal{P}_d$ with a one in the i-th dimension (note that $\alpha' \in H'$). Then $\alpha'(v-v')^T < 0$ and, thus, $\alpha' \notin H$, which is a contradiction to $H = H'$. □

Corollary 3.1
Given any finite point set $P \subset \mathbb{R}^d$ and any point $v \in P$, then for the preference polyhedron $f_P(v)$ it holds

$$f_P(v) = \{\alpha \in \mathcal{P}_d \mid \forall v' \in \mathcal{M}_P \ \alpha\left(v - v'\right)^T \leq 0\}.$$

Proof. First, note that the difference to Definition 3.6 is that only points $v' \in \mathcal{M}_P$ are considered. The statement follows from the fact that $\bigcup_{v \in \mathcal{M}_P} f_P(v) = \mathcal{P}_d$ (Lemma 3.1). □

In general, not every optimal path is part of the minimum preference cover as demonstrated in the small example above. However, one can show that we can enforce that every optimal path is in the minimum preference cover by introducing infinitesimal perturbations in the edge costs.

Thus, even though there might be shortest paths $v \in P(\Pi, c)$ with $v \notin \mathcal{M}_{P(\Pi,c)}$, the shortest paths in $\mathcal{M}_{P(\Pi,c)}$ are special and deserve a name.

Definition 3.8
Given a path set Π and a cost function $c \in \mathcal{C}_d$ for some $d \geq 1$, we call shortest paths $v \in \mathcal{M}_{P(\Pi,c)}$ extreme shortest paths *(with respect to Π and c).*

It is easy to see that a path $P(\Pi, c)$ is an extreme shortest path if and only if

$$\{\alpha \in \mathcal{P}_d \mid \forall v' \in P(\Pi, c) \ \alpha\left(v - v'\right) < 0\} \neq \emptyset.$$

This set is almost equivalent to the definition of preference polyhedra (Definition 3.6) but with a strict inequality, which is the motivation for the prefix *extreme*.

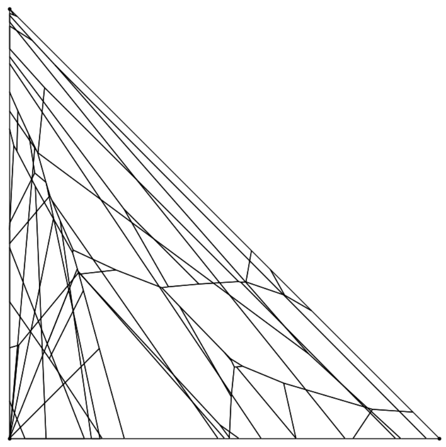


Figure 3.1: Example 3-metric preference space subdivision $\mathcal{S}$.

Figure 3.1 shows how the preference space $\mathcal{P}_3$ is divided into cells that correspond to preference polyhedra of extreme shortest paths from a source node s to a target node t. We call such an arrangement preference space subdivisions as defined in the following.

Definition 3.9
Given a finite set of points $P \subset \mathbb{R}^d$, the preference space subdivision *(PSS) $\mathcal{S}_P$ is the arrangement induced by the set $\{f_P(v) \mid v \in P\}$. We write $f_P^r \in \mathcal{S}_P$ for an r-dimensional facet of $\mathcal{S}_P$.*

A PSS is like a jigsaw consisting of preference polyhedra of extreme shortest paths because the preference polyhedra only intersect each other at their boundaries. We state two important properties of how preference polyhedra intersect each other in the following lemma.

Lemma 3.2
Given a finite set of points $P \subset \mathbb{R}^d$ with $d \geq 1$, for any two $v, v' \in \mathcal{M}_P$ we have $\dim(f_P(v) \cap f_P(v')) < d-1$. Furthermore, for any $v \in P$ there is a $v' \in \mathcal{M}_P$ with $f_P(v) \subseteq f_P(v')$.

Proof. Given two points $v, v' \in \mathcal{M}_P$, let H be the hyperplane described by $\alpha(v-v')^T = 0$, $\alpha \in \mathbb{R}^d$. For the first part of the lemma we have to show that $\dim(H \cap \mathcal{P}_d) \leq d-2$. This is the case if $\mathcal{P}_d \nsubseteq H$. The vector $v - v'$ must have at least one positive and one negative entry (otherwise, one vector would dominate the other). Thus, H cannot be parallel to $\sum_{i \leq d} \alpha_i = 1$ and $\mathcal{P}_d \nsubseteq H$ follows.

Now we come to the second part of the lemma. If $\dim(f_P(v)) = d-1$, then $v \in \mathcal{M}_P$ and we are finished. Furthermore, if $f_P(v) = \emptyset$ there is nothing to show. Thus, we may assume that $0 \leq \dim(f_P(v)) < d-1$. From Corollary 3.1 we know that

$$f_P(v) = \{\alpha \in \mathbb{R}^d \mid \alpha\left(v - v'\right)^T \leq 0 \;\forall\; v' \in \mathcal{M}_P\}.$$

Therefore, if $\dim(f_P(v)) < d-1$, there must be a point $v' \in \mathcal{M}_P$ and a hyperplane H' described by $\alpha(v-v')^T = 0$ with $f_P(v) \subseteq H'$. It follows that $f_P(v) \subseteq f_P(v')$. $\square$

In the following we introduce our notation for the complexity of the PSS.

Definition 3.10
Given a finite set of points $P \subset \mathbb{R}^d$, then $\varphi_d^r(P)$ is the number of r-dimensional facets in the PSS $\mathcal{S}_P$.

We are mainly interested in the complexity of PSS that are induced by point sets of paths.

Definition 3.11
Given a path set Π, we define $\varphi_d^r(\Pi)$ with $1 \leq r \leq d-1$ as

$$\varphi_d^r(\Pi) := \max_{c \in \mathcal{C}_d} \varphi_d^r(P(\Pi, c)).$$

Furthermore, we define $\varphi_d^r(n)$ to be the maximum of $\varphi_d^r(\Pi_{st})$ over all complete path sets Π_{st} of graphs with n nodes.

Note that, by Lemma 3.2, it holds $\varphi_d^{d-1}(\Pi) = \max_{c \in \mathcal{C}_d} |\mathcal{M}_{P(\Pi,c)}|$.

3.1.2 Parametric Shortest Paths

The notion of parametric shortest paths (PSP) is similar to our linear preference model and dates back to (at least) the 1980's. In this section we introduce PSP and discuss their similarities to α-shortest paths. We introduce PSP analogously to [GR19].

Given an acyclic, directed graph $G(V, E)$, in the context of PSP the weights w_e of the edges $e \in E$ are linear functions of the form

$$w_e(\lambda) := a_e \lambda + b_e$$

with $a_e, b_e \in \mathbb{R}$ being edge specific coefficients. The cost of a path p is then given by $C(p, \lambda) := \sum_{e \in p} w_e(\lambda)$. Clearly, if we compute the shortest path from a node $s \in V$ to a node $t \in V$ for different values of $\lambda \in \mathbb{R}$, we may get different paths. The function

$$C(\lambda) := \min_{p \in \Pi_{st}} C(p, \lambda)$$

is the shortest path cost function, which is a concave, piece-wise linear function. The maximum possible number of pieces of $C(\lambda)$ with respect to the size of G is called the *parametric shortest path complexity*.

A typical problem addressed in the context of PSP is to find bounds for the parametric shortest path complexity. For instance, it is well known that the number of pieces in $C(\lambda)$ is upper bounded by $n^{O(\log n)}$, where n is the number of nodes in G [Gus80].

In the following we show that, for non-negative values of λ, PSP and the linear preference model are equivalent with respect to shortest paths. We define the mapping $\gamma : \mathbb{R}_{\geq 0} \to \mathcal{P}_2$ as follows:

$$\gamma(\lambda) := \left(\frac{\lambda}{\lambda+1}, \frac{1}{\lambda+1} \right). \tag{3.1}$$

Furthermore, we define the cost function of G as $c(e) := (a_e, b_e)$. Thus, for each path p in G and each $\lambda \geq 0$ we have

$$C(p, \lambda) = (\lambda + 1) \cdot c(\gamma(\lambda), p). \tag{3.2}$$

Therefore, if a path π is a parametric shortest path for a $\lambda \geq 0$, it is also a $\gamma(\lambda)$-shortest path. With

$$\gamma^{-1} : \mathcal{P}_2 \to \mathbb{R}_{\geq 0},$$
$$\gamma^{-1}(\alpha) := \frac{\alpha_1}{\alpha_2},$$

the other way around holds as well except for the edge case $\alpha = (1, 0)$.

For negative λ we do not find such an equivalence because the preference vectors α must not have negative entries. The main reason is that we want to avoid negative aggregated edge costs in order to be able to apply Dijkstra's algorithm. Furthermore, we avoid negative cost cycles in this way. If a graph G exhibits negative cost cycles, there are typically infinitely many shortest paths of infinite hop length in G. In the context of PSP this issue is resolved by requiring G to be acyclic. With the linear preference model we do not need this constraint.

Another difference between PSP and the preference model is that PSP is commonly defined for two edge coefficients a_e and b_e whereas the preference model is defined for an arbitrary number of cost types. However, the notion of PSP can easily be extended to more than two dimensions as it is done, for instance, in [GR19].

3.1.3 Linear Programming

Linear programming is an important branch of optimization techniques. As linear programming is part of this work but not the focus, we only give a very brief introduction and refer to standard work such as [Van+20] for interested readers.

A d-dimensional linear program (LP) consist of an unknown vector $x \in \mathbb{R}^d$, a set of linear inequalities $Ax \leq b$ (also called constraints) and a linear objective function $\min c^T x$.

$$\begin{aligned} &\textsc{Minimize} && 3 \cdot x_1 - x_2 \\ &\textsc{subject to} && 2 \cdot x_1 + x_2 \leq 12 \\ & && -x_2 \leq 5 \\ & && -x_1 + x_2 \leq -3 \end{aligned} \tag{3.3}$$

Equation 3.3 gives a small example of an LP with two variables x_1 and x_2 and with three constraints (the matrix A is therefore a 3×2-matrix). The set

$$F := \{x' \in \mathbb{R}^d \mid Ax' \leq b\} \tag{3.4}$$

is called the *feasible region* of the LP and any point $x' \in F$ is a feasible solution to the LP. Note that, as F is the intersection of halfspaces, F is a convex object. The optimal solution x^* of the LP is defined as

$$x^* := \underset{x' \in F}{\operatorname{argmin}}\ c^T x'. \tag{3.5}$$

An LP is *unbounded* iff the feasible region has infinite volume. Otherwise, the LP is bounded. An LP does always have a (finite) optimal solution if its feasible region is nonempty and if the LP is bounded. If the LP is bounded we can restrict our search for the optimal solution x^* to the corners of its feasible region F.

This observation is used by one of the most popular algorithms to compute x^*, called the simplex algorithm [DOW+55]. Roughly speaking, the simplex algorithm jumps from corner to corner of F

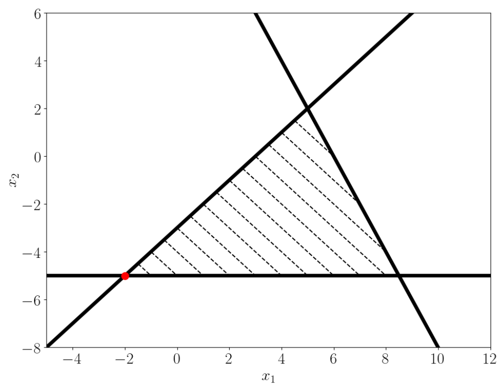


Figure 3.2: Feasible region (dashed) of LP 3.3 with optimal solution in red.

following the (opposite) direction of the objective function vector c. Each corner x' of F is described by the intersection of d hyperplanes that in turn are described by setting d constraints (or rows) A_i of the matrix A equal to their corresponding right hand side in b. These d constraints are called the basis of the corner x'. The simplex algorithm now jumps from corner to corner by replacing one constraint A_i in the basis with another constraint in A in such a way that the objective function value $c^T x$ does not increase. If none such constraint exists, one can show that the current corner $x' = x^*$. The strategy to select the constraint A_i to be replaced is called the *pivoting rule*.

There are known pivoting rules such that the simplex algorithm always finds the optimal solution x^* within finite steps [Bla77]. However, while the simplex algorithm shows good performance in practice, it is still an open question whether there is a pivoting rule that always finds the optimal solution within a number of steps that is polynomial in the problem size.

The ellipsoid method is the first known algorithm to solve any LP in polynomial time, which was shown by Leonid Khachiyan [BGT81]. The typical formulation of the ellipsoid method does not find the optimal solution x^* but any feasible solution $x \in F$ or a proof that $F = \emptyset$. One can show that, complexity-wise, this problem is as difficult as finding an optimal solution x^*.

The algorithm requires the existence of an oracle that, given any point $x \in \mathbb{R}^d$, decides whether $x \in F$ and, if $x \notin F$, returns a constraint A_i with $A_i x > b_i$. The method then starts with an ellipsoid $\mathcal{E}_0$ with center x_0 for which we know that $F \neq \emptyset \Rightarrow F \cap \mathcal{E}_0 \neq \emptyset$. It asks the oracle whether $x_0 \in F$. If so, the algorithm terminates and returns x_0. If not, the oracle returns a violated constraint A_i and the ellipsoid method computes the ellipsoid $\mathcal{E}_1$ of minimal volume that contains the intersection of $\mathcal{E}_0$ and the half-space $A_i x \leq b_i$. This procedure is repeated until either a feasible solution $x \in F$ or a proof for $F = \emptyset$ is found.

The ellipsoid method itself is not of practical importance but was inspiration for practically relevant algorithms [Kar84].

3.1.4 Shortest Path Computation

In this section we discuss the problem of finding the α-shortest path in a d-metric graph $G(E, V)$ for any preference $\alpha \in \mathcal{P}_d$, which is also called the *personalized route planning problem* [FS15a]. In principle, Dijkstra's algorithm can easily be modified to solve the personalized route planning problem. The modification additionally takes the preference as input and computes the aggregated cost of the edges on the fly. However, in this chapter we would like to conduct extensive experiments with thousands of paths in large road networks and Dijkstra's algorithm is too slow for this task.

Fortunately, the authors of [FLS17] present a modification of the CH preprocessing such that the resulting CH can answer shortest path queries correctly for all preferences. Barth et al. [BFS19] further improve this modification and call it *Multicriteria Contraction Hierarchies*. When contracting a node $v \in V$ the normal CH preprocessing has to check whether a path of the form $\pi(u,w) := (u,v)(v,w)$ is the only shortest path from u to w in the partly contracted graph $G'_{r(v)}$ as defined in Section 1.3.2. This can be done rather efficiently with the help of Dijkstra's algorithm. In the case of Multicriteria Contraction Hierarchies (MCH), though, one has to check whether π is the only α-shortest path for *any* preference $\alpha \in \mathcal{P}_d$. As there are infinitely many preferences, this task appears to be considerably more difficult.

Thus, the problem we need to solve can be phrased as follows: Given a path $\pi(u,w)$, is there a preference α such that π is an α-shortest path? If so, we need to add the shortcut (u,w). In [FLS17] this problem is expressed as the following LP, which we call the *shortest path LP*.

$$\begin{aligned} &\sum_{i=1}^{d} \alpha_i = 1 && \textit{normalization constraint} \\ &\forall i \in \{1,\ldots,d\}\colon \alpha_i \geq 0 && \textit{non-negativity contraints} \\ &\forall p \in \Pi_{uw}\colon c(\alpha,\pi) - c(\alpha,p) \leq 0 && \textit{optimality constraints} \end{aligned} \tag{3.6}$$

Our problem is equivalent to the question whether the feasible region of the shortest path LP is empty (which is the reason why the LP does not have an objective function). The set of simple paths Π_{uw} from node u to node w may have a size that is exponential in the graph size. Thus, an explicit formulation of the shortest path LP is often infeasible. Fortunately, the ellipsoid method [GLS81] does not require an explicit formulation of the LP in order to solve it in polynomial time. It merely needs to have a polynomial time oracle that, given a candidate α, decides whether it satisfies all constraints and, if not, returns a violated constraint. This oracle can be realized with Dijkstra's algorithm [FLS17] if the underlying street network of the trajectories is considered as part of the input.

Following [FLS17], we usually use an implementation of the simplex algorithm [DOW+55] and start with an LP that only consists of the normalization constraint and the non-negativity constraints shown in (3.6). We solve the LP, take its solution α, compute the α-shortest path π' and check whether $c(\alpha,\pi) - c(\alpha,\pi') = 0$. If so, preference α is a solution to the shortest path LP and we are finished. Otherwise, we found a violated constraint, add it to the LP and repeat the process until either a solution is found or the feasible region is empty.

Note that in an MCH it is often necessary to have more than one shortcut between two nodes as there is typically more than one shortest path between two nodes. In fact, this number can become rather large, which renders the space requirements of an MCH infeasible in practice. The solution proposed in [FLS17] is to stop the contraction at a certain stage (typically after contracting around 99.9% of the nodes) and to assign the same highest rank to the remaining nodes. Within this rank plateau we then fall back to a normal bidirectional search in the query phase. This issue with multi-edges is also the reason why typical customization approaches such as Customizable Contraction Hierarchies [DSW14] (see Section 2.1.6 for a description) are not directly applicable.

3.2 On the Number of Extreme Shortest Paths

We now approach the linear preference model from a theoretical point of view. The problem of bounding the parametric shortest path complexity (PSP complexity, see Section 3.1.2) can also be addressed in the context of the linear preference model. In this section, which is based on joint work with Florian Barth and Stefan Funke [BFP22], we generalize results for the PSP complexity for two and three dimensions to arbitrary dimensions.

3.2.1 Introduction

Given a finite set of points $P \subset \mathbb{R}^d$, what is the number of vertices (or extreme points) of the convex hull of P? This is a frequently asked question, for instance, in the area of Multi-objective Linear Programming [BS00] or in probability theory [BKST78]. In this work, we examine this question for the case when P is the set of cost vectors of paths in a graph $G(V, E)$ with multiple edge costs.

Figure 3.3 shows example cost vectors with two metrics. The red points are non-dominated (or Pareto-optimal) and, thus, may be the solution to constrained minimization problems. However, optimizing over all non-dominated cost vectors often turns out to be too expensive as, for instance, in the constrained shortest path problem [HZ80; MZ00]. A typical strategy in such cases is to restrict the set of possible solutions to the non-dominated extreme points of the convex hull (circled in blue). The extreme points have the property that for any convex combination of the metrics there is at least one extreme point optimal for it. Therefore, extreme points are interesting on their own and one can hope that restricting the search to extreme points will lead to a good approximation of the optimal solution.

Let P_{st} be the set of cost vectors of all simple paths from a node s to a node t in a graph G. While it is easy to see that the number of non-dominated points in P_{st} can be exponential in the size of G, there is little known about the complexity of the extreme points in P_{st}. In this work we tackle the problem of counting the extreme points in P_{st}, which we call *extreme shortest paths*. We show that the number of extreme shortest paths in P_{st} is in $n^{O\left(\log^{d-1} n\right)}$, where n is the number of nodes in G and d is the fixed number of metrics. Thus, complexity-wise there is indeed a considerable gap between extreme points and non-dominated points in P_{st}.

Related Work

The PSP complexity (see Section 3.1.2) is closely related to our problem as, complexity-wise, counting the pieces of the shortest path cost function $C(\lambda)$ is equivalent to counting extreme shortest paths from s to t. Therefore, all results regarding the parametric shortest path complexity are related to our work.

For the two-metric case, it is well known that the number of pieces in $\mathcal{C}(\lambda)$ is upper bounded by $n^{O(\log n)}$, where n is the number of nodes in G [Gus80]. This upper bound is tight [Car83a; Car83b; MS00], even for planar graphs [GR19].

Gajjar and Radhakrishnan [GR19] extend the parametric shortest path problem to three dimensions by setting

$$w_e(\lambda := (\lambda_1, \lambda_2, \lambda_3)) := a_e\lambda_1 + b_e\lambda_2 + c_e\lambda_3.$$

They show that in this case the number of extreme shortest paths is in $n^{(\log n)^2 + O(\log n)}$.

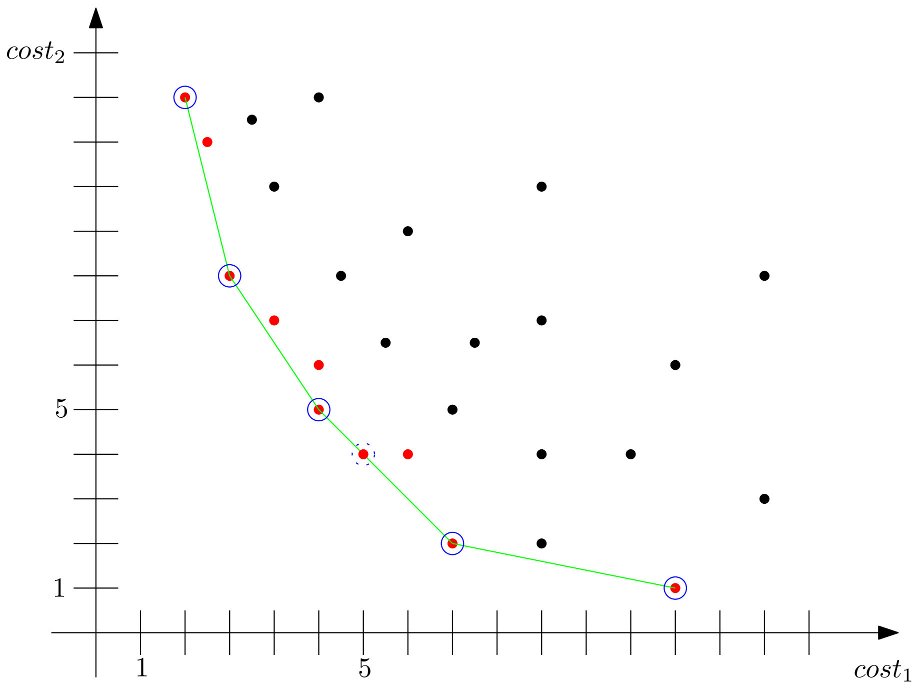


Figure 3.3: Paths in cost space (black and red dots); Pareto-optimal paths (red); (lower left part of) the boundary of the convex hull of all Pareto-optimal paths in green; extreme shortest paths/extreme points of the CH circled in blue; shortest (but not extreme) path dot-circled in blue.

We are not aware of any results regarding the parametric shortest path complexity beyond three dimensions. Parts of our way (especially Section 3.2.3.2) to prove the general upper bound are inspired by the proof for three dimensions in [GR19].

Both, Pareto-optimal paths as well as extreme shortest paths have been instrumented to create alternative route recommendations. The former approach, pursued e.g. in [DW09; KRS10], unfortunately only seems to be viable on rather small graphs due to the too rapidly growing number of Pareto-optimal paths. Restricting to extreme shortest paths, though, as in [FS15a], has been shown to be feasible in different practical application scenarios [BFP21; BFS19].

3.2.2 Preliminaries

In this section we introduce terminology that is specific to Section 3.2.

With $\Pi_{st}(l)$ we denote all simple paths from $s \in V$ to $t \in V$ (with s and t being arbitrary but fixed nodes) with at most $\lceil l \rceil$ edges (l is a real number as we need to divide it by two in a recursion later on). Even though the paths are restricted in the hop length, we call $\Pi_{st}(l)$ a complete path set.

Definition 3.12
We define $\varphi_d^r(n, l)$ to be the maximum of $\varphi_d^r(\Pi_{st}(l'))$ over all $l' \leq l$, all possible graphs $G(V, E)$ with n nodes and all possible node pairs $s, t \in V$.

3.2.3 A General Upper Bound on the Number of Extreme Shortest Paths

In this section we prove the following theorem.

Theorem 3.1
For any fixed but arbitrary $d \geq 1$ and any cost function $c \in \mathcal{C}_d$ the number of extreme shortest paths in $P(\Pi_{st}(n), c)$ is upper bounded by $n^{O\left(\log^{d-1} n\right)}$.

Note that Theorem 3.1 is equivalent to the statements $\varphi_d^{d-1}(n) \in n^{O\left(\log^{d-1} n\right)}$ (Definition 3.11). Furthermore, it directly follows from Definition 3.12 that $\varphi_d^{d-1}(n) \leq \varphi_d^{d-1}(n, n)$. We prove Theorem 3.1 by showing that $\varphi_d^{d-1}(n, n) \in n^{O\left(\log^{d-1} n\right)}$. Our approach is to find a function f such that $\varphi_d^{d-1}(n, n) \leq f\left(\varphi_{d-1}^{d-2}(n, n)\right)$. Thus, we upper bound $\varphi_d^{d-1}(n, n)$ with a recursion in the dimension d, which is an idea also used in the proof of the upper bound for $d = 3$ in [GR19].

We obtain the function f as follows. First, we show that the intersection of a d-dimensional PSS and a hyperplane can have at most the complexity of a $(d-1)$-dimensional PSS (Lemma 3.3). In fact, this observation is based on a similar result shown in [GR19] in the context of parametric shortest paths. Second, we use this insight to upper bound $\varphi_d^{d-2}(n, l)$ for special path sets (Lemma 3.4 and 3.5), which then allows us to construct a second recursion of the form $\varphi_d^{d-2}(n, l) \leq g\left(\varphi_d^{d-2}\left(n, \frac{l}{2}\right)\right)$. This leads to an upper bound for $\varphi_d^{d-2}(n, l)$ based on $\varphi_{d-1}^{d-2}(n, l)$ (Lemma 3.6). The proof of Theorem 3.1 then uses Lemma 3.6 together with an observation we discuss in Section 3.2.3.1 to construct the function f. We consider Lemma 3.4 and 3.5 in Section 3.2.3.3 as our main contributions as these ingredients allow us to generalize the ideas shown in [GR19] to arbitrary values d.

3.2.3.1 A First Upper Bound on the Number of Cells

The number of $(d-2)$-dimensional facets in a PSS is an upper bound of the number of $(d-1)$-dimensional facets in the same PSS for the following reasons. Given any finite point set $P \subset \mathbb{R}^d$ with $d > 1$. Each facet f_P^{d-2} of $\mathcal{S}_P$ supports at most two $(d-1)$-dimensional facets. Moreover, every facet f_P^{d-1} is supported by at least d $(d-2)$-dimensional facets. This is true because the preference space $\mathcal{P}_d$ is bounded itself and, thus, there is no unbounded cell in $\mathcal{S}_P$. Therefore, we have

$$\varphi_d^{d-1}(P) \leq \frac{2}{d}\varphi_d^{d-2}(P). \tag{3.7}$$

For the case $d = 1$, our task of counting extreme shortest paths is simple. It holds

$$\varphi_1^0(P) \leq 1 \tag{3.8}$$

for any finite point set P because $\mathcal{M}_P = \{\min_{v \in P} v\}$.

3.2.3.2 Bounding the Complexity of PSS Intersections

In this section we show that the complexity of the intersection of a hyperplane with a d-metric PSS is upper bounded by the complexity of a $(d-1)$-metric PSS. This observation is a crucial ingredient of our proof of Theorem 3.1 as it allows us to construct recursive upper bounds in the dimension d. The authors of [GR19] prove a similar statement in the context of parametric shortest paths. As their setting and notation slightly differ from ours, we decided to give a proof of the following lemma.

Lemma 3.3
Let $d > 1$ and let H_d be a hyperplane in d dimensions with $\mathcal{P}_d \not\subseteq H_d$ that intersects the preference space $\mathcal{P}_d$. Then for any path set Π and cost function $c \in \mathcal{C}_d$ the set $Y := \{f_{P(\Pi,c)}^{d-1} \cap H_d \mid \dim\left(f_{P(\Pi,c)}^{d-1} \cap H_d\right) = d-2\}$ has at most $\varphi_{d-1}^{d-2}(\Pi)$ elements.

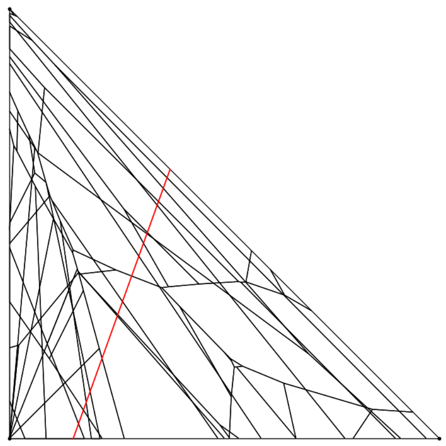


Figure 3.4: Example preference space subdivision with red hyperplane intersection

Proof. We fix an arbitrary cost function $c \in \mathcal{C}_d$ and define $P := P(\Pi, c)$. See Figure 3.4 for an example of the intersection $H_d \cap \mathcal{P}_d$. The set $Z := \{f \cap H_d \mid f \in \mathcal{S}_P\}$ looks like a $(d-1)$-metric PSS. However, it is unclear whether the complexity bounds for $(d-1)$-metric PSS also apply for sets like Z, which live in d dimensions. This lemma says that the answer is yes in the case of extreme shortest paths.

Our strategy to prove Lemma 3.3 looks as follows.

1. We show that for each $v \in P$ there is at most one $f \in Y$ with $\dim(f_P(v) \cap f) = d-2$.
2. We define mappings $\eta : \mathbb{R}^d \to \mathbb{R}^{d-1}$ and $\beta : H_d \cap \mathcal{P}_d \to \mathcal{P}_{d-1}$ such that for each $\alpha \in H_d \cap \mathcal{P}_d$ and each $v \in P$ it holds $\alpha v^T = \beta(\alpha)\eta(v)^T$.

We prove these points later on and first show that they are sufficient to prove the lemma. Let $\eta(P)$ and $\beta(f_P)$ be the mapped sets P and f_P. It follows from the second point that for any $v \in P$ with $\dim(f_P(v) \cap H_d) = d-2$ it holds

$$\beta(f_P(v) \cap H_d) \subseteq f_{\eta(P)}(\eta(v)).$$

Therefore, for each $f \in Y$ there is also an extreme shortest path $v' \in \mathcal{M}_{\eta(P)}$ with $\beta(f) \subseteq f_{\eta(P)}(\eta(v'))$ (see Lemma 3.2).

Let $v' \in \mathcal{M}_{\eta(P)}$ be any extreme shortest path with respect to $\eta(P)$. Then the set $W := \{f \in Y \mid \beta(f) \subseteq f_{\eta(P)}(v')\}$ contains at most one element. Otherwise, we could take any cost vector $v \in P$ with $\eta(v) = v'$ and show, using point two, that all elements of W are subsets of $f_P(v)$. This contradicts the first point.

In summary, for each element $f \in Y$ we find an extreme shortest path $v' \in \mathcal{M}_{\eta(P)}$ with $\beta(f) \subseteq f_{\eta(P)}(v')$. Furthermore, for each extreme shortest path $v' \in \mathcal{M}_{\eta(P)}$ we find at most one $f \in Y$ with $\beta(f) \subseteq f_{\eta(P)}(v')$. Thus, it is possible to injectively map Y to $\mathcal{M}_{\eta(P)}$. As $|\mathcal{M}_{\eta(P)}| \leq \varphi_{d-1}^{d-2}(\Pi)$ by definition and because we chose the cost function c arbitrarily, this finishes the proof.

It remains to show that the two points are correct. Assume that we find a cost vector $v \in P$ and two elements $f_1, f_2 \in Y$ with $f_1 \subseteq f_P(v)$ and $f_2 \subseteq f_P(v)$. We know from Lemma 3.2 that there is a $v' \in \mathcal{M}_P$ with $f_P(v) \subseteq f_P(v')$. Let $f_3 := f_P(v') \cap H_d$. Clearly, $f_1 \subseteq f_3$ and $f_2 \subseteq f_3$. As $f_1 \neq f_2$ at least one of them cannot be equal to f_3. W.l.o.g. let $f_2 \subset f_3$.

By definition of Y, we find a shortest path $v_2 \in \mathcal{M}_P$ with $f_2 = f_P(v_2) \cap H_d$. As $f_2 \subset H_d$ and $f_3 \subset H_d$

it follows that H_d is described by $\alpha(v' - v_2)^T = 0$. But then $f_P(v_2) \cap H_d = f_P(v') \cap H_d = f_3$, which is a contradiction to $f_2 \subset f_3$.

We come to the second point. W.l.o.g. let the intersection $H_d \cap \mathcal{P}_d$ be describable by an equation of the form

$$\alpha_1 = x_2 \cdot \alpha_2 + x_3 \cdot \alpha_3 + \cdots + x_{d-1} \cdot \alpha_{d-1} + b =: f(\alpha)$$

with $x_i, b \in \mathbb{R}$. We map each point $v := (v_1, v_2, \dots, v_d) \in P$ to

$$\eta(v) := (v_2 + v_1 \cdot \tilde{v}_{2,1} + v_d \cdot \tilde{v}_{2,d},\ v_3 + v_1 \cdot \tilde{v}_{3,1} + v_d \cdot \tilde{v}_{3,d}, \dots,\ v_d + v_1 \cdot \tilde{v}_{d,1} + v_d \cdot \tilde{v}_{d,d})$$

with $\tilde{v}_{i,1} = x_i + b$, $\tilde{v}_{i,d} = -(x_i + b)$ and $x_d = 0$. Note that η reduces the number of dimensions by one.

Let the map $\beta : H_d \cap \mathcal{P}_d \to \mathcal{P}_{d-1}$ be defined as follows:

$$\beta(\alpha) := (\alpha_2, \alpha_3, \dots, \alpha_d + \alpha_1).$$

For any $v \in P$ and $\alpha \in H_d \cap \mathcal{P}_d$ we have

$$\begin{aligned}
\alpha \cdot v^T =& \left(f(\alpha), \alpha_2, \dots, 1 - f(\alpha) - \sum_{1<i<d} \alpha_i \right) \cdot (v_1, \dots, v_d)^T \\
=& f(\alpha)(v_1 - v_d) + v_d + \sum_{1<i<d} \alpha_i (v_i - v_d) \\
=& b(v_1 - v_d) + (\alpha_1 + \alpha_d) v_d + \sum_{1<i<d} \alpha_i x_i (v_1 - v_d) + \sum_{1<i<d} \alpha_i v_i \\
=& (\alpha_1 + \alpha_d)\left(v_d + v_1 \tilde{v}_{d,1} + v_d \tilde{v}_{d,d}\right) + \sum_{1<i<d} \alpha_i \left(v_i + v_1 \tilde{v}_{i,1} + v_d \tilde{v}_{i,d}\right) \\
=& \beta(\alpha)\, \eta(v)^T.
\end{aligned}$$

□

3.2.3.3 Decomposing Path Sets

In this section we first look at path sets Π_{sut} that can be written as the pairwise concatenation of two path sets Π_{su} and Π_{ut} that end/start at a common node u. We will see that in such a case the PSS $\mathcal{S}_{P(\Pi_{sut}, c)}$ is the overlay of $\mathcal{S}_{P(\Pi_{su}, c)}$ and $\mathcal{S}_{P(\Pi_{ut}, c)}$ for any cost function c (see Figure 3.5 for an example). Thus, we can upper bound $\varphi_d^{d-2}(\Pi_{sut})$ with the help of Lemma 3.3.

Lemma 3.4

Let Π_{su} and Π_{ut} be two not necessarily complete path sets such that each path in Π_{su} ends at node $u \in V$ and each path in Π_{ut} starts at node u. Furthermore, let $\Pi_{sut} := \{\pi_1\pi_2 \mid \pi_1 \in \Pi_{su}, \pi_2 \in \Pi_{ut}\}$ be the pairwise concatenation of Π_{su} and Π_{ut}. Then it holds

$$\varphi_d^{d-2}(\Pi_{sut}) \leq \varphi_{d-1}^{d-2}(\Pi_{ut}) \cdot \varphi_d^{d-2}(\Pi_{su}) + \varphi_{d-1}^{d-2}(\Pi_{su}) \cdot \varphi_d^{d-2}(\Pi_{ut}).$$

Proof. We fix an arbitrary cost function $c \in \mathcal{C}_d$ and prove the inequality for c. Let $P_{sut} := P(\Pi_{sut}, c)$, $P_{su} := P(\Pi_{su}, c)$ and $P_{ut} := P(\Pi_{ut}, c)$. Given any $\alpha \in \mathcal{P}_d$, let v_{su} and v_{ut} be the α-shortest paths in P_{su}

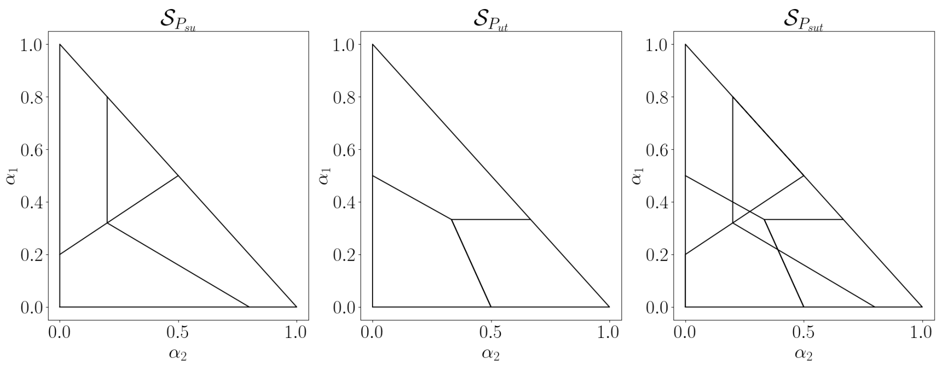


Figure 3.5: This figure illustrates the meaning of Lemma 3.4. With the two path sets Π_{su} and Π_{ut} the PSS $\mathcal{S}_{P_{sut}}$, as defined in Lemma 3.4, is the overlay of $\mathcal{S}_{P_{su}}$ and $\mathcal{S}_{P_{ut}}$. An edge $f^1_{P_{su}}$ can cause multiple edges in the PSS $\mathcal{S}_{P_{sut}}$ by intersecting the PSS $\mathcal{S}_{P_{ut}}$. Lemma 3.4 says that $f^1_{P_{su}}$ can be split into at most $\varphi^1_2(\Pi_{ut})$ many edges in $\mathcal{S}_{P_{sut}}$.

and P_{ut}. Then with $v_{sut} := v_{su} + v_{ut}$ it holds $v_{sut} \in P_{sut}$ and v_{sut} is the α-shortest path in P_{sut}. This is true because all paths in Π_{sut} go via node u. Thus, $\mathcal{S}_{P_{sut}}$ is the overlay of $\mathcal{S}_{P_{su}}$ and $\mathcal{S}_{P_{ut}}$.

Every facet $f^{d-2}_{P_{su}}$ is part of a hyperplane H_d. If we let H_d intersect the preference subdivision $\mathcal{S}_{P_{ut}}$, we know from Lemma 3.3 that this intersection contains no more than $\varphi^{d-2}_{d-1}(\Pi_{ut})$ $(d-2)$-dimensional facets. Thus, in the overlay of $\mathcal{S}_{P_{su}}$ and $\mathcal{S}_{P_{ut}}$ the facet $f^{d-2}_{P_{su}}$ can be split into at most $\varphi^{d-2}_{d-1}(\Pi_{ut})$ $(d-2)$-dimensional facets as well. An analogous statement holds for the facets $f^{d-2}_{P_{ut}}$, which finishes the proof. □

The following lemma addresses the problem of upper bounding $\varphi^{d-2}_{d}(\Pi')$ if the path set Π' is the union of multiple path sets.

Lemma 3.5

Given k path sets $\Pi_1, \Pi_2, \dots, \Pi_k$ and let $\Pi' := \bigcup_{1\le i\le k} \Pi_i$, then it holds for every $d > 1$

$$\varphi^{d-2}_{d}(\Pi') \le 2 \cdot k \cdot \varphi' \cdot \sum_{1\le i\le k} \varphi^{d-2}_{d}(\Pi_i),$$

with $\varphi' := \max_{1\le i\le k} \varphi^{d-2}_{d-1}(\Pi_i)$.

Proof. We fix an arbitrary cost function $c \in \mathcal{C}_d$ and define $P' := P(\Pi', c)$ and $P_i := P(\Pi_i, c)$ for all $1 \le i \le k$.

Now, let us fix a facet $f^{d-2}_{P'} \in \mathcal{S}_{P'}$. The facet is part of a hyperplane H that is described by $\alpha(v_1 - v_2)^T = 0$ with $v_1, v_2 \in \mathcal{M}_{P'}$ and $v_1 \ne v_2$.

We first assume that v_1 and v_2 belong to the same set P_i. As $P_i \subseteq P'$, we have $v_1, v_2 \in \mathcal{M}_{P_i}$ and $f_{P'}(v_1) \cap f_{P'}(v_2) \subseteq f_{P_i}(v_1) \cap f_{P_i}(v_2)$. Therefore, $\dim\left(f_{P_i}(v_1) \cap f_{P_i}(v_2)\right) = d-2$ and there is a facet $f^{d-2}_{P_i} = f_{P_i}(v_1) \cap f_{P_i}(v_2)$. Thus, we have at most $\varphi^{d-2}_{d}(\Pi_i)$ such pairs v_1, v_2 in $\mathcal{M}_{P_i}$.

We now assume that v_1 and v_2 belong to different sets P_i and P_j. From $P_i \cup P_j \subseteq P'$ it follows that $f_{P'}(v_1) \cap f_{P'}(v_2) \subseteq f_{P_i}(v_1) \cap f_{P_j}(v_2)$. Thus, $\dim\left(f_{P_i}(v_1) \cap f_{P_j}(v_2)\right) \ge d-2$.

Therefore, there must be a facet $f^{d-2}_{P_i} \subset f_{P_i}(v_1)$ that intersects with $f_{P_j}(v_2)$ or a facet $f^{d-2}_{P_j} \subset f_{P_j}(v_2)$ that intersects with $f_{P_i}(v_1)$ (or both).

In summary, for each facet $f_{P'}^{d-2}$ we either find a corresponding facet $f_{P_i}^{d-2}$ with $f_{P'}^{d-2} \subseteq f_{P_i}^{d-2}$ (case one) or an intersection $f_{P_i}^{d-2} \cap f_{P_j}^{d-1}$ (or $f_{P_i}^{d-1} \cap f_{P_j}^{d-2}$) with $\dim\left(f_{P_i}^{d-2} \cap f_{P_j}^{d-1}\right) = d-2$ (case two). Clearly, we can upper bound the first case with $\sum_{1\leq i\leq k} \varphi_d^{d-2}(\Pi_i)$.

The second case we handle with the help of Lemma 3.3. The intersection of the facet $f_{P_i}^{d-2}$ with any PSS $\mathcal{S}_{P_j}$ can contain at most $\varphi_{d-1}^{d-2}\left(\Pi_j\right) \leq \varphi'$ $(d-2)$-dimensional facets (Lemma 3.3). Moreover, there are at most two extreme shortest paths in P_i that are adjacent to $f_{P_i}^{d-2}$. Thus, in total the facet $f_{P_i}^{d-2}$ can contribute to at most $2 \cdot (k-1) \cdot \varphi'$ intersections of case two. Therefore, the number of facets $f_{P'}^{d-2}$ of case two can be upper bounded by $2 \cdot (k-1) \cdot \varphi' \cdot \sum_{1\leq i\leq k} \varphi_d^{d-2}(\Pi_i)$.

With $\sum_{1\leq i\leq k} \varphi_d^{d-2}(\Pi_i) + 2 \cdot (k-1) \cdot \varphi' \cdot \sum_{1\leq i\leq k} \varphi_d^{d-2}(\Pi_i) \leq 2 \cdot k \cdot \varphi' \cdot \sum_{1\leq i\leq k} \varphi_d^{d-2}(\Pi_i)$ the statement follows. □

3.2.3.4 Proving the Upper Bound via Recursion

In the following lemma we describe how we can upper bound $\varphi_d^{d-2}(n,l)$ with $\varphi_{d-1}^{d-2}(n,l)$, which we then use to prove Theorem 3.1.

Lemma 3.6

$\varphi_d^{d-2}(n,l) \leq d \cdot l^2 \cdot n^{2\log l} \cdot \varphi_{d-1}^{d-2}(n,l)^{3\log l}$ *for* $d > 1$.

Proof. We prove the inequality of Lemma 3.6 for an arbitrary complete path set $\Pi_{st}(l)$. We first introduce path sets

$$\Pi_{sut} := \{\pi \in \Pi_{st}(l) \mid u \in V \text{ divides } \pi \text{ into subpaths of at most } \left\lceil \frac{l}{2} \right\rceil \text{ edges}\}.$$

Clearly, for each path $\pi \in \Pi_{st}(l)$ we find a node u with $\pi \in \Pi_{sut}$. Thus, we can apply Lemma 3.5 and get

$$\varphi_d^{d-2}(\Pi_{st}(l)) \leq 2 \cdot n \cdot \varphi_{d-1}^{d-2}(n,l)^2 \cdot \sum_{u\in V} \varphi_d^{d-2}(\Pi_{sut}), \tag{3.9}$$

as one can easily show that $\varphi_{d-1}^{d-2}(n,l)^2 \geq \max_{u\in V} \varphi_{d-1}^{d-2}(\Pi_{sut})$.

The path sets Π_{sut} can be written as the pairwise concatenation of the path sets $\Pi_{su} := \Pi_{su}\left(\frac{l}{2}\right)$ and $\Pi_{ut} := \Pi_{ut}\left(\frac{l}{2}\right)$. Thus, they satisfy the requirements to apply Lemma 3.4 such that for each node u we get

$$\begin{aligned} \varphi_d^{d-2}(\Pi_{sut}) &\leq \varphi_{d-1}^{d-2}(\Pi_{ut}) \cdot \varphi_d^{d-2}(\Pi_{su}) + \varphi_{d-1}^{d-2}(\Pi_{su}) \cdot \varphi_d^{d-2}(\Pi_{ut}) \\ &\leq 2 \cdot \varphi_{d-1}^{d-2}\left(n, \frac{l}{2}\right) \cdot \varphi_d^{d-2}\left(n, \frac{l}{2}\right). \end{aligned} \tag{3.10}$$

If we combine (3.9) and (3.10), we get

$$\begin{aligned} \varphi_d^{d-2}(\Pi_{st}(l)) &\leq 4 \cdot n^2 \cdot \varphi_{d-1}^{d-2}(n,l)^2 \cdot \varphi_{d-1}^{d-2}\left(n, \frac{l}{2}\right) \cdot \varphi_d^{d-2}\left(n, \frac{l}{2}\right) \\ &\leq 4 \cdot n^2 \cdot \varphi_{d-1}^{d-2}(n,l)^3 \cdot \varphi_d^{d-2}\left(n, \frac{l}{2}\right). \end{aligned} \tag{3.11}$$

Note that, by Definition 3.12, we have $\varphi_{d-1}^{d-2}\left(n, \frac{l}{2}\right) \leq \varphi_{d-1}^{d-2}(n,l)$. Using this recursion in l we obtain with $\varphi_d^{d-2}(n, 1) = d$

$$\begin{aligned}\varphi_d^{d-2}(\Pi_{st}(l)) &\leq d \cdot 4^{\log l} \cdot n^{2\log l} \cdot \varphi_{d-1}^{d-2}(n, l)^{3\log l} \\ &= d \cdot l^2 \cdot n^{2\log l} \cdot \varphi_{d-1}^{d-2}(n, l)^{3\log l} .\end{aligned} \tag{3.12}$$

□

Proof of Theorem 3.1:

Proof. We need to show that $\varphi_d^{d-1}(n, n) \in n^{O\left(\log^{d-1} n\right)}$ for a fixed but arbitrary number of cost types $d \geq 1$. From Lemma 3.6 we know that $\varphi_d^{d-2}(n, n) \leq d \cdot n^{2+2\log n} \cdot \varphi_{d-1}^{d-2}(n, n)^{3\log n}$. With (3.7) we get $\varphi_d^{d-1}(n, n) \leq 2 \cdot n^{2+2\log n} \cdot \varphi_{d-1}^{d-2}(n, n)^{3\log n}$. With $\varphi_1^0(n, n) = 1$ this leads to $\varphi_d^{d-1}(n, n) \in n^{O\left((3\log n)^{d-1}\right)}$, which is, for a fixed d, equal to $\varphi_d^{d-1}(n, n) \in n^{O\left(\log^{d-1} n\right)}$. □

3.2.4 Another Upper Bound on the Number of Extreme Shortest Paths

In this section we show that a generalization of Lemma 3.4 suffices to prove that $\varphi_d^{d-1}(n, n)$ is subexponential in n.

Lemma 3.7
Let $\Pi_1, \Pi_2, \ldots, \Pi_k$ be $k > 1$ path sets such that for each $1 \leq i < k$ all paths in Π_i end at some node $u \in V$ and all paths in Π_{i+1} start at the same node u. Furthermore, let $\Pi' := \{\pi_1\pi_2\ldots\pi_k \mid \pi_i \in \Pi_i \ \forall i \in \{1, 2, \ldots, k\}\}$ be the set of concatenations of the paths in Π_1 to Π_k. Then it holds

$$\varphi_d^{d-2}\left(\Pi'\right) \leq (k-1) \cdot \varphi' \cdot \sum_{1\leq i\leq k} \varphi_d^{d-2}(\Pi_i)$$

with $\varphi' := \max_{1\leq i\leq k} \varphi_{d-1}^{d-2}(\Pi_i)$.

Proof. Let us fix an arbitrary cost function $c \in \mathcal{C}_d$ and let $P' := P(\Pi', c)$ and $P_i := P(\Pi_i, c)$ for all $1 \leq i \leq k$. For any preference $\alpha \in \mathcal{P}_d$ we find an α-shortest path v_i in P_i for every $1 \leq i \leq k$. For the same α let v' be the α-shortest path in P'. Every path in Π' is the concatenation of exactly one path per path set Π_i. Thus, $v' = v_1' + v_2' + \cdots + v_k'$ for some paths $v_i' \in P_i$ for $1 \leq i \leq k$. Therefore, we have

$$\alpha v'^T = \alpha\left(v_1' + v_2' + \cdots + v_k'\right)^T \geq \alpha\left(v_1 + v_2 + \cdots + v_k\right)^T .$$

However, the sum $v_1 + v_2 + \cdots + v_k$ is, by definition of Π', an element of P'. Thus, $v' = v_1 + v_2 + \cdots + v_k$. This means that for every $\alpha \in \mathcal{P}_d$ we obtain the α-shortest path in P' by concatenating the α-shortest paths in P_i for $1 \leq i \leq k$. Therefore, the PSS $\mathcal{S}_{P'}$ is the overlay of $\mathcal{S}_{P_1}, \mathcal{S}_{P_2}, \ldots, \mathcal{S}_{P_k}$.

We know from Lemma 3.2 that any facet $f_{P_i}^{d-2}$ can be split into at most $\varphi_{d-1}^{d-2}\left(\Pi_j\right) \leq \varphi'$ $(d-2)$-dimensional facets by intersecting it with $\mathcal{S}_{P_j}$ for any $j \neq i$ and $1 \leq j \leq k$. Thus, the facet $f_{P_i}^{d-2}$ can contribute to at most $(k-1) \cdot \varphi'$ $(d-2)$-dimensional facets in $\mathcal{S}_{P'}$. As the union of the PSS $\mathcal{S}_{P_i}$ have at most $\sum_{1\leq i\leq k} \varphi_d^{d-2}(\Pi_i)$ $(d-2)$-dimensional facets, the statement follows. □

With Lemma 3.7 we can prove the following lemma.

Lemma 3.8
For every $d \geq 1$ we have $\varphi_d^{d-1}(n, n) \in O\left(n^{4\sqrt{n}+1}\right)$.

Proof. We fix two arbitrary nodes $s, t \in V$ and a hop length limit $l \leq n$. In the following we show that $\varphi_d^{d-1}(\Pi_{st}(l)) \in O\left(n^{4\sqrt{n}+1}\right)$. Let $K := v_1v_2\ldots v_{\lfloor\sqrt{l}\rfloor}$ be any sequence of $\lfloor\sqrt{n}\rfloor$ nodes and let $\Pi_K \subseteq \Pi_{st}(l)$

be the subset of paths $\pi \in \Pi_{st}(l)$ that visit all nodes in K in the same order as in K and that have a hop length of at most $\lceil \sqrt{l} \rceil$ from one node in K to the next node in K. Clearly, for every path $\pi \in \Pi_{st}(l)$ there is a node sequence K such that $\pi \in \Pi_K$.

By the construction of Π_K we can use Lemma 3.7 to get

$$\begin{aligned}\varphi_d^{d-2}(\Pi_K) &\leq \sqrt{l} \cdot \varphi_{d-1}^{d-2}\left(n, \lceil \sqrt{l} \rceil\right) \cdot \sqrt{l} \cdot \varphi_d^{d-2}\left(n, \lceil \sqrt{l} \rceil\right) \\ &\leq n \cdot \varphi_{d-1}^{d-2}\left(n, \lceil \sqrt{n} \rceil\right) \cdot \varphi_d^{d-2}\left(n, \lceil \sqrt{n} \rceil\right).\end{aligned}$$

Clearly, between any two nodes there can be at most n^{l-1} paths of hop length l. Thus, we have

$$\varphi_d^{d-2}(\Pi_K) \leq n \cdot n^{\sqrt{n}} \cdot \varphi_d^{d-2}\left(n, \lceil \sqrt{n} \rceil\right).$$

Furthermore, for any path set Π and cost function $c \in \mathcal{C}_d$ there is at most one $(d-2)$-dimensional facet per path pair in the PSS $\mathcal{S}_{P(\Pi,c)}$, which can be upper bounded by $|\Pi|^2$. We can trivially upper bound $\varphi_d^{d-1}(n, l)$ by n^{l-1}. It therefore holds $\varphi_d^{d-2}\left(n, \lceil \sqrt{n} \rceil\right) \leq \varphi_d^{d-1}\left(n, \lceil \sqrt{n} \rceil\right)^2 \leq n^{2\sqrt{n}}$ and we obtain

$$\varphi_d^{d-2}(\Pi_K) \leq n^{3\sqrt{n}+1}.$$

Finally, there are $n^{\lfloor \sqrt{n} \rfloor}$ possible sequences K and $\varphi_d^{d-1}(\Pi_K) \leq \varphi_d^{d-2}(\Pi_K)$. Thus,

$$\begin{aligned}\varphi_d^{d-1}(\Pi_{st}) &\leq n^{\lfloor \sqrt{n} \rfloor} \cdot \varphi_d^{d-1}(\Pi_K) \\ &\leq n^{\lfloor \sqrt{n} \rfloor} \cdot n^{3\sqrt{n}+1} \\ &\leq n^{4\sqrt{n}+1}.\end{aligned}$$ □

What may be surprising about Lemma 3.8 is that its upper bound does not depend on the number of dimensions d. This is not a contradiction to Theorem 3.1, though, as the variable d has to be a fixed number. For any fixed number d and any function f it holds $f \in n^{O\left(\log^{d-1} n\right)} \Rightarrow f \in O\left(n^{4\sqrt{n}+1}\right)$. To see this, consider the functions $f_1(n) := \log^{d-1} n$ and $f_2(n) := \sqrt{n}$, which correspond to the exponents of the upper bounds. We have

$$\begin{aligned}f_1\left(n^2\right) &= 2^{d-1} f_1(n), \\ f_2\left(n^2\right) &= f_2(n)^2.\end{aligned}$$

Thus, if we take the square of n, f_1 is multiplied by a constant while f_2 is squared. Therefore, the exponent of the upper bound in Lemma 3.8 grows faster than the one in the upper bound of Theorem 3.6 irrespective of d.

3.2.5 A Note on the Complexity of Preference Polyhedra

Preference polyhedra $f_P(v)$ of a point set $P \subset \mathbb{R}^d$, as defined in Definition 3.6, are convex subsets of $\mathcal{P}_d$ for which the point v is α-shortest with respect to P. They can, for instance, be utilized to examine mobility patterns as shown in Section 3.4.

In this section we briefly discuss what we know about the number of hylperplanes that support the preference polyhedra of extreme shortest paths of complete path sets. We refer to this number as the preference polyhedron complexity. A hyperplane H supports a preference polyhedron $f_P(v)$ iff

$\dim(H \cap f_P(v)) = d-2$ and there is a point $v' \in \mathbb{R}^d$ such that for every $\alpha \in f_P(v)$ it holds $\alpha v'^T \leq 0$ and $H \cap \mathcal{P}_d = \{\alpha \in \mathcal{P}_d \mid \alpha v'^T = 0\}$. For instance, in Figure 3.1 the cells of the shown PSS correspond to preference polyhedra of extreme shortest paths. The complexity of these cells is given by the number of their boundary edges.

Definition 3.13
Given a point set $P \subset \mathbb{R}^d$ for some $d > 1$, let $h_P(v)$ for some point $v \in P$ be the number of hyperplanes supporting the preference polyhedron $f_P(v)$. We call $h_P(v)$ the complexity of preference polyhedron $f_P(v)$ with respect to the set P. Furthermore, we define $h_d(n)$ to be the maximum of $h_P(v)$ over all point sets $P \subset \mathbb{R}^d$ induced by complete path sets Π of graphs with n nodes.

For $d = 2$ the preference space is one-dimensional and the preference polyhedra of extreme shortest paths are intervals. Thus, $h_P(v) \leq 2$ for any point set $P \subset \mathbb{R}^2$ and point $v \in P$. We cannot find such a bound for $d = 3$ if P is the set of cost vectors of a complete path set.

Lemma 3.9
For $d = 3$ and any $k \in \mathbb{N}$ one can find a complete path set Π, a path $\pi \in \Pi$ and a cost function $c \in \mathcal{C}_3$ such that $h_{P(\Pi,c)}(c(\pi)) \geq k$.

Proof. We proceed in two steps. Let Π be a complete set of paths of equal hop length. In the first step we show that if $\varphi_2^1(P(\Pi, c)) = k$ for some cost function $c \in \mathcal{C}_2$ we can construct a preference polyhedron with more than k supporting hyperplanes by adding a third metric. In the second step we show that for any number k we find such complete path sets Π with $\varphi_3^2(\Pi) \geq k$.

Let us first assume we have a complete path set Π as described above and a cost function $c \in \mathcal{C}_2$ with k extreme shortest paths $M := v_1, v_2, \ldots, v_k$ in $P := P(\Pi, c)$. We refer to the j-th entry of cost vector v_i with $v_{i,j}$. Let $x := \max_{1 \leq i \leq k} v_{i,1} + v_{i,2}$ 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. We find appropriate edge costs as all paths in Π have the same number of edges. It is clear that each of the k extreme shortest paths is optimal for the preference $\alpha_3 = [0, 0, 1]$. Hence, each preference polyhedron corresponding to a $v_i \in M$ is now a triangle as shown in Figure 3.6a.

Finally, we create a new optimal path π_{k+1} with the cost vector $v_{k+1} := [2x, 2x, 2x]$. This path is clearly optimal for the preference α_3. Since $\alpha_3(v_{k+1} - v_i)^T$ 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 boundary with each of the k other optimal paths as shown in Figure 3.6b.

We now discuss that with two metrics there can be arbitrarily many extreme shortest paths in a complete path set.

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

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

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 $\alpha \in \left[\frac{2k-1}{2k}, 1\right]$. □

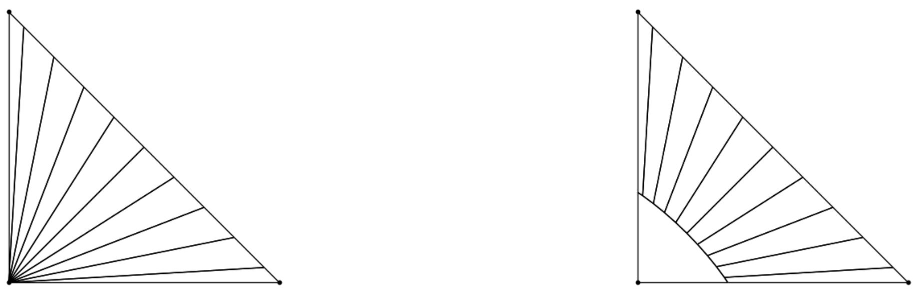

a) Before inserting new optimal path **b)** After inserting new optimal path

Figure 3.6: Example of preference polyhedra of optimal paths with the same source and target and with equal cost in the third metric

We know from [Car83a] that $\varphi_2^1 \in n^{\Omega(\log n)}$. Thus, it could be possible to prove $h_3 \in n^{\Omega(\log n)}$ using a similar idea as in the proof of Lemma 3.9. The difficulty here is to find appropriate edge costs for the third metric to achieve the required properties. In the proof above this was trivial as all paths had the same number of edges.

While the complexity of a single preference polyhedron for $d = 3$ cannot be bounded by a constant, we obtain a different result for the average preference polyhedron complexity.

Lemma 3.10
For any pair of path set Π and cost function $c \in \mathcal{C}_3$ it holds

$$\frac{1}{\varphi_3^2(P(\Pi, c))} \sum_{v \in \mathcal{M}_{P(\Pi,c)}} h_{P(\Pi,c)}(v) < 6.$$

Proof. We fix an arbitrary path set Π and a cost function $c \in \mathcal{C}_3$ and define $P := (\Pi, c)$. The PSS $\mathcal{S}_P$ can be considered as an embedding of a planar graph with the 0-dimensional facets being the nodes, the 1-dimensional facets the edges and the 2-dimensional facets the faces of the graph (see Figure 3.1 for an example). It is well known that the average degree of nodes in planar graphs is less than 6. We obtain the dual graph of $\mathcal{S}_P$ by creating one node per facet of $\mathcal{S}_P$ and by connecting these nodes with an edge iff the corresponding faces in $\mathcal{S}_P$ share an edge. One can show that this graph is planar as well. Furthermore, the node degrees in this graph correspond to the complexity of the preference polyhedra in $\mathcal{S}_P$, which completes the proof. □

Clearly, any upper bound for the number of extreme shortest paths $\varphi_d^{d-1}(\Pi)$ ($+d$ for the boundaries of the preference space) is also an upper bound for the preference polyhedron complexity. See Theorem 3.1 and Lemma 3.8 for results in this regard. However, to the best of our knowledge, there is little known about the fraction $\frac{\sum_{v \in \mathcal{M}_P} h_P(v)}{\varphi_d^{d-1}(P)}$ for general d.

3.2.6 Conclusion

In Section 3.2 we showed that the number of extreme shortest paths in a graph G is upper bounded by $n^{O\left(\log^{d-1} n\right)}$, where n is the number of nodes and d is the fixed but arbitrary number of edge costs in G. This is a generalization of previous results in the context of parametric shortest paths for two and three dimensions.

In Section 3.2.4 we showed a second, weaker upper bound on the number of extreme shortest paths using a simpler proof.

We also discussed the bounds known for the complexity of preference polyhedra (Section 3.2.5). For the cases $d = 2$ and $d = 3$ it is possible to upper bound the (average) preference polyhedron complexity by a constant. Little is known, however, for $d > 3$. We conjecture that, for a fixed d, the average preference polyhedron complexity can be upper bounded by a polynomial in n.

Another open question is whether one can generalize the matching two-dimensional lower bounds on the number of extreme shortest paths shown in [Car83a].

3.3 Robust Preference Elicitation

Driving preferences can help to improve routing services and to understand mobility patterns. However, finding an appropriate preference for a given driver is nontrivial. In this section we propose a simple and robust method to compute the preference of a driver, when trajectories of that driver are available. Thus, our approach is data-driven and does not require any explicit interaction with the driver. The work presented in this section is based on the author's contribution to [BFJP20], which is a joint publication with Florian Barth, Stefan Funke and Tobias Skovgaard Jepsen.

3.3.1 Existing Approaches

A popular approach of modeling driving preferences is to consider them as random variables. For instance, preference mining methods based on Gaussian Mixture Models are presented in [YGMJ15] and in [DYGD15]. [BJS13] present a probabilistic method that compares the input trajectory with pareto-optimal trajectory sets. [CRLB16] show how mining driving preferences can be accomplished using Bayesian learning strategies.

Our approach is non-probabilistic and defines the driving preference as the solution of an optimization problem. [DGG+15] present a similar technique, where the driving preference is chosen such that the overlap of the given and the computed trajectory is maximal. The main difference to our approach is that Delling et al. focus on geographical similarities of trajectories while we define similarity in terms of their traversal costs.

The work of [FLS16] is the most closely related work. They present a method which decides whether there exists a preference $\alpha \in \mathcal{P}_d$ such that a given trajectory is α-shortest for the traversal costs, and outputs the preference. Their method can recover the driving preferences of synthetic trajectories where such a preference is guaranteed to exist. However, this is not guaranteed for real world trajectories for various reasons such as inaccurate traversal costs or suboptimal driving behavior. We extend their method to obtain robust preferences in case a trajectory is not optimal for any preference.

3.3.2 Cost-Optimal Preferences

In this section we define the notion of *cost-optimal preferences* and show how we can compute them efficiently. Cost-optimal preferences can be used to elicit driving preferences using a single or multiple trajectories. For each set of trajectories T one can find at least one cost-optimal preference, which makes it a robust approach for preference elicitation. Think of the notion of α-shortest paths as a comparison. We can try to compute a preference α such that each trajectory in our set T is an α-shortest path. Clearly, this is another way of preference elicitation. However, what happens if - and this is the standard case when dealing with real-world trajectories - there is no such preference? Cost-optimal preferences can be considered as an extension of the notion of α-shortest paths such that we never fall into this case.

Definition 3.14
Given a set of trajectories Π in a graph $G(V, E)$ with a d-metric cost function. A preference $\alpha \in \mathcal{P}_d$ is a cost-optimal *preference of T if and only if*

$$\alpha = \operatorname*{argmin}_{\alpha' \in \mathcal{P}_d} \sum_{p \in T} c\left(\alpha', p\right) - c\left(\alpha', \pi_{\alpha',p}\right),$$

where $\pi_{\alpha',p}$ is the α'-shortest path with the same source and target as p.

Note that $\sum_{p \in T} c(\alpha', p) - c\left(\alpha', \pi_{\alpha',p}\right)$ is non-negative for any preference α' and trajectory p and zero if and only if all trajectories in T are α'-shortest paths. Thus, a preference α is always cost-optimal for a set of α-shortest trajectories.

Let Π_p be the set of all simple paths with the same source and target as trajectory p. One can rewrite Definition 3.14 as a linear program as shown in the following. We call it the *cost-optimality* LP.

$$\begin{aligned} &\textsc{Minimize} && \sum_{p \in T} \delta_p \\ &\textsc{subject to} && \forall p \in T\ \forall p' \in \Pi_p : c(\alpha, p) - c\left(\alpha, p'\right) \leq \delta_p \\ & && \sum_{i=1}^{d} \alpha_i = 1 \\ & && \forall i \in \{1, \ldots, d\} : \alpha_i \geq 0 \end{aligned} \tag{3.13}$$

The cost-optimality LP 3.13 differs from the shortest path LP 3.6 in the following aspects. First, there are new variables δ_p that replace the zero on the right hand side of the optimlaity constraints. Second, LP 3.13 has an objective function, which minimizes the sum of δ_p. Third, LP 3.13 is defined for a set of trajectories whereas LP 3.6 only considers a single trajectory. However, note that LP 3.6 could easily be extended to a set of trajectories.

The intuition behind cost-optimal preferences is that we take the preference α for which all trajectories in T are α-shortest paths if such a preference exists. If not, we take the preference such that the difference of the aggregated cost to the actual α-shortest paths is minimized on average. Thus, the relevance and meaning of cost-optimal preferences are directly derived from the preference model, which is a clear advantage compared to black-box machine learning approaches. The cost-optimality LP can be solved in polynomial time using the ellipsoid method. The argument is the same as for the shortest path LP: For given α and δ_p we can use Dijkstra's algorithm to check whether

$c(\alpha, p) - c\left(\alpha, \pi_{\alpha,p}\right) \leq \delta_p$ is satisfied for every trajectory $p \in T$. If this is the case, every constraint in the LP is satisfied as the path $\pi_{\alpha,p}$ maximizes the difference $c(\alpha, p) - c\left(\alpha, \pi \in \Pi_p\right) \leq \delta_p$ among all paths in Π_p by definition.

Instead of calculating the best preference on average one could also be interested in a worst-case optimization.

Definition 3.15
Given a set of trajectories T in a d-metric graph $G(V, E)$. A preference $\alpha \in \mathcal{P}_d$ is a worst-case cost-optimal *preference of T if and only if*

$$\alpha = \underset{\alpha' \in \mathcal{P}_d}{\operatorname{argmin}} \max_{p \in T} c\left(\alpha', p\right) - c\left(\alpha', \pi_{\alpha',p}\right),$$

where $\pi_{\alpha',p}$ is the α'-shortest path with the same source and target as p.

Note that the only difference to Definition 3.14 is that the sum is replaced by a maximization. The corresponding LP looks as follows, which we call the *worst-case cost-optimality* LP.

$$\begin{aligned} &\textsc{Minimize} && \delta \\ &\textsc{subject to} && \forall p \in T\ \forall p' \in \Pi_p : c(\alpha, p) - c\left(\alpha, p'\right) \leq \delta \\ &&& \sum_{i=1}^{d} \alpha_i = 1 \\ &&& \forall i \in \{1, \ldots, d\} : \alpha_i \geq 0 \end{aligned} \tag{3.14}$$

Again, the LP is solvable in polynomial time using the same arguments as above.

LP 3.13 and LP 3.14 can be used to elicit driving preferences. A typical application would be to take a set of recorded trajectories T of a single driver and to compute the cost-optimal preference α with respect to T. The preference α could then be used to suggest the driver new routes by computing α-shortest paths.

3.3.3 Experiments

We now evaluate our preference elicitation approach using real-world car trajectories, which are described in Section 3.3.3.1. Specifically, we evaluate our approach for each trajectory $p(s, t)$ in our trajectory set as follows. First, we solve LP 3.14 for p. Then, we use the resulting preference vector α to compute the α-shortest path $\pi(s, t)$ using an implementation of MCH as described in Section 3.1.4. Ideally, the preference vector α combined with the source s and target t is sufficient to reconstruct or recover the trajectory p. We therefore refer to π as the *recovered route* of trajectory p.

In our evaluation we measure the quality of the recovered routes with respect to the overlap with the trajectories and with respect to the cost similarity.

The entire code for our experiments is written in the Rust programming language[1] and is publicly available, together with the graph file and synthetic trajectories[2].

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

3.3.3.1 Data Set

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 OpenStreetMap 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 segment. 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 segments. In addition, each road segment 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 OpenStreetMap data and speed limits provided by Aalborg Municipality and Copenhagen Municipality [JJNT18].

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 segments $T = (e_1, \ldots, e_n)$ 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.

3.3.3.2 Routing Cost Types

From the data sets described in 3.3.3.1, 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   The vehicle trajectories in our trajectory set have the tendency to be concentrated on a few popular segments. As such, many road segments have few or no traversals in the trajectory set. We therefore require a means of estimating travel times for such road segments. To this end, we use a pre-trained machine learning model to provide travel time estimates. However, for road segments 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. the driving speed estimate of a road segment becomes increasingly less influential the more historical traversals the road segment is associated with.

We compute the travel time t_e for a road segment 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 segment $e \in E$. 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 segment cannot be below 5 kmh.

Congestion We assign a congestion level to road segment 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 segment e if a vehicle is driving at exactly the speed limit. Formally, we assign road segment e the congestion level $c_e = \max\{1 - \frac{\tau_e}{t_e}, 0\}$ 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 segment is not traversable.

The value of τ_e relies on the speed limit of road segment e. We use a speed limit data set that combines OpenStreetMap 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 segments without a known speed limit. In such cases, we use an OpenStreetMap 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 segment as in a city if either the source or destination intersection is in a city according to its attributes.

Crowdedness This routing cost type describes how 'crowded' the landscape around a road segment is. It is derived from the number of OpenStreetMap nodes in the vicinity of the road segment. We use all OpenStreetMap nodes in Denmark from a 2019 data set regardless wether 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 OpenStreetMap nodes within each cell. For each road segment, we locate the OpenStreetMap nodes that are part of its geometry in the grid. The cost per road segment 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 209m x 177m.

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

3.3.3.3 Evaluation Functions

To measure our approach's ability to model driving behavior, we use the Relative Recovered Route Overlap (RRRO):

$$\mathrm{RRRO}(p, \pi) := \frac{|p \cap \pi|}{|p|}$$

Let α be a preference vector recovered from trajectory p. Here, π is the recovered route of trajectory p. If the preference vector α used to construct π fully captures the driving preferences exhibited in p, then the route π recovered using α should be identical to p, resulting in a relative recovered route overlap of 1.

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

Table 3.1: Mean RRRO and RCRS of RDP, TTP, and BRP of different algorithms for personalised routing on the unstitched trajectory set.

	RDP	*TTP*	*BRP*
RRRO	**0.74**	0.70	0.66
RCRS	**0.87**	0.85	0.81

We also measure whether p and π have similar aggregated costs using the Relative Cost Recovery Score (RCRS), which is defined as follows.

$$\mathrm{RCRS}(p, \pi) := \frac{c(\alpha, \pi_{\alpha,p})}{c(\alpha, p)}$$

Note, that $c(\alpha, p) \geq c(\alpha, \pi_{\alpha,p})$ since $\pi_{\alpha,p}$ is the α-shortest path with the same endpoints as p. Thus, the RCRS is always between 0 and 1. An RCRS value of, e.g., 0.8, indicates that the preference vector α accounts for 80% of the personalized cost of trajectory p. If the preference vector α fully captures the driver's preferences, then the RCRS is 1.

3.3.3.4 Baselines

We refer to our approach as *Recovered Driving Preference* (RDP) and compare its performance with two baselines. The first baseline, *Travel Time Preference* (TTP), always returns a preference vector that has weight one for travel time. The second baseline, *Best Random Preference* (BRP), generates five random preference vectors for a trajectory, evaluates them and returns the preference with the best result. The BRP baseline is run independently for the two evaluation functions used in our experiments.

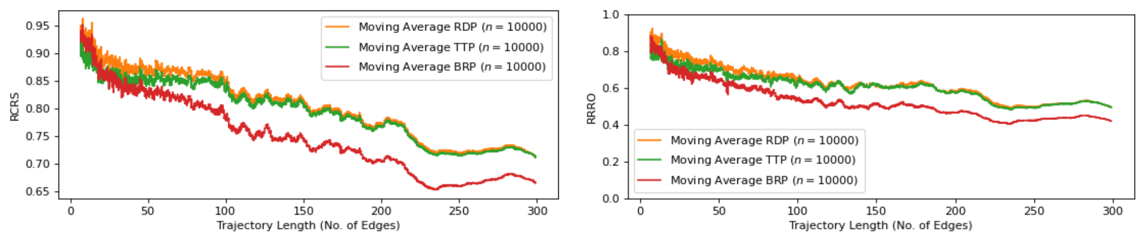
3.3.3.5 Results

We summarize the results of our experiments in Table 3.1. We computed the results of RDP with a single core with a clock speed of 2.3 GHz. The average processing time is 1.04 milliseconds per trajectory. As shown in the table, RDP achieves both the highest mean RRRO and mean RCRS that are, respectively, 5.7% and 2.4% better than the best performing baseline TTP. We expect that these figures will be even higher if more than four traversal cost types are used.

The RDP shows superior performance compared to both baseline algorithms, but the performance of both RDP and the baselines deteriorate as trajectory length increases as shown in Figure 3.7. This is not particularly surprising since longer trajectories are more likely to contain more via-points, and none of the approaches in this experiment take such information into account. In addition, TTP approaches the performance of RDP as trajectory length increases. This suggests that people are likely to prioritize travel time more on long trips which matches both our expectations and anecdotal experiences.

3.3.3.6 Discussion

Our experiments show that the RDP preference vectors explain driver behavior better than the TTP and BRP baselines. Although the average results of TTP are very similar to those of RDP, they are never better. This is a strong indication that our approach indeed finds the best preferences to describe



a) RCRS **b)** RRRO

Figure 3.7: The (a) RCRS and (b) RRRO scores of RDP, the TTP and the BRP.

the drivers' behavior. In addition, the fast processing time (and trivial parallelizability) of RDP makes it scalable to even very large trajectory data sets.

Notably, the performance gap between RDP and TTP nearly disappears for long trajectories. This matches our expectation that travel time is the most important criterion for such trajectories.

3.3.4 Conclusion

In this section we discussed the problem of eliciting driving preferences. We defined two slightly different notions of cost-optimal preferences for sets of trajectories that can be computed in polynomial time using their linear program formulations. These formulations allow robust preference elicitation as, first, the relevance of cost-optimal preferences is directly derived from the preference model and, second, there is a cost-optimal preference for every set of trajectories (even if the trajectories are no shortest paths). In our experiments we showed that cost-optimal preferences can be computed very efficiently in practice.

3.4 Preference-Based Trajectory Clustering

This section is based on [BFP21], a joint work with Florian Barth and Stefan Funke. In the entire section we assume that a graph $G(V, E)$ and a d-dimensional cost function c are given. To improve the readability we define

$$f_\pi := f_{P(\Pi_\pi, c)}(c(\pi)),$$

where π is a path in G, Π_π is the complete path set with the same endpoints as π and $f_{P(\Pi_\pi,c)}(c(\pi))$ is the preference polyhedron of π as described in Definition 3.6. Thus, f_π is the preference polyhedron of path π in graph G with cost function c. We call f_π the preference polyhedron of path π.

We consider the following problem: Given a set of shortest paths (or trajectories) T in graph G, find a discrete set of preferences $A \subset \mathcal{P}_d$ of minimum cardinality such that for every path $\pi \in T$ it holds $f_\pi \cap A \neq \emptyset$. We refer to this problem in the following as *Preference-Based Trajectory Clustering* (PTC).

PTC is motivated by the need to understand mobility patterns. Imagine we were some kind of urban planners and had the recordings of many bicyclists' trajectories in our city available. It would be crucial for us to understand the motivation (or preferences) of the bicyclists to take these routes in order to improve the infrastructure for them. Surely, not every bicyclist has the same routing preferences. Some could, for instance, strictly prefer the shortest route by distance. Others may choose to take a detour in order to avoid car traffic. But is it perhaps possible to divide the routes into few clusters such that each cluster can be "explained" by a single preference? This question can be answered by solving PTC.

In [FLS16], a sweep algorithm is introduced that computes an approximate solution of PTC. It is, however, relatively easy to come up with examples where the result of this sweep algorithm is by a factor of $\Omega(|T|)$ worse than the optimal solution. We aim to improve this result by finding practical ways to solve PTC optimally as well as approximately with better quality guarantees. Our (surprisingly efficient) strategy is to explicitly compute for each trajectory π in T the preference polyhedron and to translate PTC into a geometric hitting set problem.

Fortunately, the shortest path LP 3.6 already provides a way to compute these polyhedra. The constraints in this 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.

3.4.1 Driving Preferences and Geometric Hitting Sets

In this section we discuss how to construct preference polyhedra from given paths and reformulate PTC as a minimum geometric hitting set problem.

3.4.1.1 Exact Polyhedron Construction

A straightforward, yet quite inefficient way of constructing the preference polyhedron for a given 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.

Boundary Exploration

With the tool of linear programming at hand, one possible way of exploring the preference polyhedron is by repeated invocation of the shortest path LP but with varying objective functions. For the sake of simplicity let us assume that the preference polyhedron is full (that is, $d-1$) dimensional. We first determine d 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 k facets, this approach clearly requires $O(k)$ 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). The best known upper bound for k and an arbitrary but fixed d is $n^{O\left(\log^{d-1} n\right)}$ as discussed in Section 3.2.5. However, we conjecture that one can find better upper bounds for the average number k. While theoretically appealing, in practice this approach is not very competitive due to the overhead of repeated LP solving.

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 feasibility polyhedron via a sequence of half-space intersections. Let f_π be the preference polyhedron of path π. In the i-th iteration CCA computes a polyhedron f_i with $f_\pi \subseteq f_i \subseteq f_{i-1}$. The polyhedron f_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 f_{i-1} that has not been checked before. If no such corner exists, CCA terminates and returns $f_{CCA} := f_{i-1}$. Otherwise, it marks the corner $\alpha^{(i)}$ as checked and computes the $\alpha^{(i)}$-shortest path $\pi'_{\alpha^{(i)},\pi}$ with the same endpoints as π. If $c\left(\alpha^{(i)}, \pi\right) - c\left(\alpha^{(i)}, \pi'_{\alpha^{(i)},\pi}\right) = 0$, the corner $\alpha^{(i)}$ belongs to f_π and there is nothing to do. Otherwise, the constraint $c\left(\alpha^{(i)}, \pi\right) - c\left(\alpha^{(i)}, \pi'_{\alpha^{(i)},\pi}\right) \leq 0$ is violated by a part of f_{i-1}. We call this part f'_i. It is clear that $\alpha^{(i)} \in f'_i$. Finally, the polyhedron $f_i := f_{i-1} \backslash f'_i$ is computed by intersecting f_{i-1} with the half-space $c\left(\alpha^{(i)}, \pi\right) - c\left(\alpha^{(i)}, \pi'_{\alpha^{(i)},\pi}\right) \leq 0$. This intersection may introduce new corners, which are marked as unchecked. Afterwards, the next iteration starts.

We prove that CCA indeed computes f_π. Let f_{CCA} be the output of CCA. We first show that $f_\pi \subseteq f_{CCA}$. $f_\pi \subseteq f_0$ is trivially true. Furthermore, in each iteration i it is clear that $f'_i \cap f_\pi = \emptyset$. Hence, for each iteration i, we have $f_\pi \subseteq f_i$ and therefore $f_\pi \subseteq f_{CCA}$. The other direction $f_{CCA} \subseteq f_\pi$ follows from the fact that f_π is convex and that all corners of f_{CCA} belong to f_π as they are marked as checked.

The number of steps CCA has to conduct is upper bounded by the number of shortest paths between two nodes in G plus the number of corners of the preference polyhedron f_π. This can be shown as follows. When CCA checks a corner $\alpha^{(i)}$ two cases are possible. First, if $\alpha^{(i)} \in f_\pi$, then $\alpha^{(i)}$ is a corner of f_π. Second, if $\alpha^{(i)} \notin f_\pi$, then we find a shortest path $\pi'_{\alpha^{(i)},\pi}$ and subtract f'_i from f_{CCA}. In this way we ensure that there is no $\alpha \in f_{CCA}$ with $c(\alpha, \pi) - c\left(\alpha, \pi'_{\alpha^{(i)},\pi}\right) > 0$. Thus, in case two we will not find path $\pi'_{\alpha^{(i)},\pi}$ again.

Each corner of f_π is defined by the cost vectors of d extreme shortest paths. Furthermore, we may assume that each shortest path in G is an extreme shortest path, which could be enforced by introducing infinitesimal cost perturbations. Thus, for a fixed number of dimensions d, it follows from Theorem 3.1 that the number of steps of CCA is upper bounded by $n^{O\left(\log^{d-1} n\right)}$.

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

3.4.1.2 Minimum Geometric Hitting Set

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 3.8 shows an example of how preference polyhedra of different optimal paths could look like in case of three metrics. Note that the preference polyhedra may overlap because the paths in T do not necessarily have the same endpoints.

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 3.8 we have depicted two feasible hitting sets (white squares and black circles) for this instance. Both solutions

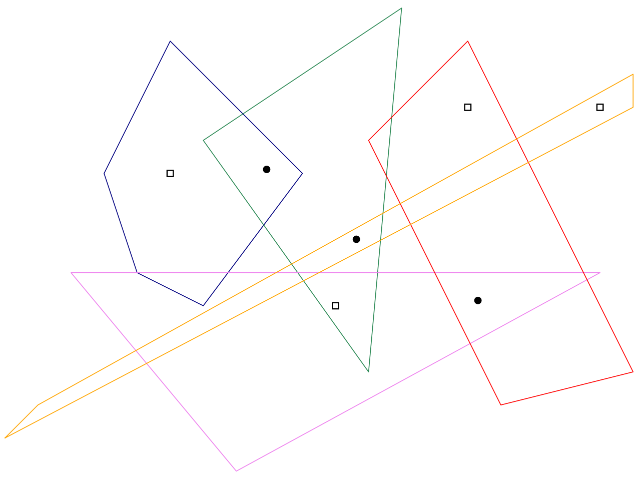


Figure 3.8: Example of a geometric hitting set problem as it may occur in the context of PTC. Two feasible hitting sets are shown (white squares and black circles).

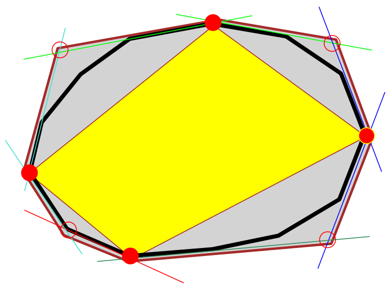


Figure 3.9: Inner (yellow) and outer approximation (grey) of the preference polyhedron (black).

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 more precisely vertices in the arrangement of preference polyhedra. We describe the computation of these candidates for the hitting sets in Section 3.4.1.3.

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 $\mathcal{U} := \{f_1, f_2, \ldots, f_k\}$ 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:

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

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.

3.4.1.3 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.

3.4.1.4 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.

Preference Polyhedron Complexity

It follows from Theorem 3.1 that the complexity of any preference polyhedron is in $n^{O\left(\log^{d-1} n\right)}$, where n is the number of nodes in the graph and d is the number of cost types. Furthermore, we know from [Car83a] that $\varphi_2^1(n,n) \in n^{\Omega(\log n)}$. We therefore expect that it is possible to show $h_3(n) \in n^{\Omega(\log n)}$ using similar ideas as in the proof of Lemma 3.2.5. However, we are not aware of any result in this direction.

For $d \leq 3$ the average complexity of preference polyhedra can be upper bounded by a constant (see Lemma 3.10). However, in Section 3.2.5 we show that for $d = 3$ the preference polyhedron complexity of a single path can be arbitrarily high. This result does not make any statement about the graph size, though.

MGHS 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.

3.4.2 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.

3.4.2.1 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 3.9. 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 3.9, 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.

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.

3.4.2.2 Approximate Instance Solving

In this section, we discuss approximation algorithms to replace the ILP shown in Section 3.4.1.2, 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 3.4.1.3.

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 3.4.1.3.

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).

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 3.4.1.2 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 f_i.

At each constraint C_i LPG checks if f_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 f_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.

3.4.3 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.

3.4.3.1 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 MCH as described in Section 3.1.4. For the Sections 3.4.3.3 to 3.4.3.5, 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 3.4.3.6, we show that the number of preferences used does not change the characteristics of our approaches.

3.4.3.2 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 3.4.1.2 and solve this ILP formulation and its LP relaxation with GLPK. Finally, we implemented the two greedy algorithms described in Section 3.4.2.2 which solve the hitting set instances in C++. Source code and data is available under `https://doi.org/10.17605/osf.io/4qkuv`.

3.4.3.3 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 3.2 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.

3.4.3.4 Geometric Hitting Set Solving

We now compare the two greedy approaches from Section 3.4.2.2 with the ILP formulation of Section 3.4.1.2. 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 3.4. 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

Table 3.2: 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

Table 3.3: 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

and outer approximations of the two instances. The ILP solver always reports the optimal solution but it 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 the run time drastically.

Table 3.4: 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

3.4.3.5 Varying Polyhedron Approximation

In Table 3.3 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 3.3 it yields a tight lower bound for eight directions while the inner approximation only achieves a tight upper bound with 128 directions.

3.4.3.6 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 3.5 and 3.6 hold the results for instances with 1,000 paths on both graphs. The size of the hitting set grows if the number of preferences is increased but perhaps not as strongly as expected. 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 3.4.3.4. Otherwise, there is no performance difference.

Table 3.5: Run times and optimal solution of instances generated with varying amount 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 3.6: Run times and optimal solution of instances generated with varying amount 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

3.4.4 Conclusion

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.

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.

3.5 Concluding Remarks

In this chapter we discussed problems related to preference-based routing. Specifically, we studied an intuitive linear preference model for routing that we defined analogously to [FS15a].

We started with tackling the problem of counting shortest paths between two nodes in the context of multidimensional edge costs (Section 3.2). Surprisingly, we were able to show that this number is in $n^{O\left(\log^{d-1} n\right)}$, where n is the number of nodes in the graph and d the number of edge costs. This is in contrast to the well known fact that the number of Pareto-optimal paths between two nodes can be exponential in the graph size. Furthermore, it is a generalization of results in the context of parametric shortest paths for $d = 2$ and $d = 3$ [GR19; Gus80].

There are several open questions in this regard. First, is it also possible to generalize the matching lower bound for $d = 2$ [Car83a; MS00]? Second, we conjecture that the average complexity of preference polyhedra is polynomial in the graph size (for a fixed d). Third, our approach to prove the results in Section 3.2 is closely related to the idea of Minkowski sums. It would be worthwhile studying if we can extend our results to other point sets apart from those induced by path sets.

In Section 3.3 and Section 3.4 we gave examples of practical applications in the context of preference-based routing. First, we showed how to identify the preference of a driver if we have access to past trajectories of the driver (Section 3.3). This preference could then, for instance, be used to compute new routes for the driver. Preference elicitation is relevant in practice but it is difficult to measure the quality of the approaches, especially if it is not possible to interact with the driver. Our approach, however, is directly motivated by the preference model itself and our resulting preferences are optimal in an intuitive, cost-oriented sense.

In Section 3.4 we showed how preference polyhedra can help to cluster a set of trajectories with respect to cost optimality. Such a clustering can help to interpret mobility patterns of a single or multiple drivers. We showed that it is practically feasible to solve our clustering problem optimally even for rather large input sizes. A disadvantage of our approach, however, is our assumption that each trajectory in the input is a shortest path. Even though one could enforce this condition by segmenting each input trajectory into shortest paths as shown in [BFJP20], it is unclear whether segmenting trajectories preserves their characteristics with respect to driving preferences. It would therefore be interesting to incorporate the ideas of Section 3.3 that allow to characterize non-optimal trajectories with respect to preferences.

Chapter 4 Discussion

4.1 Conclusion

In this work we discussed interesting problems in the domain of shortest path computation and presented new insights on the theoretical as well as on the practical level.

In the first part (Chapter 2) we focused on shortest path speed-up techniques that make use of the typical hierarchical structure of transportation networks. We labeled this kind of speed-up techniques as *hierarchical search*.

In Section 2.3 we presented a new algorithm, inspired by Contraction Hierarchies and Hierarchical Hub Labeling, that achieves considerably better tradeoffs between space consumption and query times than CH and HHL. Such an approach is especially attractive when dealing with extremely large networks. For instance, to the best of our knowledge, we are the first to present query times of below 20 microseconds for graphs with more than 1 billion edges.

Furthermore, we showed in Section 2.4 that it is possible to conduct hierarchical search without employing shortcuts. Our approach improves the query times of Dijkstra's bidirectional algorithm by roughly one order of magnitude with hardly any space overhead. The advantage of this approach is, beside the low space requirements, that the search is conducted in the original graph and not in an overlay graph as in typical hierarchical methods. In this way, for instance, it is easier to obtain the actual shortest path. However, compared to speed-up techniques with shortcuts, the query times of our approach are not competitive.

We concluded Chapter 2 with theoretical considerations regarding query times of specific CH (Section 2.5). Our results show that the number of edges traversed in a CH query can be almost quadratic in the number of visited nodes, even if the graphs show a favorable property called bounded growth [BFS21]. This insight is a contribution to the problem of finding lower bounds for the complexity of the CH query, an interesting research direction that has not received a lot of attention so far.

In Chapter 3 we addressed problems related to preference-based routing. As in Chapter 2 we presented theoretical as well as practical contributions. We used a simple and versatile linear preference model first introduced in [FS15a] as the basis of our considerations.

We started in Section 3.2 with an analysis of extreme shortest paths, a generalization of shortest paths in graphs with multidimensional edge costs. We showed that the number of extreme shortest paths between two nodes is in $n^{O\left(\log^{d-1} n\right)}$, where n is the number of nodes in the graph and d is the number of cost types. This is a rather surprising result as there can be much more Pareto-optimal paths. Our upper bound generalizes previous results in the context of parametric shortest paths for $d = 2$ [Gus80] and $d = 3$ [GR19].

We proceeded in Section 3.3 with an approach to obtain driving preferences in the case when past trajectories are given. In contrast to previous algorithms [FLS16] our approach is robust in the sense that it accepts any kind of input trajectories, not only optimal paths. Furthermore, our resulting preferences are the solution of an intuitive, cost-oriented optimization problem and, thus, are directly motivated by our preference model.

In Section 3.4 we showed how to efficiently cluster trajectories with respect to cost optimality. Such a clustering can, for instance, help to understand mobility patterns. Even though it is unclear whether our approach runs in polynomial time, we demonstrated that it is capable of computing optimal solutions even for rather large problem instances.

4.2 Future Work

There are multiple open questions and issues related to this work that are interesting subjects for future research. We go through them in the order of the sections.

We believe that the preprocessing phase of our algorithm presented in Section 2.3 is not as efficient as it could be. Especially the selection of group separators should be addressed with a more sophisticated solution. Moreover, it would be interesting to check if it is possible to incorporate the ideas of Customizable Contraction Hierarchies [DSW14] to allow frequent changes in the edge costs.

Regarding the algorithm presented in Section 2.4 it is an interesting open question whether the search space sizes of CH and LCH have common optimal node orderings or not. If not, it would be interesting to investigate how to find node orderings that are specifically well suited for LCH.

The results in Section 2.5 are only one small step towards understanding the CH query complexity. We believe that, especially with respect to lower bounds, our understanding of the subject is still rather limited. A graph property such as the highway dimension [AFGW10] that allows to upper *and* lower bound the CH query complexity would be very desirable.

We know that our results shown in Section 3.2 regarding extreme shortest paths are tight for $d = 2$ [Car83a]. However, we are not aware of any known nontrivial lower bounds for $d > 2$. It would be interesting to see whether one can show matching lower bounds for general d. Other properties such as the complexity of preference polyhedra could be investigated as well. We conjecture that the average complexity of the preference polyhedra in a preference space subdivision is polynomial in n for a fixed number of cost types d. Furthermore, it may be possible to generalize the results of Section 3.2 to point sets that are not induced by path sets but that satisfy weaker assumptions.

The trajectory clustering approach in Section 3.4 requires the input trajectories to be shortest paths. Unfortunately, this is a property that is rarely satisfied in practice. The solution to this issue could be found in the preference elicitation algorithm of Section 3.3 that can handle any input trajectories. It is, however, unclear how to reformulate the optimization problem of the trajectory clustering accordingly and whether this new problem can be tackled as efficiently.

Bibliography

[20] *OpenStreetMap*. en-US. https://www.openstreetmap.org/. 2020. (Visited on 03/29/2018) (cit. on p. 33).

[ADF+11] I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, R. F. Werneck. 'VC-dimension and shortest path algorithms'. In: *International Colloquium on Automata, Languages, and Programming (ICALP)*. Springer. 2011, pp. 690–699 (cit. on p. 44).

[ADF+16] I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, R. F. Werneck. 'Highway dimension and provably efficient shortest path algorithms'. In: *Journal of the ACM (JACM)* 63.5 (2016), pp. 1–26 (cit. on p. 44).

[ADGW11] I. Abraham, D. Delling, A. V. Goldberg, R. F. Werneck. 'A hub-based labeling algorithm for shortest paths in road networks'. In: *International Symposium on Experimental Algorithms (SEA)*. Springer. 2011, pp. 230–241 (cit. on pp. 22, 23).

[ADGW12] I. Abraham, D. Delling, A. V. Goldberg, R. F. Werneck. 'Hierarchical hub labelings for shortest paths'. In: *European Symposium on Algorithms (ESA)*. Springer. 2012, pp. 24–35 (cit. on pp. 23, 39).

[AFGW10] I. Abraham, A. Fiat, A. V. Goldberg, R. F. Werneck. 'Highway dimension, shortest paths, and provably efficient algorithms'. In: *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms (SODA)*. SIAM. 2010, pp. 782–793 (cit. on pp. 44, 46, 96).

[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. 88).

[AKT13] O. Andersen, B. B. Krogh, K. Torp. 'An open-source based ITS platform'. In: *2013 IEEE 14th International Conference on Mobile Data Management (MDM)*. Vol. 2. IEEE. 2013, pp. 27–32 (cit. on p. 80).

[AMS06] N. Alon, D. Moshkovitz, S. Safra. 'Algorithmic construction of sets for k-restrictions'. In: *ACM Transactions on Algorithms (TALG)* 2.2 (2006), pp. 153–177 (cit. on p. 87).

[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 p. 90).

[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 (CIAC)*. Springer. 2010, pp. 359–370 (cit. on p. 24).

[BCRW16] R. Bauer, T. Columbus, I. Rutter, D. Wagner. 'Search-space size in contraction hierarchies'. In: *Theoretical Computer Science* 645 (2016), pp. 112–127 (cit. on pp. 45, 46, 55).

[BDG+16] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, R. F. Werneck. 'Route planning in transportation networks'. In: *Algorithm engineering*. Springer, 2016, pp. 19–80 (cit. on p. 33).

[BFJP20] F. Barth, S. Funke, T. S. Jepsen, C. Proissl. ‘Scalable unsupervised multi-criteria trajectory segmentation and driving preference mining’. In: *Proceedings of the 9th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data (BigSpatial)*. 2020, pp. 1–10 (cit. on pp. 17, 18, 77, 93).

[BFM06] H. Bast, S. Funke, D. Matijevic. ‘Transit ultrafast shortest-path queries with linear-time preprocessing’. In: *9th DIMACS Implementation Challenge [1]* (2006) (cit. on pp. 19, 20, 39).

[BFMP22] D. Bahrdt, S. Funke, S. Makolli, C. Proissl. ‘Distance Closures: Unifying Search- and Lookup-based Shortest Path Speedup Techniques’. In: *Proceedings of the Twenty-Fourth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2022 (cit. on pp. 17, 27).

[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)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2021 (cit. on pp. 17, 18, 67, 83).

[BFP22] F. Barth, S. Funke, C. Proissl. ‘An Upper Bound on the Number of Extreme Shortest Paths in Arbitrary Dimensions’. In: *30th Annual European Symposium on Algorithms (ESA)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2022 (cit. on pp. 17, 18, 66).

[BFS19] F. Barth, S. Funke, S. Storandt. ‘Alternative Multicriteria Routes’. In: *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)*. 2019 (cit. on pp. 65, 67).

[BFS21] J. Blum, S. Funke, S. Storandt. ‘Sublinear search spaces for shortest path planning in grid and road networks’. In: *Journal of Combinatorial Optimization* 42.2 (2021), pp. 231–257 (cit. on pp. 17, 25, 45, 46, 48, 55, 95).

[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. 87).

[BGK+15] M. Babenko, A. V. Goldberg, H. Kaplan, R. Savchenko, M. Weller. ‘On the complexity of hub labeling’. In: *International Symposium on Mathematical Foundations of Computer Science (MFCS)*. Springer. 2015, pp. 62–74 (cit. on p. 27).

[BGT81] R. G. Bland, D. Goldfarb, M. J. Todd. ‘The ellipsoid method: A survey’. In: *Operations research* 29.6 (1981), pp. 1039–1091 (cit. on p. 64).

[BJS13] A. Balteanu, G. José, M. Schubert. ‘Mining driving preferences in multi-cost networks’. In: *International Symposium on Spatial and Temporal Databases (SSTD)*. Springer. 2013, pp. 74–91 (cit. on p. 77).

[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. 66).

[Bla77] R. G. Bland. ‘New finite pivoting rules for the simplex method’. In: *Mathematics of operations Research* 2.2 (1977), pp. 103–107 (cit. on p. 64).

[BS00] H. Benson, E. Sun. ‘Outcome space partition of the weight set in multiobjective linear programming’. In: *Journal of Optimization Theory and Applications* 105.1 (2000), pp. 17–36 (cit. on p. 66).

[BS20] J. Blum, S. Storandt. ‘Lower bounds and approximation algorithms for search space sizes in contraction hierarchies’. In: *28th Annual European Symposium on Algorithms (ESA)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020 (cit. on p. 46).

[Car83a] P. J. Carstensen. ‘Complexity of some parametric integer and network programming problems’. In: *Mathematical Programming* 26.1 (1983), pp. 64–75 (cit. on pp. 66, 76, 77, 87, 93, 96).

[Car83b] P. J. Carstensen. ‘The complexity of some problems in parametric linear and combinatorial programming’. PhD thesis. University of Michigan, 1983 (cit. on p. 66).

[CHKZ03] E. Cohen, E. Halperin, H. Kaplan, U. Zwick. ‘Reachability and distance queries via 2-hop labels’. In: *SIAM Journal on Computing* 32.5 (2003), pp. 1338–1355 (cit. on p. 22).

[CRLB16] P. Campigotto, C. Rudloff, M. Leodolter, D. Bauer. ‘Personalized and situation-aware multimodal route recommendations: the FAVOUR algorithm’. In: *IEEE Transactions on Intelligent Transportation Systems (T-ITS)* 18.1 (2016), pp. 92–102 (cit. on p. 77).

[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: *Proceedings of the 23rd SIGSPATIAL international conference on advances in geographic information systems*. 2015, pp. 1–9 (cit. on p. 77).

[Dij+59] E. W. Dijkstra et al. ‘A note on two problems in connexion with graphs’. In: *Numerische mathematik* 1.1 (1959), pp. 269–271 (cit. on pp. 11, 13).

[DOW+55] G. B. Dantzig, A. Orden, P. Wolfe, et al. ‘The generalized simplex method for minimizing a linear form under linear inequality restraints’. In: *Pacific Journal of Mathematics* 5.2 (1955), pp. 183–195 (cit. on pp. 63, 65).

[DSW14] J. Dibbelt, B. Strasser, D. Wagner. ‘Customizable contraction hierarchies’. In: *International Symposium on Experimental Algorithms (SEA)*. Springer. 2014, pp. 271–282 (cit. on pp. 21, 22, 65, 96).

[DW09] D. Delling, D. Wagner. ‘Pareto Paths with SHARC’. In: *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA)*. Vol. 5526. Lecture Notes in Computer Science. Springer, 2009, pp. 125–136 (cit. on p. 67).

[DYGD15] J. Dai, B. Yang, C. Guo, Z. Ding. ‘Personalized route recommendation using big trajectory data’. In: *IEEE 31st international conference on data engineering (ICDE)*. IEEE. 2015, pp. 543–554 (cit. on p. 77).

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Vol. 10. EATCS Monographs on Theoretical Computer Science. Springer, 1987 (cit. on p. 87).

[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. 80).

[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*. 2016, pp. 1–9 (cit. on pp. 7, 18, 77, 84, 96).

[FLS17] S. Funke, S. Laue, S. Storandt. ‘Personal routes with high-dimensional costs and dynamic approximation guarantees’. In: *16th International Symposium on Experimental Algorithms (SEA)*. 2017 (cit. on pp. 41, 65).

[FM19] S. Funke, T. Mendel. ‘International Symposium on Experimental Algorithms (SEA)’. In: ed. by I. Kotsireas, P. Pardalos, K. E. Parsopoulos, D. Souravlias, A. Tsokas. Springer, 2019. Chap. Improved Contraction Hierarchy Queries via Perfect Stalling, pp. 158–166 (cit. on p. 16).

[FMN05] S. Funke, K. Mehlhorn, S. Näher. ‘Structural filtering: a paradigm for efficient and exact geometric programs’. In: *Computational Geometry* 31.3 (2005), pp. 179–194 (cit. on p. 93).

[FPT81] R. J. Fowler, M. S. Paterson, S. L. Tanimoto. ‘Optimal packing and covering in the plane are NP-complete’. In: *Information processing letters* 12.3 (1981), pp. 133–137 (cit. on p. 87).

[FS15a] S. Funke, S. Storandt. 'Personalized route planning in road networks'. In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 2015, pp. 1–10 (cit. on pp. 5, 7, 11, 57, 64, 67, 93, 95).

[FS15b] S. Funke, S. Storandt. 'Provable efficiency of contraction hierarchies with randomized preprocessing'. In: *International Symposium on Algorithms and Computation (ISAAC)*. Springer. 2015, pp. 479–490 (cit. on pp. 33, 45, 46).

[FSS17] S. Funke, N. Schnelle, S. Storandt. 'URAN: A Unified Data Structure for Rendering and Navigation'. In: *W2GIS*. Vol. 10181. Lecture Notes in Computer Science. 2017, pp. 66–82 (cit. on p. 37).

[FT87] M. L. Fredman, R. E. Tarjan. 'Fibonacci heaps and their uses in improved network optimization algorithms'. In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615 (cit. on p. 13).

[Fun20] S. Funke. 'Seamless Interpolation Between Contraction Hierarchies and Hub Labels for Fast and Space-Efficient Shortest Path Queries in Road Networks'. In: *International Computing and Combinatorics Conference (COCOON)*. Springer. 2020, pp. 123–135 (cit. on pp. 28, 33, 36).

[Geo73] A. George. 'Nested dissection of a regular finite element mesh'. In: *SIAM Journal on Numerical Analysis (SINUM)* 10.2 (1973), pp. 345–363 (cit. on p. 22).

[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. 65).

[GR19] K. Gajjar, J. Radhakrishnan. 'Parametric shortest paths in planar graphs'. In: *IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2019, pp. 876–895 (cit. on pp. 7, 62, 63, 66–68, 93, 96).

[GRS13] A. V. Goldberg, I. Razenshteyn, R. Savchenko. 'Separating hierarchical and general hub labelings'. In: *International Symposium on Mathematical Foundations of Computer Science (MFCS)*. Springer. 2013, pp. 469–479 (cit. on p. 27).

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. 'Contraction hierarchies: Faster and simpler hierarchical routing in road networks'. In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2008, pp. 319–333 (cit. on pp. 11, 14, 16, 25, 39, 42, 43).

[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. 11, 14–16, 21, 33, 42, 43).

[Gus80] D. M. Gusfield. 'Sensitivity analysis for combinatorial optimization'. PhD thesis. University of California, Berkeley, 1980 (cit. on pp. 7, 62, 66, 93, 96).

[Gut04] R. J. Gutman. 'Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks.' In: *ALENEX/ANALC* 4 (2004), pp. 100–111 (cit. on pp. 19, 20, 39).

[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. 66).

[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 (T-ITS)* (2020), pp. 1–12 (cit. on p. 80).

[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: *IEEE International Conference on Big Data (Big Data)*. IEEE. 2018, pp. 3422–3431 (cit. on pp. 80, 81).

[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. 89).

[Kar84] N. Karmarkar. ‘A new polynomial-time algorithm for linear programming’. In: *Proceedings of the sixteenth annual ACM symposium on Theory of computing (STOC)*. 1984, pp. 302–311 (cit. on p. 64).

[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 p. 67).

[KV17] A. Kosowski, L. Viennot. ‘Beyond highway dimension: small distance labels using tree skeletons’. In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2017, pp. 1462–1478 (cit. on p. 45).

[LUYK17] Y. Li, L. H. U, M. L. Yiu, N. M. Kou. ‘An Experimental Study on Hub Labeling Based Shortest Path Algorithms’. In: *Proc. VLDB Endow.* 11.4 (Dec. 2017), pp. 445–457. URL: `https://doi.org/10.1145/3186728.3164141` (cit. on p. 39).

[Mil12] N. Milosavljevi’c. ‘On optimal preprocessing for contraction hierarchies’. In: *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS)*. 2012, pp. 33–38 (cit. on pp. 24, 44).

[MR09] N. H. Mustafa, S. Ray. ‘PTAS for geometric hitting set problems via local search’. In: *Proceedings of the twenty-fifth annual symposium on Computational geometry (SoCG)*. 2009, pp. 17–22 (cit. on p. 87).

[MS00] K. Mulmuley, P. Shah. ‘A lower bound for the shortest path problem’. In: *Proceedings 15th Annual IEEE Conference on Computational Complexity (CCC)*. IEEE. 2000, pp. 14–21 (cit. on pp. 66, 93).

[MZ00] K. Mehlhorn, M. Ziegelmann. ‘Resource constrained shortest paths’. In: *European Symposium on Algorithms (ESA)*. Springer. 2000, pp. 326–337 (cit. on p. 66).

[Ope18] OpenStreetMap contributors. *Street Network of South America*. `https://www.openstreetmap.org`. 2018 (cit. on p. 41).

[PR21] C. Proissl, T. Rupp. ‘On the Difference between Search Space Size and Query Complexity in Contraction Hierarchies’. In: *SIAM Conference on Applied and Computational Discrete Algorithms (ACDA)*. SIAM. 2021, pp. 77–87 (cit. on pp. 17, 46).

[Pro22] C. Proissl. ‘Light Contraction Hierarchies: Hierarchical Search Without Shortcuts’. In: *Proceedings of the International Symposium on Combinatorial Search (SoCS)*. Vol. 15. 1. 2022, pp. 234–238 (cit. on pp. 17, 39).

[RF20] T. Rupp, S. Funke. ‘A lower bound for the query phase of contraction hierarchies and hub labels’. In: *International Computer Science Symposium in Russia (CSR)*. Springer. 2020, pp. 354–366 (cit. on p. 46).

[SS05] P. Sanders, D. Schultes. ‘Highway hierarchies hasten exact shortest path queries’. In: *13th European Symposium on Algorithms (ESA)*. Springer. 2005, pp. 568–579 (cit. on p. 20).

[SS06] P. Sanders, D. Schultes. ‘Engineering highway hierarchies’. In: *European Symposium on Algorithms (ESA)*. Springer. 2006, pp. 804–816 (cit. on pp. 20, 39).

[SS07] D. Schultes, P. Sanders. ‘Dynamic highway-node routing’. In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2007, pp. 66–79 (cit. on pp. 15, 21, 39).

[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. 90).

[Van+20] R. J. Vanderbei et al. *Linear programming*. Springer, 2020 (cit. on p. 63).

[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. 90).

[Whi15] C. White. ‘Lower bounds in the preprocessing and query phases of routing algorithms’. In: *European Symposium on Algorithms (ESA)*. Springer, 2015, pp. 1013–1024 (cit. on pp. 44, 46).

[YGMJ15] B. Yang, C. Guo, Y. Ma, C. S. Jensen. ‘Toward personalized, context-aware routing’. In: *The VLDB Journal* 24.2 (2015), pp. 297–318 (cit. on p. 77).