Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Personalized route planning using machine learning

Temirlan Kurbanov

Supervisor: doc. Ing. Michal Jakob, Ph.D.
Field of study: Open Informatics
Subfield: Artificial Intelligence
May 2020

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Kurbanov Temirlan**	Personal ID number:	**420157**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence**		

II. Master’s thesis details

Master’s thesis title in English:

Personalized route planning using machine learning

Master’s thesis title in Czech:

Personalizované plánování tras využívající strojového učení

Guidelines:

Personalized route planning is a challenging open problem. The aim of personalized route planning is to leverage data on past travel behaviour to generate routes that pass through popular locations and/or route segments or which pass through locations and/or route segments that have similar characteristics as the popular ones. When solving the problem, please proceed as follows:
1. Familiarize yourself with existing approaches to personalized route planning.
2. Formalize personalized route planning problem
3. Design an algorithm for personalized route planning.
4. Implement the designed algorithm.
5. Evaluate the performance of the algorithm on real-world data.

Bibliography / sources:

[1] D. Delling, A. V. Goldberg, M. Goldszmidt, J. Krumm, K. Talwar, and R. F. Werneck. 2015. Navigation made personal: inferring driving preferences from GPS traces. In Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL ’15). New York, NY, USA, Article 31, pp. 1–9.
[2] J. Dai, B. Yang, C. Guo and Z. Ding. 2015. Personalized route recommendation using big trajectory data. In Proceedings of the IEEE 31st International Conference on Data Engineering, Seoul, pp. 543-554.
[3] S. Funke and S. Storandt. 2015. Personalized route planning in road networks. In Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL ’15). NY, USA, Article 45, pp. 1–10.

Name and workplace of master’s thesis supervisor:

doc. Ing. Michal Jakob, Ph.D., Artificial Intelligence Center, FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **04.02.2020** Deadline for master's thesis submission: **22.05.2020**

Assignment valid until: **30.09.2021**

doc. Ing. Michal Jakob, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I would like to thank my supervisor doc. Ing. Michal Jakob, Ph.D. for his insight and all the advice. I would also like to express my sincere gratitude to Ing. Pavol Žilecký for providing me with all the necessary guidance and support throughout the research.

Declaration

Abstract

The majority of modern navigation systems are mainly focused on minimizing the travelling time of a planned route. While travelling time is undeniably important, it is not always the main priority of a user. Although a limited set of planning customizations is available, these must be set manually and do not provide distinction between different types of users, such as car drivers and cyclists.

This thesis studies the topic of navigation personalization through the examination of previous user routes. The theoretical part addresses the task of path planning in road networks and the utilization of optimization algorithms in route preference computation. The practical part of the thesis is dedicated to the implementation of an algorithm capable of deducing a user's priorities from the history of their routes. The algorithm has been tested both on artificially generated and real-world trajectories.

Keywords: navigation, route planning, personalization, machine learning

Supervisor: doc. Ing. Michal Jakob, Ph.D.

Abstrakt

Majorita současných navigačních systému je zaměřená na plánování nejrychlejší cesty. Zatímco důležitost doby cesty nelze popřít, není vždy hlavní prioritou uživatele. Ačkoli existuje limitované množství dostupných úprav plánování, mají být zadány ručně a nezajišťují rozlišení různých typů uživatelů, jako jsou řidiči a cyklisti.

Tato diplomová práce se zabývá studiem personalizace navigace prostřednictvím zkoumání předcházejících tras uživatele. Teoretická část je věnována úloze plánování cest v dopravních sítích a využití optimalizačních algoritmů k výpočtu drahových preferencí. Praktickou částí práce je implementace algoritmu schopného získávání uživatelských priorit z historie jeho cest. Algoritmus je otestován na uměle vytvořených a skutečných trajektoriích.

Klíčová slova: navigace, plánování tras, personalizace, strojové učení

Překlad názvu: Personalizované plánování tras využívající strojového učení

Contents

1 Introduction **1**

2 Related work **5**

2.1 Shortest path search 5

2.1.1 Naive methods 6

2.1.2 Informed methods 7

2.1.3 Separator-based methods 9

2.1.4 Hierarchical methods 11

2.1.5 Path extraction 12

2.2 Optimization algorithms 12

2.2.1 Linear Programming 13

2.2.2 Reference trajectories 16

3 Problem statement **19**

4 Proposed method **21**

4.1 Shortest path search 21

4.2 Preference optimization 25

4.3 Overall approach 27

5 Experimental evaluation **31**

5.1 Preprocessing 31

5.2 Preference optimization 33

5.2.1 Artificial data 33

5.2.2 Real-world data 36

6 Conclusion **39**

Bibliography **41**

Figures

4.1 SCD flowchart 29

5.1 Average loss for different training set sizes 36

5.2 An exemplary optimization progression at $k = 32$ 37

Tables

5.1 Overlay graph C parameters for varying k values 32

5.2 Pruning percentage by different techniques 32

5.3 Road parameters and their possible values 35

Chapter 1

Introduction

Navigation in its broad definition is the process of controlling the position and movement of an object on its path from an origin to a destination. The term can also refer to a set of special skills used to determine one's location. Generally, four categories of navigation are distinguished: space navigation, aerial navigation, land navigation, and marine navigation.

The emergence of navigation as a form of art can be traced back to 3500 B.C., when the first records of trading ships appeared. These boats followed the coast and used visible landmarks at day and major constellations at night to monitor their position. Obviously, maps and charts were not available then, and the only instrument available was sandglass. One of the first known crafted navigation instruments were the mariner's compass, the earliest example of a magnetic compass, and the lead line, which was a tool for measuring the water depth and the landscape of the bottom. The efficiency and ease of use of compasses were greatly increased by the Mercator Projection, the first accurate spherical model of the planet's surface. On this projection, the compass bearing could be represented in a straight line. As a result, the ability to precisely determine the longitude became crucial for consistent navigation. Since seamen already had the means to measure a ship's speed, an accurate time-keeping tool was necessary. It was invented by John Harrison[T.G23]. Born in England, this self-educated clockmaker designed a chronometer accurate to one-tenth of a second per day. James Cook used a copy of this device in his third voyage (1776-1780) [Hou94] to create the first accurate charts. Although expensive, the ship chronometers were irreplaceable due to their precision. However, after the worldwide spread and adoption of radio in the 20th century, regularly provided time updates made a usual wrist watch sufficient for precise longitude calculation. Eventually, a number of radio systems were developed, including Gee, Loran, and Decca — hyperbolic radio navigation systems introduced during World War II. These systems were later replaced by GPS (Global Positioning System), a satellite radio navigation system designed by the U.S. Department of Defense in 1973.

GPS operates to this day along with analogical systems (GLONASS, Galileo, BeiDou) and can provide location data with accuracy within 0.5 meters. Moreover, these systems are accessible not only for mariners and militaries, but virtually for anyone. A personal navigation assistant device can be bought at any consumer electronics store, most smartphones have preinstalled navigation applications such as Google Maps and Apple Maps. These provide users with highly detailed interactive maps and 3D models of countries, cities, and even streets. Moreover, traffic information is processed and updated continuously, including jams, accidents, and roadworks. Using this information, the navigation devices are able to quickly build a fastest route to a desired destination considering different means of transport. These features make navigation applications valuable not only for professional drivers, but also for vehicle owners, commuters, travelers, and tourists.

Despite the efficiency and usage simplicity of navigation applications, they have some drawbacks. Namely, in the overwhelming majority of cases, the main planning criterion is the time a route takes. Although the travelling time is undeniably important, it is not always the main priority. Some other route properties such as road surface and elevation may be of greater significance to an individual user. Moreover, the ratios of importance of road qualities can influence the desired path choice significantly. The individual dependencies of a user must also depend on the type of transport they use. For example, the route choice of a person driving a car is unlikely to be dictated by elevation degree, while it is of great importance to a cyclist. On the other hand, a cyclist takes the surface quality and bicycle infrastructure into great consideration, but these are less relevant to a driver. Even if one decides to build a navigation system capable of considering and prioritizing multiple route parameters, not every user is able or willing to take time to express their preferences explicitly. Therefore, the aim of this thesis is to implement and algorithm capable of computing and employing the route preferences of individual users based on the history of their commutes.

The main challenge of this thesis project is the extraction of an individual's route preference coefficients. The expected input is a set of a user's previous commute trajectories. Analyzing these commutes, the program extracts the "preference weights" of the route features in question. It should be emphasized that in order for the algorithm to make a correct conclusion, the input trajectories must reflect the user's priorities correctly. In other words, the routes have to be built by an informed user (i.e. familiar with the area) according to their personal preferences. If the input trajectories are produced by a navigation planner instead of being chosen by the user, the results of the algorithm will reflect the planner's priorities instead of the personal preferences of a user. Simultaneously, if the input trajectories are built by a user in an area they are not well aware of, the trajectories will be random to significant extent. After the preference analysis is performed, the algorithm can use the extracted coefficients to plan new routes according to the user's priorities.

For a route search algorithm to be able to run on a road network, its map should be modelled into a directed weighted graph. Processing a digital map for graph extraction is a non-trivial task on its own, so for this thesis project, a previously built graph of Prague's road network has been used. The project can be divided into two subtasks: preference computation from trajectory data and route search in a road network, which is necessary to evaluate the quality of proposed paths during preference optimization and to be able to plan further routes once the preferences are extracted.

Chapter 2

Related work

Since automatic navigation is not a new concept, there is a significant amount of articles and algorithms dedicated to it. Nevertheless, navigation tailored to individual preferences is a relatively new concept. Therefore, there are still relatively few advances in the topic. The work related to personalized navigation can be separated into two parts: shortest path query in road networks and optimization algorithms capable of deducing the preference ratios. Instead of briefly listing the researches conducted on the topic, this section provides a more detailed description of the algorithms applied to these tasks and their performance.

2.1 Shortest path search

As it is stated above, a road network can be modelled as a directed weighted graph, where intersections correspond to vertices, the road segments connecting them correspond to edges, and the edge weights are resembled by chosen metrics (e.g. distance, traversal time, fuel consumption). After the directed graph is built, the route search task can be narrowed down to a special case of the search of a shortest point-to-point path in a graph. Shortest path problem in graph theory is a task of finding a path between two vertices of a graph such that it has the minimum sum of the weights of its component edges. Graphs built from road networks usually consist of thousands and millions of vertices. Hence, time efficiency of the shortest path algorithm is crucial for online usage of the navigation system. Some of the algorithms developed for route query in road networks employ preprocessing techniques, which are capable of accelerating the path searches by tens of times. However, in the case of personalized navigation, one should take into account the continuous adjustment of preference ratios (and final edge costs) during the optimal preference extraction. These adjustments may necessitate repetitive preprocessing, resulting in a significant increase of the overall running time. Therefore, methods both with and without preprocessing should be considered [BDG+15].

2.1.1 Naive methods

Dijkstra's algorithm [Dij59] is the basic and probably the most standard shortest path algorithm. It is a one-to-all shortest path algorithm developed by Edsger Dijkstra in 1956. The method works by building a priority queue of vertices ordered by minimum distances from the source node s. Initially, the priority queue only contains s, and the distance from the starting vertex to every other one is set to infinity, marking these vertices as unprocessed. At every iteration, the algorithm extracts the *current* minimum distance node u from the priority queue and updates the distances of the adjacent vertices. For every adjacent vertex v, it calculates the distance $dist(s,u) + e(u,v)$, where $e(u,v)$ is the cost of the edge between u and v. If this new distance is smaller than the current distance from s to v, an update is performed: $dist(s,v)$ is set to the new value and vertex v is added to the priority queue. Once the current vertex u is processed (all its neighbor vertices are updated), it is considered closed and the minimum distance $dist(s,u)$ is correct. Thanks to that label-setting property, the algorithm can be stopped when the destination vertex is processed for point-to-point queries, which are more frequent in navigation. The asymptotic complexity of standard Dijkstra's algorithm is $\mathcal{O}((|V|+|E|)\log|V|)$ or $\mathcal{O}(|E|+|V|\log|V|)$ for priority queues based on binary heaps and Fibonacci heaps respectively, where $|V|$ is the number of vertices in the graph and $|E|$ is the number of edges. For point-to-point, the number of vertices processed can be reduced significantly by applying bidirectional search. It is performed by running forward search from the starting vertex and backward search from the destination vertex simultaneously. The bidirectional search can terminate when a vertex has been closed by both the forward and backward searches.

An alternative one-to-all shortest path technique is the Bellman-Ford algorithm [BEL58]. Instead of maintaining a priority queue of minimum distances, it processes all of the edges in each of the $|V|-1$ cycles. For every edge (u,v) with cost $e(u,v)$, if the condition $dist(u) + e(u,v) < dist(v)$ is satisfied, the distance to v is set to the new value. In contrast to Dijkstra's algorithm, the Bellman-Ford algorithm is a label-correcting algorithm, since every vertex is scanned multiple times. In every iteration, at least one vertex is assigned correct minimum distance. Hence, the algorithm is bound to assign optimal distances to all vertices in $|V| - 1$ iterations. This bound makes the algorithm useful for negative weight cycle detection. A cycle in a graph is said to have negative weight, if the sum of its component edges is less than 0. If after $|V| - 1$ iterations the algorithm can still decrease the distance of a vertex in the graph, the vertex is a part of a negative cycle. Although important in a number of problems, this property is of no significant use in most road network planning tasks, since all road segment costs are assumed to be positive. The worst-case asymptotic complexity of Bellman-Ford is $\mathcal{O}(|V||E|)$, although it can be modified slightly to decrease the number of iterations. If an iteration has been performed without decreasing the distance of any of the nodes, the algorithm can terminate, since the optimal solution has been found. This modification makes the technique competitive to Dijkstra's algorithm in some tasks.

The third algorithm that should be noted in this subsection is the Floyd-Warshall algorithm [Flo62], sometimes called the Roy-Warshall algorithm. As opposed to previous techniques, this method is an all-to-all shortest path algorithm, meaning it calculates minimum distances for all pairs of vertices in a graph. This is achieved by comparing all possible paths between each pair of vertices. The asymptotic complexity of the algorithm is $\Theta(|V|^3)$, which, however, usually yields better execution time than $|V|$ iterations of Dijkstra's algorithm for a dense graph. As one can assume, this algorithm is rarely used on road networks, since those usually contain thousands and millions of vertices.

2.1.2 Informed methods

All of the algorithms discussed above compute shortest distances to all vertices. For a point-to-point navigation, the most effective approach would be terminating Dijkstra's algorithm after it scans the destination node. Nevertheless, it still processes all the nodes with distances smaller than the one of the destination vertex. In contrast to that, informed or goal-directed search methods employ techniques directing them towards the goal, thus decreasing the number of nodes processed.

Probably the most well known informed shortest path search algorithm is A* [HNR68], which can be considered an extension of Dijkstra's algorithm. The distinction consists in the application of a heuristic function $h : V \rightarrow \mathbb{R}$ which estimates the potential distance from a vertex to the destination node. Similar to Dijkstra's algorithm, A* maintains a priority queue of most promising vertices to be expanded. However, the vertices are ordered not by their distance from the source vertex, but by the function $f(u) = g(u) + h(u)$, where $g(u)$ is the distance from the source vertex to a vertex u, and $h(u)$ is a heuristic estimate of the cost of the shortest path from u to the target vertex. This ordering allows A* to expand its search tree mainly in the direction of the target node. The heuristic function is problem-specific, and its formulation affects the efficiency of the algorithm dramatically. If h is admissible, meaning it never overestimates the cost of reaching the destination, the algorithm is guaranteed to find the minimum-cost path from start to destination. Moreover, if the heuristic estimate is exact, meaning it always returns a true distance to destination, the algorithm will only process and expand the vertices on the shortest path from start to target vertex. If A* is applied to a classic road network navigation problem with cost denoted by distance, a standard heuristic estimate function would be a straight-line distance (for relatively small search areas) or a great-circle distance (the minimum distance on a sphere surface). If the cost metric is travel time, geographical distance between vertices divided by the maximum speed occurring in the road network is an admissible heuristic. In practice, however, the time gain given by these estimates is relatively small. A more aggressive heuristic function has a chance of improving the performance, but a non-admissible heuristic cannot always return an optimal solution. The asymptotic time complexity of the algorithm can be expressed as $\mathcal{O}(b^d)$, where b is the branching factor(the average number of successors per vertex) and d is the depth of the solution (the number of vertices in the shortest path). There is a number of major improvements and modifications to the standard A* method, some of which use preprocessing. Namely, LPA* (Lifelong Planning A*) is an incremental version of A* effective on non-stationary graphs. LPA* is able to adapt to changes in a graph without reprocessing it entirely. Theta* [DNKF14] is one of any-angle path planning algorithms, which are tailored for path search in a space. Theta* accepts transitions between any vertices connected by a straight line without obstacles. This modification results in a relatively direct path with few turns. ALT (A*, landmarks, and triangle inequality) [GH05] includes a preprocessing phase where a small subset of vertices is selected as landmarks. The path costs between these landmarks and all vertices in the graph are precomputed and used in triangle inequality during path queries to estimate distance to destination. However, this modification is ineffective in preference optimization, since the preprocessing phase must be repeated after every cost function adjustment.

Another informed path search algorithm is Geometric Containers. Similar to ALT, it has a preprocessing phase. For every edge $e = (u, v)$, it labels a set of vertices the shortest paths from u to which start with e. In its core, the preprocessing phase is an all-to-all shortest path search. However, in case of application on road networks, the preprocessing phase can be accelerated by replacing a standard all-to-all path search with heuristic approximation based on vertex coordinates. Unfortunately, if the cost metric is not based on travel distance or time, this heuristic is unsuitable. The query phase is a graph traversal guided by the label information: for every node, if a label of an incident edge e does not contain the target vertex, the search tree should not be expanded via e. Unfortunately, the algorithm cannot be used for preference optimization, since every weight adjustment necessitates repetitive preprocessing, and coordinate heuristic cannot be applied due to consideration of other road parameters apart from distance and traversal time.

The Arc Flags [HKMS09] method resembles Geometric Containers to some extent despite not using coordinate heuristic. The preprocessing phase of Arc Flags splits the graph into a predefined number K of parts, also called cells. The partition is performed so as to balance the cells by assigning an approximately equal number of vertices to every cell and minimize the number of border vertices. Every edge is the assigned a vector of K flags, where the i-th flag of is set to true if the edge is a part of the shortest path to the cell i. There are multiple possible ways of flag computation, the standard one being backward shortest path search from border vertices for every cell. After all flags are set, the edges without true flags can be safely pruned. During a query, edges that do not contain true flag for the destination cell can be ignored. For large graphs multiple nested partitions can be built, which generally improves the query performance. Although Arc Flags yields relatively fast shortest path queries, time requirements of the preprocessing stage are relatively high, making it especially inefficient for cost function adjustment — whenever the preference coefficients are changed, flag assignment must be performed again.

2.1.3 Separator-based methods

The planar separator theorem [LT77] states that any planar graph can be partitioned into smaller subgraphs through a removal of a small number of vertices. The removal of $\mathcal{O}(\sqrt{n})$ nodes from a graph with n vertices can split the graph into separate subgraphs containing at most $\frac{2}{3}n$ nodes each. Although road graphs are not planar (due to tunnels and overpasses), the separator property can be applied to them [DGRW11, EG08] for shortest path search acceleration in both vertex separator and edge separator forms.

A vertex separator is a subset of vertices whose removal splits the graph into several disjoint components. The separator subset should be as small as possible while keeping the sizes of the resulting components roughly equal. The separators are used to produce an overlay graph, while shortcut edges [Vli78] are added to it in a manner that preserves the initial distances between all pairs of separator vertices. The produced overlay graph is then incorporated in shortest path search. The separator vertices play the role of landmarks — important nodes that have to be traversed if the origin and the destination belong to different components. There are multiple possible specifications to the general approach. Namely, the separator vertex choice can be based on different properties, for example the number of adjacent edges. The shortcut edges offer a trade-off between search speed and preprocessing requirements. A nested multilevel model of the overlay graph is possible to.

Arc separator approach follows the same logic, but in a different order. First, the graph is partitioned in a set of cells so that it is roughly balanced and the number of separator edges is minimized. A separator edge connects two border vertices belonging to different cells. Afterwards, one can add shortcut edges connecting pairs of border vertices belonging to the same component while preserving distances between them. An outstanding arc separator algorithm in context of road networks is Customizable Route Planning (CRP) [DGPW11]. CRP can be considered the current "state-of-the-art" shortest path search algorithm designed specifically for application to road networks [DGPW15]. Similar to the algorithms described above, the preprocessing phase starts with the generation of a balanced cell partition of the initial graph G. After that, it builds an overlay graph H consisting of border vertices and the edges connecting them. When the border structure is extracted, a clique is built inside every partition: for every pair of boundary vertices in a cell, an edge is added with the cost of the shortest path between these vertices. It should be noted that preprocessing becomes metric-dependent at this stage. Traditionally, the clique of a cell can be built by running standard Dijkstra's algorithm from each of its border vertices one by one. After the cell cliques are produced, H becomes the overlay graph of G. Nested levels of the overlay graph can be built in a recursive manner for query acceleration. A query in CRP is performed using the bidirectional version of Dijkstra's algorithm. For an origin vertex s and a destination vertex t, the query runs as follows. If s is a border vertex, no preparations must be performed. If s does not belong to the set of border vertices, it is connected to H via the border vertices of its cell $C(s)$ in the same way the cell cliques are produced (i.e. Dijkstra's algorithm is run from s to the border vertices of $C(s)$). After that, t is analyzed and prepared similarly. The resulting structure is called the search graph, and bidirectional Dijkstra's algorithm is run. It returns a shortest path from s to t consisting of three parts: a prefix transition inside $C(s)$, a path in H, and a suffix part in $C(t)$. Since H is an overlay graph preserving the distances between pairs of vertices, the distance of the path is equal to that of the shortest path in G. To further improve the running time of queries a pruning technique can be applied. Not necessarily all inner shortcuts are needed for shortest distance preservation in the overlay graph, since only the paths that are fully inside a cell are used in clique generation. These unnecessary edges can be easily removed using Dijkstra's algorithm. For every vertex $u \in H$, Dijkstra's algorithm is run until it processes all nodes adjacent to u. Afterwards, for every neighbor v of u connected by an edge $e(u, v)$ with cost l, if $dist(v) < l$, then e can be removed from H. Since CRP has been designed specifically for operation on road networks, it incorporates a number of distinctive features. Namely, it is able to consider turn restrictions and impose additional costs on undesired types of turns (e.g. left turns, U-turns). This is done by associating a turn table T with every vertex. If a vertex u has p incoming edges and q outgoing edges, its turn table T_u takes form $p \times q$, where $T_u[i, j]$ is the cost of turning from the incoming road segment i to the outgoing segment j. In practice, the number of possible turn tables is dramatically smaller than the number of intersections. Thus, it is more memory-efficient to store turn tables only once and assign each vertex a pointer to the turn table it is associated with. Besides, CRP handles metric adjustment better than most other algorithms with preprocessing. The reason is that preprocessing can be divided into two phases. The first stage only considers the topology of the graph and is metric-independent. During this stage, the initial graph is partitioned into multiple levels of cells and arc separators are determined. Since the topology of a road networks is static, these operations can only be performed once. The second phase of preprocessing assigns costs to edges of the overlay graph, for which the actual cost function is necessary. Fortunately, this stage can be accelerated through parallelization and performed on GPUs, since the clique of every cell is built using only the internal paths.

2.1.4 Hierarchical methods

The techniques of this category exploit the hierarchical topology of road networks, since long routes tend to converge to a subset of roads consisting of highways and primary roads. One of the most efficient hierarchical algorithms is Contraction Hierarchies (CH) [GSSD08]. It implements the idea of skipping unimportant vertices via addition of shortcuts. This is done by repetitive node contraction. For a graph G, a vertex v is contracted by removing it from the network while preserving the shortest paths in G. The preservation is achieved by replacing paths (u, v, w) with shortcut edges of the form (u, w). Since it is necessary to preserve only the shortest paths, an edge (u, w) must be added only if the unique shortest path from u to w goes through v. The contraction of nodes is performed in a specific order based on a heuristic estimate of their importance. A heuristic provided in [GSSD08] is based on edge difference: the number of shortcut edges added during the contraction of v minus the number of edges incident to v. This heuristic helps the algorithm minimize the overall number of edges after preprocessing. The queries are performed using a modified version of bidirectional Dijkstra's search. Both forward and backward searches only visit nodes in ascending importance order. Both searches are bound to process the highest ranked vertex on the shortest path from origin to destination. Hence, among all of the vertices processed by both searches, the one minimizing the sum of distances to origin and destination represents the shortest path. However, both searches have to process all nodes, since the first node processed by both the forward and backward search is not necessarily on the shortest path.

2.1.5 Path extraction

Since a navigation program must not just calculate the cost of the minimum path, but provide the actual sequence of road segments to traverse, the algorithms listed above must be modified to return the minimum cost routes. Fortunately, these modifications are relatively simple to implement and cause no substantial performance decline. The methods that do not utilize shortcuts can be augmented to maintain the parent pointers for each of the vertices and update them every time a shorter route to a vertex is found. After the search is complete, traversing the parent pointers one by one from the destination vertex until the starting vertex is met gives the reversed shortest route. When considering the algorithms that use shortcuts (e.g. CRP and CH), one must bear in mind that such approach will only return the "compact" form of the route represented by the shortcut vertices traversed. In order to extract the full representation of the shortest route, one can store the entire sequences of vertices corresponding to each of the shortcut edges or run a local shortest path search algorithm for the endpoints of the shortcut to be restored.

2.2 Optimization algorithms

In order for a shortest path search algorithm to be able to tailor routes to a user's desires, it must have access to a mathematical representation of the user's preferences. These can be represented as a vector of coefficients $\alpha \in \mathbb{R}^n$. Using the coefficients and a function $c(\alpha, p) : \mathbb{R}^n_+ \times \mathbb{R}^n_+ \to \mathbb{R}_+$ which maps coefficients and an edge parameter vector $p \in \mathbb{R}^n$ to a scalar edge cost, a shortest path algorithm will be able to plan a route from any origin to any destination in accordance with the user's priorities. As it was stated earlier, a user's preferences should be computed based on a set of his preceeding commutes. Several optimization algorithms have been successfully applied to this task.

2.2.1 Linear Programming

In their research, Funke et. al. [FLS16] attempted to solve the task using a linear programming model. The cost function they have experimented on is of linear form $c(\alpha, p) := \sum_{i=1}^{n} \alpha_i p_i$. The cost function of a full path P can then be expressed as $c(\alpha, P) := \sum_{e \in P} \alpha^T p^e$, and the parameter vector of P is denoted as $c(P) = \sum_{e \in P} p^e$. In order to limit the search space of possible preference coefficients, all coefficients have been locked on the conditions of $\forall \alpha_i \in \alpha : \alpha_i \in \langle 0, 1 \rangle$ and $\sum_{i=1}^{n} \alpha_i = 1$. This assumption does not cause the loss of generalization, since any vector α which does not satisfy these conditions can be replaced with $\alpha' = \frac{\alpha}{\sum_{i=1}^{n} \alpha_i}$ without changing the respective ratios of any of the coefficients.

User preference computation from previously traversed routes is formalized using the definition of preferential feasibility: a path P from a vertex s to a vertex t is preferentially feasible if there is a preference vector α such that $\pi(s, t, \alpha) = P$, where $\pi(s, t, \alpha)$ is a route built by a shortest path algorithm in accordance with α. A set of paths $\mathcal{P}$ is preferentially feasible if there exists a preference vector α such that $\forall P \in \mathcal{P} : \pi(s^P, t^P, \alpha) = P$. As one can anticipate, in order for an optimization algorithm to be able to find a vector of coefficients α^* which accounts for all input trajectories, the set of input trajectories must be preferentially feasible.

The linear program solving the preferential feasibility uses real positive variables $\alpha_{1..n}$. The previously defined constraints these variables must satisfy can be expressed as

$$\begin{gathered} \alpha_1 + ... + \alpha_n = 1 \\ \alpha_1 \geq 0 \\ ... \\ \alpha_n \geq 0 \end{gathered} \tag{2.1}$$

The linear definition of preferential feasibility is rather straightforward. For every $P \in \mathcal{P}$, it is necessary to guarantee the shortest path algorithm is not able to find a route $\pi(s^P, t^P)$ from s^P to t^P that has smaller cost than P under the coefficient vector α. The linear constraint that implements this condition is:

$$\begin{gathered} \forall P \in \mathcal{P} : \forall \pi(s^P, t^P) : \\ \sum_{i=1}^{n} (c_i(P) - c_i(\pi(s^P, t^P)))\alpha_i \leq 0 \end{gathered} \tag{2.2}$$

Any feasible solution α^* yielded by these constraints fully satisfies $\mathcal{P}$. Furthermore, a feasible solution will be found only if $\mathcal{P}$ is preferentially feasible.

However, this approach requires the enumeration of all possible alternative paths for all $P \in \mathcal{P}$, which implies an exponential number of constraints even for a single trajectory.

This issue has been solved by dynamic constraint addition. The algorithm starts without any further constraints, attempting to find a feasible solution. When a solution is found, it is checked against all provided paths. In order to do that, Dijkstra's algorithm or any other shortest path search algorithm is run for every pair $(s^P, t^P), P \in \mathcal{P}$. If a produced solution α satisfies all paths, meaning $\forall P \in \mathcal{P} : \pi(s^P, t^P, \alpha) = P$, the algorithms stops and the desired preference vector is found. If, on the other hand, the equality does not hold for a path in the set, a new constraint of the form (2.2) is added to the linear program. After the linear program is updated, it is optimized again and the solution is checked against the input path set. The cycle continues until solution reproducing all paths has been found or the linear program is proved to be unsatisfiable, in which case $\mathcal{P}$ is preferentially infeasible. If $\mathcal{P}$ is proven to be preferentially infeasible, the feasibility of single paths $P \in \mathcal{P}$ can be checked by rerunning the algorithm with $\mathcal{P} = P$. After the infeasible paths have been detected, they can be removed from the initial set or one could to attempt to extract preferentially feasible subpaths.

One can attempt to minimize the number of distinctive coefficient values in the preference vector by introducing binary variables e_{ij} for $\forall i, j : i < j, i \in \{1, ..., n\}, j \in \{1, ..., n\}$, where $e_{ij} = 0$ if $\alpha_i = \alpha_j$ and $e_{ij} = 1$ if $\alpha_i \neq \alpha_j$. In the linear model, this relation is enforced by constraints:

$$\alpha_i - \alpha_j \leq e_{ij}$$
$$\alpha_j - \alpha_i \leq e_{ij}.$$

Then, the model can be led to equalize individual coefficients if possible by setting the objective function to:

$$\min \sum_{i,j=\{1,...,n\}, i<j} e_{ij}$$

However, turning the model into a mixed integer linear program significantly complicates solving it. Therefore, the researchers have proposed a way of excluding the integer variables through the replacement of $e_{ij} = \{0, 1\}$ by $0 \leq e_{ij} \leq 1$. Although this solution does not return an exact minimum of the distinctive preference values, it is claimed to yield near-optimal solutions due to the fact that it still attempts to minimize the sum of the absolute differences of coefficient pairs.

It is also possible that the whole set of individually feasible paths cannot be explained by a single preference vector. For example, one would probably

LP Preference Estimator

```
Input:
  input routes $\mathcal{P}$, graph $G = (V, E)$, edge parameters $p_e : E \to \mathbb{R}^n_+, e \in E$,
cost function $c(\alpha, \cdot) : \mathbb{R}^n_+ \times \mathbb{R}^n_+ \to \mathbb{R}_+; \alpha, \cdot \in \mathbb{R}^n_+$

LP ← an empty linear model                                  ▷ initialize the model
LP.set_variables($\alpha = (\alpha_1, ..., \alpha_n)$)
for $i = \{1, ..., n\}$ do
    LP.add_constraint($\alpha_i \geq 0$)
end for
LP.add_constraint($\sum_{i=1}^{n} \alpha_i = 1$)
LP.set_objective (minimize 0)
LP.solve()

all_explained ← False
while all_explained = False do                              ▷ optimize preferences
    all_explained ← True
    for $\forall P \in \mathcal{P}$ do
        $\Pi \leftarrow PathQuery(G, s^P, t^P, \alpha)$
        if $c(\alpha, P) > c(\alpha, \Pi)$ then                 ▷ preferences are not optimal
            all_explained ← False
            LP.add_constraint($\sum_{i=1}^{n} \alpha_i (p_i^P - p_i^\Pi) \leq 0$)
        end if
    end for
    if all_explained = False then
        LP.solve()
    end if
end while
return $\alpha$
```

choose the fastest route for work commutes, but would prefer a more scenic route during a vacation trip. The authors of the research addressed this scenario to. The proposed approach uses the non-modified version of the initial algorithm to find the minimum number of importance ratio vectors satisfying the whole input set. The paths are processed in an order $P_1..P_m$ and sorted into buckets based on their feasibility. First, the algorithm finds a preference vector α_1 for $\mathcal{P} = \{P_1\}$. Next, α_1 is checked against P_2. If P_2 is satisfied by α_1, it is added to the bucket with P_1. Else, the algorithm is run on $\mathcal{P} = \{P_1, P_2\}$. If a feasible solution is found, P_1 and P_2 are stored in a common bucket, and the solution is saved. Otherwise, new importance coefficients are computed for $\mathcal{P} = P_2$ and it is put to a new bucket. Every next P_i is checked on the subject of preference similarity to all of the existing buckets, and if it can't be added to any of those without feasibility violation, it is put into a new one. When all of the input paths are processed, the minimum possible number of individual groups of routes with corresponding preference vectors is produced.

For a graph $G = (V, E)$ with n parameters per edge, the basic algorithm runs in time $\mathcal{O}(|V| \log |V| + n|E|)$ per path $P \in \mathcal{P}$, which implies polynomial asymptotic complexity. An advantage of the designed technique is that it is able to incorporate any shortest path search algorithm, which gives acceleration opportunities.

2.2.2 Reference trajectories

There has been some research on navigation personalization approaches that do not use explicit preference functions in shortest path search. Dai et. al. [DYGD15] designed a method which provides personalized routes using big trajectory data arrays gathered from other drivers. The building block of the approach is an Inverted Trajectory Index (ITI) structure which for a given edge e returns the set of all available routes that contain e. Additionally, the edges are indexed based on their geographical data. The graph is partitioned into a uniform grid, and every cell of the grid contains the array of all edges inside of it. An edge is inside a cell if its starting vertex or sink vertex is inside it. Another helpful stricture is a mathematical representation of a driver's priorities. Given two parameters p_i and p_j of a path P (e.g. distance and travel time), the preference ratio with respect to parameters i and j is $pr_{i,j} = \frac{p_i}{p_j}$. For n considered road parameters, a full preference ratio vector $PR = (pr_1, ...pr_m)$ contains $m = \binom{n}{2}$ random variables, where each random variable pr_i on the interval $\langle 0, 1 \rangle$ denotes the distribution of a preference ratio. Using a set of a user's trajectories, one can calculate a set of corresponding preference ratio value vectors. Gaussian Mixture Models are employed on this data to derive a vector of random variable distributions expressing the driver's preference ratios. The Personalized Satisfaction Score Function $\mathcal{F}$ uses the preference vector to compute the degree of satisfaction of a route P for the driver:

$$\mathcal{F}(P, PR) = \sum_{i=0}^{m} \int_{r_i - \Delta}^{r_i + \Delta} pr_i(c)dc,$$

where r_i is the preference ratio of P with respect to pr_i, and $\Delta \in \mathbb{R}_+$ is a small real value used for neighborhood definition. The value returned by the function is directly proportional to the driver's content.

During a query from a vertex s to t, these data are used to extract a set of reference trajectories. The idea is that it is inefficient to consider all trajectories of varying relevance to build a route of optimal preference. Instead, the set of available trajectories is pruned to identify the most relevant ones. The first of the proposed pruning filters is called spatial filter. It extracts the set E_s of outgoing edges of s and the set E_t of incoming edges of t. For every pair $(e_s, e_t), e_s \in E_s, e_t \in E_t$, it searches for an intersection $ITI(e_s) \cap ITI(e_t)$. These intersections consist of trajectories that visited s and t and are therefore considered relevant to the query. Additionally, one can check timestamps of these trajectories to make sure s is visited before t. The resulting set of trajectories is then processed by the temporal filter, which crosses out trajectories that occurred in a time period different from that of the query. This filter is based on the observable time-dependence of traffic. Lastly, the preference filter calculates the satisfaction scores of the given trajectories and removed routes whose scores are below a predefined threshold. Since the computation of satisfaction scores requires a history of the user's previous trajectories to deduce their preference ratios, this filter can be omitted if the necessary data are not available. Using the set of reference trajectories, the algorithm builds a local reference graph $G_{ref} = (V_{ref}, E_{ref})$, where V_{ref} is the set of vertices occurring in the reference trajectories and E_{ref} is the set of edges included in ones. It is important to note that thanks to the spatial filter G_{ref} is connected and contains a path from s to t. To further guide a shortest path search algorithm, the edges of the local reference graph are weighted by applying the PageRank [ML04] algorithm to the dual graph $\overline{G}_{ref} = (\overline{V}_{ref}, \overline{E}_{ref})$, where every vertex $\overline{v} \in \overline{V}_{ref}$ corresponds to an edge in E_{ref} and every edge $\overline{e} \in \overline{E}_{ref}$ corresponds to a vertex in V_{ref}.

Chapter 3

Problem statement

In order to formally define the problem in question, it is necessary to define the setting first. As is usual in this domain and is it was stated earlier, the road network is modelled as a directed graph. Let the road network be represented as a *graph* $G = (V, E)$, where every *vertex* $v \in V$ corresponds to an intersection or an end of a road, and every *edge* $e = (u, v) \in E$ is a road segment that connects intersections u and v. A road segment can be one-way or two-way, so the edges must be directed. Hence, a one-sided path from u to v is coded as a directed edge (u, v). If the road connecting these intersections is two-sided, two edges (u, v) and (v, u) are added to the graph. The starting vertex of an edge e is denoted as s^e, and its sink is denoted as t^e. Since the thesis project is aimed to process multiple road parameters, every edge $e \in E$ is associated with a static *parameter vector* $p^e \in \mathbb{R}^n_+$ of positive real values reflecting the details of the corresponding road segment. Thus, the algorithm operates on a directed weighted graph with multiple edge weights.

A *route* or a *path* $P = (e_1, e_2, ..., e_k), k \geq 1$ in G is a sequence of edges that connect a sequence of distinct vertices, where $\forall i = \{1, ..., k-1\} : t^{e_i} = s^{e_{i+1}}$, meaning every pair of consecutive edges shares a vertex. In the context of real-world road networks it is not always the case that a driver approaching an intersection can take a turn to any of the outgoing roads, since some turns can be forbidden. However, the data on permitted turns are difficult to obtain. Therefore, it is assumed in this thesis that any vertex has an available turn between any pair of its incoming and outgoing edges. Since the input routes are expected to be in the form of sequences of vertices $(v_1, ..., v_{k+1})$, there is no need in the usual definition of a trajectory as a sequence of GPS trace coordinates. Hence, the term *trajectory* has the same meaning as the terms route and path in this thesis. The *parameter vector of a path* P can be calculated as $p^P = \sum_{e \in P} p^e$.

The priorities of a user are simulated using a vector of positive real *preference weights* $\alpha \in \langle 0,1\rangle^n$. The higher is a coefficient α_i, the higher is the importance of the associated road parameter to the driver. As it was stated in [FLS16], limiting the interval of possible values to $\langle 0,1\rangle$ causes no loss of generality, since any preference vector α which does not satisfy this condition can be replaced by the corresponding satisfactory vector $\alpha' = \frac{\alpha}{\sum_{i=1}^{n} \alpha_i}$ with equal preference ratios. In order to be able to incorporate these preferences in a route search algorithm, a *cost function* $c(\alpha, p) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ mapping the preference and edge parameter vectors to a scalar is required.

In graph theory, the *shortest path* problem is the task of computing the path P^* with minimum cost between a pair of vertices. In the context of multiple edge parameters, for some given preference vector α, the shortest path P^* is such a path that for any other path P from $s \in V$ to $t \in V$, the condition $c(\alpha, p^{P^*}) \leq c(\alpha, p^P)$ holds.

To be able to make a conclusion on how well a path accounts for the preferences of a user, a *path similarity function* has to be introduced. For two paths $P = (p_1, ..., p_k)$ and $Q = (q_1, ..., q_l)$ connecting two vertices $s, t \in G$, their similarity can be evaluated using the similarity function

$$sim(P, Q) = \frac{\sum\limits_{i=1}^{\min(k,l)} [p_i = q_i]}{\min(k, l)},$$

$$\text{where } [p_i = q_i] = \begin{cases} 1, & \text{if } p_i = q_i \\ 0, & \text{else.} \end{cases}$$

Finally, the formal definition for the preference computation task can be formulated. In order to deduce a driver's preferences, an optimization algorithm must compute the preference vector $\alpha^* = \arg\min_{\alpha \in \langle 0,1\rangle^n} \mathcal{L}(\mathcal{P}, \alpha)$ given a set of the driver's previous routes $\mathcal{P} = \{P_1, ..., P_m\}$. The *loss function* $\mathcal{L}(\mathcal{P}, \alpha)$ expresses the average similarity lack between the given trajectories $\mathcal{P}$ and the ones built by a shortest path algorithm using a set of preference coefficients α:

$$\mathcal{L}(\mathcal{P}, \alpha) = 1 - \frac{\sum\limits_{i=1}^{m} sim(P_i, SP(s^{P_i}, t^{P_i}, \alpha))}{m}, \tag{3.1}$$

where $SP(s^{P_i}, t^{P_i}, \alpha)$ is the path from $s^{P_i} \in G$ to $t^{P_i} \in G$ produced by a shortest path algorithm using the preference vector α for edge cost computation.

Chapter 4

Proposed method

As it is stated in previous chapters, the two main building blocks of the algorithm are a shortest path search algorithm and an optimization algorithm for preference computation. Hence, the overall solution can be described by defining the solutions of these subtasks first and describing their interaction afterwards.

4.1 Shortest path search

It is stated in the related work section chapter that various graph preprocessing methods can significantly decreasing the time requirements of routing algorithms. Since the optimization function $\mathcal{L}(\mathcal{P}, \alpha)$ operates by comparing the input trajectories with the built ones, the running time of the full project will be influenced greatly by the speed of the routing algorithm. Hence, a preprocessing phase is desirable, though not necessary. However, a preprocessing approach incorporating the cost function would have to be repeated after every preference adjustment, hence a method that is independent of preferences has been chosen.

During the preprocessing phase, a k-Path Cover [FNS14, FS15] of the given graph is built. Given a graph $G = (V, E)$ and a natural value $k \in \mathbb{N}$, a k-Path Cover (k-PC) of G is a subset of vertices $C \subseteq V$ such that for every simple path $P = (v_1, ..., v_k)$ in G it holds $\exists i = \{1, ..., k\} : v_i \in C$. There is also a modification of the structure which is built on the basis of shortest path only.

It is called k-Shortest-Path-Cover (k-SPC) and consists of a subset of vertices of G such that for every shortest path there is at least one cover node in it. This modification yields an even smaller subset of the initial graph than standard k-PC and hence better running times. Unfortunately, it is metric-dependent and becomes ineffective during continuous preference adjustments. Therefore, the standard k-PC approach has been used. Intuitively, a smaller number of cover nodes is expected to yield better performance, but the problem of minimum k-PC is proved to be of APX-hardness (subsuming NP-hardness), therefore the project does not aim to find the minimum k-PC for a given graph.

The basic method of k-PC construction consists in building for every vertex all paths of size k that start at the vertex and retrieving a feasible cover using a greedy approach. Unfortunately, this approach is highly ineffective on road networks due to their large size and hence high time and storage requirements. For that reason, Funke et. al. have proposed a more efficient approach [FNS14], the pseudocode of which is laid out below.

```
k-PC construction
Input:
  graph $G = (V, E)$

$C \leftarrow V$
for $v_i \in V, i = \{1, ..., |V|\}$ do
    $\mathcal{P}_i \leftarrow$ the set of all paths starting at $v_i$ such that $\forall P \in \mathcal{P}_i : P \cap C = \{v_i\}$
    if $\exists P \in \mathcal{P}_i : |P| = k$ then
        continue
    end if
    for $\forall P \in \mathcal{P}_i$ do
        $Pr \leftarrow$ the longest path ending at $v_i$ such that $Pr \cap (C \cup P) = \{v_i\}$
        if $|Pr \cup P| = k$ then
            continue
        else
            $C \leftarrow C \setminus \{v_i\}$
        end if
    end for
end for
return $C$
```

The core idea of the method is to include all vertices in the cover and then prune the unnecessary ones. The nodes are examined one by one in a predefined order. To decide whether a vertex should be kept in C or not one must attempt to find a path of size k that contains no other vertices from the cover. If no such path has been found, the vertex should be removed from the cover. The path searching operation is divided into two stages. First, the algorithm builds and stores all the paths that start at the vertex and contain no other cover vertices. This can be implemented through a procedure resembling forward depth-first-search, but whenever another cover vertex is visited, the current path is rebuilt and saved. Since it is possible that a k-sized path can start at the vertex, the sizes of all the outgoing paths should be checked before continuation. If no paths of satisfactory length are found, the algorithm attempts to augment them using the incoming paths, which are built through backward depth-first-search. Since the paths must be simple, the augmentation of every outgoing path is performed separately. At every vertex iteration, the maximum depth of the search tree is k, since C is a valid cover before the vertex is processed, which can be easily shown via a proof by induction. A nested version of covers is also possible, though not used in this thesis.

After the vertices of the overlay graph are determined, the edges connecting them are produced. An edge between two cover vertices u and v is added to the overlay graph if there is a simple path from u to v that contains no other vertices from C. To build the set of overlay edges, a breadth-first-search is run from every cover vertex. Whenever the search hits another cover node, an edge between these vertices is built, and the corresponding search tree branch is pruned. Obviously, this approach can produce more than one path between a pair of vertices. Therefore, for every found path the parameter vector and the vertex sequence of the path are computed and added to the set of parameter vectors and node sequences associated with the corresponding overlay edge. Due to the storage of all path parameter vectors the preprocessing phase is independent of preference coefficients and can only be run once. The described procedure of building overlay edges returns exponential number of possible paths and corresponding parameter vectors, the majority of which cannot become the shortest irrespective of the preferences. For that reason, several pruning techniques are used. The set S of parameter vectors associated with a single edge is pruned using three rules: domination pruning, convex pruning, and triangle pruning.

Domination pruning is the most simple and straightforward pruning strategy, while being possibly the most efficient ones. It effectively removes all the excessively long paths. A parameter vector $p \in \mathbb{R}^n$ dominates vector $p' \in \mathbb{R}^n$ if $\forall i = \{1, ..., n\} : v_i \leq v'_i$ and $\exists i = \{1, ..., n\} : v_i < v'_i$. A naive algorithm of performing domination pruning consists in comparing every vector $p \in S$ to all other vertices and pruning out the vectors that are dominated by it. However, the asymptotic complexity of this approach is $\mathcal{O}(n|S|^2)$. Alternatively, one could select a subset of n promising vectors and prune other vectors by these, which would yield $\mathcal{O}(n|S|)$. A good rule for the selection of promising vectors is to pick for each parameter the vector that has the minimum respective value.

Convex pruning is based on the lemma provided and proved in [FS15]. This lemma states that a vector $p \in \mathbb{R}^n$ can be pruned from a set of vectors $S \subset \mathbb{R}^n$ if a convex combination p' of at most n other vectors from S dominates p. The subset $p^1, ..., p^n$ of promising vectors that have been chosen during domination pruning can also be used as a control set in convex pruning. The procedure of checking if a parameter vector p should be pruned from S consists essentially in searching a feasible solution for a system of linear inequalities:

$$\begin{gathered}
\gamma_1 p_1^1 + ... + \gamma_n p_n^1 \leq p_1 \\
\gamma_1 p_1^2 + ... + \gamma_n p_n^2 \leq p_2 \\
... \\
\gamma_1 p_1^n + ... + \gamma_n p_n^n \leq p_n \\
\gamma_1 + ... + \gamma_n = 1
\end{gathered}$$

Unlike previous methods of pruning, triangle pruning utilizes the structure of the overlay graph instead of processing the edges individually. In order to further prune parameter vectors of an edge, the technique examines all triangles induced by it. The set of parameter vectors induced by these pairs is computed and used to prune the vectors from the original edge through any of the pruning methods described above. Triangle pruning can even modify the structure of the graph: if all of the parameter vectors of an edge are removed, the edge can be deleted from the graph. After the pruning is complete, the preprocessing phase ends.

The shortest path query is performed using a modified version of Dijkstra's algorithm, which can be separated into two stages: connection of the origin and destination vertices to the overlay graph, and the shortest path search. The origin is connected to the overlay graph through Dijkstra's algorithm, which operates on the initial graph, begins with the starting vertex, and operates until all nodes in the priority queue are cover nodes. The destination node is connected in the same way, but the graph search traverses edges in the reversed direction. The second phase of the query is conducted on the overlay graph. Dijkstra's algorithm starts with all cover nodes to which the origin is connected. For every inspected edge, it computes the actual costs of the parameter vectors and selects the one which returns the minimum value for the given preferences, and the corresponding vertex sequence is labelled for the reconstruction of the actual path. The search continues until all of the cover nodes connected to the destination vertex are settled. The overall path is chosen so as to minimize the cost sum of the path segments from origin to the destination's adjacent cover node and from the cover node to the destination itself.

In [FS15], the performance of the k-PC method has been compared to those of CRP and CH. The experiments have shown that out of the three, k-PC yields the least acceleration compared over the standard Dijkstra's algorithm. However, there is a number of reasons why it has been chosen for this project. For example, there are fewer articles researching the extent of efficiency of k-PC on road networks than there are of CRP and CH. Additionally, k-PC appears to be more universal than other algorithms. The preprocessing phase of the algorithm is built purely on the structure of the graph in question. In contrast to CRP, which incorporates the coordinates of nodes for cell partitioning, it requires no information about the geographic locations of the vertices. CH is the most efficient on large scale tasks: it works well with long-range routes since those usually go through important roads like highways. Unfortunately, this advantage of CH is of no significant importance if for example it is necessary to build a route for a cyclist. On the other hand, the road hierarchy is not an inherent part of k-PC, and it can work equally well in small and large scales. Moreover, k-PC can be adjusted to work with large graphs and routes if necessary by building multiple nested levels of the overlay graph. And lastly, as it is stated in [FS15], k-PC is the most lightweight algorithm out of the three — the auxiliary data generated by its preprocessing phase have the smallest memory requirements. This is a huge advantage if the program is deployed on a less powerful device such as a smartphone, which is an ordinary case nowadays since the majority of drivers and commuters use navigation systems on their portable devices, often in offline mode.

4.2 Preference optimization

Although preference extraction via linear programming appears to be an effective optimization approach, it has only been tested on artificial data. The condition (2.2) of linear program limits the scope of possible cost functions to linear ones. Unfortunately, there is no guarantee that routes of a real driver can be explained by a linear function, so an optimization approach capable of incorporating cost functions of any form would be preferable. Therefore, a modified version of random or stochastic coordinate descent (SCD) [RT11] has been chosen as an optimization algorithm for this project.

The standard random coordinate descent is an optimization algorithm that works on smooth convex functions. It has been later generalized to work on composite functions such as sums of smooth convex and convex block-separable functions. The smoothness of a function $f(x)$ implies the coordinate-wise Lipschitz continuity of its gradient with constants $L_1, ..., L_n$. This condition can be written as $\forall x \in \mathbb{R}^n, h \in \mathbb{R} : |\nabla_i f(x + he_i) - \nabla_i f(x)| \leq L_i|h|$. The pseudocode of the full algorithm is laid out below.

```
Random Coordinate Descent
  Input:
      starting point $x^0 \in \mathbb{R}^n$

  $x \leftarrow x^0$
  while termination criterion not met do
      $i \leftarrow$ random value from the discrete uniform distribution over $\{1, ..., n\}$
      $x_i \leftarrow x_i - \frac{1}{L_i} \nabla f_i(x)$
  end while
  return $x$
```

Since the objective function (3.1) is non-differentiable, the standard stochastic coordinate descent algorithm cannot be applied to it. Instead, a version of SCD has been proposed and tested specifically on driver preference optimization in [DGG+15]. Since the optimization process cannot be guided by the gradient of the loss function, three different operations have been used to improve its convergence: local search, perturbation, and specialized sampling. The algorithm starts with an initial solution and attempts to improve through local search. Local search is the main learning procedure of the algorithm, systematically exploring and examining the search space. It operates in several iterations and terminates when several consecutive iterations fail to improve the solution. During every round, a random permutation of the parameter examination order is generated. Then for every parameter in the given order, a better coefficient value is attempted to be found. Since the objective function cannot be differentiated, a set of random candidate coefficients is generated with a bias towards its current value. Then every candidate value is evaluated. During the evaluation, the candidate value replaces the true one in the coefficient vector and the loss of the modified solution is computed. If the candidate solution yields a better loss than the initial one, it is accepted as the current best coefficient vector. If a solution decreasing the objective function has been found, the local search iteration is considered successful. On the other hand, if all the parameters have been processed and no improvement of the objective function has been achieved, the iteration of the local search terminates with failure.

Local search is able to gradually minimize the loss function until it reaches a local minimum. At that point, it is bound to fail since it is only able to accept improving solutions. Intuitively, the objective function has many local minima, some of which take the form of flat plateaus. These are caused by the fact that many coefficient combinations can yield identical shortest routes. When such a plateau is met, an adjustment of a single parameter coefficient cannot improve the solution, and perturbation is performed to escape it. Analogically to local search, perturbation operates in several iterations on randomly ordered parameters. Its core idea is in a simultaneous alteration of several coefficients, and, in contrast to local search, it accepts non-improving solutions. For every parameter p_i and its corresponding weight α_i, a set of candidate values is generated randomly. All of the candidates are evaluated on the loss function, and the minimum $\alpha_i^{\min}$ and maximum $\alpha_i^{\max}$ weight candidates yielding the same loss are saved. These are then used as limits for the algorithm to generate new weight candidates at random in the interval $\langle\alpha_i^{\min}, \alpha_i^{\max}\rangle$ until a candidate is found that does not worsen the objective function. If a non-improving candidate is found, it is accepted as the new solution and perturbation proceeds to the next parameter. Alternatively, if an improving candidate is found, it is saved and perturbation terminates with success.

If perturbation fails to decrease the loss, the algorithm is assumed to be trapped in a deep local minimum, and specialized sampling is performed to make a drastic move in the search space. In order to shift the focus to another area of the search space, it evaluates how well the current solution fits individual input trajectories. Based on the matching of the input trajectories to the artificially generated ones, all of the input trajectories are assigned importance weights: the lower is the matching ratio of a route, the higher is the weight it is assigned. This is done in order to make the algorithm focus on the least explained routes. After the trajectories are weighted, the algorithm attempts to maximize the weighted sum of their similarity scores.

The full optimization algorithm works as follows. The maximum number of consecutive iterations is set for each of the three operations (i.e. ML, MP, MS respectively). The value of the loss function on the initial coefficient vector is computed. After that, local search starts. It is performed repeatedly until ML of consecutive failed iterations occur. When local search fails to improve the solution further, perturbation starts. It is run for at most MP rounds. If during one of its iterations it manages to further decrease the objective function, the algorithm returns to local search and sets counters for all operations to zero. Otherwise, specialized sampling is performed at most MS times. Similarly, if one of its iterations is successful, the algorithm returns to the beginning, and if not, optimization terminates and the current best solution is returned.

4.3 Overall approach

The full algorithm starts with the preprocessing phase of k-PC. After the overlay graph is constructed and the excess edges are pruned, the optimization is ready to begin. In order to evaluate the current loss at a single state, the algorithm runs k-PC queries with current preference weights for the origin-destination pairs of every input route. When the optimization process is finished, the program can use the computed preferences to plan personalized routes for any desired origin-destination pair.

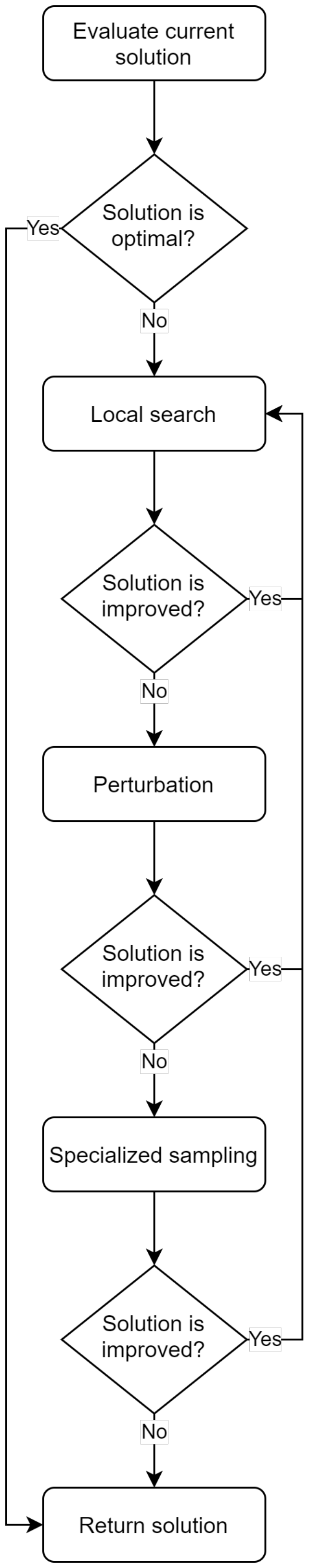

Figure 4.1: SCD flowchart

Chapter 5

Experimental evaluation

The project has been implemented in C++ and compiled with Visual C++ 2019. The experiments have been conducted on a PC running Windows 10. The CPU is Intel Core-i7 7700k with 4 cores, 8 threads, 4.2 GHz frequency, 4×32 KB L1 cache, 4×256 L2 cache, and 8 MB L3 cache. The size of 2666-DDR4 RAM is 32 GB. Parallelization has been performed on the CPU and used during overlay edge pruning, local search and perturbation operations using OpenMP. Since convex pruning requires a linear program solver, Gurobi has been incorporated. The graph used in the project is a segment of Prague's road network consisting of 235970 vertices and 633658 edges. 8 road parameters have been considered: distance, elevation gain, elevation loss, lane number, maximum allowed speed, bicycle infrastructure, road type, and surface quality. The maximum iteration numbers for SCD procedures are $ML = 5$, $MP = 8$, $MS = 8$.

5.1 Preprocessing

As it was stated earlier, the preprocessing phase of k-PC extracts cover vertices in such a manner that every simple path of size k in the graph contains at least one cover vertex. There are two parameters to be specified in the algorithm. First of all, the vertex examination order. This choice influences the resulting number of nodes in the cover graph. In [FNS14], different vertex ordering rules have been tested, including the ascending and descending vertex IDs, number of incoming and outgoing edges, etc. The resulting difference in overlay graph size, although varying, is not dramatic.

Therefore, the vertices have been ordered in the ascending ID order in this project. The second parameter to be specified is the value of k. Intuitively, the increase in this value decreases the number of cover vertices, increasing the time requirements in return. The table 5.1 presents data on the produced cover graphs for different values of k: number of cover vertices $|V^C|$, relative cover size $|V^C|/|V|$, number of overlay edges before pruning, number of overlay edges after pruning, percentage of pruned edges, and the overlay graph construction time in seconds. The tested k values are $2^i, i = \{1, .., 5\}$. Higher values have not been tested due to large running time requirements.

k	$\|V^C\|$	$\|V^C\|/\|V\|$	$\|E^C\|$	$\|E^C\|$ pruned	pruning %	time
2	133040	56.38 %	566369	546940	3.43 %	37
4	84750	35.92 %	633667	542686	14.38 %	27
8	54909	23.27 %	1021645	624644	38.86 %	29
16	38675	16.39 %	3639494	863488	76.27 %	92
32	29408	12.46 %	84867600	1546274	98.18 %	3549

Table 5.1: Overlay graph C parameters for varying k values

As the table shows, a significant increase of k after 16 yields a relatively small size adjustment of the cover vertex number, while dramatically increasing the cover edge number and hence time and memory requirements. Interestingly, the preprocessing phase runs faster for $k = 4$ than for $k = 2$. A possible reason for that is a communications bottleneck. For $k = 8$, the vertex number is decreased by 4 while maintaining the edge number on the same level. The vertex number decrease implies the decrease in the number of iterations of a shortest path search, and only requires 30 seconds of preprocessing without GPU usage. Most road networks developers operate on are of significantly bigger sizes, containing millions of vertices, and the relative graph compression given by the method is likely to increase even further with the size of the graph. For further experiments, k has been set to 16, since it appears to be the optimal value based on the gain/requirements ratio.

k	domination pruning	convex pruning	triangle pruning
2	99.75	0.17	0.08
4	95.66	0.64	3.7
8	90.32	1.36	8.32
16	91.92	2.03	6.05
32	97.97	1.04	0.99

Table 5.2: Pruning percentage by different techniques

Apart from that, it is also important to analyze the contribution of individual pruning methods. The table 5.2 provides the information about the portions of edges pruned by each method for varying values of k. It is important to note that pruning is performed in the order: domination pruning, convex pruning, triangle pruning. The reason for that is domination pruning requires no additional procedures such as linear program solvers and is straightforward and simple in implementation. On the other hand, triangle pruning requires multiple times more operations than other methods due to the fact that for every edge it has to examine all pairs of adjacent edges that form triangles. As can be seen on the table, in all of the examined cases, domination pruning on its own accounts for more than 90% of pruned edges. The method was expected to prune a significant part of the cover edges due to the fact that for every pair of adjacent cover vertices, the edge building algorithm saves all possible simple paths between them, including the winding ones that are excessively and needlessly long. However, the fact that domination pruning is capable of decreasing the cover edge number by a factor of 10 all by itself is surprising. This observation hints that the preprocessing stage of k-PC can be accelerated with no significant slowdown of its queries by only performing domination pruning.

5.2 Preference optimization

The optimization algorithm has been tested both on artificially generated data and real-world trajectories. The aim of experiments on artificial data is to test the learning capabilities of the method and its requirements. Further, experiments on paths generated by real users should give the answer to the question of sufficiency of the cost function and parameters in consideration for simulation of the reasoning of a real person when selecting a route.

5.2.1 Artificial data

The first series of experiments has been conducted in a controlled environment. The idea is to generate randomly a vector of preferences and use it in a shortest route search algorithm to build a set of preferentially guided routes. These routes can then be sent to the optimization algorithm for it to attempt to recreate them by extracting the generated preferences. This setup aims to evaluate the learning capabilities of the approach and observe the extent to which the original preferences can be reconstructed. The advantage of experiments on artificial data is that while the road qualities a real driver is concerned with during route selection are unknown and may be out of the scope of this project, the artificial data are produced using exactly the named parameters. Moreover, when generating artificial data one can be confident the routes can be explained by the chosen cost function and a vector of perfectly fitting preferences (the perfect solution) actually exists.

Another advantage is that while real-world data are hard to acquire, namely due to privacy concerns, the artificial data can be generated in any desirable quantity, hence, training and testing route sets of varying sizes can be used for convergence evaluation.

A set of m artificial traces can be generated as follows. A preference vector is generated either randomly or manually to guide the route generation. Then, m vertex pairs are picked randomly from the graph as origin-destination pairs for future routes. For every such pair, a shortest path search algorithm (in this implementation — k-PC query procedure) is applied with the preference vector to return the route from the origin to the destination that has the minimum cost under the given preferences. 8 road parameters have been processed in this project: distance, elevation gain, elevation loss, lane number, maximum speed, bicycle infrastructure, road type, and surface quality. The distance of a road segment is the direct distance between start and end vertices calculated using the projected latitude and longitude coordinates. Elevation gain of a road segment (u, v) is the vertical distance between u and v, calculated as $max(0, a(v) - a(u))$, where $a(\cdot)$ is the altitude of the vertex $\cdot$. The reason the elevation gain cannot be negative is that elevation loss is also considered and calculated as $max(0, a(u) - a(v))$. Lane number is the expression of the width of the road segment in lanes which is provided in the graph in the form of a natural number. Maximum speed is also provided in the graph, but is unknown for some segments. In this case, the maximum speed for the road segments without available maximum speed is set to some average value (e.g. 45 km/h). Bicycle infrastructure, road type and surface quality parameters have some predefined values which have been mapped to natural numbers to be used in the cost function. The table 5.3 provides the information about the possible values of the parameters.

For the experiments, randomly generated preference vectors have been used to build training sets of varying sizes and test sets consisting of 500 routes. Evaluating the loss on a test set helps estimate the number of training trajectories necessary to replicate the preferences of an abstract user with a desired precision. The cost function used in these experiments is a linear function $c(\alpha, p) := \sum_{i=1}^{n} \alpha_i p_i$. Furthermore, the road parameters have been normalized. Since, for example, the distance parameter generally has higher average value than the number of lanes, it is bound to have a significantly higher influence on the final cost of the road segment for equal coefficient intervals. Therefore, the normalization should balance out the influence of the parameters themselves during cost calculation, thus making the generated routes more diverse.

As can be seen in the figure 5.1, an increase of the size of the training set decreases the testing loss as expected. 64 training trajectories seem

parameter	possible values
distance	$\mathbb{R}_+$
elevation gain	$\mathbb{R}_+$
elevation loss	$\mathbb{R}_+$
lane number	$\mathbb{N}$
maximum speed	$\mathbb{N}$

bicycle infrastructure	track = 1
	zone = 2
	lane = 3
	sharrow = 4
	none = 5

road type	primary = 1
	secondary = 2
	tertiary = 3
	service = 4
	residential = 5
	cycleway = 6
	offroad = 7
	footway = 8
	crossing = 9
	steps = 10
	unknown = 5

surface quality	excellent = 1
	good = 2
	intermediate = 3
	bad = 4
	horrible = 5
	impassable = 6
	unknown = 4

Table 5.3: Road parameters and their possible values

sufficient for the algorithm to reconstruct the target preferences for the 8 parameters in question with 95% accuracy. In fact, the majority of the loss optimization steps has been performed by local search, an example of the optimization process being figure 5.2. Hence, at least for the linear cost function, the maximum consecutive number of perturbation and specialized sampling iterations can be decreased without loss of the efficiency while decreasing the expected execution time. Since the pairs of vertices for route generation have been picked randomly without the consideration of their respective distances, the average size of the generated paths is relatively high and exceeds 200 vertices. The k-PC queries have taken at average less than 50 milliseconds while standard Dijkstra's algorithm runs for approximately 74 milliseconds per query. A closer examination of the query process shows that

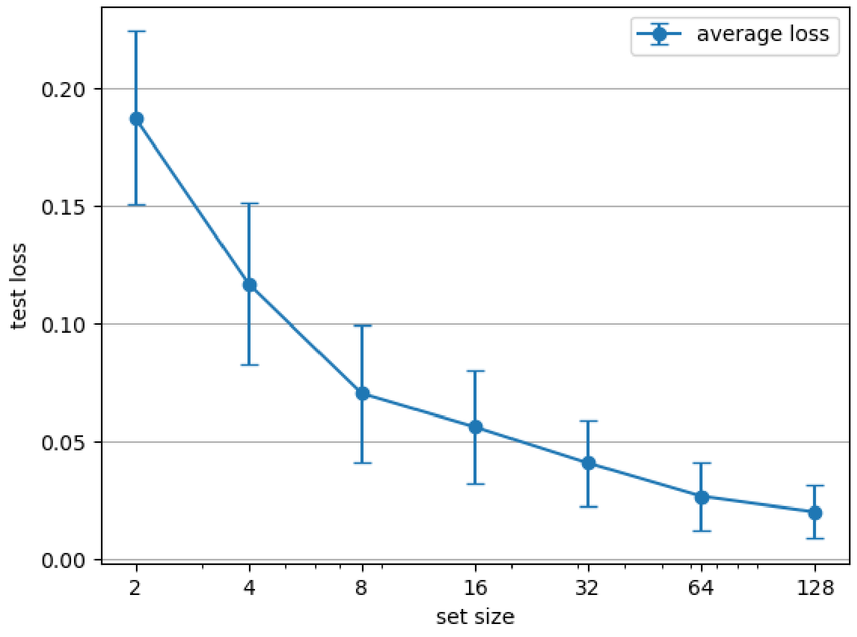

Figure 5.1: Average loss for different training set sizes

during a search on the overlay graph, more than half of the running time is spent on edge cost calculation, since for every overlay edge the algorithm has to evaluate all of the possible parameter combinations and pick the one which yields the minimum overall cost under current preferences. On the other hand, a k-PC query processes several times less vertices than standard Dijkstra's algorithm in every iteration. This hints that the relative acceleration provided by k-PC preprocessing is probable to increase for larger graphs. On the other hand, since k-PC queries spend a lot of time on cost calculation, they are likely to accelerate even further during route searches with known preferences. After the preference coefficients have been extracted, all the edges in the graph can be processed relatively fast to calculate their final minimum costs. Thus, k-PC queries will no longer have to spend the majority of the time evaluating the edges.

5.2.2 Real-world data

After the algorithm has been tested on artificial data, experiments on routes of real users have been conducted. Unfortunately, the personal trajectories are not easy to obtain mainly due to privacy concerns. Nevertheless, sets of 20 trajectories of 6 cyclers have been provided to experiment on. Experi-

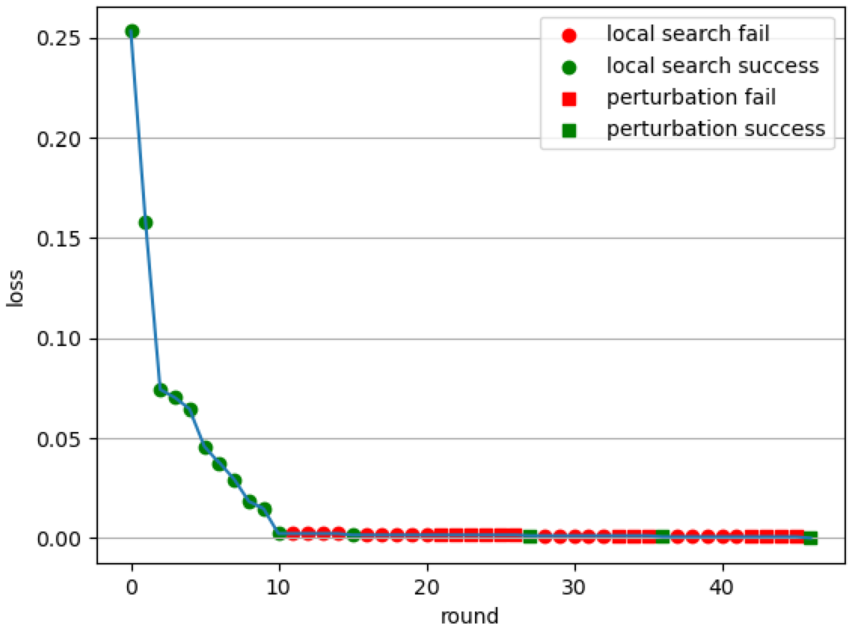

Figure 5.2: An exemplary optimization progression at $k = 32$

ments have shown varying degrees of convergence with the average loss being approximately 50%. Due to such an unsatisfactory result, the routes have been examined individually. Optimization on single routes showed their final loss values are scattered dramatically despite being produced by one cyclist. Some of the routes could be perfectly explained by considering only the distance parameter, others could not be reproduced by optimizing on various combinations of the 8 listed parameters. To test the preferential feasibility of individual routes as it is defined in the Linear Programming section of the Related Work chapter, the linear programming algorithm described in [FLS16] has been implemented. The algorithm showed that only a fraction of real routes are in fact preferentially feasible and can be reproduced by a linear function. Therefore, several possible modifications of the cost function have been tested, for example

$$c(\alpha, p) = p_d(\alpha_d + \sum_{i \in \{1,\ldots,n\} \setminus d} \alpha_i p_i), \tag{5.1}$$

where p_d is the distance parameter of a road segment and α_d is its corresponding coefficient. Unfortunately, none of the tested cost functions managed to decrease the average loss below 40%.

There are multiple possible explanations to the poor convergence on real trajectories. First, the cyclists whose trajectories have been provided may consider during route selection a parameter that has not been addressed in this project, for instance how scenic, crowded or ”green” (i. e. the amount of vegetation) a road is. On the other hand, the preferences of a user can change based on the time of the day or the momentary goal of theirs. When considering the route of a work commute, one would intuitively choose the fastest path, the possible important parameters being distance, maximum speed, surface quality. In contrast, a recreational walk can be guided by scenery, crowdedness, noise level, or greenery. Unfortunately, this assumption could not be tested since the provided routes had no timestamps and were not categorized by their goals. Another possible reason is that a route of an actual cyclist (or a driver) cannot be reproduced by a linear function. Unfortunately, no alternative cost functions were found in related research articles. Namely, in [DGG$^+$15] a non-linear function incorporating turn costs among other parameters has been used with success, but it has not been described due to being proprietary. Moreover, not only different preferences, but even varying function forms may be necessary to build routes in different time periods. Thus, even though the implemented algorithm is capable of learning preference coefficients, further research on the topic of cost functions and parameters to be considered is required.

Chapter 6

Conclusion

The thesis addresses the topic of personalized navigation in road networks. Namely, the aim of the project was to solve the task of preference extraction from an exemplary set of routes.

The theoretical part consists of a research of various shortest path search algorithms and their application to road networks. Multiple preprocessing techniques aimed to accelerate the route search have been examined. Besides that, several optimization methods and their efficiency in the computation of navigational preferences have been considered and researched. Chapter 3 contains a comprehensive formal definition of the preferential optimization of planned routes. A lightweight yet flexible method for the optimization of preferences and consequent route planning has been described in Chapter 4.

The practical part of the thesis is devoted to the implementation of the proposed algorithm and the verification of its efficiency. The project has been implemented in C++ and tested both on artificially generated and real-world data. Experiments on synthetic routes showed that the method in question is indeed capable of making accurate conclusions on the likings of a notional user and incorporating them during the creation of subsequent paths. Tests conducted on a sample of trajectories of actual cyclists revealed that the parameters and cost functions in consideration are not enough to reproduce their preferences with sufficient quality, thus indicating the direction for further improvement.

The thesis has revealed multiple possible topics for further research. Most importantly, a comprehensive study of non-linear cost functions and road parameters satisfactory for the accurate simulation of a real user's reasoning is required. The varying convergence ratios of individual routes indicate that the consideration of time periods may also be required. Another task to be addressed is the optimization of the actual algorithm's efficiency, in particular through the adjustment of the hyperparameters and interior procedures such as pruning methods, which is of substantial importance since the algorithms of the kind are expected to operate mainly on portable devices.

Bibliography

[BDG$^+$15] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck, *Route planning in transportation networks*, CoRR **abs/1504.05140** (2015).

[BEL58] RICHARD BELLMAN, *On a routing problem*, Quarterly of Applied Mathematics **16** (1958), no. 1, 87–90.

[DGG$^+$15] Daniel Delling, Andrew V. Goldberg, Moises Goldszmidt, John Krumm, Kunal Talwar, and Renato F. Werneck, *Navigation made personal: Inferring driving preferences from gps traces*, Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems (New York, NY, USA), SIGSPATIAL '15, Association for Computing Machinery, 2015.

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck, *Customizable route planning*, Experimental Algorithms (Berlin, Heidelberg) (Panos M. Pardalos and Steffen Rebennack, eds.), Springer Berlin Heidelberg, 2011, pp. 376–387.

[DGPW15] Daniel Delling, Andrew Goldberg, Thomas Pajor, and Renato Werneck, *Customizable route planning in road networks*, Transportation Science **51** (2015), 150522062514007.

[DGRW11] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck, *Graph partitioning with natural cuts*, In IPDPS. IEEE Computer Society, 2011.

[Dij59] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numer. Math. **1** (1959), no. 1, 269–271.

[DNKF14] Kenny Daniel, Alex Nash, Sven Koenig, and Ariel Felner, *Theta*: Any-angle path planning on grids*, J. Artif. Intell. Res. (JAIR) **39** (2014).

[DYGD15] J. Dai, B. Yang, C. Guo, and Z. Ding, *Personalized route recommendation using big trajectory data*, 2015 IEEE 31st International Conference on Data Engineering, 2015, pp. 543–554.

[EG08] David Eppstein and Michael T. Goodrich, *Studying (non-planar) road networks through an algorithmic lens*, Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (New York, NY, USA), GIS '08, Association for Computing Machinery, 2008.

[Flo62] Robert W. Floyd, *Algorithm 97: Shortest path*, Commun. ACM **5** (1962), no. 6, 345.

[FLS16] Stefan Funke, Sören Laue, and Sabine Storandt, *Deducing individual driving preferences for user-aware navigation*, Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (New York, NY, USA), SIGSPACIAL '16, Association for Computing Machinery, 2016.

[FNS14] Stefan Funke, André Nusser, and Sabine Storandt, *On k-path covers and their applications*, Proc. VLDB Endow. **7** (2014), no. 10, 893–902.

[FS15] Stefan Funke and Sabine Storandt, *Personalized route planning in road networks*, Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems (New York, NY, USA), SIGSPATIAL '15, Association for Computing Machinery, 2015.

[GH05] Andrew V. Goldberg and Chris Harrelson, *Computing the shortest path: A search meets graph theory*, Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (USA), SODA '05, Society for Industrial and Applied Mathematics, 2005, p. 156–165.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling, *Contraction hierarchies: Faster and simpler hierarchical routing in road networks*, 05 2008, pp. 319–333.

[HKMS09] Moritz Hilger, Ekkehard Köhler, Rolf Möhring, and Heiko Schilling, *Fast point-to-point shortest path computations with arc-flags*, vol. 74, pp. 41–72, 07 2009.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems Science and Cybernetics **4** (1968), no. 2, 100–107.

[Hou94] Richard Hough, *Captain James Cook*, Holder and Stroughton, 1994.

[LT77] Richard J. Lipton and Robert E. Tarjan, *A separator theorem for planar graphs*, 1977.

[ML04] Carl D. Meyer and Amy N. Langville, *Deeper inside pagerank*, Internet Mathematics **1** (2004), no. 3.

[RT11] Peter Richtárik and Martin Takáč, *Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function*, 2011.

[T.G23] Rupert T.Gould, *The Marine Chronometer, its history and development*, J. D. Potter, 1923.

[Vli78] Dirck [Van Vliet], *Improved shortest path algorithms for transport networks*, Transportation Research **12** (1978), no. 1, 7 – 20.