Multimodal Journey Planning and Assignment in Public Transportation Networks

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Tobias Zündorf

Tag der mündlichen Prüfung: 4. Dezember 2020

Erste Referentin: Prof. Dr. Dorothea Wagner
Zweiter Referent: Prof. Dr. Matthias Müller-Hannemann

Acknowledgements

First of all, I would like to thank Dorothea Wagner for inviting me to join her group. During my time as a PhD student, I benefited greatly from her experience, the support she provided, and the friendly and cooperative work environment she created. I thank the German Research Foundation (DFG) for funding my research as a part of the research unit FOR 2083. I would also like to thank all the members of the research group for their wonderful collaboration and the productive workshops we had together. In particular, I would like to thank Matthias Müller-Hannemann who not only supported me as part of the research unit but also agreed to be the second referee for my thesis.

I want to give a special thanks to all my co-authors, Moritz Baum, Lars Briem, Valentin Buchhold, Sebastian Buck, Daniel Delling, Julian Dibbelt, Andreas Gemsa, Nicolai Mallig, Thomas Pajor, Jonas Sauer, Peter Vortisch, and Dorothea Wagner, who supported me with many insightful discussions and their expertise. I am no less grateful for the administrative and technical support provided by the current and former staff, Lilian Beckert, Isabelle Junge, Ralf Kölmel, Laurette Lauffer, and Tanja Wehrmann. Without their help, I would have had significantly less time for research and this thesis. Furthermore, I would like to thank Daniel Delling, Julian Dibbelt, and Thomas Pajor for the opportunity to join them at Apple for an incredibly insightful and inspiring internship.

Also, I want to thank all my co-workers at the Institute of Theoretical Informatics who always created a joyful atmosphere. I cannot think of a better place to do research. I would especially like to thank my fellow "route planners", Moritz Baum, Valentin Buchhold, Julian Dibbelt, Jonas Sauer, Sascha Witt, and Tim Zeitz, with whom I had many fruitful discussions about novel ideas for journey planning algorithms.

Moreover, I thank my office mate Valentine Buchhold for all the help he offered, not only with algorithmic issues but also with other important matters, such as optimizing notation or typography. I also fondly remember our regular table football games and thank Guido Brückner and Jonas Sauer for always responding to our game challenges.

Last but not least, I want to thank my family for supporting me throughout the years and for encouraging me to pursue my interest in computer science. I especially thank Albert and Sabine for proofreading my thesis. Finally, I want to thank my wonderful wife Monica for her encouragement and love throughout my PhD. Without her support, the completion of this thesis would certainly have been more stressful.

Abstract

Timetable information systems (such as the DB Navigator) and navigation systems for road networks (such as Google Maps) have become an integral part of everyday life. The widespread use of such journey planning applications has been enabled by algorithmic developments of recent decades. Considering road networks, a shortest path across all of Europe can be computed in well below one millisecond. Similarly, timetable information systems are able to find optimal journeys throughout Germany in less than 50 milliseconds. However, if these two network types are combined into a single, multimodal network, where the mode of transportation can be switched arbitrarily, then computing optimal journeys requires significantly more time.

In this thesis, we consider several variants of journey planning problems in multimodal transportation networks. In contrast to many other works, we do not only consider the passengers point of view, but also the point of view of the public transport operator. Algorithms that compute optimal journeys between two given locations in the multimodal network are particularly relevant for a passenger. Public transportation operators, on the other hand, are often interested in the overall passenger flow through a network. The computation of passenger movements from a given list of demands lies at the core of *traffic assignment* problems. The result of the traffic assignment can then, for example, be used to evaluate the utilization of trains or other vehicles in the network.

Single Source Single Target Journey Planning. The first part of this work focuses on the problem of finding optimal journeys between a single source and target location within a multimodal network. Computing such a journey on a country sized

network can take seconds, even if the multimodal network only consists of public transit and one additional mode of transportation (e.g., walking). Faster journey planning algorithms that can consider both, walking and public transit, of course already exist. However, all of them achieve their efficiency by limiting the maximal distance that can be traveled by walking. Since this can be seen as an approximation of the problem with unlimited walking, the question arises, to what extent journeys can benefit from unlimited walking.

In order to answer this question we develop a first multimodal profile algorithm. That is, an algorithm that computes for every possible departure time a set of optimal journeys from a source location to a target location. Our algorithm is based on the idea of iteratively reducing the size of the time interval for which the profile is unknown, which we do with the help of existing journey planning algorithms that require a fixed departure time. Through careful algorithm engineering, this approach is capable of computing profiles, which comprise a whole day, in a few seconds.

We use our novel profile algorithm to assess the impact of the permitted walking distance on the overall travel time of optimal journeys. To this end, we evaluate and compare profiles for several hundreds of source-target-pairs in both scenarios, with and without limited walking. As a result, we find that allowing unlimited walking can significantly reduce the travel of optimal journeys, which reinforces the need for multimodal journey planning algorithms. However, we also observe that traveling long distances by walking in between rides with public transit vehicles is rarely required. Instead, walking is mainly needed to reach the first public transit stop from the source location and to reach the target location from the last public transit stop.

Based on this observation we develop a novel preprocessing technique, which we call ULTRA (UnLimited TRAnsfers), that enables fast multimodal journey planning. The main idea of our approach is to process walking between public transport stops differently from walking towards the target or from the source. Since walking between stops is only rarely required, it is feasible to precompute all pairs of stops where it occurs as part of an optimal journey. On the other hand, possible paths for walking from the source or towards the target can be explored efficiently at query time. In an extensive experimental evaluation we show that this approach outperforms any existing multimodal journey planning algorithm. Furthermore, we demonstrate that our approach is not only capable of handling walking as additional transportation mode, but also any other non-schedule based mode of transportation, such as cycling or using a car.

Finally, we acknowledge that up to this point we only solved bi-modal journey planning problems. We change this by addressing a more complex scenario, where public transportation is combined with unlimited walking and bike sharing. In order to solve the journey planning problem in this truly multimodal scenario, we present two possible approaches and develop an additional speed-up technique that signifi-

cantly reduces the computational overhead of handling bike sharing. In combination with ULTRA this yields the fastest known multimodal journey planning algorithm.

Public Transit Traffic Assignments. The second part of this work focuses on solving traffic assignment problems for transportation networks that include public transit. These problems are not concerned with finding an optimal journey for a single pair of source and target locations. Instead, the objective is to predict the behavior of passengers for millions of origin-destination-pairs.

Although this problem has many similarities with traditional journey planning, algorithmic advances that have been made for the single source single target problem have not yet been applied to the assignment problem. Thus, we initially focus on accelerating the computation of assignments for networks that solely consist of public transit. Analyzing the problem and its structure reveals that the Connection Scan Algorithm (CSA), which was originally proposed for journey planning in pure public transit networks, is particularly well suited for solving the assignment problem. Adapting this algorithm in a way that exploits the special structure of the assignment problem, yields a new algorithm that can compute assignments in less than one minute, on problem instances where previous approaches took about half an hour.

Finally, we explore to what extent the approaches and results from the first part of the work can be applied to the assignment problem and our CSA-based approach. To this end, we demonstrate how the basic idea of ULTRA and its preprocessing can be adopted to the assignment problem. In combination with novel approaches for handling the multitude of origin and destination locations that are integral to the assignment problem, we are able to develop a fast, multimodal assignment algorithm.

Contents

Abstract iii

1 Introduction 1
- 1.1 Main Contributions . . . 5
- 1.2 Thesis Outline . . . 7

2 Literature Overview 11
- 2.1 Journey Planning . . . 11
  - 2.1.1 Algorithms for Road Networks . . . 11
  - 2.1.2 Public Transit Algorithms . . . 14
  - 2.1.3 Multimodal Techniques . . . 17
- 2.2 Traffic Assignments . . . 19

3 Fundamentals 23
- 3.1 Network Models . . . 23
  - 3.1.1 Connection-Based Model . . . 24
  - 3.1.2 Route-Based Model . . . 25
  - 3.1.3 Transfer Graphs . . . 26
  - 3.1.4 Journeys and Profiles . . . 27
  - 3.1.5 Minimum Change Times and Departure Buffer Times . . . 31
- 3.2 Journey Planning and Assignment Problems . . . 33
- 3.3 Algorithms . . . 35
  - 3.3.1 Shortest Paths in Non-Timetable Networks . . . 35
  - 3.3.2 Journey Planning in Timetable Networks . . . 40

4 Benchmark Datasets **47**
4.1 Data Sources . . . 47
4.2 Additional Preparations and Sanitizing . . . 50
4.3 Transitively Closed Instances . . . 54

5 Multimodal Profiles **59**
5.1 Profile Algorithm . . . 61
5.1.1 Earliest Arrival Profiles . . . 61
5.1.2 Pareto Profiles . . . 63
5.2 Experiments . . . 64
5.2.1 Performance Experiments . . . 67
5.2.2 Travel Time Comparison . . . 69
5.3 Final Remarks . . . 77

6 UnLimited TRAnsfer Shortcuts **79**
6.1 Shortcut Computation . . . 81
6.1.1 Overview . . . 81
6.1.2 Implementation Details . . . 82
6.1.3 Proof of Correctness . . . 86
6.2 Query Algorithms . . . 86
6.2.1 Basic Query Algorithm. . . . 87
6.2.2 Running Time Optimizations. . . . 88
6.3 Integration with Trip-Based Routing . . . 89
6.3.1 Trip-Based Preprocessing . . . 89
6.3.2 Improved Query . . . 92
6.4 Experiments . . . 96
6.4.1 Preprocessing . . . 96
6.4.2 Queries . . . 103
6.5 Final Remarks . . . 110

7 Bike Sharing **113**
7.1 Preliminaries . . . 114
7.2 Models for the Bike Sharing Problem . . . 115
7.2.1 Operator-Dependent Model . . . 115
7.2.2 Operator-Expanded Network . . . 117
7.3 Operator Pruning . . . 118
7.4 Extended Scenarios . . . 120
7.5 Experiments . . . 121
7.5.1 Preprocessing . . . 122
7.5.2 Queries . . . 123
7.6 Final Remarks . . . 127

8 Assignments **129**
8.1 Preliminaries . . . 130
8.1.1 Perceived Arrival Time . . . 131
8.1.2 Decision Models . . . 134
8.1.3 Problem Statement . . . 135
8.2 Public Transit Assignment . . . 136
8.2.1 Perceived Arrival Time Computation . . . 137
8.2.2 Passenger Assignment . . . 140
8.2.3 Cycle Elimination . . . 141
8.2.4 Implementation Details . . . 142
8.3 Multimodal Assignment . . . 143
8.3.1 Departure Buffer Times . . . 144
8.3.2 ULTRA-Based Passenger Assignment . . . 144
8.3.3 Improved Passenger Grouping . . . 147
8.4 Experiments . . . 148
8.4.1 Public Transit Assignment . . . 149
8.4.2 Multimodal Assignment . . . 151
8.5 Final Remarks . . . 156

9 Conclusion **159**
9.1 Summary . . . 159
9.2 Outlook . . . 161

Bibliography **163**

List of Figures **179**

List of Tables **181**

List of Acronyms **183**

List of Symbols **187**

A Curriculum Vitæ **195**

B List of Publications **199**

C Deutsche Zusammenfassung **203**

1 Introduction

Journey planning systems are widely available in our modern world. Many cars have built-in navigation systems for road networks. Web services like Google Maps or Bing Maps provide free journey planning and support multiple modes of transportation, such as using a car, walking, or public transit. Journey planning for public transportation networks is also available in the form of specialized timetable information systems, such as DB Navigator or the various applications provided by local public transportation operators.

These systems are based on fast journey planning algorithms that have been developed throughout the last decades [Bas+16]. Many of these algorithms achieve their practical efficiency by augmenting the network data with additional information that only has to be computed once and can then be used for faster query answering. A prominent example that utilizes this approach are Contraction Hierarchies (CHs), a speed-up technique for journey planning in road networks [GSSD08]. Through a relatively lightweight preprocessing step (the road network of Europe can be processed in a few minutes) CHs achieve query times of about 0.1 milliseconds. This corresponds to a speed-up of about 10 000, when using Dijkstra's algorithm as a baseline. Even faster queries are possible if more extensive preprocessing is used. Currently the fastest algorithm for journey planning in road networks is Hub Labeling (HL), which achieves query times of less than one microsecond on the Europe network [ADGW11].

Unfortunately, techniques that perform well on road networks are often not suitable for public transportation networks [BDW11]. Therefore, specialized algorithms have been developed for public transit, which allow for significantly faster query times than Dijkstra's algorithm on a graph-based representation of the network.

However, the speed-ups achieved for public transit networks are much smaller than the aforementioned speed-ups for road networks. One of the fastest public transit journey planning algorithms, that does not require preprocessing, is RAPTOR (Round-bAsed Public Transit Optimized Router) [DPW15a], which achieves a speed-up of 5.3 compared to a variant of Dijkstra's algorithm that is suitable for public transit networks. An additional speed-up of 4.5, compared to RAPTOR, is achieved by Trip-Based Public Transit Routing [Wit15], which requires a few minutes of preprocessing. In order to achieve even greater speed-up factors, extensive preprocessing phases are required. The fastest currently known algorithm for journey planning in public transit networks is Transfer Patterns [Bas+10], which is 136 times faster than Trip-Based Routing. However, Transfer Patterns requires several days of preprocessing in order to accomplish this speed-up.[1]

Combining both, public transit and road networks, results in a multimodal transportation network. Research considering such networks has yielded even fewer results than research on journey planning in public transit networks. Most successful approaches for journey planning in multimodal networks are based on combining a journey planning algorithm for road networks with a public transit algorithm. Notable examples for this are MCR (multiModal multiCriteria RAPTOR) [Del+13], which combines Dijkstra searches with RAPTOR, and HLRAPTOR [PV19], which combines Hub Labeling with RAPTOR. However, while Hub Labeling on its own yields significant speed-ups on road networks, its combination with RAPTOR is only two to three times faster than MCR.

Multimodal Journey Planning. In this thesis, we address several *multimodal* journey planning problems, i.e., journey planning problems in networks that comprise multiple modes of transportation. In this context, a mode of transportation refers to a means of transportation that can be used by a traveler, such as riding a bicycle, walking, or using public transit. However, we do not explicitly distinguish different forms of public transit (e.g., trains, buses, etc.), since, as long as they follow a fixed schedule, they are equivalent from an algorithmic standpoint. Based on this, we consider multimodal journey planning to be the task of finding optimal (fast) journeys that utilize multiple transportation modes. In particular, we are interested in algorithms that compute journeys where the mode of transport is changed en route. This is in contrast to many available systems that can compute an optimal unimodal journey for each of the available modes of transportation.

Within this work, we distinguish between two general classes of problems related to multimodal journey planning. First, we consider problems as they occur from a traveler's point of view. That is, we are given a source and a target location and want to recommend one or several good journeys from the source to the target.

[1] Note that each speed-up factor reported in this paragraph is based on a different network, since the presented algorithms were not evaluated on a common network.

We study variants of this problem with fixed and with flexible departure times, and with various combinations of available transportation modes. The second class of problems arises from the planning and analysis of transportation networks, where the overall passenger flow is of interest. In this case, the objective is to estimate, for a large number of passengers, which journeys will be used, such that the utilization of public transit vehicles can be predicted. Problems of this kind are also known as traffic assignment problems.

Difficulty of Combined Networks. Before we introduce our approaches for solving the aforementioned multimodal journey planning problems, we want to give an intuition why journey planning is more difficult in multimodal and in public transit networks than it is in road networks. A major reason for this are structural differences between the network types. Many speed-up techniques for road networks achieve their performance by exploiting the inherent hierarchy of the network. However, public transit networks are generally much less hierarchical than road networks [Bas09]. Moreover, public transportation networks are significantly more dense than road networks, which has a negative effect on the performance of the journey planning algorithms [BDGM09].

In addition to these structural differences, public transport networks are generally time-dependent, i.e., travel times change over time. Of course, road networks are also to some extent time-dependent. However, an optimal journey in a road network changes only slightly, if the departure time of the journey is shifted [SWZ20c]. In contrast, public transit journeys can change drastically with a shift of the departure time, for example, if a journey is no longer possible because a train already departed. Hence, many geographically different journeys can be optimal during the course of a day, which impedes preprocessing-based speed-up approaches. Still, specialized algorithms have been developed, which allow for relatively efficient journey computation in public transit networks.

To some extent, journey planning becomes even more difficult when public transit networks are augmented by road networks. The reason for this is that the resulting multimodal scenario requires a single algorithm that can handle both networks types. However, algorithms that are optimized for road networks usually perform poorly on public transit networks, and specialized public transit algorithms are often not suitable for road networks. Therefore, approaches for multimodal journey commonly use a combination of two algorithms, which handle the two parts of the network (e.g., MCR or HLRAPTOR). But, approaches that combine two algorithms come with their own disadvantages, such as an enormous preprocessing overhead (e.g., HLRAPTOR) or the fact that parts of the network have to be processed multiple times (e.g., the scanning of the road network in MCR).

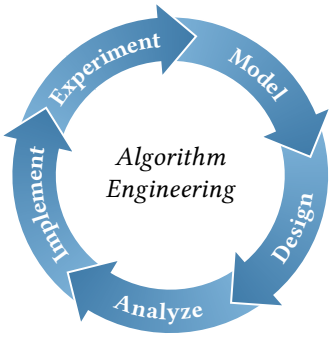


Figure 1.1: A visualization of the algorithm engineering methodology as introduced by [San09] and [MS10]. Algorithm engineering is a cyclic process consisting of modeling the problem, followed by the design and analysis of an algorithm, which is subsequently implemented and experimentally evaluated, before the next cycle begins.

Methodology. This work is focused on developing multimodal journey planning algorithms that achieve good practical performance. Accordingly, we will not evaluate our algorithms in terms of asymptotic worst case running time, but via experiments using real world data. While we are not interested in provable runtime bounds, we nevertheless want to guarantee that the journeys computed by our algorithms are optimal[2].

We develop algorithms that comply with these requirements by following the algorithm engineering methodology [San09, MS10], which establishes a cyclic process for the development of algorithms, as shown in Figure 1.1. The development of an algorithm usually starts with choosing an appropriate model for the problem, which in our case also includes decisions about the network representation. This step is followed by the design and analysis of an algorithm that solves the problem. Many theoretical works on algorithms finish at this point. However, the algorithm engineering methodology continues with the implementation and experimental evaluation of the algorithm. The evaluation of the algorithm on realistic data often yields valuable insights about the problem structure, which can be used in another iteration of the algorithm engineering cycle, in order to further improve the algorithm.

In practice, not all iterations of the algorithm engineering cycle have the same impact. Some iterations may only yield negligible progress, while other iterations entail significant results. Moreover, it is not necessary to perform all steps of the cycle within each iteration. In order to keep this work concise, we will not report every single iteration that was conducted. Instead, we focus on the most important results and findings, i.e., the algorithms we found to perform well in praxis.

An example for large-scale iterations of the algorithm engineering cycle are chapters 5, 6, and 7. The experimental evaluation of the algorithm presented in Chapter 5 leads to structural insights into the problem structure, which sparks the development of an efficient preprocessing technique for multimodal journey planning in Chapter 6. This novel preprocessing technique in turn enables us to consider a problem with even more transportation modes in Chapter 7.

[2] The notion of *optimal* is formally defined in Section 3.

1.1 Main Contributions

The major contributions of this thesis comprise structural insights into multimodal journeys, several efficient algorithms for multimodal journey planning, and fast assignment algorithms for public transit. We introduce all algorithms in detail in chapters 5 through 8, where we also experimentally evaluate their performance. Prior to this, we present a brief summary of the most important results in this section.

Multimodal Profile Algorithm. Our first contribution is an efficient multimodal profile algorithm. That is, a journey planning algorithm that does not require a concrete departure time as input, but a whole interval of possible departure times. The algorithm then has to compute optimal journeys for all departure times within the specified interval. While several profile algorithms have been proposed for time dependent road networks as well as public transit networks, the problem has (to the best of our knowledge) not yet been studied for multimodal networks.

We show how a journey planning algorithm for specific departure times can be extended to a profile algorithm. Using the MCR algorithm as a concrete example for this, we demonstrate with an experimental evaluation that our approach indeed yields a practical profile algorithm. For the multimodal network of Switzerland our approach can compute a full 24 hour profile in about 30 seconds even for long range queries.

Multimodal Journey Structure. Given the multimodal profile algorithm we developed, it becomes practical to analyze multimodal travel times between a large number of stops, throughout a day. Thus, we use our profile algorithm to compare optimal travel times in a network where walking is possible between all pairs of stops and networks where this is not the case. As a result we find, that the ability to walk besides using public transit can reduce travel times significantly. However, most importantly we observe that walking is predominantly required between the source location and the first stop and between the last stop and the target location of a journey. In contrast, it is only rarely necessary to walk from one stop to another.

Fast Multimodal Queries through ULTRA. We exploit the aforementioned property of multimodal journeys (i.e., the fact that walking between two stops is only rarely necessary) within ULTRA (UnLimited TRAnsfers), our novel preprocessing technique for fast multimodal journey planning. The basic idea of ULTRA is to determine all pairs of stops (and their distance), between which walking is required as part of an optimal journey. This information can then be used by most existing public transit journey planning algorithms (with minor modifications) in order to compute optimal multimodal journeys.

We prove the validity of our approach with an extensive experimental evaluation. In detail, we demonstrate that the preprocessing phase of ULTRA is well suited for parallelization, such that the preprocessing takes only a few minutes for smaller networks like London or Switzerland and a few hours for larger networks like Germany. Furthermore, we show that the results of the preprocessing enable highly efficient journey planning in multimodal networks. Optimal journeys can be computed in only 5 milliseconds on smaller networks and in less than 100 milliseconds on the Germany network. Overall, our algorithm is about one order of magnitude faster than the best existing multimodal journey planning algorithm (MCR). The running time of our approach is even comparable to state-of-the-art journey planning algorithms for public transit networks. Moreover, our experiments show that ULTRA cannot only handle walking between stops but also other modes of transportation like cycling or taking a taxi.

Journey Planning with Bike Sharing. The excellent performance of ULTRA allows us to consider more complex scenarios. Therefore, we examine networks that feature bike sharing, where bicycles can be rented at bike sharing stations and have later to be returned at bike sharing stations of the same operator. We present two basic approaches for modeling networks with bike sharing and we show how optimal journeys can be computed in both models. Furthermore, we adapt ULTRA to this extended scenario and also develop a new preprocessing step that can be combined with ULTRA to enable even faster queries.

An experimental evaluation of our approach demonstrates that the basic idea of ULTRA also works in this more complex scenario. Furthermore, we show that the special preprocessing step, which we developed for bike sharing, can reduce the preprocessing time of ULTRA by more than an order of magnitude. Moreover, we can show that our query algorithms are also more than an order of magnitude faster than our baseline. Overall, using our algorithms, optimal journeys can be computed in about 20 milliseconds for the smaller networks (Stuttgart, London, and Switzerland), while queries on the much larger network of Germany take about 650 millisecond on average.

Highly Efficient Traffic Assignment. All contributions mentioned thus far consider queries between single pairs of source and target locations. Another important class of problems related to journey planning are assignment problems, which require the prediction of journeys for millions of passengers. However, standard speed-up techniques for journey planning have thus far not been used to improve assignment algorithms. We fill this gap and demonstrate how the Connection Scan Algorithm (CSA) can be adapted to the traffic assignment scenario. As a result we

present the CSA-Based Assignment (CBA) algorithm, which can compute an assignment of 1.2 million passengers in about 34 seconds, a task that takes half an hour using state-of-the-art commercial applications.

Multimodal Traffic Assignment. As a final contribution we extend the assignment problem to multimodal networks. We achieve this by combing our preprocessing approach for multimodal networks (ULTRA) with our efficient assignment algorithm (CBA). The main challenge in implementing this combination is the fact that ULTRA is inherently an algorithm for computing journeys between a single pair of source and target locations while CBA gains its efficiency from computing journeys for multiple targets simultaneously. We overcome this problem through careful engineering of the data structures used to represent partial journeys. As a result we obtain the first efficient multimodal traffic assignment algorithm, which is capable of assigning 1.2 million passengers in about 17 seconds.

1.2 Thesis Outline

We continue by outlining the structure of the remainder of this thesis. While doing so, we especially point out that parts of this work have previously been published in conference proceedings and technical reports [Bri+17, WZ17, Bau+19a, Bau+19b, SWZ19a, SWZ19b, SWZ20a, SWZ20b].

Chapter 2 provides an overview of the literature related to this work. In particular, we present the state-of-the-art regarding algorithm for journey planning in road networks, public transit networks, and multimodal networks, as well as models and approaches for computing traffic assignments.

Chapter 3 introduces fundamental concepts and notations used throughout this thesis (Section 3.1). Building upon this, in Section 3.2 we give precise definitions for the various journey planning problems considered in this work. The algorithms we develop in order to solve these problems are partially based on existing algorithms, which we briefly explain in Section 3.3.

Chapter 4 introduces the real world data sets, which we use to evaluate all algorithms presented in this work. We state the source of all data sets in Section 4.1. Afterwards, Section 4.2 describes the necessary preprocessing steps to obtain reliable and consistent multimodal transportation networks from the raw data. In order to be able to compare our results with existing public transit algorithms, we also compile networks that are suitable for these algorithms in Section 4.3. This chapter is partially based on joint work with Dorothea Wagner [WZ17].

Chapter 5 considers multimodal profiles. Section 5.1 describes how journey planning algorithms, which assume a fixed departure time, can be extended to solve profile queries. We continue with an experimental evaluation of our approach in Section 5.2. In this context we analyze the efficiency of our approach, as well as the structure and travel time of the computed journeys, in comparison to journeys in pure public transit networks. The chapter is concluded in Section 5.3 with some final remarks on the discovered properties of multimodal journeys. This chapter is based on joint work with Dorothea Wagner [WZ17].

Chapter 6 presents ULTRA, a preprocessing technique for networks with unlimited transfers. The chapter begins with the introduction of shortcuts, which are precomputed in order to reduce the complexity of unlimited transfers, in Section 6.1. Afterwards, in Section 6.2, we describe how these shortcuts can be used, in combination with preexisting public transit algorithms, to compute multimodal journeys. We continue by discussing a special case, the combination of our shortcuts with the Trip-Based query algorithm, in Section 6.3. This combination yields the fastest known algorithm for multimodal journey planning, which we demonstrate with an extensive experimental evaluation in Section 6.4. We conclude the chapter with some final remarks in Section 6.5. This chapter is based on joint work with Moritz Baum, Valentin Buchhold, Jonas Sauer, and Dorothea Wagner [Bau+19a, Bau+19b, SWZ20b].

Chapter 7 considers journey planning in multimodal networks that comprise public transit, walking, and bike sharing. We start with a formal definition of this extended scenario in Section 7.1. Afterwards, we present two possible approaches for modeling journey planning algorithms that can solve this problem in Section 7.2. In this context we also describe how ULTRA can be used to compute journeys with bike sharing. We continue by introducing an additional speed-up technique that is tailored to the special properties of networks with bike sharing in Section 7.3. The chapter is concluded with a discussion of extended scenarios and an experimental evaluation of the algorithms for bike sharing in Sections 7.4 and 7.5. This chapter is based on joint work with Jonas Sauer and Dorothea Wagner [SWZ20a].

Chapter 8 considers assignment problems, which are highly relevant for planning and analyzing public transit networks. We start with introducing some additional concepts and notation, which we will use throughout the chapter, in Section 8.1. Additionally, we present a detailed definition of the assignment problem within this section. We continue with introducing a novel assignment algorithms for public transit algorithm in Section 8.2. Afterwards, in Section 8.3,

we demonstrate how this algorithm can be combined with ULTRA, which yields the first efficient assignment algorithm for multimodal networks. We conclude this chapter with an experimental evaluation of our two assignment algorithms in Section 8.4. This chapter is based on joint work with Lars Briem, Sebastian Buck, Holger Ebhart, Nicolai Mallig, Jonas Sauer, Ben Strasser, Peter Vortisch, and Dorothea Wagner [Bri+17, SWZ19a, SWZ19b].

Chapter 9 concludes this work by summarizing its key results and its contribution to the state-of-the-art in journey planning. Finally, we discuss possibilities for extending the algorithms presented in this work to some interesting open problems.

2 Literature Overview

In this chapter we present an overview of existing work related to multimodal journey planning and traffic assignments. We focus especially on techniques and approaches that are relevant for this thesis, either because they serve as basis for the algorithm developed in this work or because they solve similar problems, which enables us to compare results. A much broader overview of state-of-the-art journey planning algorithms is presented in [Bas+16] and a detailed introduction into public transit traffic assignments is given in [GN16].

2.1 Journey Planning

We begin our literature overview by discussing journey planning algorithms, where we distinguish three types of algorithms. First, we discuss algorithms for road networks, followed by techniques for public transit, and finally, we consider the multimodal scenario. For algorithms designed for pure road networks or public transit we additionally provide a brief assessment of their applicability in multimodal scenarios.

2.1.1 Algorithms for Road Networks

Journey planning algorithms for road networks have seen remarkable advances over the past decades. Typically, journey planning problems in road networks are formulated as classical shortest path problems in a graph that represents the network. Using this approach the problem can for example be solved with the well-known

algorithm of Dijkstra [Dij59]. However, for larger networks the running time of Dijkstra's algorithm is too slow for many applications. Thus, many algorithms have been developed that aim at improving the time required to compute a shortest path. One of the simplest speed-up techniques for Dijkstra's algorithm is a bidirectional search, where a forward search from the source and a reverse search from the target are performed simultaneously [Dan63, Nic66]. However, even such a simple approach cannot be directly transferred to public transport networks, since performing a reverse search would require the arrival time at the target, which is not known.

Preprocessing-Based Speed-up Techniques. More sophisticated speed-up techniques commonly use a two phase (or even a three phase) approach, where early phases compute additional information that is later used to reduce the running time of the last phase, i.e., the actual query. Many algorithms that utilize a preprocessing phase can furthermore be categorized as *goal-directed* or *hierarchical*. In goal-directed techniques, the data computed during the preprocessing is used to guide the search towards the target. In contrast, hierarchical techniques aim at skipping unimportant parts of the network, which is often achieved through the usage of shortcuts. In what follows, we list notable examples for both approaches.

Goal-Directed Techniques. An early example for a goal-directed preprocessing technique is Arc-Flags [Lau04, Möh+06, HKMS09]. During the preprocessing for Arc-Flags, the road graph is partitioned into regions. Afterwards a flag is computed for every pair of directed edge and region, which indicates if the edge is part of an optimal journey that ends in the region. These flags are then used at query time to speed-up Dijkstra's algorithm by restricting it to scan only edges that are flagged for the region of the target.

Another example of a goal-directed technique is ALT (A*, Landmarks, Triangle inequality) [GH05, EP13], which is a variant of the A* algorithm [HNR68] that does not require any additional input data besides the road graph. The A* algorithm uses lower bounds on the distance to the target to direct a Dijkstra search towards the target. In particular, it changes the order in which vertices are settled by Dijkstra's algorithm, favoring vertices that are expected to be located on a short path from the source to the target (based on the lower bound for the distance to the target). In ALT the lower bounds required by the A* algorithm are obtained by utilizing the triangle inequality and distances from and to a few landmark vertices. For this, ALT requires a preprocessing phase, during which a small set of landmark vertices is selected and the distances between all vertices and the landmarks are computed.

While it is possible to adapt goal-directed techniques for networks containing public transportation, the resulting speed-up is quite limited [BDGM09]. The reason

for this is the different structure of public transit networks, which leads to imprecise lower bounds when using ALT and requires that almost all edges are flagged when using Arc-Flags.

Hierarchical Techniques. A prominent example for a hierarchical speed-up technique is Contraction Hierarchies (CHs) [GSSD08, GSSV12], which is based on vertex contractions. A vertex is contracted by removing it from the road graph and replacing it with *shortcut* edges, such that the shortest path distance between all other vertices remain unchanged. The main idea of CH is then to iteratively contract vertices until the whole graph is contracted. Afterwards, the original graph and the shortcuts are used within the highly efficient CH query, which is a bidirectional variant of Dijkstra's algorithm that only considers edges (and shortcuts) that lead from vertices that were contracted earlier to vertices that were contracted later.

On road networks the CH approach yields excellent performance, with a query that is about 10 000 times faster than Dijkstra's algorithm, while the required shortcuts can be computed in a few minutes. However, as with many other speed-up techniques for road networks, CHs are not particularly well suited for networks containing public transit. One reason for this is that the CH query is inherently bidirectional, which is a problem as this requires knowledge of the arrival time. Nonetheless, variants of CH for time dependent networks [BDSV09, BGNS10] and public transportation networks [Gei10] have been developed. Instead of using a bidirectional query, these approaches unpack the search space of the backward search, such that it can be explored within the forward search. However, these approaches still yield significant longer preprocessing times and query times than CH for road networks.

A possible solution for problems that are too complex to be solved with CH alone are *core*-based approaches [Bau+10b, Del+13, DPW15b, Bau+15]. The basic idea of these approaches is to contract only some vertices and to keep the most difficult parts of the network uncontracted. This uncontracted part of the network is then called the core, which is generally much smaller than the original network and can be handled by a specialized query algorithm. Core-based approaches have been successfully applied to networks containing public transit by keeping parts of the network related to public transit in the core (MCR [Del+13], UCCH [DPW15b]).

CHs have also been adapted to some important extended scenarios. One such scenario are one-to-many queries, where multiple targets are specified and the distance to all of them has to be computed. Since CH uses a bidirectional query, this would normally require one query per target. To overcome this, Bucket-CHs have been proposed [Kno+07, GSSD08, GSSV12], where the backward search space of the targets is processed during preprocessing, such that a single forward search is sufficient to compute the distance to all targets. Another import journey planning problem arises in dynamic scenario, where travel time can change (e.g., through congestion).

In order to solve this problem with CHs the preprocessing has do be repeated whenever the travel times change, which is quite costly. This shortcoming is solved with CCH (Customizable Contraction Hierarchies) [DSW16], which extends the two phase approach of CHs to a three phase approach. This first phase of CCH determines the order in which vertices are contracted without considering travel times. Afterwards, a much faster second preprocessing phase can compute the travel times for all shortcuts, once the actual travel times of the edges are known. Finally, CHs have also been adapted for other means of transport, such as bicycles, where the height difference of the journey should be optimized, in addition to the travel time [Sto12].

Besides CH, another important hierarchical speed-up technique is Multilevel Dijkstra (MLD). Interestingly, this technique was originally proposed for public transit networks [SWZ02], and was only later adopted for road networks [JP02, Del+09, HSW09]. However, MLD is much more relevant for journey planning in road networks, where it achieves much higher speed-ups than in public transportation networks. Independent of the network type, the basic idea of MLD is to partition the network into regions, which can be skipped through the use of shortcuts, if neither the source nor the target is located within the region. Similar to CCH, MLD can be extended to a three phase approach for handling dynamic scenarios [DGPW17].

Other notable examples for hierarchical speed-up techniques are Transit Node Routing (TNR) and Hub Labeling (HL). TNR is based on the observation that long distance journeys pass through a small number of important vertices (called transit nodes), e.g., slip roads. Thus, precomputing the distances between all pairs of such vertices can be used to speed-up query algorithms, which compute journeys in three hops (source to first important vertex, first to second important vertex, and second important vertex to target) [BFSS07, BFM09]. Using CHs enables an efficient implementation of this idea [ALS13]. Furthermore, this approach can also be applied to public transit networks [DPW09a, AW12], where it is not nearly as efficient as for road networks. HL is based on a very similar idea, as it precomputes distances between all vertices and some important vertices (called hubs) [ADGW11, ADGW12, DGSW14]. given these distances, the query algorithm finds optimal journeys by trying all possible two hop journeys between the source and the target, which is extremely efficient. This approach was also applied to public transit networks [DDPW15].

2.1.2 Public Transit Algorithms

We continue with an overview of journey planning algorithms that have been developed for public transportation networks. Many of these algorithms also consider the possibility of walking between neighboring stops and can therefore in theory handle at least two modes of transportation (e.g., public transit and walking). However, in

this work we only consider an algorithm to be multimodal, if it has been shown that the algorithm can handle networks where walking is possible between most stops and not only between a limited subset of stops.

For journey planning in public transit networks two general classes of algorithms can be differentiated. The first class models the problem as a classical shortest path problem in a graph representing the network. This approach enables in theory the usage of techniques developed for road networks (or the usage of shortest path algorithms in general). However, in practice speed-up techniques developed for road networks often perform poorly on public transportation networks [BDW11]. An alternative approach are algorithms that were developed to work directly on the timetable and aim at exploiting its structure. In the following we list notable examples for both approaches.

Graph-Based Approaches. Research on journey planning for public transportation networks has yielded a wide range of algorithms that utilize a graph-based model of the timetable. In this section we present some of the most important results for graph-based approaches. A more extensive overview of existing techniques is given in [MSWZ07]. In general, graph-based approaches can be subdivided into *time-dependent* and *time-expanded* techniques. Time-dependent techniques use vertices to model locations and represent all options for traveling between two locations with a single edge, where the travel time of the edge is a function of the departure time [BJ04, PSWZ04]. In contrast, time-expanded techniques unroll the time dimension of the network and represent every departure or arrival of a public transit vehicle at some location with a unique vertex. Thus every vertex represents a time and a location and two vertices are connected if traveling between them is possible (either by waiting or by using a public transit connection) [PS98].

An advantage of the time-expanded approach is that it can easily be extended to more complex scenarios, such as optimizing ticket costs as an additional criterion [MS07]. The efficiency of the time-expanded model can be further improved by reducing the number of edges needed in the graph and by using goal directed techniques, such as Arc-Flags or ALT [DPW09b]. Handling dynamic updates of the timetable is also possible with a time-expanded approach and only requires a few microseconds if the time-expanded graph has been optimized for updates [Cio+17]. Applying HL to a time-expanded representation of timetable yields one of the fastest known journey planning algorithms for public transportation networks, which is called Public Transit Labeling (PTL) [DDPW15]. However, a drawback of this approach is, that it is only viable if the number of stops between which walking is possible is small. Furthermore, PTL requires several days of preprocessing, even for relatively small networks, such as the metropolitan area of London.

Nonetheless, handling dynamic updates of the timetable is possible with PTL by using a fast algorithm that updates the preprocessed data if some public transit vehicles are delayed [DK19]. Another speed-up technique that builds upon the time-expanded network representation is Transfer Patterns [Bas+10]. It is based on the observation that many optimal journeys use the same stops for transferring between public transit vehicles. Precomputing the patterns of these transfers enables extremely fast journey planning at the cost of a very time consuming preprocessing phase. An improved version of this approach, called Scalable Transfer Patterns, can be used to reduce the time consumption and the space consumption of the preprocessing phase [BHS16].

The time-dependent approach is generally more space efficient than the time-expanded approach and for simple scenarios (e.g., optimizing only the travel time) it also leads to faster algorithms than the time-expanded approach [PSWZ08, BDW11]. In [DMS08] it is shown how multiple criteria can be optimized in the time-dependent representation of the network by using a multi-criteria version of Dijkstra's algorithm. Careful engineering of the time-dependent graph can be used to enable dynamic updates of the timetable data and faster query algorithms [BGM10]. Finally, the time-dependent approach has been combined with HL to speed-up queries [Wan+15]. However, this approach is not as efficient as PTL for time-expanded graphs.

Timetable-based Approaches. A huge disadvantage of modeling the timetable as a graph is that parts of the timetable's structure are lost (e.g., the existence of bus lines or train lines that are repeated periodically). Thus, many algorithms have been developed that operate on an explicit representation of the timetable and aim at exploiting its structure. One example for such an approach is RAPTOR (Round-bAsed Public Transit Optimized Router), which exploits the existence of lines (which are often called routes in this context) in the timetable [DPW12, DPW15a]. The basic idea behind RAPTOR is that only the first reachable vehicle of a route has to be considered during journey planning. The reason for this is that taking a later vehicle of a route never leads to improved arrival times. As its name suggests, RAPTOR operates in rounds, where the i-th round discovers journeys that use i vehicles. Thus, RAPTOR inherently optimizes two criteria: the travel time of the journey and the number of trips used by the journey. While RAPTOR on its own is already quite fast, it can be combined with a partition-based speed-up technique, similar to MLD, to enable even faster queries [DDPZ17]. However, since public transit networks are not as hierarchical as road networks, the speed-up achieved by this approach is quite limited.

Another algorithm that operates directly on the timetable data is the Connection Scan Algorithm (CSA) [DPSW13, DPSW18]. The fundamental idea of CSA is to split the trips of the public transit vehicles into connections between consecutive stops. These connections are then sorted chronologically, such that optimal journeys can be

found through a single linear scan of the sorted array of connections. This approach is extremely memory efficient and therefore yields a fast query algorithm. Using CSA, it is also possible to find journeys with a minimum expected arrival time (MEAT) based on the delay probability of the public transit vehicles [DSW14]. Finally, a speed-up technique similar to MLD has been developed for CSA, which reduces the running time of the journey computation significantly [SW14]. However, this speed-up technique is only viable for pure public transit networks and cannot be used for networks that allow for walking between neighboring stops.

Further examples for journey planning algorithms that exploit the structure of the timetable include Frequency-Based Routing and Trip-Based Routing. Frequency-Based Routing is based on the observation that many public transit trips are repeated with a fixed frequency, which can be used to compress the timetable and enables fast journey planning [BS14]. This approach has also been used to reduce the preprocessing time of Transfer Patterns. The basic idea of Trip-based Routing is to precompute all pairs of trips between which transferring is reasonable [Wit15]. With this information a graph is constructed that contains one vertex for every trip of a public transit vehicle. Two of these vertices are connected by directed edges if a reasonable transfer between the corresponding trips was found during the preprocessing phase. Given this graph, optimal journeys can be computed with a slightly modified breadth-first search. The running time of this search can be significantly reduced by precomputing and compressing the search trees [Wit16]. Using these condensed search trees yields one of the fastest journey planning algorithms for public transit networks.

2.1.3 Multimodal Techniques

A journey planning algorithm is called multimodal if it can compute optimal journeys in a network that combines public transit with one or more other transportation networks. A particularly simple example of such a combined network consists of public transit and a footpath-graph that enables walking between arbitrary public transit stops. Of course, other transportation modes might also be added, such as cycling, using electric scooters, taking a taxi, or any other mode of transportation that does not have a fixed schedule. As before, the public transit part of the multimodal network can either be modeled as a graph (together with the modes of transportation) or it can be modeled by using a direct representation of the timetable. In the following, we present the state-of-the-art for both approaches.

Graph-Based Approaches. Similar to public transit, a multimodal network can be modeled as a time-dependent graph or a time-expanded graph [HJ13, GPZ19]. However, with an additional mode of transportation that is available, the size of

the graph increases significantly. Thus, journey planning algorithms that do not use speed-up techniques are often too slow in practice. An example for a multimodal speed-up technique is Access Node Routing [DPW09a], which combines a time-dependent representation of the network with TNR. In particular, this approach identifies all stops that are used to transition between the road networks and the public transit network. These stops are called access nodes and they are used, similar to transit nodes in TNR, in order to speed-up the journey computation. However, Access Node Routing was only evaluated for relatively small public transit networks, where it already required a comparatively large number of access nodes. Thus, it is questionable whether applying Access Node Routing to larger networks is practical.

Another example for a multimodal speed-up technique is State Dependent ALT, which combines a time-dependent graph representation of the multimodal network with ALT [Kir13]. This approach was evaluated for the networks of New York and Île-de-France, where it achieves a significant speed-up. Furthermore, State Dependent ALT can also handle additional modes of transportation, such as bike sharing or using a car. However, only networks with a single bike sharing operator are supported, i.e., bicycles can be returned at any bike sharing station.

Currently, the best known graph-based multimodal journey planning algorithm is UCCH (User-Constrained Contraction Hierarchies) [DPW15b]. It combines a time-dependent graph representation of the network with a core-CH. In particular, UCCH keeps all public transit stops and all vertices where the transportation mode can be changed in the core. The core-based approach enables the usage of the bidirectional CH query for public transit, which normally is not viable as it requires knowing the arrival time. The reason for this is, that the backwards search can be restricted to the contracted part of the network, which does not contain any public transportation.

An alternative approach for fast journey planning in complex multimodal networks are heuristic algorithms. In this case the provable correctness is sacrificed in favor of speed. Nonetheless, a carefully engineered heuristic algorithm can often find the optimal solution in practice [BJR16].

Timetable-based Approaches. Only a few multimodal journey planning algorithms that operate directly on the timetable have been developed. The first such approach is MCR (multiModal multiCriteria RAPTOR), which extends RAPTOR to the multimodal scenario [Del+13]. Overall, the approach of MCR is similar to the approach of UCCH, as it is also based on a core-CH where the public transit network remains uncontracted. After the core-CH has been computed in a preprocessing step, the query of MCR is mostly equivalent to RAPTOR. The main difference is, that every relaxation of transfer edges in RAPTOR is replaced with a Dijkstra search on the core graph. An advantage of MCR is that additional criteria besides travel time and

number of used public transit vehicles can be optimized (e.g., walking distance).

Most recently HLRAPTOR and HLCSA have been proposed, which combine HL with RAPTOR and CSA, respectively [PV19]. Both of these algorithms apply the speed-up technique (HL) only to the non-schedule based part of the multimodal network, i.e., the public transit network is not affected. This is quite similar to the approach of UCCH and MCR. HL is integrated into the query algorithm of RAPTOR and CSA by replacing the relaxation of transfer edges in these algorithms with multiple HL queries. Because of this, HLRAPTOR and HLCSA ar not nearly as fast as a single HL query on a road graph. However, they still achieve the best known running times for journey planning in multimodal networks.

Multicriteria optimization. Multimodal journey planning is often combined with multicriteria optimization. That is, besides travel time, other criteria have to be optimized, such as travel time, ticket cost, or walking distance. In general, an optimal route is no longer uniquely defined if more than one criterion is optimized. Thus, the goal of multicriteria optimization is to find a Pareto-set of journeys. While these Pareto-sets can get quite large in theory, they only contain a few entries in practice if only two criteria are optimized [MW01].

Almost all public transit algorithms and multimodal algorithms covered in this chapter support Pareto-optimization for two criteria: travel time and number of used public transit vehicles. Some algorithms support even more criteria. However for three or four criteria, the size of the Pareto-set increases significantly and quickly becomes impractical. Therefore, several approaches for finding meaningful subsets of a Pareto-set have been proposed [MS07, BBS13]. A notable example for such an approach is RAPTOR/MCR, which has been combined with fuzzy logic in order to determine the most significant journeys within a Pareto-set. Another example is Bounded McRAPTOR which introduced the notion of restricted Pareto-sets [DDP19]. These restricted Pareto-sets have the additional advantage that the Pareto-sets can be reduced during the execution of the algorithm, which also improves the running time.

2.2 Traffic Assignments

In the final section of our literature overview we consider traffic assignment algorithms for public transportation networks. In general, the traffic assignment problem can be subdivided into two subproblems [She85]. The first problem is to generate for each individual demand (i.e., pair of origin and destination) a set of journeys that could be used to fulfill the demand. This set is usually called the *choice set* and should not only contain optimal journeys, but all reasonable journeys. The objective of the

second subproblem is to determine for every journey in the choice set the percentage of passengers that would use that journey. Since passengers are *assigned* to journeys in this second step, the overall problem is called the traffic assignment problem.

While the first subproblem, at its core, has many similarities with journey planning problems, hardly any of the results covered in the last section were applied to the traffic assignment problem. Many assignment algorithms simply use Dijkstra's algorithm (modified such that additional suboptimal journeys can be found) to compute the journeys for the choice sets [TW99, Møl99, NF06]. However, this can be quite slow, since it is not uncommon that the demand consists of millions of origin-destination pairs. A more sophisticated approach uses branch and bound to compute extended shortest path trees that also represent slightly suboptimal journeys [FHW01].

Discrete Choice Models. Most research on public transit traffic assignments focuses on the second part of the problem, i.e., deciding which journeys in the choice set are used by which percentage of the passengers. Probably the simplest solution for this subproblem are all-or-nothing assignment, where all passengers are assigned to the best journey from the choice set (given a suitable definition of best journey) [SLM07]. While this simple approach already yields good results for traffic assignments in road networks, it does not produce realistic assignments for public transit networks [She85]. This is because in many cases public transit passengers will not agree on single best journey (e.g., some passengers prefer fast journeys, while other passengers prefer journeys that do not require many transfers between different vehicles).

To solve this issue, discrete choice models can be used, which stochastically assign the passengers to multiple journeys in the choice set. A comprehensive overview of various discrete choice models used for public transit assignments is given in [Tra09]. For this work, we focus on Random Utility Models (RUMs), which are one class of discrete choice models [Mar60, McF73]. A RUM rates the usefulness of every option (i.e., journey in the choice set) with a *utility*, which is a random variable. The probability that a certain option is chosen is then equivalent to the probability that the random utility of this option is greater than the random utility of each other option.

A commonly used RUM is the Logit model, which is based on the assumption that the random utilities have a Type-I Extreme Value distribution [DM75]. An advantage of this model is that the computation of the probabilities for all journeys in the choice set is comparatively simple. However, the resulting probabilities can become unrealistic if the choice sets are very large or if many journeys are similar (e.g., use the same vehicle for parts of the journeys). In order to overcome this shortcomings, many extensions to the Logit model have been proposed.

One example for an extended Logit model is the C-Logit model, which adds a commonality factor to the utility of the journeys [CNRV96, RV03]. This factor is

proportional to similarity of two journeys and adjusts their probability in order to reflect that they are not independent choices. However, the commonality factor has to be computed for all pairs of journeys, which is quite time consuming. Another approach for making the probabilities of similar journeys more realistic are nested Logit models [Dal87]. These models are based on the idea that the decision about which journey to choose from the choice set can be broken down into several independent decisions. For example: first deciding between using a tram or using a bus and afterwards deciding which specific line of the chosen vehicle type should be used. An extension of this idea is the sequential route choice model [GP06]. Within this model, every journey is split into several small sections (e.g., walking between to stops or taking a train from one stop to another). Afterwards, the probability of a journey is determined by evaluating a sequence of discrete choice models (one for every section) and multiplying their probabilities.

3 Fundamentals

In this chapter we present a detailed introduction of the basic concepts and notation used throughout this thesis. We start by formally introducing multimodal transportation networks and all of their components in Section 3.1. Next, we define the different journey planning problems covered in this thesis in Section 3.2. Finally, Section 3.3 highlights the most important algorithmic approaches, which are used as a basis for the solutions developed in this thesis. An additional quick guide to the notation can be found on page 187.

3.1 Network Models

Throughout this thesis we will encounter several different variants of journey planning problems. Since no single algorithm exists that achieves the best performance for all problem variants, we will use different algorithms, depending on the problem at hand. Moreover, many of the algorithms that are relevant for this thesis require a representation of the public transit network that is tailored to the algorithm. Thus, we have to handle several slightly different network models. In the following section we will introduce the most important network representations. For this, we first introduce a quite detailed base model, and then derive the other models from it.

On a fundamental level, we can model a public transit network based on its *stops*, *vehicles*, and *events* (where we differentiate between departure events and arrival events). The stops are all locations where it is possible to enter or leave a public transit vehicle (i.e., bus stops, train stations, platforms, ferry ports, etc.). These stops

are connected by the vehicles (buses, trains, ferries, etc.), which travel between them. The network contains one arrival event for every time when a vehicle arrives at a stop, such that passengers may leave the vehicle. Similarly, the network contains one departure event for every time when a vehicle departs from a stop, with the possibility of passengers entering it beforehand. We can represent both types of events using triples containing a stop, a time, and a vehicle. In order for the transit network to be valid, we require that all events of one vehicle, when sorted by time, form an alternating sequence of departure and arrival events, starting with a departure event and ending with an arrival event. Furthermore, the stop of each departure event has to be the stop of the preceding arrival event, except for the first departure event, which has no preceding arrival event.

3.1.1 Connection-Based Model

The first concrete network model we introduce is the one used by the Connection Scan Algorithm [DPSW13]. It revolves around the idea that the shortest amount of time a passenger can meaningfully spend in a public transit vehicle is the time between a departure event and the subsequent arrival event of the same vehicle. The model thus aggregates pairs of consecutive departure and arrival events into *connections*, which are also the origin of the name of the network model and the algorithm building upon it. Formally, a connection c is a 5-tuple $(v_{\text{dep}}(c), v_{\text{arr}}(c), \tau_{\text{dep}}(c), \tau_{\text{arr}}(c), T(c))$. It represents a vehicle driving from a *departure stop* $v_{\text{dep}}(c)$ to an *arrival stop* $v_{\text{arr}}(c)$ without any intermediate stops. The vehicle is scheduled to depart from $v_{\text{dep}}(c)$ at the *departure time* $\tau_{\text{dep}}(c)$ and arrives at $v_{\text{arr}}(c)$ at the *arrival time* $\tau_{\text{arr}}(c)$, which we require to be greater than $\tau_{\text{dep}}(c)$. Thus, a connection can be seen as an atomic part of a public transit journey, since a passenger can either use a connection entirely or not at all (it is not possible to stay inside a vehicle for a fraction of a connection). The *trip* $T(c)$ of the connection is an abstraction of the vehicle that drives the connection. Two connections c, c' are part of the same trip (i.e., $T(c) = T(c')$) if and only if a passenger can use both connections without leaving the vehicle in between. Thus, two connections being part of the same trip implies that they are served by the same physical vehicle. However, a single physical vehicle may serve several trips, for example when the vehicle returns to the depot in between.

The set of all connections that are present in the public transit network is denoted by $\mathcal{C}$. All stops served by the connections in $\mathcal{C}$ are compiled into the set of stops $\mathcal{S} := \{v_{\text{dep}}(c) \mid c \in \mathcal{C}\} \cup \{v_{\text{arr}}(c) \mid c \in \mathcal{C}\}$. Similarly, we define the set $\mathcal{T}$ of all trips in the network as $\mathcal{T} := \{T(c) \mid c \in \mathcal{C}\}$. Using this, we finally define the public transit network N, in its connection-based form, as the triple $N := (\mathcal{C}, \mathcal{S}, \mathcal{T})$.

Some algorithms (e.g. traffic assignments, which we cover in Chapter 8) require that all connections departing from the same stop have a well-defined order of

departure, i.e., $\forall c, c' \in \mathcal{C}\colon\ v_{\text{dep}}(c) = v_{\text{dep}}(c') \implies \tau_{\text{dep}}(c) \neq \tau_{\text{dep}}(c')$. Because of this, we establish a unique order of departures by perturbing the departure times of concurrent connections by some $\varepsilon > 0$. However, this has no effect on the journeys, which we will define in Section 3.1.4. In particular it does not change weather a journey is possible or not within the connection-based model.

3.1.2 Route-Based Model

The algorithms of the RAPTOR-family [DPW12] require a slightly different representation of the timetable data, while ultimately yielding the same results as other algorithms. Instead of focusing on the connections between subsequent stops, the route-based model has the interaction of a vehicle with a single stop at its core. To this end, we define a *stop event* ϵ as a triple $(\tau_{\text{arr}}(\epsilon), \tau_{\text{dep}}(\epsilon), v(\epsilon))$, which encapsulates a vehicle arriving at the stop $v(\epsilon)$ at the arrival time $\tau_{\text{arr}}(\epsilon)$ and subsequently departing from the same stop at the departure time $\tau_{\text{dep}}(\epsilon)$. Similar to a connection, a stop event also is the combination of an arrival and a departure event. However, while a connection combines a departure event with the subsequent arrival event at the next stop, the stop event combines an arrival event and the subsequent departure event at the same stop. Since we initially defined that the sequence of events belonging to a vehicle starts with a departure event and ends with an arrival event, we have to introduce special cases for the first and the last stop event of a vehicle. We do so by defining the arrival time of the first stop event of a vehicle as $\tau_{\text{arr}}(\epsilon) \coloneqq -\infty$ and the departure time of the last stop event as $\tau_{\text{dep}}(\epsilon) \coloneqq \infty$.

Similar to the connection-based model, we use *trips* as an abstraction of physical vehicles. For the route-based model, we define a trip T as a sequence $\langle \epsilon_0, \ldots, \epsilon_k \rangle$ of stop events that are served consecutively (without interruptions) by the same vehicle, such that a passenger may ride along all of them without requiring a transfer. For these trips we also introduce some additional notation: First, we define the length of a trip $T = \langle \epsilon_0, \ldots, \epsilon_k \rangle$ as $|T| \coloneqq k$. Thus, the length of a trip is the number of stop events it contains minus one, which corresponds to the number of connections in the trip when using the connection-based model. Secondly, we denote the i-th stop event of a trip T by $T[i]$. Thirdly, we define the stop sequence $v(T)$ of a trip T as $v(T) \coloneqq \big\langle v(T[0]), \ldots, v(T[k]) \big\rangle$. And finally, we say that the trip $T_a \in \mathcal{T}$ overtakes the trip $T_b \in \mathcal{T}$ if both trips have the same stop sequence and there exist two indices $i < j$, such that T_a arrives at or departs from $v(T_a[i])$ before T_b and T_a arrives at or departs from $v(T_a[j])$ after T_b.

The crucial idea of the route-based model is to partition the set of trips $\mathcal{T}$ into *routes*, such that any two trips of the same route have the same stop sequence and do not overtake each other. The rationale behind this is that a passenger that could use several trips of a route, only needs to consider the first trip of the route he can reach.

The reason for this is that all later trips reach the same stops at later points in time. In order to obtain a maximal benefit from this, we require that the set of trips is partitioned into as few routes as possible. We also require that every trip in the network is part of exactly one route. The resulting set of routes is denoted by $\mathcal{R}$. Note that the partition of trips into a minimal number of routes is not necessarily unambiguous.

Using the trips and routes we introduced above, we define the public transit network N, in its route-based form, as the triple $N := (\mathcal{S}, \mathcal{T}, \mathcal{R})$. As before, for the connection-based model, $\mathcal{S} := \{v(T[i]) \mid T \in \mathcal{T}, 0 \leq i \leq |T|\}$ denotes the set of all stops in the network.

3.1.3 Transfer Graphs

Since this thesis is focused on multimodal journey planning, we, of course, also consider other transportation modes besides public transit. We model non-schedule-based modes of transportation (such as walking, cycling, or driving) using *transfer graphs*. A transfer graph G is a weighted, directed graph, which we define as a tuple $G := (\mathcal{V}, \mathcal{E})$, consisting of a *vertex set* $\mathcal{V}$ and an *edge set* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. The vertices represent locations in the network. We require that they are a superset of the stops in the public transit network ($\mathcal{S} \subseteq \mathcal{V}$). By doing so, we establish a natural connection between the transfer graph and the public transit network. Each edge $e = (v, w) \in \mathcal{E}$ represents a street or pathway in the network that allows passengers to move from vertex v to vertex w. The *transfer time* required to move along the edge e in G is denoted by $\tau_{\text{tra}}(e)$. Since we do not intend to introduce time travel in this work, we limit the co-domain of the transfer time function to the non-negative real numbers. Formally, we define $\tau_{\text{tra}} \colon \mathcal{E} \to \mathbb{R}_0^+$. For simplicity of notation we also use $\tau_{\text{tra}}(v, w)$ to denote the transfer time of an edge $e = (v, w) \in \mathcal{E}$. We call the graph $G = (\mathcal{V}, \mathcal{E})$ *transitively closed* if, for each pair $(v, w), (w, x) \in \mathcal{E}$ of edges, a third edge $(v, x) \in \mathcal{E}$ exists such that $\tau_{\text{tra}}(v, x) \leq \tau_{\text{tra}}(v, w) + \tau_{\text{tra}}(w, x)$.

A *path* P in G is a sequence of vertices $\langle v_1, \ldots, v_k \rangle$, such that an edge $(v_i, v_{i+1}) \in \mathcal{E}$ exists for every $0 < i < k$. The number of vertices in the path P is denoted by $|P| := k$. We extend the notion of transfer time from edges to paths $P = \langle v_1, \ldots, v_k \rangle$, using the definition $\tau_{\text{tra}}(P) := \sum_{i=1}^{k-1} \tau_{\text{tra}}(v_i, v_{i+1})$. We also allow paths to consist of only a single vertex and define $\tau_{\text{tra}}(\langle v \rangle) := 0$ for this case. We call the path $P = \langle v_1, \ldots, v_k \rangle$ an s-t-path if $v_1 = s$ and $v_k = t$ holds.

An s-t-path P is called a *shortest path* if $\tau_{\text{tra}}(P) \leq \tau_{\text{tra}}(P')$ holds for all s-t-paths P' that exist in G. Using this definition of shortest paths, we introduce a distance measure $\delta_\tau \colon \mathcal{V} \times \mathcal{V} \to \mathbb{R}_0^+ \cup \{\infty\}$ on the vertices of a graph. To this end, we define $\delta_\tau(v, w)$ as the transfer time $\tau_{\text{tra}}(P)$ of a shortest v-w-path P in G, if such a path exists. If the vertices v and w are not connected by any path in the graph, we define $\delta_\tau(v, w) := \infty$. Thus, $\delta_\tau(v, w)$ is the minimal time required to transfer from vertex v to w. We can

use this for an alternative characterization of transitively closed graphs. A graph G is transitively closed if for every pair of vertices $v, w \in \mathcal{V}$ with $\delta_\tau(v, w) \neq \infty$ an edge $(v, w) \in \mathcal{E}$ exists such that $\delta_\tau(v, w) = \tau_{\text{tra}}(v, w)$ holds.

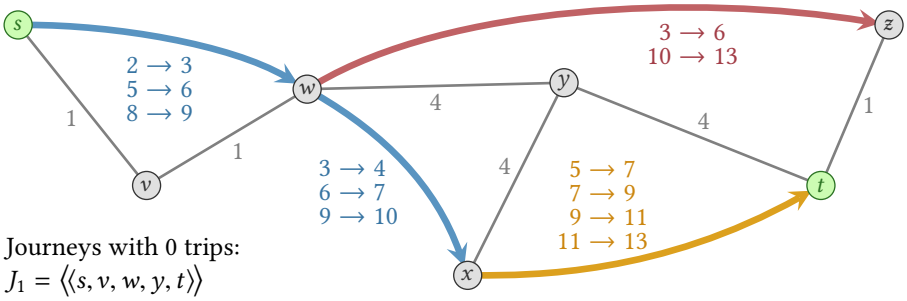
We call a graph $G^c = (\mathcal{V}^c, \mathcal{E}^c)$ a *core graph* of G if its vertices are a subset of the vertices from G and the transfer times of shortest paths in G are preserved in G^c for pairs of vertices from $\mathcal{V}^c$. More formally, we define that $G^c = (\mathcal{V}^c, \mathcal{E}^c)$ is a core graph of $G = (\mathcal{V}, \mathcal{E})$ if $\mathcal{V}^c \subseteq \mathcal{V}$ and $\forall v,w \in \mathcal{V}^c$: $\delta_\tau(v, w)$ in $G = \delta_\tau(v, w)$ in G^c holds. Note that G^c is sometimes also called *overlay graph* in the literature.

3.1.4 Journeys and Profiles

With the multimodal network being well-defined, we can now focus on the objects we want to compute, given a network. Most relevantly, we want to describe the movement of a passenger through the network. For this purpose, we first require the notion of a *trip leg*. A trip leg T^{ij} is a subsequence of the trip T, representing a passenger boarding the trip T at the i-th stop and disembarking at the j-th stop. Hence, for a given trip $T = \langle \epsilon_0, \ldots, \epsilon_k \rangle$, we define $T^{ij} := \langle \epsilon_i, \ldots, \epsilon_j \rangle$. In addition, we define the departure time of T^{ij} as the departure time at the first stop of the trip leg T^{ij}, i.e., $\tau_{\text{dep}}(T^{ij}) := \tau_{\text{dep}}(T[i])$. Similarly, the arrival time of the trip leg is defined as $\tau_{\text{arr}}(T^{ij}) := \tau_{\text{arr}}(T[j])$.

Two trip legs can be connected using an *intermediate transfer*. Given two trip legs T_a^{ij} and T_b^{mn}, we define an intermediate transfer as a path P in the transfer graph G with the following properties: First, the path P begins with the vertex of the last stop event of the trip leg T_a^{ij}, i.e., $v(T_a[j])$. Secondly, the path P ends at the vertex of the first stop event of the trip leg T_b^{mn}, i.e., $v(T_b[m])$. Thirdly, the transfer time $\tau_{\text{tra}}(P)$ of the path is sufficient to reach T_b^{mn}. The transfer time is sufficient if, after vacating T_a^{ij}, there is enough time to transfer to the departure stop of T_b^{mn} before T_b^{mn} departs. We can express this formally as $\tau_{\text{arr}}(T_a[j]) + \tau_{\text{tra}}(P) \leq \tau_{\text{dep}}(T_b[m])$. An *initial transfer* before a trip leg T^{ij} is a path $P = \left\langle s = v_0, \ldots, v_k = v(T[i]) \right\rangle$ in G from a source vertex s to the first stop of T^{ij}. Correspondingly, we define a *final transfer* after a trip leg T^{ij} as a path $P = \left\langle v(T[j]) = v_0, \ldots, v_k = t \right\rangle$ in G from the last stop of T^{ij} to the target t. We use the term *transfer* on its own to denote the union of all transfer types, or if the actual type of the transfer can be deduced from context.

Journeys. Building upon trip legs and transfers, we can finally introduce journeys within the multimodal network. We define a *journey* $J = \langle P_0, T_0^{ij}, \ldots, T_{k-1}^{mn}, P_k \rangle$ as an alternating sequence of transfers and trip legs. Note that some or all of the transfers may be empty, i.e., consist of a single stop only. We call a journey J an s-t-journey if the first vertex of P_0 is s and the last vertex of P_k is t. The set



Journeys with 0 trips:
$J_1 = \langle\langle s, v, w, y, t\rangle\rangle$

Journeys with 1 trip:
$J_2 = \langle\langle s, v, w\rangle, \langle 3 \to 6\rangle, \langle z, t\rangle\rangle$
$J_3 = \langle\langle s\rangle, \langle 2 \to 3\rangle, \langle w, y, t\rangle\rangle$
$J_4 = \langle\langle s, v, w\rangle, \langle 10 \to 13\rangle, \langle z, t\rangle\rangle$

Journeys with 2 trips:
$J_5 = \langle\langle s\rangle, \langle 2 \to 3, 3 \to 4\rangle, \langle x\rangle, \langle 5 \to 7\rangle, \langle t\rangle\rangle$
$J_6 = \langle\langle s\rangle, \langle 5 \to 6, 6 \to 7\rangle, \langle x\rangle, \langle 7 \to 9\rangle, \langle t\rangle\rangle$
$J_7 = \langle\langle s\rangle, \langle 8 \to 9, 9 \to 10\rangle, \langle x\rangle, \langle 11 \to 13\rangle, \langle t\rangle\rangle$

Figure 3.1: A small example of a multimodal network. The public transit network contains 3 routes (colored arrows) and 12 connections, which are annotated with $\tau_{\text{dep}} \to \tau_{\text{arr}}$. The transfer graph contains 6 edges, which are annotated with their transfer times. Below the network we list the s-t-journeys (s and t are marked in green) in a Pareto-set, with respect to departure time, arrival time, and number of trips.

of all s-t-journeys is denoted by $\mathcal{J}_s^t$. For our objective of journey planning, we need to assess the quality of a journey. To this end, we introduce some properties of journeys that are of special interest: First, we define the departure time of a journey $J = \langle P_0, T_0^{ij}, \ldots, T_{k-1}^{mn}, P_k\rangle$ as $\tau_{\text{dep}}(J) := \tau_{\text{dep}}(T_0^{ij}) - \tau_{\text{tra}}(P_0)$, and the arrival time as $\tau_{\text{arr}}(J) := \tau_{\text{arr}}(T_{k-1}^{mn}) + \tau_{\text{tra}}(P_k)$. Using these two values, we can see that the overall travel time of the journey is $\tau_{\text{tra}}(J) := \tau_{\text{arr}}(J) - \tau_{\text{dep}}(J)$. Finally, we are interested in the number of trips (i.e., vehicles) used by the journey, which we denote as $|J| := k$. An important special case is a journey $J = \langle P_0\rangle$ that consists solely of a path in the transfer graph. Since such a journey does not rely on any trip, it can be traveled at any time. Thus, its departure time $\tau_{\text{dep}}(J)$ has to be stated separately. Given the departure time, the corresponding arrival time is defined as $\tau_{\text{arr}}(J) := \tau_{\text{dep}}(J) + \tau_{\text{tra}}(P_0)$.

Since multiple properties affect the quality of a journey, it is not possible in general to establish a total order on a set of journeys. However, we can define a partial ordering, using the concept of *Pareto dominance*. To this end, let $\mathcal{F}$ be a set of functions that map journeys to some totally ordered set (e.g. $\mathcal{F} = \{\tau_{\text{arr}}(\cdot), |\cdot|\}$). We then say that a journey J *weakly dominates* a journey J' with respect to $\mathcal{F}$ if $f(J) \leq f(J')$ holds for all f in $\mathcal{F}$. The journey J *strictly dominates* J' with respect to $\mathcal{F}$ if J weakly dominates J' and $f(J) < f(J')$ holds for at least one function $f \in \mathcal{F}$. Let $\mathcal{J}$ be a set of

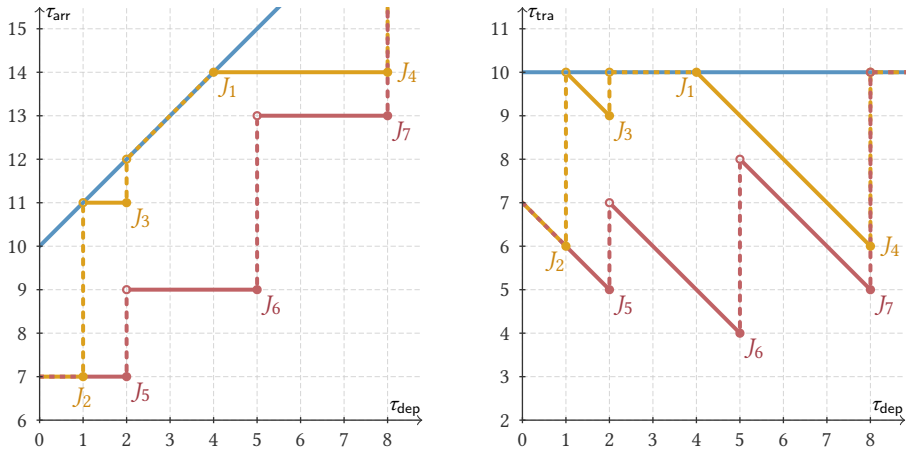


Figure 3.2: A trip-dependent arrival time s-t-profile f^+_{arr} (left) and a trip-dependent travel time s-t-profile f^+_{tra} (right) for the multimodal network from Figure 3.1. The blue plots represent $f^+_{arr}(\tau, 0)$ and $f^+_{tra}(\tau, 0)$. The yellow plots represent $f^+_{arr}(\tau, 1)$ and $f^+_{tra}(\tau, 1)$. The red plots represent $f^+_{arr}(\tau, 2)$ and $f^+_{tra}(\tau, 2)$. The filled points correspond to the journeys in the Pareto-set of s-t-journeys.

journeys. We call a journey $J \in \mathcal{J}$ *Pareto-optimal* (with respect to $\mathcal{J}$ and $\mathcal{F}$) if no journey that strictly dominates J exists in $\mathcal{J}$. Furthermore, a subset $\mathcal{J}' \subseteq \mathcal{J}$ of minimal cardinality, such that each journey in $\mathcal{J}$ is weakly dominated by a journey from $\mathcal{J}'$, is called a *Pareto-set* of $\mathcal{J}$ (with respect to $\mathcal{F}$). Note that the Pareto-set is not necessarily unique. Common examples for the set $\mathcal{F}$ are $\mathcal{F} = \{\tau_{tra}(\cdot), |\cdot|\}$, where we want to minimize the travel time and number of used trips, and $\mathcal{F} = \{-\tau_{dep}(\cdot), \tau_{arr}(\cdot), |\cdot|\}$, where we optimize for journeys that depart as late as possible, arrive as early as possible, and use as few trips as possible. In order to keep the notation simple, we will not specify the set of functions $\mathcal{F}$ explicitly every time that we work with Pareto-optimization. Instead we simply mention (in text form) the properties of the journeys we want to optimize. An example of a small multimodal network and an accompanying Pareto-set of s-t-journeys is shown in Figure 3.1.

Profiles. Closely related to Pareto-sets that consider departure time as one of their optimization criteria is the notion of profiles. An *arrival time s-t-profile* is a function $f_{arr} : \mathbb{R} \to \mathbb{R} \cup \{\infty\}$ that maps each departure time τ to the minimal arrival time of any s-t-journey J with $\tau_{dep}(J) \geq \tau$. The arrival time profile is a piecewise linear function consisting of segments with a slope of either 0 or 1. Segments with slope 0

indicate that the best journey uses at least one trip and requires some additional waiting time before it departs. Segments with slope 1 correspond to transferring directly to the target (without using the public transit network). Since this can be done at any departure time and since the time required for the transfer is independent of the departure time, the slope is 1. The breakpoints of the arrival time s-t-profile function correspond exactly to the journeys in a Pareto-set of s-t-journeys with respect to departure time and arrival time, since the profile function returns the minimal arrival time for every departure time. A slightly different view on the same data is given by the *travel time s-t-profile* function $f_{\text{tra}} : \mathbb{R} \to \mathbb{R}_0^+ \cup \{\infty\}$, which is defined as $f_{\text{tra}}(\tau) \coloneqq f_{\text{arr}}(\tau) - \tau$.

A generalization of the aforementioned profiles, which also considers the number of trips used by the journeys, are trip-dependent profiles. A *trip-dependent arrival time s-t-profile* is a function $f_{\text{arr}}^+ : \mathbb{R} \times [0, \ldots, |\mathcal{R}|] \to \mathbb{R} \cup \{\infty\}$ that maps each departure time τ and maximal number of trips n to the minimal arrival time of any s-t-journey J with $\tau_{\text{dep}}(J) \geq \tau$ and $|J| \leq n$. As before, this function is piecewise linear, if we fix its second parameter. Moreover, the breakpoints of this profile function once again correspond to the journeys in a Pareto-set. In particular, the breakpoints of a trip-dependent arrival time s-t-profile are precisely the journeys in a Pareto-set of s-t-journeys with respect to departure time, arrival time, and number of trips. We can also see that the trip-dependent profile is a true generalization of the normal profile, as $f_{\text{arr}}(\tau) = f_{\text{arr}}^+(\tau, |\mathcal{R}|)$ holds. The reason for this is that every Pareto-optimal journey will use at most one trip of every route in a public transit network. Thus, asking for the minimal arrival time of any journey with at most $|\mathcal{R}|$ trips is equivalent to asking for the minimal arrival time of any journey. Finally, we can again define a representation of the profile that focuses on travel time instead of arrival time. We do so by defining the *trip-dependent travel time s-t-profile* as a function $f_{\text{tra}}^+ : \mathbb{R} \times [0, \ldots, |\mathcal{R}|] \to \mathbb{R}_0^+ \cup \{\infty\}$ with $f_{\text{tra}}^+(\tau, n) \coloneqq f_{\text{arr}}^+(\tau, n) - \tau$. An example of a trip-dependent arrival time profile and the corresponding travel time profile is given in Figure 3.2.

Often we are not interested in the optimal journeys for every departure time, but only for a small interval $I = [\tau_{\text{min}}, \tau_{\text{max}}] \subseteq \mathbb{R}$ of departure times. We achieve this for every profile type by simply restricting the domain of the function. However, it is important to note that such a restriction does not carry over directly to the corresponding Pareto-set of the profile. In detail, this means that the set of breakpoints of a profile restricted to the departure time interval $I = [\tau_{\text{min}}, \tau_{\text{max}}]$ is not equivalent to the Pareto-set of journeys J with $\tau_{\text{dep}}(J) \in I$. The reason for this is that an optimal journey for the departure time τ_{max} might involve some waiting time at the source, such that the actual departure of the journey is greater than τ_{max}. However, we can still formulate a slightly altered correlation: The breakpoints of the trip-dependent arrival time s-t-profile restricted to departure times in $I = [\tau_{\text{min}}, \tau_{\text{max}}]$ correspond

exactly to the journeys in the union of the following two sets. The first set is the Pareto-set of $\{J \mid J \in \mathcal{J}_s^t,\ \tau_{\mathsf{dep}}(J) \in I\}$ with respect to departure time, arrival time, and number of trips. The second set is the Pareto-set of $\{J \mid J \in \mathcal{J}_s^t,\ \tau_{\mathsf{dep}}(J) \geq \tau_{\mathsf{max}}\}$ with respect to arrival time and number of trips. An analogous correlation can be formulated for all other profile types.

3.1.5 Minimum Change Times and Departure Buffer Times

We obtain multimodal transportation networks by combining a public transit network with one or multiple transfer graphs. Doing this does not require any additional steps, as the stops of the public transit network are a subset of the vertices in the transfer graph by definition. For this work, we introduced journeys within the multimodal network in a way that allows switching from trips in the public transit network to paths in the transfer graph at stops without any constraints. However, this is not the only formulation of journeys used in the literature.

Minimum Change Times. The most common approach specifies an additional *minimum transfer time* or *minimum change time* $\tau_{\mathsf{ch}}(v)$, which has to be observed when transferring between two trips at the stop v of the public transit network [DPW12, Wit15, DPSW18]. More specifically, a journey $J = \langle \ldots, T_a^{ij}, \langle v \rangle, T_b^{mn}, \ldots \rangle$ is only valid if the difference between the departure time $\tau_{\mathsf{dep}}(T_b^{mn})$ of the trip T_b at v and the arrival time $\tau_{\mathsf{arr}}(T_a^{ij})$ of the trip T_a at v is greater or equal to the minimum change time $\tau_{\mathsf{ch}}(v)$ of the stop. The minimum change time was introduced, because a single stop v is occasionally used to represent a whole station instead of individual platforms. In order to transfer between trips at a large station, it can be necessary to change from one platform to another, which requires additional time. In this case, the minimum change time ensures that two trips can be part of a single journey only if the time between them is sufficient for the transfer.

It is important to note that the minimum change time has to be observed only when transferring between two trips at the same stop. The minimum change time does not apply to cases where a path in the transfer graph is used to transfer between trips. It is also not used when transferring from the source vertex to the departure stop of the first trip in a journey. The reason for this is that minimum change times were introduced before complex transfer graphs have been considered. Most works that use minimum change times only allow transfers between trips, which consist of at most one edge $e = (v, w)$ in the transfer graph. In this case, the time required for leaving the station v or entering the station w is simply added to the transfer time of the edge connecting them.

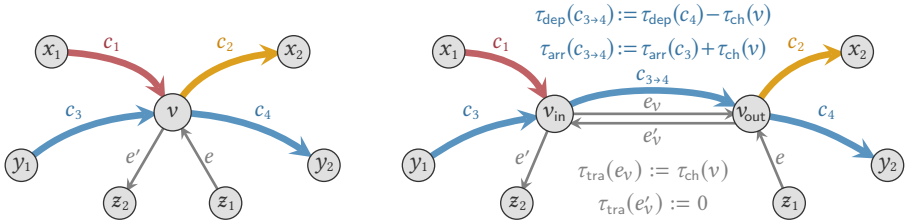


Figure 3.3: An example of a network with minimum change time $\tau_{\text{ch}}(v)$ at the stop v (left) and an equivalent network without minimum change times (right). Connections share the same color if they are part of the same trip. For the construction of the right network, the stop v is split into two stops v_{in} and v_{out}. These new stops are connected by two new edges $e_v = (v_{\text{in}}, v_{\text{out}})$ and $e'_v = (v_{\text{out}}, v_{\text{in}})$.

Departure Buffer Times. An approach better suited for the use within multimodal scenarios is based on *departure buffer times* $\tau_{\text{buf}}(v)$. Here, a journey is only valid if the departure time $\tau_{\text{dep}}(T^{ij})$ of each trip leg T^{ij} in the journey is greater or equal to the arrival time at that stop plus the departure buffer time $\tau_{\text{buf}}(v_{\text{dep}}(T^{ij}))$. This requirement is independent of the mode of transportation that was used to arrive at the departure stop. The departure buffer time can be interpreted as the time required for a passenger to orient himself and find the way to the departure platform.

Network Transformations. While these two approaches make the overall network model more realistic, they do not increase the complexity of any journey planning problem. In fact, we can replicate the effects of both, minimum change times and departure buffer times, with small modifications to the public transit network. If we want to handle departure buffer times, we simply reduce the departure time $\tau_{\text{dep}}(\epsilon)$ of each stop event ϵ in the network by $\tau_{\text{buf}}(v(\epsilon))$, which yields the new stop event $\epsilon' := (\tau_{\text{arr}}(\epsilon), \tau_{\text{dep}}(\epsilon) - \tau_{\text{buf}}(v(\epsilon)), v(\epsilon))$. It is easy to see that these modified stop events lead to the same results as departure buffer times, since they require an arrival time that is less or equal to $\tau_{\text{dep}}(\epsilon) - \tau_{\text{buf}}(v(\epsilon))$ in order to embark the trip of ϵ', just like the departure buffer time would.

Handling minimum change times is more involved, since we need to be able to differentiate whether a stop was reached by trip (in which case the minimum change time has to be observed) or by a path (in which case it is immediately possible to enter a trip). We achieve this by replacing each stop v of the network with two stops v_{in} and v_{out}, which are connected by two new edges $e_v = (v_{\text{in}}, v_{\text{out}})$ and $e'_v = (v_{\text{out}}, v_{\text{in}})$, as shown in Figure 3.3. Inbound connections of the original stop v are connected to v_{in} and outbound connections are connected to v_{out}. Thus, transferring between trips at

the stop v is only possible by using the edge e_v. By defining the transfer time of this edge as $\tau_{\text{tra}}(e_v) := \tau_{\text{ch}}(v)$ we account for the minimum change time, which is required for the transfer. Edges like $e' = (v, z_2)$ that originate at v are substituted with edges that start at v_{in}. Therefore, a trip leg can be succeeded by a path in the transfer graph without incurring the minimum change time. Similarly, edges like $e = (z_1, v)$ that end at v are substituted with edges that end at v_{out}, enabling the usage of trips after a path without applying the minimum change time. We ensure that paths, which pass through v, can still be represented in the new network by defining the transfer time of the edge e'_v as $\tau_{\text{tra}}(e'_v) := 0$. Finally, in order to prevent gaps within trips, an additional connection from v_{in} to v_{out} is added to the network for every trip that passes through v.

The construction of this network reveals another problem of minimum change times that arises when they are combined with a transfer graph. Assume that z_1 and z_2 from Figure 3.3 are the same vertex. In this case, transferring from v_{in} to v_{out} is not only possible via the edge e_v but also via the path through z_1. This path can be used to evade the minimum change time if its transfer time is less than $\tau_{\text{ch}}(v)$. The reason for this is the definition of the minimum change time, by which the minimum change time only has to be observed if no path in the transfer graph is used to transfer between trips.

It is important to note that for both approaches, the constructed network may contain stop events where the vehicle departs before it arrives. However, this is not a problem since journeys that arrive at the same vertex that they started from, but at an earlier time, are still not possible. We therefore adjust our definition of public transit networks to allow stop events with a negative duration. We furthermore specify that the order of stop events in a trip is still determined with respect to the original time of the stop events.

Throughout this work, we will use networks with departure buffer times, since this is the most appropriate approach for multimodal scenarios. However, we will not discuss the effect of departure buffer times explicitly for every algorithm presented in this work. Instead, we simply assume that the departure buffer times are integrated into the stop events of the public transit network.

3.2 Journey Planning and Assignment Problems

Throughout the literature, several different journey planning problems have been considered for networks as those, which we defined in the previous section. In this section we introduce the journey planning problems addressed in this work and differentiate them from common other journey planning problems. To this end, we present a brief classification of some common journey planning problems. We classify journey planning problems on the basis of four properties: the number of sources, the number of targets, the optimization goal, and the timeframe.

Number of Sources/Targets. For both, the number of sources and the number of targets, we distinguish between three options. The first option is that a single vertex is specified as source or target location of the journey. We denote this case by using the word *one* within the problem definition. Alternatively, a set of vertices can be specified as source or target locations. In this case the objective is to find an optimal journey for each pair of source and target location. We use the word *many* to denote this within the problem formulation. Finally, we distinguish the special case where optimal journeys have to be found for all vertices of the network, which we denote by *all*.

The number of sources and targets can be chosen independently of each other and each combination results in a separate journey planning problem. For denoting these problems we use the notation: *source-type*-to-*target-type*. An example for this is the one-to-many problem, where a single source vertex and multiple target vertices are given. Accordingly, the objective of this problem is to find optimal journeys from the source vertex to all target vertices.

Optimization Goal. We further differentiate journey planning problems on the basis of the criteria with regard to which the journeys are optimized. In particular, we consider two types of optimization problems in this thesis. The first type is the *earliest arrival* problem, which asks for the minimal possible arrival time at the target. Finding a journey that actually achieves the minimal possible arrival time is not considered to be a separate problem within this work, since almost all earliest arrival algorithms also provide the corresponding journey. It is important to note that a journey not necessarily minimizes the travel time if it minimizes the arrival time. Most earliest arrival algorithms require some additional steps in order to find a journey with minimal travel time. However, finding such a journey generally does not require much additional time, once the earliest arrival time has been computed.

In addition to earliest arrival problems we also consider *bicriteria* problems in this work, where the arrival time and the number of used trips are optimized. In particular, the goal of the bicriteria problem is to find a Pareto-set of journeys with respect to arrival time and the number of used trips. Similar to earliest arrival problems we mainly focus on finding the Pareto-optimal values for the arrival time and the number of used trips. However, most bicriteria algorithms can also compute the actual journeys that are part of the Pareto-set.

Timeframes. The last part of our problem definition is the timeframe for the departure of the journeys. For this we distinguish two cases: a fixed departure time and a departure time interval. The objective of planning problems with fixed departure time is to find optimal journeys among all journeys with a departure time greater or equal to the given value. In the case of a departure time interval the objective of the

problem is to find a profile for the specified interval. Thus, we denote the problem as *profile* problem in this case. If the timeframe of the problem is not explicitly stated we assume a problem with fixed departure time.

Problems Addressed in this Work. We now briefly list the problems considered in this work. All of them assume that a public transit network N and a transfer graph G are given. The first journey planning problem, which we address in Chapter 5, is the *one-to-one bicriteria profile* problem. Afterwards we consider the *one-to-one earliest arrival* problem and the *one-to-one bicriteria* problem in chapters 6 and 7.

Finally, we consider *assignment* problems in Chapter 8. These problems do not fit in our classification of journey planning problems, as they involve stochastic discrete choice models. However, the assignment problems have many similarities with classical *many-to-many* problems. We will present a precise definition of the assignment problems considered in this work in Chapter 8.

3.3 Algorithms

We proceed with describing existing journey planning algorithms and concepts, which we will use as basis for our algorithms. In particular we will use ideas from Dijkstra's algorithm and Contraction Hierarchies in order to handle the non-schedule-based parts of the multimodal networks (e.g., walking, cycling, or driving). Regarding public transit, we will use and combine ideas from three different timetable-based algorithms: CSA, RAPTOR, and Trip-Based Routing.

3.3.1 Shortest Paths in Non-Timetable Networks

A fundamental idea shared by many shortest path algorithms is the approach of iteratively extending optimal journeys until a journey from the source to the target has been found. Algorithms based on this approach commonly use *labels* to represent the partial journeys that have already been found. One of the earliest shortest path algorithm based on this approach is Dijkstra's algorithm [Dij59].

Dijkstra's Algorithm. Dijkstra's algorithm uses one label per vertex in the graph to represent shortest paths, which start at the source vertex of the query. During the execution of the algorithm, the label of a vertex v represents the shortest path found thus far that starts at the source and ends at the vertex v. Each label is either a single key value or a pair of parent pointer and key, depending on whether only the length of the shortest path or the shortest path itself has to be computed. In both cases the key of the label is equivalent to the property of the journey that has to be

optimized. Classically, this is the distance or the travel time of the journey. In this work, we will often use the arrival times of the journeys as the key of the label, since we want to solve earliest arrival problems.

The algorithm starts with a single label for the source vertex that represents a path of length zero (i.e., a path that only contains the source vertex). This journey is represented by a label, which has either 0 or the departure time of the query as key value, depending on whether travel time or arrival time should be minimized. If the optimal path should also be calculated, then the source vertex itself is used as value of the parent pointer.

Starting with the label of the source vertex, Dijkstra's algorithm iteratively creates new labels by extending existing labels. For this, the algorithm maintains a priority queue of labels that have not yet been processed. This queue initially contains only the label of the source vertex and uses the key of the labels as priority criterion. After the queue has been setup the algorithm continues with iteratively processing the label with the minimal key in the queue. Processing a single label is called *settling* the label and starts with removing the label from the queue. Afterwards, all edges $e = (v, w)$ that start at vertex v that corresponds to the label are *relaxed*. Relaxing the edge creates a new label for the vertex w, where the key of this new label is the sum of the travel time of the edge e and the key of the original label (i.e., the label corresponding to the vertex v). If another label with a strictly smaller key has previously been computed for the vertex w, then the new label is discarded. Otherwise, the new label is used for the vertex w and added to the queue, replacing any previous label of w. Finally, if parent pointers are used, then v is used as the value of the parent pointer of the new label.

The algorithm ends when the queue is empty. At this point the computed labels correspond to the optimal paths to all vertices in the graph. If only the optimal path to a certain target vertex has to be computed, then the algorithm can be terminated once the target has been removed from the queue. In order to find the optimal path, the parent pointers can be traced back to the source node.

An important factor for the running time of Dijkstra's algorithm is the implementation of the priority queue. The best known worst case bound for the running time is achieved by using a Fibonacci heap [FT87]. However, it has been shown that Fibonacci heaps are outperformed by 4-ary heaps on realistic networks [CGR96]. Thus, we use 4-ary heaps as priority queues in this work.

Bidirectional Search. A simple yet versatile speed-up technique for Dijkstra's algorithm is based on the idea of searching simultaneously from the source and the target [Nic66]. In order to implement this, a second set of vertex labels and a second priority queue is used for the additional *backward search*. This backward search differs in two points from the *forward search*, which we described above.

First, it is initialized with one label for the target vertex instead of the source vertex. Secondly, it relaxes edges in the opposite direction, i.e., the edge (w, x) is relaxed during the settling of the vertex x and creates a new label for the vertex w.

After both searches (forward and backward) have been initialized, the bidirectional algorithm continues with settling the labels in the queues. For this purpose, it has to be decided whether the next label to be settled should be extracted from the forward queue or the backward queue. Common approaches for choosing the next queue are: selecting the queue with the smaller minimal key, selecting the queue that contains fewer entries, or alternating between the two queues, which we do in all bidirectional algorithms presented in this work.

Finally, the bidirectional search algorithm finds optimal journeys by combining labels from the forward search with labels of the backward search. For this purpose, the algorithm keeps track of the *tentative key* of the best journey that has been found, which is initialized as ∞. Every time a new label for some vertex w is created (while relaxing the edge (v, w) in the forward search or the edge (w, x) in the backward search), the algorithm checks whether a path through w can improve the tentative key. In particular, the tentative key is updated with the sum of the keys of the forward and backward label of w, if both these labels exist and if the sum of their keys is smaller than the tentative key. The algorithm terminates when the sum of the minimal keys of the two queues is larger than the tentative key.

Contraction Hierarchies. One of the most efficient speed-up techniques for road networks is Contraction Hierarchies (CHs), which is based on a bidirectional search algorithm that uses precomputed shortcut edges to reduce the size of the search space [GSSV12]. We use CHs in this work as they, and especially core-based variants of them, are particularly well suited for complex journey planning scenarios, such as multimodal journey planning.

The CH preprocessing phase is based on vertex *contractions*. A vertex w is contracted in two steps: First, the vertex w is temporarily (until the end of the preprocessing phase) removed from the graph. Secondly, *shortcuts* are added to the graph, such that the minimal travel time between the remaining vertices is equivalent to the minimal travel time in the original graph. The shortcuts required for this are determined by iterating over all pairs of an incoming edge (v, w) and an outgoing edge (w, x) of the vertex w, which is contracted. For each of these pairs, the CH preprocessing algorithm checks, whether a shortcut edge (v, x) with travel time $\tau_{\text{tra}}(v, x) \coloneqq \tau_{\text{tra}}(v, w) + \tau_{\text{tra}}(w, x)$ has to be added to the graph. The shortcut has to be added if the minimal travel time from v to x increases due to the removal of w. In order to check this, the bidirectional variant of Dijkstra's algorithm is used to compute a shortest path from v to x in the graph without w. If the travel time of this path

is longer than the travel time of the shortcut, then the shortcut is added to the graph, otherwise it is omitted. In the case that the graph already contains an edge from v to x, the travel time of this edge is simply updated instead of adding the shortcut edge.

While the contraction of a single vertex is a fast operation, contracting millions of vertices can take quite a long time in practice. In order to improve the running time, the bidirectional search, which accounts for the largest portion of the overall running time, can be terminated early. In this case it is simply assumed that no path exists and the shortcut is added to the graph. This might lead to superfluous shortcuts in the graph, but it does not affect the correctness of the algorithm. The CH implementation used in this work adds shortcuts to the graph if the bidirectional search did not find a shorter path before settling more than 400 vertices.

The goal of the CH preprocessing phase is to contract all vertices in the graph while adding as few shortcuts as possible. In order to minimize the number of shortcuts, the vertices have to be contracted in a suitable order. However, finding a contraction order that minimizes the number of shortcuts is NP-hard [Bau+10a]. Thus, most implementations use some heuristical approach for determining the order in which the vertices are contracted. Most commonly a greedy algorithm is used, where the next vertex to be contracted is chosen depending on a linear combination of multiple factors. In this work we consider two factors to rate a vertex v: The first factor is the exact number of shortcuts that have to be added if v is contracted divided by the number of edges incident to v. The second factor is the level of v, where the level of a vertex is zero if none of its neighbors is contracted and otherwise the level is one plus the maximal level of any of its contracted neighbors. We weight the first factor four times as much as the second factor and contract the vertex that minimizes this linear combination.

The CH query algorithm is a variant of the bidirectional search, which uses the contraction order and the shortcuts to skip unimportant vertices. To this end, the query algorithm only relaxes edges (and shortcuts) that lead to vertices that have been contracted after the vertex that is currently settled. In contrast to the standard bidirectional search, the CH query can only terminate after the minimal keys of both queues are larger than the tentative key.

Core-CH. For road networks the preprocessing phase of CH only takes a few minutes. However, in more complex networks the preprocessing phase can become much slower or even be infeasible. A common reason for this are vertices that produce a large number of shortcuts if they get contracted. Adding a large number of shortcuts increases the average vertex degree in the uncontracted graph, which in turn increases the number of shortcuts that have to be considered when contracting further vertices. In such a case the preprocessing can take several weeks which renders it impractical. Moreover, a large number of shortcuts can also lead to a slowdown of the query phase.

These problems can be resolved by aborting the preprocessing phase before all vertices have been contracted. In this case the vertices that have not been contracted form a *core* graph. Due to its construction the optimal travel times in the core are equivalent to the optimal travel times in the original graph. Thus, applying a journey planning algorithm to the core yields correct results and is usually faster than using the original graph.

A mostly unmodified version of the CH query algorithm can be used to compute journeys between contracted vertices. It only needs to be ensured that core vertices and edges between them are considered by the query. One possible approach for this is to relax edges between core vertices in the forward search, which corresponds to performing Dijkstra's algorithm on the core. Alternatively, edges between core vertices can be relaxed in both, the forward search and the backward search, which corresponds to performing a bidirectional search on the core. A third option is to restrict the CH query to settle only vertices that are not part of the core and use an entirely different algorithm to handle the core. An example for this is core-ALT [Bau+10b].

Bucket-CH. The standard CH query algorithm only solves a one-to-one problem, since it is based on a bidirectional search. Solving a one-to-many problem with CHs would thus require multiple queries, or at least multiple backward searches. However, additional preprocessing can be used to improve one-to-many queries, if the targets are known in advance and remain unchanged between queries [Kno+07, GSSV12]. In particular, the backward searches can be performed during the preprocessing phase, such that the query only consists of a single forward search.

The main idea of Bucket-CH is to represent the results of the backward searches with buckets. The goal of this approach is to enable efficient access to the results of the precomputed backward searches during the query. For this purpose the algorithm maintains for every vertex in the graph a bucket, which contains pairs of target vertex and travel time. These buckets can be implemented as dynamic arrays and are empty at the begin of the preprocessing phase. Next, the backward search of the CH query is performed once for every target. Every time that the backward search for the target t settles a vertex v, the algorithm adds t to the bucket of v. Finally, the key of the label of v (which at this point is equal to the minimal travel time from v to t) is used as travel time value for the new bucket entry.

The Bucket-CH query is a modified forward search that interprets the bucket entries as additional edges. To this end, an entry (t, τ_{tra}) in the bucket of some vertex v is interpreted as an edge from v to t with travel time τ_{tra}. Within the query algorithm these edges are handled as follows. Every time a vertex v is settled during the forward search all entries in the bucket are relaxed in addition to the edges and shortcuts in the graph.

3.3.2 Journey Planning in Timetable Networks

We proceed with introducing the public transport journey planning algorithms, which form the basis of the algorithms developed in this thesis. In particular, we build upon ideas from three different public transit algorithms: CSA, RAPTOR, and Trip-Based Routing. All of them operate on a public transit network $N = (\mathcal{C}, \mathcal{S}, \mathcal{T})$ and an additional transfer graph $G = (\mathcal{V}, \mathcal{E})$. However, since they are only public transit algorithms and not multimodal algorithms, they cannot handle arbitrary transfer graphs. In order to be suitable for the three algorithms mentioned above, the vertices of the transfer graph have to be equivalent to the stops of the network ($\mathcal{V} = \mathcal{S}$) and the transfer graph has to be transitively closed.

CSA. The first public transit algorithm that we describe in detail is CSA, which solves the one-to-all earliest arrival problem [DPSW18]. Besides the network data, the input for CSA consists of a source stop and a desired departure time. CSA is based on the observation that there exist only one possible order in which the connections of the network can be used in a journey, i.e., in chronological order. Consequently, the preprocessing phase of CSA inserts all connections of the network into an array, which is then sorted by the departure time of the connections.

The CSA query algorithm uses one label per vertex to maintain earliest arrival times, similar to Dijkstra's algorithm or CHs. Additionally, the query algorithm maintains one reachability flag per trip in the network, which is initialized as not reachable. The algorithm starts with initializing the label of the source stop with the desired departure time and relaxing all outgoing edges of the source stop. Next, a binary search on the sorted connection array is used to find the first connection with a departure time greater or equal to the departure time of the query. Starting with this connection, the algorithm proceeds with scanning all later connections of the network using a single linear sweep over the connection array. The scanning of a single connection is done as follows: First, the algorithm checks whether the connection is reachable. This is the case if the trip of the connection is flagged as reachable. Alternatively, the connection is also reachable if the departure time of the connection is greater or equal to the earliest arrival time of the connection's departure stop, which is maintained by the label of the stop. In the case that the connection is reachable, the following two steps are performed: First, the trip of the connection is flagged as reachable. Secondly, the algorithm checks whether the connection's arrival time is smaller than the current arrival time maintained by the label of the connection's arrival stop. If this is the case then the label of the connection's arrival stop is updated with the connection's arrival time and all outgoing edges of the connection's arrival stop are relaxed. Afterwards, the algorithm continues with scanning the next connection in the sorted array.

CSA terminates after all connections have been scanned. However, if the algorithm is used to solve a one-to-one problem instead of a one-to-all problem, then an earlier termination is possible. In this case the algorithm only needs to scan connections that depart before the minimal arrival time at the target. Thus, the algorithm can stop scanning connections as soon as the departure time of a scanned connection is greater or equal to the earliest arrival time at the label of the target.

Important properties of CSA are that it solves the one-to-all problem and that it builds journeys by using connections as fundamental building blocks. Because of these properties, CSA is particularly well suited for the efficient computation of assignments, as we will see in Chapter 8. CSA can also be extended to compute Pareto-optimal journeys with respect to arrival time and number of trips. However, doing so increases the running time of the algorithm considerably. But this has no effect on the assignment computation, which does not require Pareto-optimal journeys.

RAPTOR. The second public transit algorithm that is relevant to this work is RAPTOR, which solves the one-to-all bicriteria problem [DPW15a]. Just like CSA, the input for the algorithm consists of a source stop and a desired departure time in addition to the public transit network and transfer graph.

RAPTOR operates on a partition of the public transit trips into routes and exploits the fact that only one trip per route has to be considered within optimal journeys. The algorithm operates in *rounds*, where the i-th round finds journeys that use exactly i different trips. It does this by extending journeys that have been found in the previous round with an additional trip. RAPTOR also uses labels to represent the computed journeys, similar to CSA or Dijkstra's algorithm. However, instead of using a single label per vertex, it maintains for every round one label per vertex.

The algorithm starts by initializing the labels for round 0. In particular, this means that the desired departure time of the query is assigned to the arrival time of the source stop's label. Furthermore, the algorithm relaxes all outgoing edges of the source stop. All other labels (i.e., labels that correspond neither to the source, nor to a vertex reachable from the source in the transfer graph) initially have an arrival time of ∞.

After the initialization, the algorithm continues with round one. Each round consists of three steps: collecting routes that where reached in the previous round, scanning these routes, and relaxing transfer graph edges. In order to collect the routes during the first step, RAPTOR keeps track of the stops whose labels were updated in the previous round. The algorithm then collects all routes that contain one of these stops. Afterwards, each of these routes is scanned in the second step of the round. The scan starts with the first stop v of the route that was reached. For this stop the algorithm determines the earliest trip of the route that can be boarded using the arrival time in the stop's label from the previous round. Afterwards, all stops that

follow after v are processed in the order in which they appear in the route. When processing a stop w, the algorithm first checks whether the current trip can improve the arrival time of the stop's label for the current round. If this is the case, the label of w is updated accordingly. Afterwards, the algorithm checks whether an earlier trip of the route can be boarded at w using the arrival time in the label of w from the previous round. If this is the case, the current trip is replaced with the earliest trip that is reachable. Once all routes have been scanned, the algorithm continues with the third step of the round. In this step, the algorithm relaxes the outgoing edges of all stops whose labels were updated during the route scanning. The algorithm terminates after the first round that did not update any stop labels.

The running time of RAPTOR can be improved when solving a one-to-one problem instead of a one-to-all problem. In contrast to CSA and Dijkstra's algorithm, this does not affect the criterion for terminating the algorithm. Instead, the algorithm is altered in such a way that labels are only updated with an arrival time if this arrival time is smaller than the arrival time in the target's label.

Because of its round-based structure the RAPTOR algorithm implicitly solves the bicriteria problem. (Every round finds an optimal arrival time for a different number of used trips.) On realistic inputs, the scanning of a route is a very efficient operation and the number of required rounds is usually small. Thus, the overall algorithm is quite efficient and a good starting point for the development of our multimodal algorithms in chapters 5 through 7.

Range RAPTOR. Range RAPTOR (rRAPTOR) is an extension of the RAPTOR algorithm that can compute profiles. The rRAPTOR algorithm operates in *iterations*, where each iteration performs RAPTOR for one departure time.

In detail, the rRAPTOR algorithm works as follows. First, it collects all meaningful departure times for the source stop. That is, the departure times of trips departing directly from the source and the departure times minus the transfer times of trips departing from stops that are reachable from the source by using the transfer graph. Afterwards, the collected departure times are sorted and the algorithm performs one iteration for each of them, in descending order. An iteration of rRAPTOR essentially consists of performing RAPTOR for the corresponding departure time. However, the labels of RAPTOR are not reset between iterations. Thus, later RAPTOR iterations operate on labels, which contain the arrival times that were computed in earlier iterations. Since rRAPTOR processes departure times in descending order, the arrival times from previous iterations are valid upper bounds for the arrival times of optimal journeys that are computed in subsequent iterations. This technique is called *self-pruning* and ensures that journeys that are not Pareto-optimal because they are dominated by journeys with a later departure time are pruned as early as possible.

Algorithm 3.1: rRAPTOR

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{R})$, transfer graph $G = (\mathcal{S}, \mathcal{E})$, a source stop s, and a departure time interval $I = [\tau_{\text{min}}, \tau_{\text{max}}]$
Output: Profiles from s to all other stops, which cover the intervall I

```
1  Reset the label for every stop and round
2  DT ← Collect departure times at s
3  for each τ_dep ∈ DT in descending order do                    // rRAPTOR iteration
4      S' ← {s}
5      Set the arrival time in the round 0 label of s to τ_dep
6      Relax outgoing edges for stops in S', add updated stops to S'
7      while S' is not empty do                                  // RAPTOR round
8          Collect routes containing stops from S', clear S'
9          Scan routes, add updated stops to S'
10         Relax outgoing edges for stops in S', add updated stops to S'
11     Generate profile entries from round labels
```

High level pseudo-code of rRAPTOR is given in Algorithm 3.1. As mentioned before, the labels used by the algorithm are only reset once per profile query, in line 1. The core of the algorithm are the iterations, which correspond to the loop in line 3. Within each iteration a RAPTOR query is performed, which corresponds to lines 4 through 10. RAPTOR uses the set $\mathcal{S}'$ to keep track of the updated stops and operates in rounds, which correspond to the loop in line 7. Each round consists of three steep: collecting routes, scanning routes, and relaxing edges. The first step (line 8), collects all routes that contain stops from $\mathcal{S}'$ and clears $\mathcal{S}'$. Afterwards, $\mathcal{S}'$ is filled with the stops updated during steps two and three. At the end of each iteration, the journeys found by RAPTOR are added to the profiles, which form the output of the algorithm.

Multimodal Multicriteria RAPTOR (MCR). With MCR, the RAPTOR algorithm has been extended to multimodal scenarios and to more than two optimization criteria [Del+13]. The bicriteria variant of MCR with unrestricted walking (MR-∞) is particularly relevant for this work, since it solves the multimodal journey planning problem, which we consider in Chapter 6. For the most part, the MR-∞ algorithm is equivalent to RAPTOR. The only difference is, that the third step of each RAPTOR round (the relaxation of transfer edges) is replaced with Dijkstra's algorithm. In detail this means that all stops, which are updated during the route scanning step, are added to a Dijkstra queue. Afterwards, Dijkstra's algorithm is performed on the labels of the current round, until the queue is empty. Additionally, MR-∞ uses a Core-CH, where all stops are part of the core, as speed-up technique for the query.

Trip-Based Routing. The last public transit algorithm that this work builds upon is Trip-Based Routing, which solves the one-to-one bicriteria problem [Wit15]. This algorithm is based on a breadth-first search (BFS), which operates on a *trip-based graph* $G^{\mathrm{t}} = (\mathcal{V}^{\mathrm{t}}, \mathcal{E}^{\mathrm{t}})$ specifically constructed for it. The vertices of this graph are the stop events of the public transit network ($\mathcal{V}^{\mathrm{t}} := \{T[i] \mid T \in \mathcal{T}, 0 \leq i \leq |T|\}$) and the edges represent transfers between them. In order to compute this graph, Trip-based Routing requires a small preprocessing phase. During the preprocessing phase, the algorithm first generates all possible edges between stop events, i.e., all pairs of stop events (ϵ, ϵ'), such that transferring between the corresponding stops ($v(\epsilon)$ and $v(\epsilon')$) is possible in the transfer graph. Transferring between the stop events is possible if $\tau_{\mathsf{arr}}(\epsilon) + \delta_\tau(v(\epsilon), v(\epsilon')) \leq \tau_{\mathsf{dep}}(\epsilon')$ holds. Afterwards, a *transfer reduction* step is performed, which discards unnecessary edges (e.g., edges that represent U-turns or edges that can never be part of Pareto-optimal journeys).

In contrast to the other algorithms presented in this chapter, the Trip-Based query algorithm does not compute earliest arrivals for the vertices in the network, but for the trips. Thus, it does not maintain a label for each vertex, but a label for every trip. The label of a trip T keeps track of the trip's last stop event index that has not been processed by the algorithm, and is initially $|T|$. Similar to RAPTOR, the Trip-Based query algorithm operates in rounds, where the i-th round computes journeys that use i trips. For each of the rounds, the algorithm maintains a first-in-first-out (FIFO) queue of trip legs that have to be processed within the round.

The algorithm starts by identifying all trips that are reachable directly from the source stop or from a neighbor of the source stop in the transfer graph. For each of these trips T that is reachable at index i, the trip leg T^{ij} is added to the queue of round one, where j is the index from the label of T. Additionally, the indices in the labels of T and all later trips in the route of T are reduced to $i - 1$.

After all trips that are reachable from the source stop have been added to the queue, the first round starts. Each round of the Trip-Based query algorithm consists of scanning the trip legs in the queue that belongs to the round. Each of the trip legs is scanned by processing its stop events in the order in which they appear in the trip leg. A stop event ϵ is processed in two steps: First, it is checked whether the minimal arrival time at the target can be improved by disembarking from the trip at the stop event and using the transfer graph to reach the target. If this is the case then the algorithm has found a new Pareto-optimal journey, which is added to the result Pareto-set. Secondly, all outgoing edges of ϵ in the trip-based graph are relaxed. When relaxing an edge $(\epsilon, T[i])$, the algorithm adds the trip T with reachable index i to the queue of the next round. This is done in the same way as for the initially reachable trips. After all trip legs in the queue have been scanned, the algorithm proceeds with the next round. The algorithm terminates after the first round that did not add any trip legs to the queue of the next round.

With some minor modifications Trip-Based Routing can also be used to compute one-to-one bicriteria profiles. For this, an approach similar to rRAPTOR is used, where the standard Trip-Based query algorithm is performed once for every possible departure time, in descending order. Just like rRAPTOR, this algorithm does not reset labels between processing different departure times, and thus benefits from self-pruning. However, in order to be correct, the Trip-based profile algorithm requires one label per vertex and round.

4 Benchmark Datasets

The previous chapter formally introduced multimodal transportation networks and how they can be modeled for the purpose of journey planning. In this chapter we consider the practical aspects of public transit network data, which we will need in order to test our algorithms, following the algorithm engineering methodology. To this end, we collected real world timetable and road network data from various sources and show how it is processed into a meaningful and sound benchmark dataset.

4.1 Data Sources

In order to cover a wide range of network types and structures, we collected data from several different sources. All of the networks considered in this work have been used in other works before. We therefore ensure comparability with many other journey planning algorithms.

London. The first network we consider, is the public transit network of the greater region of London. This network was first used to evaluate the RAPTOR algorithm in [DPW12]. Moreover, this network has already been evaluated in a multimodal scenario with the state-of-the-art MCR algorithm in [Del+13]. In order to ensure comparability with the aforementioned algorithm, we use the same timetable data, which covers a single Tuesday of the periodic summer schedule of 2011. It contains most of the public transit available in London, including subways (tubes), buses, and trams. The data was originally extracted from the web presence of Transport for London (TfL)[3], were it is publicly available.

[3] `http://data.london.gov.uk`

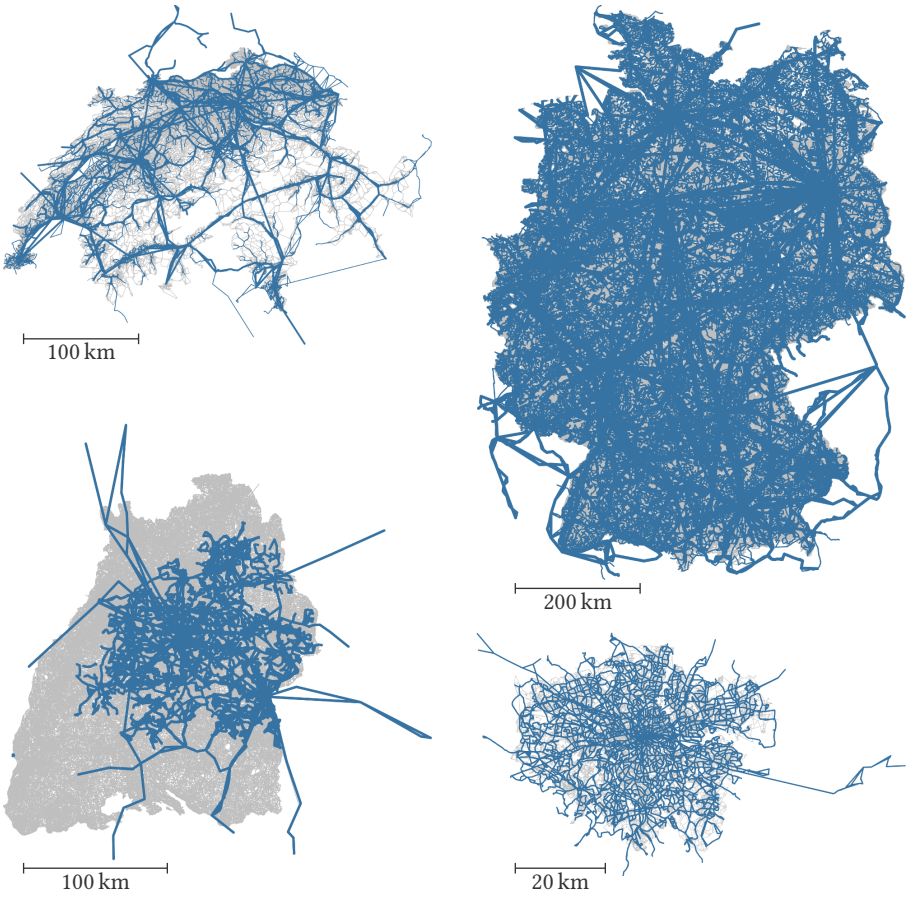


Figure 4.1: The four networks considered in this Work: Switzerland (top left), Germany (top right), Stuttgart (bottom left), and London (bottom right). The edges of the transfer graph are depicted using gray lines. The routes of the public transit network are depicted using blue lines.

Switzerland. The second network we use to evaluate our algorithms is the public transportation network of Switzerland. Similar to the London network, this network also has been considered by many journey planning works before [Bas+10, DDPW15]. The timetable of the Switzerland network used in this work was extracted from a publicly available GTFS feed[4], which contains data provided by the federal office for traffic

[4] http://gtfs.geops.ch/

in Switzerland. The data contains a wide variety of public transit types, including local buses and trams, regional trains, express trains, and even more unusual means of transportation like rack railways and ski lifts. From this data, we extracted two successive business days: the 30th and the 31st of May 2017, which were also used in [WZ17].

Germany. We use the public transportation network of Germany as a third benchmark instance. The timetable data of the Germany network which we consider in this work was first used to evaluate CSA in [SW14]. The network data was provided by Deutsche Bahn (DB)[5] for research purposes and is unfortunately not publicly available. Just like the Switzerland network, the Germany network also contains most of the available public transportation, including local, regional, and long distance traffic. The timetable data used in this work covers two successive days from the winter 2011/2012 schedule. However, we ignore the days of operation of the individual vehicles (i.e., that trips of all days are merged). We do this to ensure comparability, as the same approach was used in [SW14] and other works.

Stuttgart. The last network considered in this work, is a network covering the greater region of Stuttgart in Germany. While this network is in theory a subset of the Germany network, we still include it as its own instance, as it provides additional data, that is not available to us for the entirety of Germany. Namely, we have real world estimates for the overall passenger flow through this network, which will be important for the evaluation of assignment algorithms in Chapter 8. This data includes all passengers traveling through Stuttgart, also including commuters. Thus, the network does not only contain local buses and trams, but also regional trains and long distance trains reaching as far as Frankfurt, Munich, or Zürich. The timetable data of the network covers two identical business days and was previously used in [MKV13]. The network was originally presented in [SHP11] and is not publicly available.

Transfer graphs. We extracted transfer graphs from Open Street Map (OSM)[6] to accompany our four networks. To this end, we gathered data on roads, pathways, pedestrian zones, and staircases together with their length and speed limits (if they exist). This data is then used to construct transfer graphs with varying travel speeds, depending on the mode of transportation that we want to represent (e.g. walking or cycling). For our four networks we extracted data of the regions: Greater London, Switzerland, Germany, and the state of Baden-Württemberg. The extent of the first three regions matches quite well with the corresponding public transit network. However, the transfer graph for the Stuttgart network covers an area that is quite a bit larger than the public transit network. Due to the special structure of the Stuttgart network (dense in the center, incomplete in the surroundings), this is still the best fit. The overall shape and extent of the resulting four multimodal networks is shown in Figure 4.1.

[5] http://www.bahn.de/

[6] https://download.geofabrik.de/

4.2 Additional Preparations and Sanitizing

Since we create our networks from real world data, it is to be expected that they contain some irregularities or even erroneous data. We therefore sanitize all networks before we work with them, such that the resulting data conforms to our network definition. This step also includes the implementation of the connection between the public transit networks and the transfer graphs.

Preparation Procedure. An outline of all the steps performed during our data preparation is given in Algorithm 4.1. As input we use the timetable of the network, represented as a set of trips $\mathcal{T}'$, and the stops $\mathcal{S}'$, for which the data source provides additional information like coordinates and minimum change times or departure buffer times. In addition to this, we also use the transfer graph G', that was extracted from OSM, and a bounding box B as input.

The first part of the preparation procedure (lines 1 - 6) sanitizes the timetable data. First, we remove trips from the network that contain undefined stops (e.g. where the coordinates are unknown) or that leave the specified bounding box. We do this, since we are interested in the performance of algorithms on densely interconnected networks. Therefore, single trips leaving the main network are not of interest to us. An example for a trip removed by this, is the Germany network, which originally contained a single trip from Berlin to Moscow. We further reduce the set of trips by removing all trips that would allow for time travel in line 2. These trips are removed, as their data is obviously incorrect. Moreover, they have to be removed because many journey planning algorithms simply break down, if a network contains such trips. The set of trips is thinned out one final time in line 3, where we reduce the timetable to its maximal connected component. We do this, because we are interested (as mentioned before) in the performance of algorithms on densely interconnected networks. Having parts of the network that are not reachable at all might distort performance measurements. Therefore we remove such parts from the network. As a last step in the preparation of the trips, we adjust their departure times to implicitly reflect departure buffer times as outlined in Section 3.1.5. Next, we collect all stops actually used by some trip into the set $\mathcal{S}$. We do this mainly for comparability reasons, since the original set of stops $\mathcal{S}'$, which is provided as input, often contains unused stops, which let the network appear larger than it actually is. The preparation of the public transit network concludes with the partitioning of the trips into routes in line 6. For this, we use a simple greedy approach: For each trip T we check if we have already created a route with the same stop sequence, such that T can be added to the route without overtaking any of the trips in the route or being overtaken by any of them. If such a route exists, we add the trip T to it, otherwise we create a new route for T.

Algorithm 4.1: Data Preparation

Input: Set of stops $\mathcal{S}'$, set of trips $\mathcal{T}'$, graph $G' = (\mathcal{V}', \mathcal{E}')$, and bounding box B
Output: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{R})$ and a transfer graph $G = (\mathcal{V}, \mathcal{E})$

1 $\mathcal{T} \leftarrow \mathcal{T}' \setminus \{$Trips with stops not in $\mathcal{S}'$ or $B\}$
2 $\mathcal{T} \leftarrow \mathcal{T} \setminus \{$Trips that arrive at a stop before departing from a preceding stop$\}$
3 $\mathcal{T} \leftarrow \{$Maximal subset of $\mathcal{T}$ that is connected$\}$
4 $\mathcal{T} \leftarrow \{$Trips in $\mathcal{T}$ with τ_{buf} subtracted from their departure times$\}$
5 $\mathcal{S} \leftarrow \{$Stops used by trips in $\mathcal{T}\}$
6 $\mathcal{R} \leftarrow$ Greedy partitioning of $\mathcal{T}$ into routes

7 $\mathcal{V} \leftarrow \{$Vertices from $\mathcal{V}'$ that are within $B\}$
8 $G \leftarrow$ Maximum connected component of the subgraph of G induced by $\mathcal{V}$

9 **for each** $v \in \mathcal{S}$ **do**
10 $\quad w \leftarrow$ Nearest vertex to v from $\mathcal{V} \setminus \mathcal{S}$
11 $\quad x \leftarrow$ Nearest stop to w from $\mathcal{S}$
12 $\quad$ **if** $v = x$ **and** $\|v - w\| < 5\,\text{m}$ **then**
13 $\quad\quad$ Replace w with v (in $\mathcal{V}$ and in all $e \in \mathcal{E}$)
14 $\quad$ **else**
15 $\quad\quad$ Add v to $\mathcal{V}$
16 $\quad\quad$ **if** $\|v - w\| < 100\,\text{m}$ **then**
17 $\quad\quad\quad$ Add (v, w) and (w, v) to $\mathcal{E}$

18 Contract degree 2 vertices (excluding stops) in G

The second part of our network preparation (lines 7 - 8) affects the transfer graph. Similar to the processing of the timetable data, we start by removing all vertices that are not located within the bounding box. Afterwards, we look at the graph that is induced by the remaining vertices. In this graph, we compute a strongly connected component of maximal size, which we use as our final transfer graph G. As we mentioned before, this is done in order to simplify the analysis and evaluation of the algorithms presented in this work.

During the third step of the preparation phase (lines 9 - 17), we establish the connection between the public transit network and the transfer graph (i.e., we ensure that $\mathcal{S}$ is a subset of $\mathcal{V}$). For this purpose we identify for every stop $v \in \mathcal{S}$ its nearest (regarding linear distance) vertex $w \in \mathcal{V}$, which can be done efficiently using a k-d tree [Ben75]. If the nearest stop to the vertex w is also v and their distance is less than 5 meters, then we assume that v and w designate the same location.

We implement this formally by replacing the vertex w in G with the stop v in line 13, thereby ensuring that $v \in \mathcal{V}$ holds. During this process it is quite important to check whether v and w are mutual nearest neighbors, since v could otherwise be replaced by another stop later on. In the case, that we cannot identify the stop v directly with an existing vertex in $\mathcal{V}$, we add v as a new vertex to $\mathcal{V}$. Afterwards, we check if the linear distance to the nearest vertex w of v is less than 100 meters. If so, we connect w and v with two new edges.

As the final step of the network preparation we contract vertices of degree two in the transfer graph (line 18). We do this in order to get a meaningful value for the size of the transfer graph. Since we use OSM, which primarily focuses on map visualization, as the source of our transfer graph, we expect quite a large number of vertices with degree two. These vertices are used to model the shape of streets (e.g. in curves or serpentines). However, they have only a minor impact on the difficulty of journey planning problems, because if one of them is part of a shortest path, their neighbors are also part of that shortest path. The only exception to this rule occurs if a vertex of degree two is the first or the last vertex in a shortest path. Therefore, a graph with many vertices of degree two might look quite large, while solving a shortest path problem within the graph is relatively easy. This phenomenon is discussed in detail in [DSW15].

Resulting Dataset Overview. The sizes of the four multimodal networks, after they have been prepared as described above, are listed in Table 4.1. The visualizations of the networks in Figure 4.1 (page 48) also corresponds to the networks after they have been sanitized. Please note, that sizes of the networks used in this work may differ slightly from the ones used in other works, even if they are based on same data source. The reason for this is of course the network preparation, which removes some of the trips and stops from the networks. However, we argue that this does not hinder comparability with other works, as the networks are structurally still the same. The intended effect of the network preparation used in this work is, that the resulting network sizes, which are reported in Table 4.1, reflect the true dimension and complexity of the networks more closely.

When comparing the sizes (Table 4.1) and the overall structure (Figure 4.1 on page 48) of the networks, we observe that the four networks are quite different. Obviously, the two city networks are much smaller than the two country networks. But even within both of these network classes there are huge differences. Geographically, the Stuttgart network is much larger than the London network, which is also reflected in its larger transfer graph. However, London is of course the larger city, and thus its network contains significantly more public transit stops and trips. Overall, the network of London is much denser and more interconnected than the Stuttgart

Table 4.1: Sizes of the public transit networks that are considered in this work. Additionally, we report the number of transfers that are available with and without using a transfer graph from a different source. First, we state the number of *Original Transfers*, which were extracted from the source of the respective public transit network. Secondly, we state the number of vertices and edges in the *Transfer graph*, which we extracted from OSM.

	Stuttgart	London	Switzerland	Germany
Stops	13 583	20 595	25 125	243 071
Routes	12 350	2 107	13 785	230 216
Trips	91 298	125 436	350 006	2 380 966
Stop events	1 561 912	4 970 428	4 686 865	48 368 190
Connections	1 470 614	4 844 992	4 336 859	45 987 224
Original transfers	33 500	44 840	12 806	92 748
Transfer graph vertices	1 166 593	183 025	603 691	6 870 354
Transfer graph edges	3 680 930	579 888	1 853 260	21 366 756

network. Even the size of public transit network of the whole country of Switzerland is roughly comparable to the London network. Its structure, on the other hand, is vastly different and unique among the four networks. The Switzerland network is the only network that is quite inhomogeneous, with a relatively dense northern part and a much more loosely interconnected south (due to the Alps). Lastly, the Germany network is roughly a factor of 10 larger than the Switzerland network and has a relatively uniform structure in comparison. Overall, we observe that the four networks cover a broad spectrum of different network types and structures. This indicates that the presented collection of networks offers a good test bed for analyzing and comparing journey planning algorithms.

In addition to the size of the transfer graphs extracted from OSM, Table 4.1 also report the number of original transfers, that were specified by the source of the public transit network. These transfers are only defined for pairs of stops and do not involve any additional vertices. Thus $\mathcal{S} = \mathcal{V}$ holds for the transfer graph $G = (\mathcal{V}, \mathcal{E})$ if the original transfers are used. The original transfers are used, when we are not interested in a multimodal scenario, but want to consider a pure public transit network instead. This is useful for the comparison with algorithms that were not developed for multimodal scenarios and thus have only been evaluated using the original transfers. Moreover, we will use the original transfers in Chapter 5, in order to evaluate the impact of multimodal scenarios.

4.3 Transitively Closed Instances

Many public transit journey planning algorithms support the usage of transfer graphs, but require some sort of limitation. A common restriction limits the maximal duration of transfers that can be part of a journey [BHS16, BS14, DDPW15]. Another frequently used approach for limiting transfers is to require that the transfer graph is transitively closed [DPW15a, DPSW18, Wit15]. In this section we evaluate, how much of the transfer graph, which we introduced in the previous section, can be preserved if the transfers have to be limited. Wo do this for the special case that the transfer graph represents walking (assuming an average walking speed of 4.5 km/h), as this is the most common scenario in the literature. To this end, we analyze how the size of a transitively closed transfer graph changes, depending on the maximal walking duration that should be preserved. Afterwards, we construct transitively closed transfer graphs with large, but still feasible, walking limits for all four networks. These graphs will be used in the following chapters to analyze algorithms that require a limited transfer graph. This section is based on joint work with Dorothea Wagner [WZ17].

Transitively Closed Graphs with Guaranteed Walking Time. Let $G = (\mathcal{V}, \mathcal{E})$ be a transfer graph with transfer times $\tau_{\text{walk}} : \mathcal{E} \to \mathbb{R}_0^+$ that represent walking. Furthermore, let $\bar{\tau}_{\text{walk}} \in \mathbb{R}_0^+$ be the maximal walking time that should be guaranteed. From these, we want to construct a new transfer graph $G' = (\mathcal{S}, \mathcal{E}')$ that is transitively closed, has as few edges as possible, and preserves all walking times up to $\bar{\tau}_{\text{walk}}$. That is, for all pairs of stops $v, w \in \mathcal{S}$ with a walking distance $\tau_{\text{walk}}(P) \leq \bar{\tau}_{\text{walk}}$ (where P is the shortest v-w-path in G) there exists an edge $e = (v, w) \in \mathcal{E}'$ with $\tau_{\text{walk}}(e) = \tau_{\text{walk}}(P)$. Furthermore, if an edge $e = (v, w) \in \mathcal{E}'$ exists, then $\tau_{\text{walk}}(e) \geq \tau_{\text{walk}}(P)$ has to hold, where P is once again the shortest v-w-path in G. The graph G' can be constructed quite easily as follows: First we add an edge $e = (v, w)$ to $\mathcal{E}'$ for every pair of stops $v, w \in \mathcal{S}$ with a walking distance in G that is less than or equal to $\bar{\tau}_{\text{walk}}$. Afterwards, we compute the transitive closure of these edges.

Figure 4.2 shows the size and structure of the resulting graph G' depending on the guaranteed walking time $\bar{\tau}_{\text{walk}}$ for our four networks. The figure reveals that the number of edges needed for the transitive closure increases drastically with the guaranteed walking time. However, even with a relatively high guaranteed walking time, the graphs still consist of many distinct connected components. The relation between guaranteed walking time and the structure of the graph is quite similar for all four networks. With all networks, we observe a significant change, for both number of edges and number of isolated stops, if the guaranteed walking time is increased from 0 to values slightly above 0. The reason for this are sets of stops, which together model a larger station, e.g. the platforms of a train station or two bus stops on opposing sides of a street. This effect is especially pronounced for the

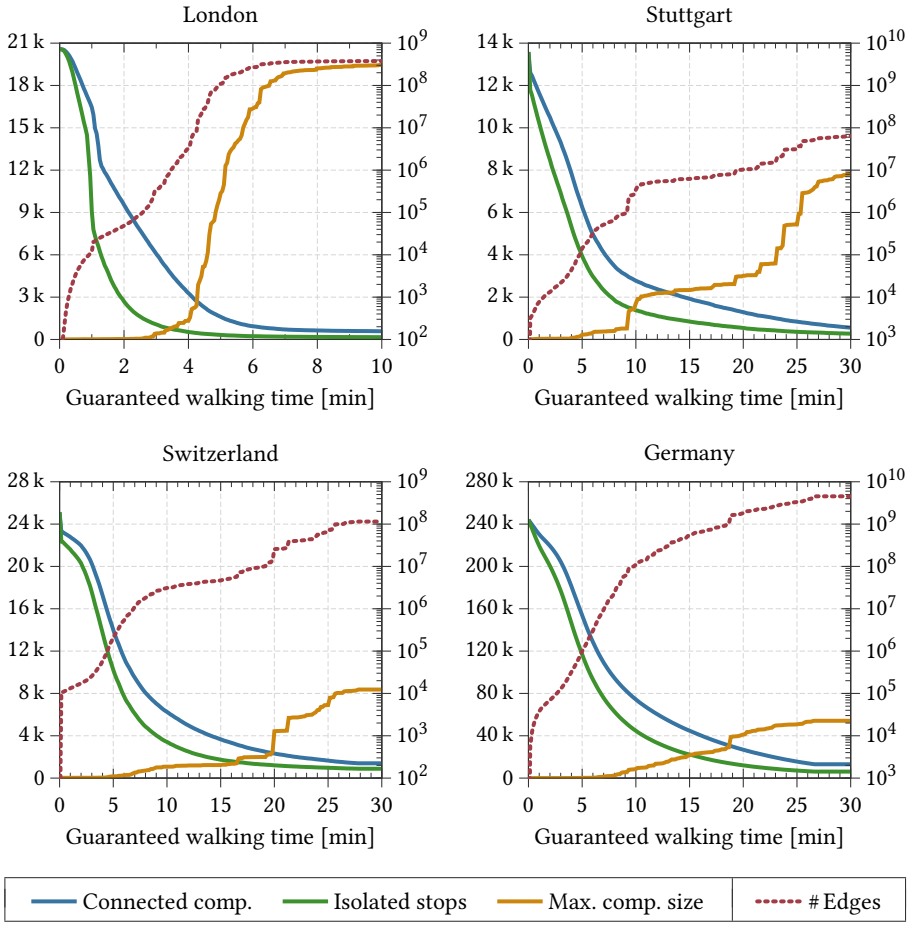


Figure 4.2: Sizes of the transitively closed transfer graphs depending on the guaranteed walking time. Solid lines are plotted using the left y-axis, dotted lines use the right y-axis. The transitively closed transfer graph is constructed as follows: Two stops $v, w \in \mathcal{S}$ are connected with an edge $e = (v, w)$ if the walking time of the shortest path from v to w is less than or equal to the guaranteed walking time. Afterwards, the transitive closure is computed. The plots show that the number of connected components (blue) remains high, even if the threshold for walking is rather high. Many stops are not connected to any other stops (green) and even the largest connected component remains comparatively small (yellow). However, the number of edges required for the transitive closure (red) increases drastically with the guaranteed walking time.

Table 4.2: Sizes of the transitively closed transfer graphs. The guaranteed walking time was chosen as an integral number of minutes, such that the mean stop degree is close to 100. We also report the number of stops that are still isolated.

Measured value	Stuttgart	London	Switzerland	Germany
Guaranteed τ_{walk}	9 min	4 min	9 min	8 min
Number of edges	945 514	3 755 200	2 639 402	23 880 322
Mean stop degree	69.6	182.3	105.5	97.8
Isolated stops	1 575	529	4 023	62 659

Switzerland network. After this initial jump, the connectivity of the network increases comparably smoothly with an increasing guaranteed walking time. An exception to this pattern is the London network, were the connectivity of the transitively closed graph increases drastically at a guaranteed walking time of about 5 minutes. This indicates that the public transit network of London is much denser than the public transit networks of the other three instances.

In order to properly evaluate the algorithms presented in this work, we need to compare them with existing algorithms, which often require a limited transfer graph. For this purpose we construct a transitively closed transfer graph that contains as much of the original graph's structure as possible, while still being feasible (In terms of memory consumption of the transfer graph and running time of the algorithms). As shown by Figure 4.2, this requires a severely limited guaranteed walking time. For this work, we decided to construct transitively closed transfer graphs with an average vertex degree of about 100. This results in graphs that are much denser than the graphs typically used in journey planning, while still being feasible. The sizes of the resulting graphs for the four networks are reported in Table 4.2. For most networks we can use a guaranteed walking time of 8 or 9 minutes. Only the London network requires a much lower guaranteed walking time of only 4 minutes. As mentioned before, this is due to the London network being much denser than the other networks.

Partial Non-Transitive Transfers. As an alternative to the transitively closed transfer graph one could impose a limit on the transfer time without considering the transitive closure. While such an approach drastically reduces the number of edges required to represent the transfers, it too has severe disadvantages. The biggest downside is, that journeys found by such an approach can be inconsistent. Figure 4.3 illustrates this using a small example network and a limit for the transfer time of 10. Within this network there exist three possible s-t-Journeys. However, the best journey (J_3) requires two transfer edges ((v, w) and (w, x)) with a total transfer

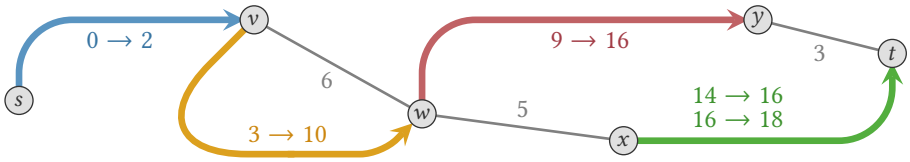


Figure 4.3: Effect of transfers that are not transitively closed. There exist three possible s-t-journeys in this network: $J_1 = \langle\langle s\rangle, \langle 0 \to 2\rangle, \langle v, w\rangle, \langle 9 \to 16\rangle, \langle y, t\rangle\rangle$, with an arrival time of 19, $J_2 = \langle\langle s\rangle, \langle 0 \to 2\rangle, \langle v\rangle, \langle 3 \to 10\rangle, \langle w, x\rangle, \langle 16 \to 18\rangle, \langle y, t\rangle\rangle$, with an arrival time of 18, and $J_3 = \langle\langle s\rangle, \langle 0 \to 2\rangle, \langle v, w, x\rangle, \langle 14 \to 16\rangle, \langle y, t\rangle\rangle$, with an arrival time of 16. The journey J_3 dominates the other two journeys. However, if transfers are limited to maximum duration of 10 and if transfers are not transitively closed, then J_3 is not found.

time of 11. Thus, a journey planning algorithm would only report journeys J_1 and J_2. Looking at these two journeys, it becomes obvious that it is possible to transfer from v to w and that it is also possible to transfer from w to x. Therefore, the question arises why the combined transfer from v to x has not been considered. Another inconsistence can be observed if a passenger decides to follow the journey to J_1. If such a passenger recomputes the optimal journey while waiting for the next trip at w, the algorithm will suddenly find a journey $J_4 = \langle w, x\rangle, \langle 14 \to 16\rangle, \langle y, t\rangle\rangle$. This again raises the question why this option was not found initially.

These problems can of course be solved by a simple search for transfers in the non transitive graph, instead of only using single edges. However, many algorithms have not been designed to consider such a search. Moreover, an additional search in the transfer graph will increase the overall running time of the algorithm. Because of all these reasons, we only consider transitively closed transfer graphs as means to limit transfers in this work.

5 Multimodal Profiles

In this chapter we address the problem of answering one-to-one profile queries in multimodal networks. In particular, we consider public transit networks that allow for unrestricted walking between the stops of the network. Given a source location, a target location, and a departure time interval, we want to find a Pareto-set of journeys with respect to travel time and number of transfers for every departure time in the interval. In order to solve this problem, we introduce a novel profile algorithm that, unlike most state-of-the-art algorithms, can compute profiles efficiently in a network that allows for arbitrary walking. Using our algorithm, we show in an extensive experimental study that allowing unrestricted walking, significantly reduces travel times, compared to settings where walking is restricted. This chapter is based on joint work with Dorothea Wagner [WZ17].

Related Work. To the best of our knowledge, no efficient profile algorithm for multimodal networks exists. However, many different profile algorithms have been developed for public transit networks. Notable examples of such algorithms are the Self-Pruning Connection-Setting algorithm [DKP12], which is a graph based approach, and rRAPTOR [DPW15a], which is the profile variant of the RAPTOR algorithm. Other timetable-based algorithms, such as CSA [DPSW18] and Trip-Based Routing [Wit15] can also be used to compute profiles.

All of these algorithms support networks that combine a public transit timetable with a transfer graph for walking between stops. However, none of the algorithms is suitable for transfer graphs where walking is possible between arbitrary pairs of stops. A common restriction imposed by journey planning algorithms for public

transportation networks is the requirement that the transfer graph has to be transitively closed. One of the first techniques based on this restriction is the RAPTOR algorithm [DPW15a]. A transitively closed graph is also required for CSA [DPSW13] and its accelerated version [SW14]. Another technique depending on transitively closed footpaths is trip-based public transit routing [Wit15].

Other approaches for public transit journey planning do not explicitly state requirements on the transfer graph. However, the problems arising from detailed transfer graphs are often neglected. Either the used transfer graph is not specified, or the algorithms are only evaluated on rather sparse and unconnected transfer graphs. In both cases it is unknown how the techniques would perform on a public transit network that features a complete transfer graph, which connects most stops in the network. An example of a technique where no information about the size of the used transfer graph is known, was presented in [DMS08]. Another technique where the used transfer graph is not specified, is transfer patterns [Bas+10]. However, for the accelerated version of this technique, which is called scalable transfer patterns [BHS16], it was specified that stops are connected by a footpath if their distance is below 400 meters. This corresponds to a walking time of 8 minutes or less (assuming a walking speed of 4.5 km/h), which leads to a rather sparse transfer graph. Similarly, frequency-based search for public transit networks [BS14] was only evaluated using a limited number of transfers. Here, two variants, one allowing up to 5 minutes walking, the other up to 15 minutes, were evaluated. Even fewer footpaths are considered in works that only consider the transfers specified in the source of the public transit network. This is the case for PTL [DDPW15], SUBITO [BGM10], or the graph based techniques presented in [PSWZ08]. Finally there are algorithms, like delay robust routing using MEAT [DSW14] or CSA accelerated [SW14], that omit footpaths altogether.

Self-Pruning Profile Algorithms. A common approach used by many efficient profile algorithms for public transit networks is *self-pruning*. It is based on the observation that a profile cannot contain more journeys than the number of trips departing from the source (each departing trip is part of at most one journey). Thus, algorithms based on self-pruning simply collect all possible departure times at the source stop (and at stops that are reachable from the source by walking). Afterwards, the self-pruning algorithm proceeds by computing optimal journeys for each of these departure times in descending order. During the repeated computation of the journeys for different departure times, the labels of the algorithm are not cleared. Since journeys are searched in decreasing order regarding departure time, labels of the previous search can be used to prune the current search, which leads to very efficient algorithms. Examples for algorithms based on this approach are the Self-Pruning Connection-Setting algorithm [DKP12], rRAPTOR [DPW15a], and the profile

variant of Trip-Based Routing [Wit15]. However, this approach loses its efficiency in networks that allow unrestricted walking. In such networks every stop can be reached by walking from the source. Therefore, journeys have to be computed for every departure in the entire network, which is not feasible.

Chapter Overview. In this chapter, we reevaluate the common practice of imposing restrictions on the transfer graph. To this end, we present a novel algorithm that can compute profiles for public transit networks with unrestricted walking. Using this algorithm we can efficiently evaluate the travel times between given source and target stops over the course of a whole day. In order to evaluate the practicality of our approach and the impact of unrestricted walking we compare three network variants: The first variant uses a transfer graph that only contains transfers specified by the source of the public transit network. The second variant uses additional transfers, which are chosen such that the transitively closed graph still has a practical size. The third variant uses an unrestricted transfer graph. By evaluating the same set of profile queries for all variants of the network, we show that travel times are significantly improved by allowing unrestricted walking.

5.1 Profile Algorithm

We now introduce our new profile algorithm for public transit networks with unrestricted walking, which is based on MCR [Del+13]. As mentioned before, we cannot use self-pruning based approaches, such as rRAPTOR, since every trip in the network could potentially be the first trip of an optimal journey. However, we can still use repeated executions of a multimodal journey planning algorithm, such as MCR, in order to compute a complete profile. We begin by describing a simplified variant of our algorithm, that only computes earliest arrival profiles. Afterwards, we show how this basic approach can be extended to compute bicriteria profiles.

5.1.1 Earliest Arrival Profiles

In the following, we assume that source and target vertices $s, t \in \mathcal{V}$, as well as a time interval $I = [\tau_{\text{min}}, \tau_{\text{max}}]$ are given. In order to compute the s-t-profile for the interval I we start with one execution of MCR with τ_{min} as departure time. As result of this query we obtain a journey with minimal possible arrival time τ_{arr} at the target t. However, we do not know the travel time of this journey, since we do not know the latest departure time from s that still allows to reach t at τ_{arr}. We determine the latest possible departure time from s by performing a backward MCR query from t, starting with the arrival time τ_{arr}. As result of these two queries we know one pair of

departure time τ_{dep} and arrival time τ_{arr}, such that τ_{arr} is the earliest possible arrival time at t if departing from s at τ_{min} and τ_{dep} is the latest possible departure time at s that allows to reach t at τ_{arr}. Therefore, the pair $(\tau_{\text{dep}}, \tau_{\text{arr}})$ is the first entry of the profile function from s to t. Furthermore, the s-t-profile is already complete for the interval $I' = [\tau_{\text{min}}, \tau_{\text{dep}}]$. This means that we now only need to compute the profile for the interval $I'' = [\tau_{\text{dep}} + \varepsilon, \tau_{\text{max}}]$, where the departure time $\tau_{\text{dep}} + \varepsilon$ indicates that a potential passenger just missed the journey that departs at τ_{dep}.

The remaining profile for I'' can now be computed using the same approach as for the original interval I. We repeat this process, until we are left with an empty interval. In particular, this means that the backward search results in a departure time τ_{dep} that is greater or equal to the maximum departure time τ_{max} of the interval.

Figure 5.1 illustrates one cycle of forward and backward search of our algorithm. Initially (left plot), the profile function has already been computed for the interval [0:00,2:00]. Thus, the earliest arrival time for the departure time 2:00 + ε has to be computed. In our example, this arrival time is 5:00, which establishes the y-coordinate of the next breakpoint of the profile (middle plot). The algorithm proceeds with a backward search for the arrival time 5:00, which yields 3:00 as latest departure time. Thus, the breakpoint (3:00, 5:00) is added to the profile (right plot) and the algorithm proceeds with the next cycle, i.e., a forward search for the departure time 3:00 + ε.

Direct Walking. The profile algorithm we described so far will perform exactly one forward and backward query for every entry of the profile. However, the approach fails if an optimal s-t-journey contains no trips at all, i.e., the optimal journey corresponds to walking directly from s to t. In this case the forward search started for a departure time of τ_{dep} will result in an arrival time of τ_{arr}. Afterwards a backward search is performed starting with the arrival time τ_{arr}. This backward search will then result with the latest possible departure time being τ_{dep}. This means the size of the interval did not decrease, except by an ε. Even worse, repeating the procedure for a departure time of $\tau_{\text{dep}} + \varepsilon$ will have the same result. In order to solve this issue we use a slightly modified version of the basic query algorithm (in our case MCR). We demand that the query algorithm only returns journeys that contain at least one trip, i.e., direct walking from s to t is prohibited. This can easily be achieved by pruning the initial exploration of the footpaths graph (within MCR) if it reaches t. Apart from this, the profile algorithm remains for the most part unchanged. As before we perform alternating forward and backward searches in order to determine one profile entry at a time. However, the resulting profile might contain entries that are dominated by a pure walking journey. We remove these entries in a simple postprocessing step. For this we compute the walking time from s to t using Dijkstra's algorithm [Dij59]. Afterwards we remove all entries with a travel time that exceeds the walking time from the profile.

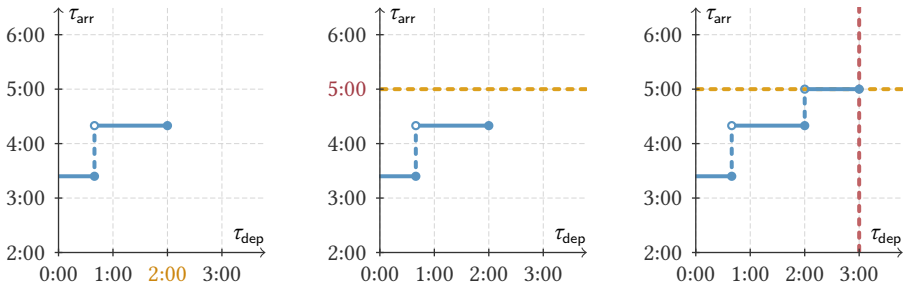


Figure 5.1: An example depicting one iteration of our profile algorithm. *Left*: The profile (blue) is already known for the interval [0:00, 2:00]. Thus, the next forward query uses $2{:}00 + \varepsilon$ as departure time. *Middle*: The forward query provides the arrival time of the next profile entry (marked by the yellow line). In this example the arrival time is 5:00, which is used as input for the backward query. *Right*: The backward query provides the departure time of the next profile entry (marked by the red line). The new entry is added to the profile, which is now known for the interval [0:00, 3:00].

5.1.2 Pareto Profiles

So far we have shown how an earliest arrival profile can be computed. However, besides arrival time, the number of transfers is another important property of a journey. Thus, a profile that does not only contain all journeys with minimal arrival time, but all journeys that are Pareto-optimal with respect to arrival time and number of transfers is often desired. Both, RAPTOR and MCR, naturally support queries that compute all Pareto-optimal journeys (regarding arrival time and number of transfers) for given source vertex, target vertex, and departure time. Thus, we only have to adapt our profile algorithm so that it can take all Pareto-optimal journeys found by MCR into account. As before, the algorithm starts with a forward search from s for the departure time τ_{min}, when computing a profile for the interval $I = [\tau_{min}, \tau_{max}]$. The result of this forward query is a set of Pareto-optimal journeys, containing up to one journey for every possible number of transfers. Each of these journeys has a different arrival time and eventually we will perform one backward query for each of these arrival times. We use a priority queue to organize all arrival times for which we still have to perform a backward search. As long as this queue is not empty, our algorithm extracts the minimum arrival time τ_{arr} and performs a backward search starting from the target with τ_{arr} as arrival time. As before, the result of the backward search is a departure time τ_{dep}, which, together with τ_{arr}, defines an entry of the profile. Following the backward search, we perform a forward search with departure time $\tau_{dep} + \varepsilon$, which possibly adds new arrival times to the queue. The advantage of this procedure is, that

one backward search can potentially generate several profile entries. This is the case if several Pareto-optimal journeys differ only in their number of transfers, but have the same arrival time. Thus, all these journeys can be found by a single backward search.

Implementation Details. In our implementation of the profile algorithm we use MCR for the forward and backward queries. However, the general approach of our algorithm can be used together with any algorithm that computes optimal s-t-journeys for a fixed departure time. The performance of our algorithm depends on the number of entries in the computed profile and the performance of the underlying query algorithm. More precisely, the underlying query algorithm will be invoked at most twice for every entry added to the profile.

The efficiency of the backward searches can be increased by implementing additional pruning rules. In particular, we propose two pruning strategies: The first uses the labels of the corresponding forward search, while the second uses the labels of the past backward searches. In detail, our first pruning strategy is based on a common approach for computing journeys with minimal travel time, which works as follows. The search can be pruned at a vertex v if the latest departure found by the backward search for v is smaller than the earliest arrival time at v found by the forward search. In this case we can conclude that a journey with minimal travel time cannot contain v and therefore the backward search does not need to settle v.

Our second pruning strategy for the backward search is based on the self-pruning approach. We observe that arrival times for which we perform backward searches are monotonically increasing. Thus, the latest departure times computed by a backward search should always be greater than the latest departure times computed by past backward searches. We exploit this fact by not resetting the labels of the backward search during the profile computation. This is equivalent to the implementation of self-pruning in other profile algorithms, such as rRAPTOR [DPW15a].

5.2 Experiments

We implemented our algorithm in C++17 compiled with GCC version 8.2.1 and optimization flag -O3. Experiments were conducted on a machine with two 8-core Intel Xeon Skylake SP Gold 6144 CPUs clocked at 3.50 GHz, with a boost frequency of up to 4.2 GHz, 192 GiB of DDR4-2666 RAM, and 24.75 MiB of L3 cache. Before we continue with the performance analysis of our algorithm, we provide a detailed description of the networks and the queries, which we used in our experiments. Afterwards, we conduct an extensive comparison of the profiles computed for our example networks, showing that many journeys benefit from unlimited walking.

Table 5.1: Comparison of the transfer graph sizes of the three network variants. The graphs are constructed following the approach from Chapter 4. The vertices of the *Original* and *Transitive* variant only contain the public transit stops, while the *Multimodal* graph also contains vertices that represent additional locations in the network.

Measured value	Stuttgart	London	Switzerland	Germany
Original $\|\mathcal{V}\|$	13 583	20 595	25 125	243 071
Original $\|\mathcal{E}\|$	33 500	44 840	12 806	92 748
Transitive $\|\mathcal{V}\|$	13 583	20 595	25 125	243 071
Transitive $\|\mathcal{E}\|$	945 514	3 755 200	2 639 402	23 880 322
Multimodal $\|\mathcal{V}\|$	1 166 593	183 025	603 691	6 870 354
Multimodal $\|\mathcal{E}\|$	3 680 930	579 888	1 853 260	21 366 756

Network Variants. We evaluate our novel profile algorithm on all four networks that were introduced in Chapter 4. In order to analyze the impact of unlimited walking, we consider three different transfer graphs, which provide various levels of connectivity, for each of the four networks. The number of vertices and edges in these graphs are listed in Table 5.1. All graphs represent walking as transfer mode, where an average walking speed of 4.5 km/h is assumed. The first transfer graph variant, which we call *original*, is the transitive closure of the transfer edges that are specified by the source of the public transit data. Therefore, the vertices of this graph only represent the stops of the public transit network. The second variant, which we call *transitive*, corresponds to the transitive closed graphs, which we described in Section 4.3. The idea behind these graphs is to connect as many stops as possible while the size of the graph still remains feasible for public transit journey planning algorithms. Our last transfer graph variant, is the *multimodal* variant, which contains all roads and pathways that were available in OSM. These graphs contain many more vertices than the first two variants and connect most pairs of public transit stops via walking paths. Using this graph variant in combination with the public transit networks yields a truly multimodal scenario, which can no longer be handled by public transit algorithms.

Queries and Experimental Setup. We want to analyze how the results of realistic queries change with respect to the three variants of our networks. A query can of course only be evaluated for all three network variants if the source and target of the query are part of all three network variants. Thus, we only consider queries, where the source and target vertices are actual stops, as additional vertices of the 'multimodal' transfer graph are not contained in the 'original' and 'partial' graphs. Our algorithm can of course handle arbitrary source and target vertices.

Another important problem regarding the evaluation of public transit routing algorithms that we have not yet addressed, is the generation of representative queries. An approach that is commonly used in the literature generates random queries where source and target stops are picked uniformly at random. However, this approach does not reflect query distributions that can be expected in real applications. It is reasonable to assume that the users of a real application will predominantly make search queries where the source stop and target stop are located within metropolitan areas. In contrast, picking source stops and target stops uniformly at random will often result in queries between rural locations.

The choice of the queries can have a significant influence on the results of the experimental evaluation. One reason for this is that stops in rural areas are typically served by far fewer trips than stops in metropolitan areas. Therefore, queries are potentially simpler and can be answered faster if the source and target stop are located in rural areas. Moreover, if a stop is only infrequently served by trips, then walking might be required more often as part of an optimal journey. Thus, using queries that were picked uniformly at random could lead to overestimating the importance of walking.

In order to avoid these problems, we do not pick the source and target stops for the queries, which we use in our experiments, uniformly at random. Instead, we argue that the number of trips that serve a stop reflects the number of passengers that want to travel to or from this stop. Thus, we expect that in a real application stops, which are served by a large number of trips, will occur more often as source stop or target stop of a query than stops that are only used by a few trips. We take this consideration into account during the generation of the random queries for our experiments. Instead of picking source stops and target stops using a uniform random distribution, we pick a stop v with a probability that is proportional to the number of trips that contain the stop v.

Distance Ranks. Another aspect that heavily influences the result of a query is the distance between the source and the target of the query. We address this issue by partitioning the queries with respect to their *distance rank*. The distance rank of a query, is the number of vertices v with the property that the distance from the source of the query to v is smaller than the distance from the source of the query to the target. As distance metric we use the length of shortest paths in the 'multimodal' transfer graph. In order to obtain representative queries for every distance rank, we first pick random source stops (where the probability of a stop is again proportional to the number of trips containing the stop). Afterwards we pick one target for every distance rank 2^r with $r \in \mathbb{N}$. The target stop for a query with distance rank 2^r is randomly picked from all stops with a distance rank between 2^r and 2^{r-1} (as before the probability of a stop is proportional to the number of trips containing the stop).

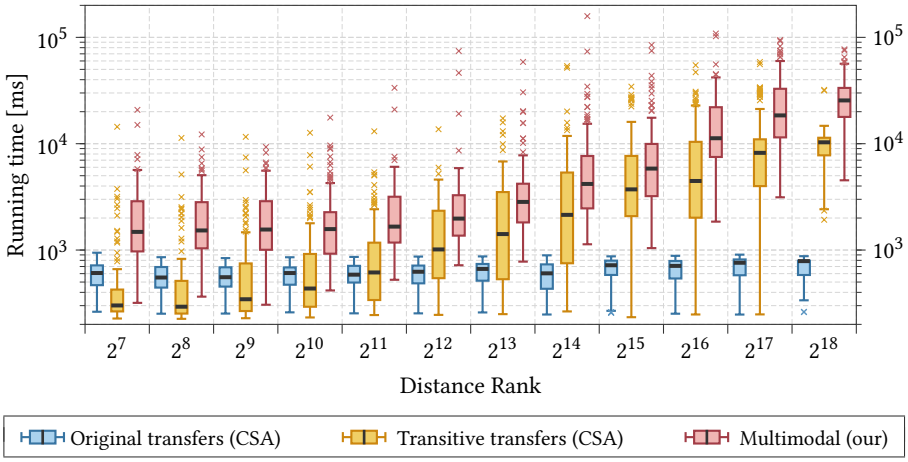


Figure 5.2: The running time of profile algorithms depending on the distance rank. We compare the three different variants of the *Switzerland* network. The 'original' and 'transitive' network variants are transitively closed, therefore, a state-of-the-art profile algorithm, such as CSA, can be applied. For the 'multimodal' variant we use our new algorithm. We evaluated 100 random queries per distance rank.

5.2.1 Performance Experiments

Our first experiment is focused on the performance of profile algorithms. We compare the time required to compute complete 24 hour profiles (containing all Pareto-optimal journeys with respect to travel time and number of transfers) depending on the three variants of our networks. For this we evaluated 100 random queries for every distance rank 2^r with $r \in \mathbb{R}$. We discuss the resulting running times in detail for the Switzerland network (Figure 5.2). The running times for the other three networks are shown in Figure 5.2 and are quite similar. For the network variants that contain only the original transfers we use CSA [DPSW18]. It is clearly visible that the running time of CSA is independent of the distance rank for the 'original' variant. The reason for this is, that the transfer graph contains only very few edges, and therefore the running time is dominated by scanning the connections. Since the algorithm always scans all connections, the running time is independent from the distance rank of the query.

Computing profiles for the 'transitive' variants can also be done using CSA, since the transfer graph is transitively closed. The resulting running times, however, differ significantly from the running times of the 'original' variant. For the highest distance rank, running times are increased by an order of magnitude, resulting in query

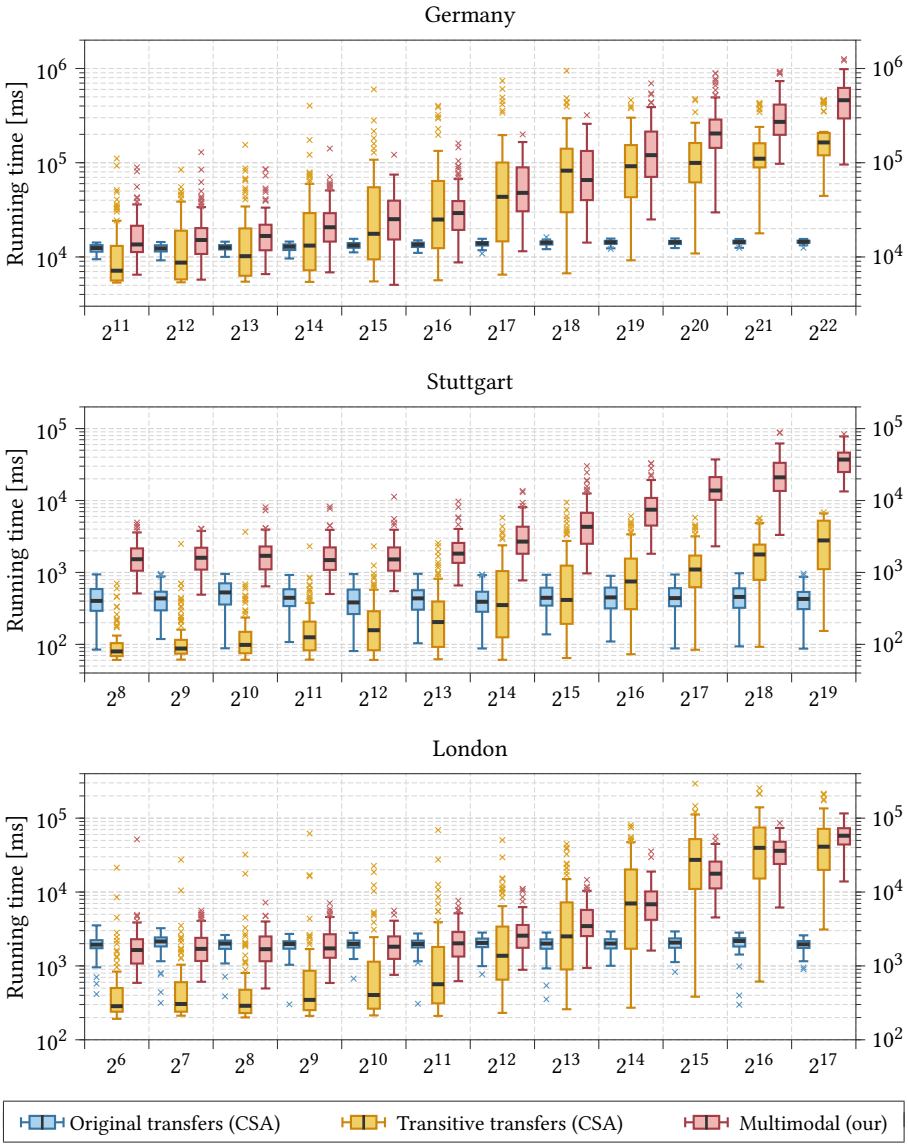


Figure 5.3: Running times for the networks of Germany, Stuttgart, and London. We compare profile-CSA on the 'original' and 'transitive' variants with our algorithm on the 'multimodal' variant. For each distance rank 100 random queries were evaluated.

times of several seconds for the Switzerland network and of about 2 minutes for the Germany network. The query time decreases with decreasing distance rank, as a result of target pruning. For small distance ranks, the running time even falls below the running time for the 'original' variant. The reason for this is most probably a high number of queries where walking is the optimal solution. This decreases the complexity of the profile functions, which leads to decreased running times.

Finally we examine the running time for the 'multimodal' variant of our networks. Since the transfer graph of these variants is not transitively closed, we have to use our new algorithm. Computing a profile using our algorithm takes about 6 minutes on average. Despite the fact that our algorithm computes profiles for more complex networks with unrestricted walking, running times are only a factor 2 to 4 slower than CSA on the 'transitive' variant. Similar to CSA, the running time of our algorithm decreases with decreasing distance rank. The reason for this is the underlying search algorithm (in our case MCR), which is faster for local queries due to target pruning.

The results for the other three networks (Figure 5.3) are quite similar to the results for the Switzerland network. Of course the absolute running times differ depending on the size of the network. A notable difference in the relative running times can be seen for the Stuttgart network. Here, the difference between our algorithm and CSA on the 'transitive' variant is much more pronounced than on the other networks. The reason for this is that the difference in size between the 'transitive' variant and the 'multimodal' variant of the transfer graph is also the greatest for this network. Note that we only report the running times for the eleven highest distance ranks of every network, as these are the most interesting queries. Moreover, the running times for low distance ranks hardly differ from each other.

5.2.2 Travel Time Comparison

Finally, we analyze how the travel times of optimal journeys change, depending on the used transfer graph. For this we compare the results of the same 100 random queries per distance rank, which we used for the performance experiments in the previous section. Our evaluation focuses only on the earliest travel time, i.e., we ignore Pareto-optimal solutions that use fewer transfers than the fastest journey. This leads to a conservative estimation for the importance of walking, since walking is even more indispensable if the number of transfers is limited.

As before, we begin with an exemplary discussion of the results for the Switzerland network. Afterwards, we address special aspects and deviations that occur in the other three networks. In all following experiments we consider the travel times that are achievable in the networks that combine public transit with the 'multimodal' transfer graphs as ground truth. We then compare these travel times with the travel times that are achievable if only the 'original' or 'transitive' transfer graph is available.

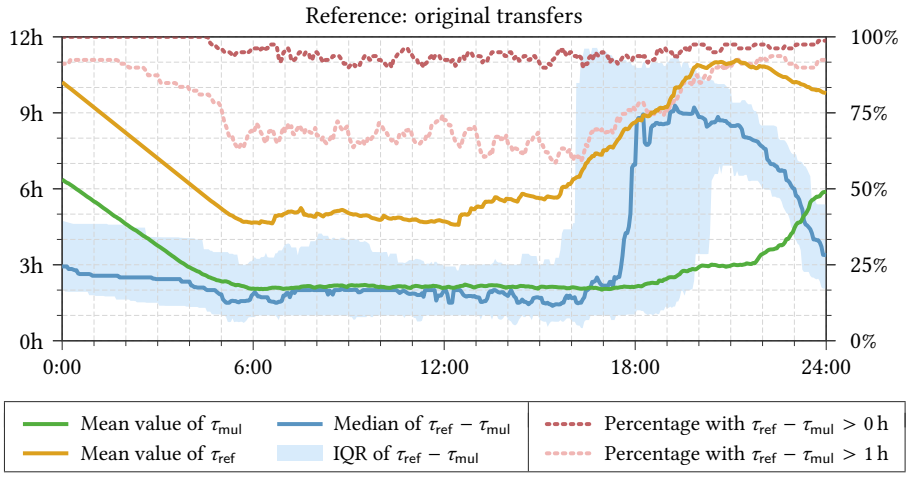


Figure 5.4: Comparison of optimal travel times throughout the day for the *Switzerland* network. We compare the travel time τ_{mul} in the network that uses the 'multimodal' transfer graph with the travel time τ_{ref} in a reference network, which in this case is the network that uses the 'original' transfer graph. We evaluated 100 random queries with distance rank 2^{16}, which correspond to an average travel time of about 2 hours. The x-axis states the departure time of the queries. The mean of τ_{mul} over the 100 queries is plotted in green and the mean of τ_{ref} is plotted in yellow. The median of the difference between the two travel times is plotted in blue and the interquartile range (IQR) of this difference is depicted in light blue. The dark red dotted curve (using the right y-axis) indicates the percentage of queries where τ_{mul} and τ_{ref} are not equal. The light red dotted curve (using the right y-axis) indicates the percentage of queries where the difference between the two travel times is more than 1 hour.

Travel Times with Original Transfers. We examined the difference in travel times between the 'multimodal' and the 'original' transfer graph variant independently for all distance ranks. Overall, we found that the relative difference in travel times is consistent over all distance ranks. Therefore, we confine our analysis of the results to one single distance rank. For the Switzerland network we have chosen a distance rank of 2^{16}, which roughly corresponds to an average travel time of 2 hours. The resulting evaluation for this distance rank is shown in Figure 5.4. In this plot (and in the subsequent plots in this section), the green curve corresponds to the mean travel time in the case that the 'multimodal' transfer graph is used. We compare this travel time to a reference travel time, which is plotted in yellow.

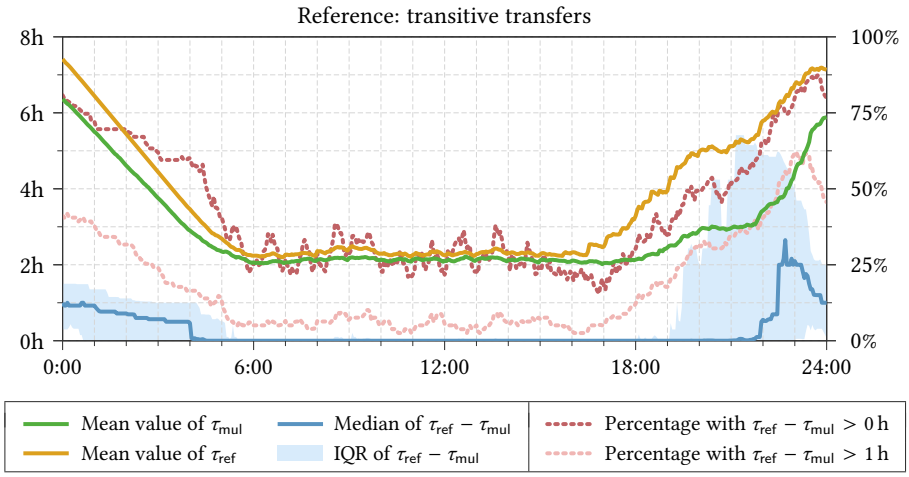


Figure 5.5: Comparison of optimal travel times throughout the day for the *Switzerland* network as in Figure 5.4. In this plot, the reference travel time τ_{ref} corresponds to the minimal travel time in the network variant that uses the 'transitive' transfer graph. As before this travel time is compared to the multimodal travel time τ_{mul}.

For our first experiment (Figure 5.4) the reference travel time corresponds to the minimal travel time in the network variant that uses the 'original' transfer graph. The plots demonstrate that using only the 'original' transfers leads to journeys with travel times that surpass optimal 'multimodal' travel times by several hours.

The importance of unrestricted walking becomes even more noticeable when looking at the percentage of queries where using the 'original' transfer graph leads to increased travel times. Looking at the dark red dotted curve, which depicts the percentage of these queries, we can see that travel times of almost all queries can be improved by using the 'multimodal' transfer graph instead of the 'original' transfer graph. Moreover, we observe that percentage of queries where the travel time can be improved by more than one hour is about 70%, as depicted by the light red dotted curve.

Travel Times with Transitive Transfers. So far we have only looked at the original transfers, which are admittedly quite limited. Figure 5.5 shows the result for the same experimental setup, but using the 'transitive' transfer graph instead of the 'original' transfers. In this case we observe that the travel times in the 'transitive' network are much closer to the travel times in the 'multimodal' network, at least during the day. However, in the evening and during the night, unrestricted walking

still improves the travel time significantly. The difference is particularly significant at around 23:00, where half of the queries can be improved by more than 2 hours, as indicated by the blue curve for the median of the travel time difference.

Considering the percentage of queries where the travel time in the two network variants differs (dark red dotted curve), we observe considerably lower values than in the previous experiment. However, about 25% of all queries can still be improved by unrestricted walking during day time. Moreover, we observe that the travel times of less than 10% of the queries can be improved by more than one hour, for departure times between 5:00 and 17:00. However, the minimal travel times still differ by more than one hour for about 50% of the queries during the night.

Overall, these results demonstrate that public transit journey planning can provide reasonable results for queries during the day. However, there exist queries that benefit significantly from multimodal journey planning. Especially at night time, public transit journey planning will often yield travel times that can be reduced by several hours if unrestricted walking is considered. These results, which we first published in [WZ17] have since been confirmed by [PV19].

Travel Times with Transitive Intermediate Transfers. In order to better understand why unrestricted walking has such a strong influence on optimal travel times, an extended experiment was proposed in [Sau18]. In this work, travel times in a multimodal network have been compared to travel times in a network where walking is only restricted for some parts of a journey. In particular, arbitrary paths in the 'multimodal' graph can be used to transfer from the source of a query to the stop where the first trip is boarded. Similarly, arbitrary paths can be used to transfer from the last trip of a journey to the target of the query. However, for transferring between two trips, a transfer within the 'transitive' graph has to be used.

Figure 5.6 shows the results of comparing optimal travel times in this scenario with optimal travel times in the 'multimodal' network variant. We observe that in this experiment the optimal travel times are quite similar for most departure times. Even during the night the mean travel time in the 'multimodal' network variant is not much lower than the mean travel time of queries where the intermediate transfer are restricted to the 'transitive' transfer graph. We also observe, that the percentage of queries, which lead to a travel time difference of more than one hour, is much smaller than in the previous two experiments.

From these results we conclude that long transfers are particularly important for reaching the first public transit stop from the source and for traveling from the last public transit stop of a journey to the target. In contrast, long transfers are much less important for transferring between two public transport trips. This means that traveling within the public transport network, for the most part, only requires short transfers, while reaching the public transit network can require quite long transfers.

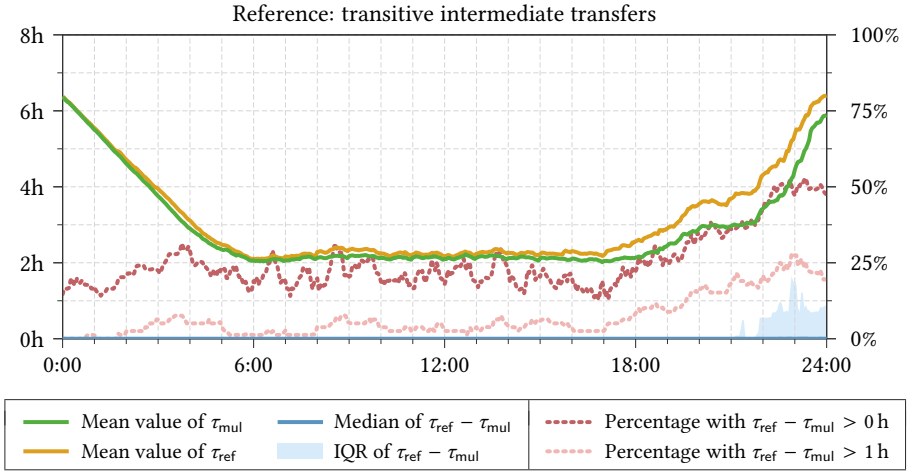


Figure 5.6: Comparison of optimal travel times throughout the day for the *Switzerland* network as in Figure 5.4. In this plot, the reference travel time τ_{ref} corresponds to the minimal travel time in a network that uses both, the 'transitive' and the 'multimodal' transfer graph. In particular, the 'transitive' graph is used for transfers between two public transit trips, while the 'multimodal' graph is used for the transfer from the source to the first trip of the journey and the transfer from the last trip of the journey to the target. As before this travel time is compared to the multimodal travel time τ_{mul}.

Results for the Other Networks. We also performed all three travel time comparisons for our other networks. We present the results for the networks of Germany, Stuttgart, and London in Figures 5.7, 5.8, and 5.9, respectively.

Overall, the results for these networks are quite similar to the results for the Switzerland network. We observe for all networks that the travel times of a significant percentage of the queries can be improved by using a multimodal journey planning algorithm instead of a public transit journey planning algorithm. Furthermore, all four networks have in common, the travel time difference mostly disappears in the scenario where only the intermediate transfers are restricted.

The only network that differs notably from the other is the London network. For this network we evaluated queries with a distance rank of only 2^{15}, since the network is much smaller than the other networks. We observe that the travel time differences for the London network are significantly smaller than for the other networks. The reason for this is that London is the only metropolitan network. Because of this, the network is much more densely interconnected and thus long transfers are needed less often.

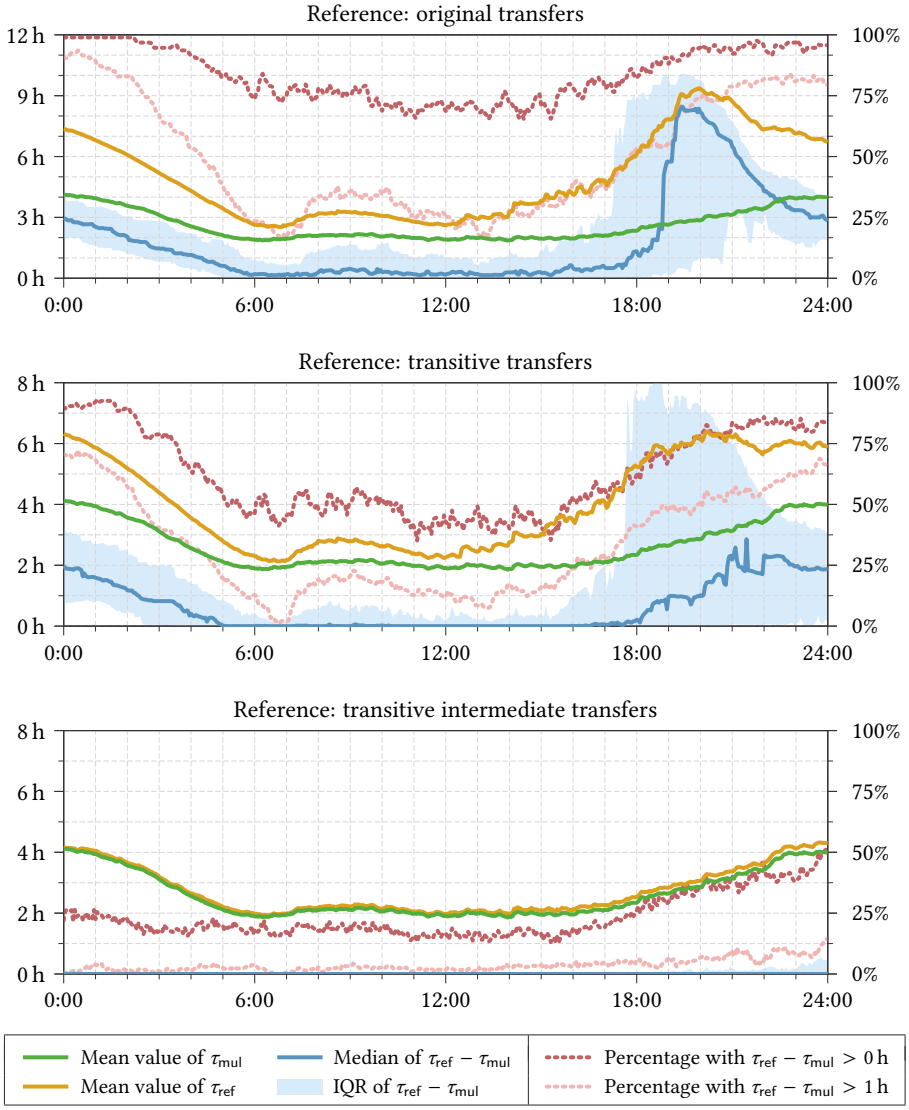


Figure 5.7: Travel time comparisons for the *Germany* network using the setup from Figure 5.4. The results are based on 100 random queries with a distance rank of 2^{16}. We compare the 'multimodal' network to the network that uses 'original' transfers (top), 'transitive' transfers (middle), and 'transitive' intermediate transfers (bottom).

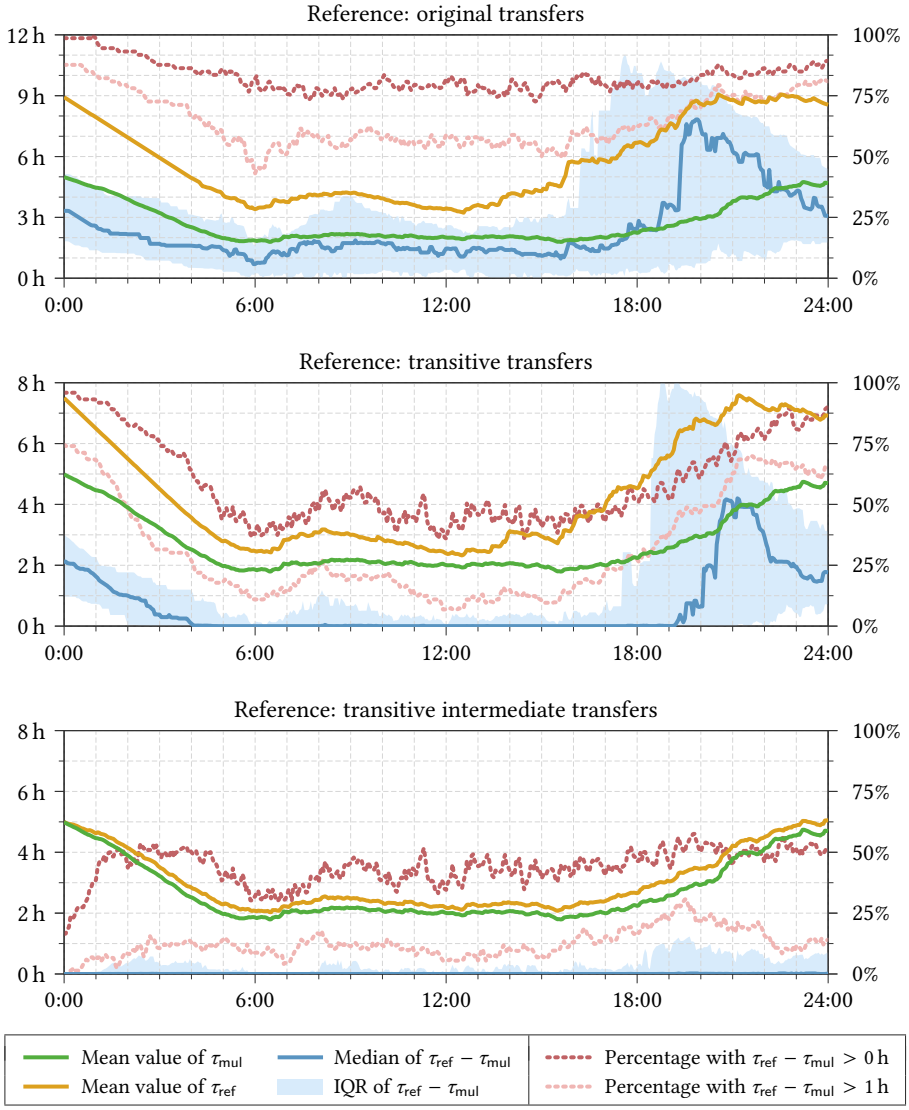


Figure 5.8: Travel time comparisons for the *Stuttgart* network using the setup from Figure 5.4. The results are based on 100 random queries with a distance rank of 2^{16}. We compare the 'multimodal' network to the network that uses 'original' transfers (top), 'transitive' transfers (middle), and 'transitive' intermediate transfers (bottom).

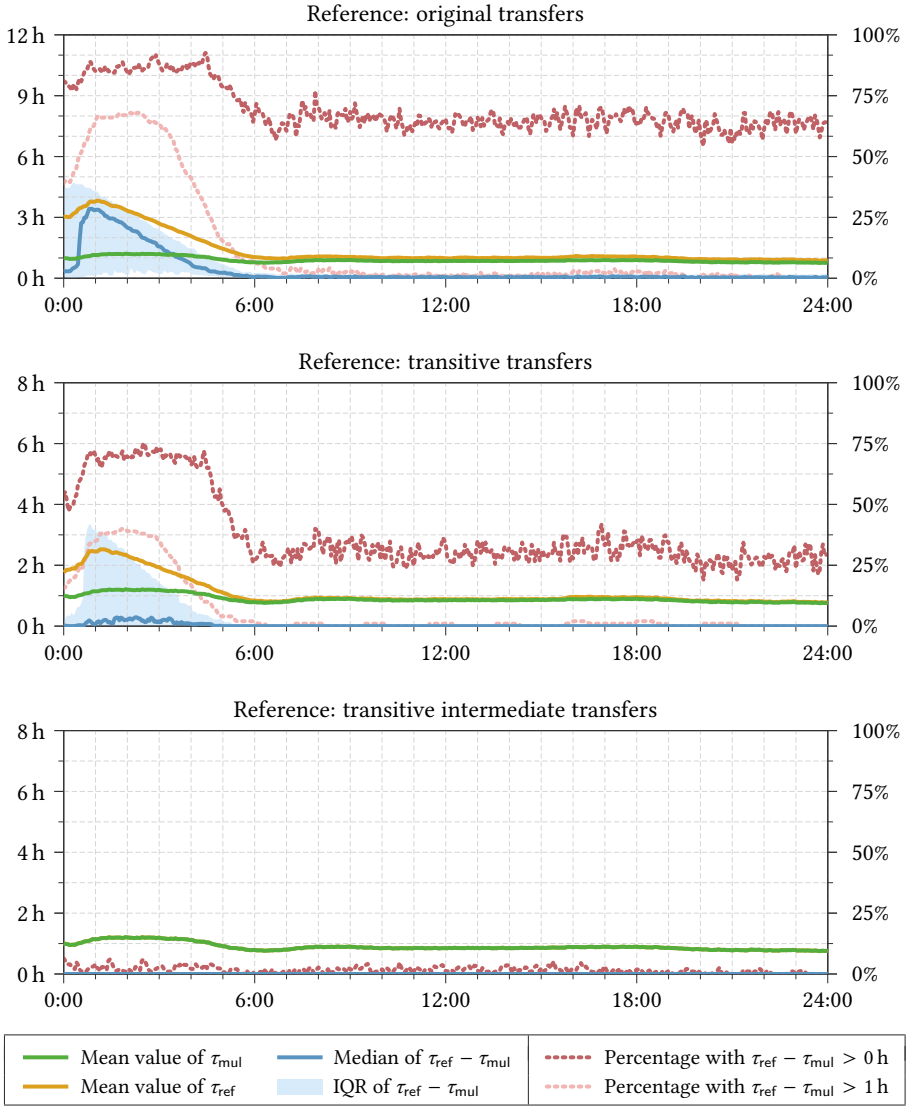


Figure 5.9: Travel time comparisons for the *London* network using the setup from Figure 5.4. The results are based on 100 random queries with a distance rank of 2^{15}. We compare the 'multimodal' network to the network that uses 'original' transfers (top), 'transitive' transfers (middle), and 'transitive' intermediate transfers (bottom).

5.3 Final Remarks

In this chapter, we studied the problem of computing bicriteria profiles in multimodal transportation networks. To this end, we demonstrated how the profile problem can be solved by successively computing optimal journeys for fixed departure times. Building upon this, we presented our novel profile algorithm for multimodal network, which is based on the MCR algorithm for fixed departure time queries.

We proved the viability of our approach with an experimental evaluation on four real-world networks. Since, to the best of our knowledge, no previous algorithm exists that can compute multimodal profiles, we compare our algorithm to public transit profile algorithms. While the multimodal networks are significantly larger than the public transit networks, our new profile algorithm still achieves running times that are comparable to a state-of-the-art public transit algorithm.

Finally, we used our new multimodal profile algorithm to analyze the importance of walking as transfer mode in public transit networks. Here, our results demonstrated that considering walking without any restrictions can have a significant impact on a journey's travel times. Even if an extensive transitive transfer graph is considered, optimal travel times can often be improved by more than one hour if a multimodal network is considered instead. However, our experiments also show that walking is not equally relevant for all parts of a journey. Long walking transfers are most often required for the first or the last transfer of a journey. In contrast, short transfers are often sufficient for transferring between two public transit trips.

6 UnLimited TRAnsfer Shortcuts

In the previous chapter we confirmed that allowing multimodal journeys has a significant impact on travel times (even if only the modes public transit and walking are considered). This result strengthens the need for an efficient multimodal journey planning algorithm. However, we also observed that the relevance of walking transfers strongly depends on their position within the overall journey. In this chapter we present a novel approach for multimodal journey planning that takes advantage of this fact. This chapter is based on joint work with Moritz Baum, Valentin Buchhold, Jonas Sauer, and Dorothea Wagner [Bau+19a, Bau+19b].

Problem Setting. In this chapter we consider a multimodal journey planning problem in a network consisting of public transit and an unrestricted secondary transportation mode, which is represented using a transfer graph. While the transfer graph might represent any non-schedule-based type of transportation (such as using a taxi or a bike), we will focus on walking as representative transportation mode. In addition to the multimodal network, which is known in advance, a query consists of a source stop, a target stop, and a desired departure time. The objective is to compute a Pareto-set of journeys from source to target that depart not earlier than the desired departure time, where arrival time and the number of trips are used as optimization criteria.

Proposed Solution. As discussed before, the main obstacle to solving such a multimodal journey planning problem efficiently are the time-consuming searches in the transfer graph. In order to perform these searches efficiently, we propose a novel speed-up technique for public transit journey planning algorithms, which we call

ULTRA (UnLimited TRAnsfers). Our approach is based on the observation, that long intermediate walking transfers (i.e., walking from one trip to another) are only occasionally required. This suggests that the number of unique paths in the transfer graph, which occur as intermediate transfers of a Pareto-optimal journey, is small. Based on this insight, we propose a preprocessing phase, during which we compute a small number of transfer shortcuts that are provably sufficient for computing correct Pareto-sets. A query algorithm can then use these shortcuts in order to find possible destinations of intermediate transfers instead of searching through the transfer graph. This approach obviously still requires searches in the transfer graph, in order to find initial and final transfers. However, since both of these transfer types have one of their end points fixed (either the source or the target), they can be found using one-to-many searches, which are very efficient. Thus, our algorithm can handle all types of transfers efficiently: Intermediate transfers through lookups in the shortcut graph and initial and final transfers through the usage of fast one-to-many queries. While the idea for this approach is based on observations that were made for walking as transfer mode, we show in Section 6.4 that the performance of ULTRA is independent of the transfer mode.

Chapter Overview. In Section 6.1 we describe the preprocessing algorithm that, given a public transit network and transfer graph as input, computes the transfer shortcuts. We start by outlining a general approach for finding the required shortcuts. Afterwards, we carefully engineer the preprocessing algorithm to ensure that the number of discovered shortcuts remains small and the required running time is low.

We continue by showing that the precomputed transfer shortcuts can be integrated into a variety of state-of-the-art public transit algorithms in Section 6.2. Thus, we do not present a single query algorithm, but establishing a whole family of ULTRA-query algorithms. Among other algorithms, this leads to ULTRA-CSA, the first efficient multimodal variant of CSA.

This family of query algorithms naturally includes ULTRA-Trip-Based. However, Trip-Based public transit routing allows for a much more sophisticated integration with ULTRA, which we present in Section 6.3. We show how the preprocessing steps of ULTRA and Trip-Based routing can be combined in order to reduce running time and the number of computed shortcuts. Moreover, we present a variant of the Trip-Based query algorithm that is optimized for a scenario with unlimited initial and final transfers.

We evaluate the performance of the preprocessing phase and the different query algorithms in Section 6.4. Here we show that ULTRA enables unlimited transfers for all presented query algorithms without sacrificing query speed, yielding the fastest known algorithms for multimodal journey planning. This is true not only for walking, but also for other transfer modes such as cycling or driving.

6.1 Shortcut Computation

The preprocessing phase of the ULTRA approach aims at finding a small number of transfer shortcuts that are sufficient to answer every point-to-point query correctly. This is achieved if there exists a journey J' for every Pareto-optimal journey J with the same departure time, arrival time, and number of trips, which only uses the precomputed shortcuts to transfer between trips. Next, we present a high-level overview of the ULTRA preprocessing, followed by an in-depth description of important algorithmic details.

6.1.1 Overview

The basic idea of the ULTRA preprocessing phase is quite simple. We enumerate all possible journeys that use exactly two trips and require neither an initial nor a final transfer. The transfers between the two trips of these journeys are then considered to be *candidates* for shortcuts. Accordingly, the journeys containing them (i.e., journeys with two trips and no initial and final transfer) are called *candidate journeys*. For each of these candidate journeys, we check if there exists another journey that weakly dominates it. If this is the case, we can replace the candidate journey with the dominating journey without losing Pareto-optimality. Note that if the candidate journey is contained in a longer journey, then it still can be replaced without affecting the Pareto-optimality of the longer journey. We call such a dominating journey a *witness* since its existence proves that the candidate shortcut is not needed. Unlike the candidate journey, the witness journey can make use of the transfer graph before the first trip or after the second trip. If no witness is found, then the candidate shortcut is added to the resulting shortcut graph.

A naive implementation of this idea would be to first enumerate all candidate journeys and subsequently search for witnesses. However, this would be impractical due to the sheer number of possible journeys. Therefore, we propose to interweave the candidate enumeration and the witness search, with the goal of eliminating as many candidates as early as possible. Pseudo code for the result of these considerations is given by Algorithm 6.1 on page 83. The algorithm resembles invoking rRAPTOR [DPW15a] once per stop, restricted to the first two rounds per iteration. Remember that the original rRAPTOR algorithm already solves one-to-all range queries. Restricting this algorithm to the first two rounds enables an efficient enumeration of candidate journeys starting at one stop. Thus, adding a loop over all possible first stops allows us to enumerate al candidate journeys efficiently. Moreover, many dominated candidates are eliminated early on, due to self-pruning. We will now continue with a detailed discussion of Algorithm 6.1, showing step by step what has changed in comparison to the original rRAPTOR and how this helps with computing the transfer shortcuts.

6.1.2 Implementation Details

A first important difference between our preprocessing algorithm and rRAPTOR is due to the fact that rRAPTOR requires a transitively closed transfer graph. As we want to allow arbitrary transfer graphs, we replace the RAPTOR that is invoked in every iteration of rRAPTOR with MR-∞, the variant of MCR that optimizes arrival time and number of used trips. Because of this change, the relaxation of transfers in lines 8 and 11 is not done by relaxing outgoing edges of updated stops. Instead, Dijkstra's algorithm [Dij59] is performed in order to propagate arrival times found by the preceding route scanning step. Furthermore, MCR would also use Dijkstra's algorithm in order to collect all routes reachable from the source stop in line 6. In the context of rRAPTOR this leads to many redundant computations, as the source stop does not change between iterations. Therefore, we compute distances from the source stop to all other stops once in line 3, again using Dijkstra's algorithm. These distances can then be used in line 6.

Departure Time Collection. In line 4, standard rRAPTOR would collect all departure events that are reachable from the source stop s. However, given a transfer graph without any restrictions, this could quite possibly be every departure event in the network. Since we are primarily interested in finding candidate journeys, which do not have initial transfers, we collect only those departure events which depart directly at the source stop s. However, in order to find witness journeys, we still need to explore initial transfers in line 6. For this purpose, a naive implementation would iterate over all stops $v \in \mathcal{S}$ that are reachable from s and over all routes containing the stop v. Each of these routes has then to be checked for the existence of a trip that was not scanned in a previous iteration and can be reached given the departure time τ_{dep} at s. Since initial transfers are unlimited, this would possibly lead to each stop and each route being processed in line 6.

A more efficient approach combines lines 4 and 6 into a single operation, that has not to be repeated for every rRAPTOR iteration. For this, we first sort all departure triplets $(v, \tau_{\text{dep}}, R)$ of departure stop v, departure time τ_{dep}, and route R by their corresponding departure time at the source, $\tau_{\text{dep}} - \tau_{\text{tra}}(s, v)$. Afterwards, we iterate through this sorted list in descending order of departure time. If the next triplet to be processed has a departure stop $v \neq s$, then its route is added to a set $\mathcal{R}'$, and we immediately continue with the next triplet. In the case that the next triplet $(v, \tau_{\text{dep}}, R)$ actually has the source stop as its departure stop ($s = v$), we again add R to $\mathcal{R}'$, but afterwards we proceed with lines 6 through 12. Now the routes that have to be collected in line 6 are exactly the routes in $\mathcal{R}'$ plus the route of the current triplet. Thus, we simply scan all routes in $\mathcal{R}'$ and then reset $\mathcal{R}' = \emptyset$ for the next iteration.

Algorithm 6.1: ULTRA transfer shortcut computation.

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{R})$, unlimited transfer graph $G = (\mathcal{V}, \mathcal{E})$
Output: Shortcut graph $G^s = (\mathcal{S}, \mathcal{E}^s)$

```
1  for each s ∈ S do
2      Clear all arrival labels and Dijkstra queues
3      τ_tra(s, ·) ← Compute transfer times in G from s to all stops
4      DT ← Collect departure times of trips at s
5      for each τ_dep ∈ DT in descending order do          // rRAPTOR iteration
6          Collect routes reachable from s at τ_dep        // first RAPTOR round
7          Scan routes
8          Relax transfers
9          Collect routes serving updated stops            // second RAPTOR round
10         Scan routes
11         UC ← Relax transfers, thereby collecting unwitnessed candidates
12         E^s ← E^s ∪ UC
```

Limited Transfer Relaxation. Another part of ULTRA that differs from rRAPTOR is the final relaxation of transfers in line 11. This is the part of the algorithm where we actually determine the candidate journeys for which we have not found a witness. As usual, relaxing the transfers is done by Dijkstra's algorithm, initialized with the arrival times from the preceding route scanning step. Whenever a stop is settled during this execution of Dijkstra's algorithm, we look at the corresponding journey and check whether it is a candidate journey, i.e., does not require initial or final transfers. If so, we know that there is no witness journey that weakly dominates this candidate, because otherwise the search would have reached the stop via this witness journey instead. Thus, we extract the intermediate transfer of the found candidate journey and add it as an edge to the shortcut graph.

We further increase the practical performance of our algorithm by adding a stopping criterion to the final transfer relaxation in line 11. For this purpose, we count the number of stops which were newly reached via a candidate journey in the preceding route scanning step. Whenever such a stop is settled in line 11, we decrease our counter. Once the counter reaches zero, we can stop settling further vertices as we know that no more candidates can be found in this iteration. We can apply a similar stopping criterion to the intermediate transfer relaxation in line 8. In this case, we count the stops which were reached via a route directly from s, without an initial transfer, since only these stops can later become part of a candidate journey. As in line 11, we can stop settling vertices as soon as no such stops are left in the queue of Dijkstra's algorithm. This does not affect the correctness of the algorithm, as we still

process all candidates. However, it might cause some witnesses to be pruned and thus it can lead to superfluous shortcuts in the result. To counteract this, we take the arrival time τ_{arr} of the last stop representing a candidate that is settled. Instead of stopping the transfer relaxation immediately, we continue until the queue head has an arrival time greater than $\tau_{\text{arr}} + \bar{\tau}_{\text{wit}}$ for some parameter $\bar{\tau}_{\text{wit}}$ (which we call *witness limit*). With these changes, the only remaining part of the algorithm that performs an unlimited search on the transfer graph is the initial transfer relaxation in line 3, which is only performed once for every source stop (i.e., every stop in the network).

The success of our pruning rule for the transfer relaxation in lines 8 and 11 depends on the presence of candidate journeys in the Dijkstra queues. Fewer candidate journeys could therefore lead to an earlier application of the pruning rule. We exploit this by further restricting the notion of candidate journeys. As before, a candidate journey must not contain any initial or final transfers. In addition, we now only classify such a journey as a candidate journey if its intermediate transfer is not contained in the set of already computed transfer shortcuts. This reduces the number of candidate journeys we have to consider. However, it does not affect the correctness of our approach, since the missing candidates would only produce shortcuts that are already part of the output anyway.

Cyclic Witnessing. Since witnesses are only required to dominate candidate journeys weakly, there may exist journeys J, J' that dominate each other. If two such journeys act as witness for each other, we could miss a required shortcut. If J has an initial transfer of length > 0, then J without the initial transfer is not dominated by J' extended by the reverse initial transfer. Therefore, the shortcut required by J will be added. Thus, cyclic domination is only problematic between journeys with initial transfers of length 0. We prevent this by temporarily contracting groups of stops with transfer distance 0 during the preprocessing.

Transfer Graph Contraction. As shown for MCR [Del+13], the transfer relaxation is often the bottleneck of multimodal journey planning algorithms. Since ULTRA only needs to compute journeys between stops, rather than journeys between arbitrary vertices of the transfer graph, only transfers that start and end at stops are relevant. Therefore, any overlay graph that preserves the distances between all stops can be used instead of the transfer graph in our preprocessing algorithm. An easy way of obtaining such an overlay graph is to construct a partial CH that only contracts vertices that do not correspond to stops of the public transit network. This, of course, leads to a suboptimal contraction order and thus makes it infeasible to contract all vertices that are not stops. As done in many other algorithms [Bau+10b, DPW15b, Del+13, Bau+15, BBDW16], we therefore stop the contraction once the uncontracted core graph surpasses a certain average vertex degree.

Parent pointers. We use parent pointers during all searches in order to be able to retrieve the intermediate transfers of candidate journeys, if we want to collect them as shortcut. Parent pointers are usually maintained by assigning parent[w] $\leftarrow v$ whenever relaxing an edge $e = (v, w)$ leads to an improved arrival time at the vertex w. We modify this behavior and assign parent[w] $\leftarrow$ parent[v] instead. Thus, the parent pointer does not represent the preceding vertex within a path but the origin vertex of the path. Because of this, we do not need to retrace parent pointers in order to obtain the intermediate transfer of a candidate journey. Furthermore, we observe that parent pointers are only needed for candidate journeys, since we are not interested in the vertices used by candidate journeys. We exploit this fact by assigning a special value to the parent pointers of the origin vertices of witness journeys. This allows us to efficiently determine if a given journey is a witness journey or a candidate journey, as we only need to inspect the parent pointer of the last vertex in the journey.

Data Structures. For an optimal performance of the preprocessing phase it is indispensable that efficient data structures and a streamlined memory layout are used. To this end, we arrange data that is accessed through a loop sequentially within the memory. This affect edges of the transfer graph with a common start vertex, stop events within a trip, and trips within routes.

For the three Dijkstra searches within the transfer graph (initial transfer search, transfer relaxation in line 8, and transfer relaxation in line 11) we use 4-ary heaps as queues, since they tend to yield the best performance in practice [CGR96]. Furthermore, we use separate queues and labels (encapsulating the arrival time and the parent pointer) for each of the three searches. This allows us to not clear the queues in between rRAPTOR iterations. Otherwise queue entries from the second or third search of an earlier rRAPTOR iteration might interfere with queue entries from the first or second search of the current iteration).

Parallelization. Finally, we observe that ULTRA allows for trivial parallelization. Our preprocessing algorithm searches for candidate journeys once for every possible source stop (line 1 of Algorithm 6.1). As these searches are mostly independent of each other, we can distribute them to parallel threads and combine the results in a final sequential step. Only the usage of the restricted candidate notion introduces a dependence between the searches for different source stops. As this is only a heuristic performance optimization, we simply relax the notion of candidate journeys again: In the parallelized version of the preprocessing phase we classify a journey as candidate journey, if it has neither an initial nor a final transfer and its intermediate transfer is not equivalent to a shortcut that has already been found by the same thread.

6.1.3 Proof of Correctness

Before continuing with the query algorithms, we want to justify that ULTRA computes a shortcut graph that is sufficient to answer all queries correctly.

Theorem 6.1. Every Pareto-optimal journey either uses solely intermediate transfers that are contained in the shortcut graph or is weakly dominated by another journey that uses solely intermediate transfers that are contained in the shortcut graph.

Proof. Assume that a journey $J = \langle P_0, T_0^{ij}, \ldots, T_{k-1}^{mn}, P_k \rangle$ with the following two properties exists: First, J requires an intermediate transfer that is not contained in the shortcut graph. Secondly, J cannot be replaced with a journey of equal travel time and number of trips, which solely uses transfers from the shortcut graph. In this case, the journey J must contain at least two trips, since otherwise it would not contain any intermediate transfers. Since the journey contains two or more trips, it can be disassembled into candidate journeys $\langle T_0^{ij}, P_1, T_1^{gh} \rangle, \langle T_1^{gh}, P_2, T_2^{pq} \rangle, \ldots, \langle T_{k-2}^{uv}, P_{k-1}, T_{k-1}^{mn} \rangle$. As J requires a transfer that is not contained in the shortcut graph, at least one of these candidates must also contain a transfer not contained in the shortcut graph. Let $J^c = \langle T_x^{gh}, P_{x+1}, T_{x+1}^{pq} \rangle$ be such a candidate journey. Since the main loop of the ULTRA preprocessing algorithm is executed once for every stop in the network, it was also executed for the source stop $v(T_x[g])$ of this candidate journey. Derived from the correctness of rRAPTOR, we know that for a given source stop our algorithm computes Pareto-optimal arrival labels for all stops reachable with two trips or less. Thus we also reached the target stop $v(T_{x+1}[q])$ of the candidate journey, since it can be reached from $v(T_x[g])$ with two trips. The journey J' corresponding to the target's arrival label is in this case either the candidate journey or a journey that dominates the candidate journey. In the first case, we have added the transfer P_{x+1} of the candidate journey to the shortcut graph. In the second case, the candidate journey J^c can be replaced by the journey J' corresponding to the target's arrival label, leading to a journey that is not worse than the original journey and does not require the missing transfer. Therefore, both cases contradict our assumption. □

6.2 Query Algorithms

The shortcuts obtained from the ULTRA preprocessing can, in principle, be combined with any public transit query algorithm that normally requires a transitively closed transfer graph, such as RAPTOR [DPW15a], CSA [DPSW13, DPSW18], or Trip-Based Routing [Wit15]. The basic idea of our query algorithm is to simply apply one of these algorithms to a network that uses the precomputed shortcut graph instead of

Algorithm 6.2: Query algorithm, using ULTRA transfer shortcuts

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{R})$, shortcut graph $G^s = (\mathcal{S}, \mathcal{E}^s)$, Bucket-CH of the original transfer graph G, source vertex s, departure time τ_{dep}, and target vertex t

Output: All Pareto-optimal journeys from s to t for departure time τ_{dep}

1 $\tau_{\text{tra}}(s, \cdot) \leftarrow$ Run Bucket-CH query from s
2 $\tau_{\text{tra}}(\cdot, t) \leftarrow$ Run reverse Bucket-CH query from t
3 $\tilde{G}^s \leftarrow (\mathcal{S} \cup \{s, t\}, \mathcal{E}^s)$
4 **for each** $v \in \mathcal{S}$ **do**
5 Add edge (s, v) to $\tilde{G}^s$ with travel time $\tau_{\text{tra}}(s, v)$
6 Add edge (v, t) to $\tilde{G}^s$ with travel time $\tau_{\text{tra}}(v, t)$
7 Run black box public transit algorithm on $(\mathcal{S} \cup \{s, t\}, \mathcal{T}, \mathcal{R}), \tilde{G}^s$

the original transfer graph. However, our shortcut graph only represents transfers between two trips and does not provide any information for transferring from the source to the first trip or for transferring from the last trip to the target. In this section we describe how the aforementioned public transit algorithms can be modified in order to handle initial and final transfers efficiently.

6.2.1 Basic Query Algorithm.

Our approach is based on the observation that for both, initial and final transfers, one endpoint of the transfer is fixed. All initial transfers start at the source vertex of the query and all final transfers end at the target vertex of the query. Therefore, we can use two additional one-to-many queries (one of them performed in reverse) to cover initial and final transfers. These queries have to be performed on the original transfer graph, where they compute the distances from the source to all stops and from all stops to the target. While any one-to-many algorithm might be used to perform this task, we decided to use Bucket-CH, as it is one of the fastest known one-to-many algorithms and allows for optimization of local queries. Pseudo code for the resulting ULTRA query algorithm using Bucket-CH and our precomputed transfer shortcuts is shown in Algorithm 6.2.

Our query algorithm begins with performing the two Bucket-CH queries: a forward search from the source vertex in line 1 and a backward search from target vertex in line 2. Afterwards a temporary copy $\tilde{G}^s$ of the shortcut graph G^s, which contains the source and the target of the query as additional vertices, is initialized. In lines 5 and 6, this temporary graph is complemented with edges from the source to all other

stops and edges from all stops to the target, using the distances obtained from the Bucket-CH queries. Finally, a public transit algorithm is invoked as a black box on the public transit network with the temporary graph instead of the shortcut graph in line 7. The temporary graph is sufficient for the query to yield correct results, as it contains edges from the source to any possible first stop, all edges required to transfer between trips, and edges from any possible last stop to the target. Since there are no additional requirements on the black box public transit algorithm, it is easy to see that any existing public transit algorithm can be used with our shortcuts.

6.2.2 Running Time Optimizations.

We can further improve the performance of this query algorithm in practice by introducing some adjustments. First, we observe that we actually do not need edges from the source to every other stop. If the distance $\tau_{\text{tra}}(s, v)$ from the source s to a stop v is greater than the distance $\tau_{\text{tra}}(s, t)$ from s to t, then every journey that requires a transfer from s to v is dominated by simply transferring directly from s to t. Thus, we do not need to add the edge (s, v) to the temporary graph in this case. The same argument can be made for edges from some stop w to the target t if the distance $\tau_{\text{tra}}(w, t)$ is greater than $\tau_{\text{tra}}(s, t)$. Moreover, if we know that a stop v is further away from the source than the target, then we do not even need to compute the actual distance $\tau_{\text{tra}}(s, v)$. We can use this fact to prune the search space of the Bucket-CH queries in lines 1 and 2. For this purpose, we first perform a standard bidirectional CH query from source to target that stops settling vertices from the forward (respectively backward) queue if the corresponding key is greater than the tentative distance from the source to the target. As a result we obtain the distance $\tau_{\text{tra}}(s, t)$, as well as the partial forward (backward) CH search space from s (t), containing no vertices that have a greater distance from s (to t) than $\tau_{\text{tra}}(s, t)$. We then perform the second phase of the Bucket-CH query (i.e., scanning the buckets) only for the vertices in the partial search spaces of the CH query. Furthermore, we store the entries in each bucket sorted by the distance to their target. Thus, we can stop scanning through the bucket of a vertex v once we reach a stop w within the bucket with $\tau_{\text{tra}}(s, v) + \tau_{\text{tra}}(v, w) \geq \tau_{\text{tra}}(s, t)$. Doing so can drastically improve local queries, as we do not need to look at all stops, but only at stops that are close to the source or target.

If we do not treat the underlying public transit algorithm as a black box, we can further improve practical performance by omitting the construction of the temporary graph $\tilde{G}^s$. Instead of adding edges from s to stops v, we can directly initialize the tentative arrival times used by most public transit algorithms with $\tau_{\text{dep}} + \tau_{\text{tra}}(s, v)$. Instead of adding edges to t, we try to update the tentative arrival time at the target with the arrival time at v plus $\tau_{\text{tra}}(v, t)$ whenever the arrival time at v is updated.

6.3 Integration with Trip-Based Routing

Trip-Based public transit routing can be used with the generic query algorithm, which we presented in the previous section, without any modification. However, some parts of the Trip-Based approach are particularly suitable for further optimization.

Unlike RAPTOR and CSA, Trip-Based Routing on its own already requires a preprocessing step, even if it is used without ULTRA. Thus, combining it with ULTRA leads to a three-phase algorithm: The first phase is the normal ULTRA preprocessing, the second phase is the Trip-Based preprocessing, which uses the ULTRA transfer shortcuts as input, and the third phase is the ULTRA-Trip-Based query. Of these three phases, the two preprocessing steps have several parts in common. Therefore, integrating them and removing redundant parts yields a single and overall more elegant preprocessing step that produces fewer shortcuts.

Furthermore, the Trip-Based query algorithm can also be optimized for networks with unlimited transfers. The original query, as introduced in [Wit15], is optimized for a use case where only a small number of stops is reachable with transfers from the source or the target. However, with unlimited transfers, we expect that almost every stop is reachable from the source and the target. Therefore, we propose to restructure the query, such that the huge number of possible initial and final transfers can be processed more efficiently.

6.3.1 Trip-Based Preprocessing

The preprocessing phases of ULTRA and Trip-Based Routing have many similarities, despite the fact that Trip-Based Routing requires transitively closed transfers, which ULTRA does not. Both of them compute shortcuts, which are later used to accelerate the query algorithm. However, ULTRA computes time-independent shortcuts (connecting pairs of stops), while the Trip-Based shortcuts are time-dependent (connecting pairs of stop events). This means that a shortcut, which is needed at one time during the day, is available at all times when using ULTRA, while Trip-Based Routing is aware that the shortcut is only needed at a certain time.

Both approaches identify unnecessary shortcuts by enumerating journeys with at most two trips in order to find witness journeys which prove that a potential shortcut is not necessary. The Trip-Based preprocessing does this in a “transfer reduction” step, after all potential shortcuts have been generated. Since this is no longer feasible with unlimited transfers, ULTRA interleaves the generation and pruning of shortcuts. Another difference is the type of journeys that are considered as witnesses. In the Trip-Based preprocessing, witness journeys must start with the same trip from which the shortcut originated, whereas the ULTRA preprocessing also considers witness journeys that start with an initial transfer. Furthermore, the Trip-Based preprocessing

does not guarantee that a witness journey is found before the shortcut it could prune has already been added to the output, since this depends on the order in which the shortcuts are explored. Overall, ULTRA has more options for pruning candidate journeys and thus produces fewer shortcuts.

Since both preprocessing phases enumerate journeys for similar purposes, we propose to integrate them and remove redundant parts. We implement this by keeping the general approach of the ULTRA journey enumeration, which can handle unlimited transfer graphs and prunes more shortcuts overall. In order to produce time-dependent shortcuts, we switch from computing shortcuts between stops to computing shortcuts between stop events, which makes the Trip-Based preprocessing phase obsolete. Achieving this requires some alterations to the original ULTRA preprocessing phase, which we describe in detail in the remainder of this section.

Candidate Journeys. The original ULTRA preprocessing includes an optimization that dismisses candidate journeys if their intermediate transfer was already added as a shortcut before. In the context of ULTRA, this has a significant impact on the preprocessing time because time-independent shortcuts are likely to be used multiple times during the day. However, when switching to time-dependent shortcuts, it becomes much less likely for a new candidate journey to use a previously found shortcut. Thus, the expected benefit of potentially dismissing the candidate no longer outweighs the work required to look up the shortcut. Therefore, we do not prune candidate journeys with already found shortcuts.

Parent Pointers. In order to determine the shortcut that corresponds to a candidate journey, the ULTRA preprocessing algorithm maintains parent pointers for the stops of the candidate journeys. These parent pointers point to earlier stops within the same journey and can thus be used to find the intermediate transfer of a journey by tracing them back, starting from the last stop of the journey. Since we want to compute shortcuts between stop events instead of stops, we also change the parent pointers from stops to stop events. As before, in the original ULTRA preprocessing, we propagate parent pointers by assigning $\text{parent}[w] \leftarrow \text{parent}[v]$, whenever relaxing an edge (v, w) leads to an improved arrival time at w. Doing this enables an efficient retrieval of the shortcut corresponding to the intermediate transfer of a candidate journey. Assume that a candidate journey J ends at the stop t. In this case, the shortcut corresponding to the intermediate transfer of J is $(\text{parent}_1[v(\text{parent}_2[t])], \text{parent}_2[t])$, where $\text{parent}_k[v]$ is the parent for reaching v using k trips (i.e., within the k-th RAPTOR round). As before, witness journeys are distinguished from candidate journeys by assigning a special value to the parent pointers of witness journeys.

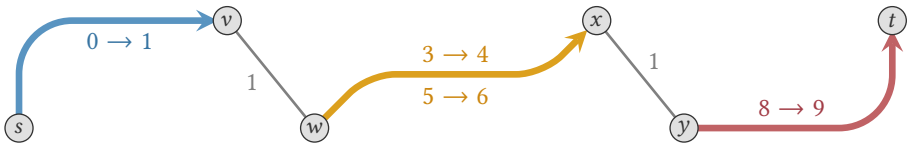


Figure 6.1: An example network that demonstrates how using weak domination in the ULTRA-Trip-Based preprocessing leads to missing shortcuts. Transfer edges (gray) are labeled with their travel time, while trips (colored) are labeled with $\tau_{\text{dep}} \to \tau_{\text{arr}}$. With weak domination of candidates, the preprocessing only finds two shortcuts: $(0 \to 1, 3 \to 4)$ and $(5 \to 6, 8 \to 9)$. However, these two shortcuts are not sufficient for finding an s-t-journey. If candidate journeys are only dismissed if they are strictly dominated by a witness journey, then an additional shortcut $(3 \to 4, 8 \to 9)$ is found during the preprocessing. Using this shortcut, the s-t-journey $\langle\langle s\rangle, \langle 0 \to 1\rangle, \langle v, w\rangle, \langle 3 \to 4\rangle, \langle x, y\rangle, \langle 8 \to 9\rangle, \langle t\rangle\rangle$ can be computed.

Initial Transfer and Strict Dominance. The most important modification of the algorithm is required due to the fact that the ULTRA preprocessing allows witness journeys with initial transfers (unlike Trip-Based). In combination with weak domination of candidates, this can lead to missed shortcuts between stop events, as demonstrated in Figure 6.1. In this example, only two shortcuts will be found: $(0 \to 1, 3 \to 4)$ and $(5 \to 6, 8 \to 9)$. However, these two shortcuts are not sufficient for finding a journey from s to t with the Trip-Based query algorithm. The algorithm will only find journeys starting at s that reach the only trip of the blue route $(0 \to 1)$ and the first trip of the yellow route $(3 \to 4)$. No further journeys can be found, since there is no transfer shortcut from the blue route to the second trip of the yellow route $(0 \to 1, 5 \to 6)$ and no transfer from the first trip of the yellow route to the red route $(3 \to 4, 8 \to 9)$. Either one of these shortcuts would be sufficient for the computation of journeys from s to t. We argue that, considering these two options, adding $(3 \to 4, 8 \to 9)$ as a shortcut is preferable. The reason for this is that passengers using the blue route would have no reason to wait for the second trip of the yellow route if they can also continue with the first trip of the yellow route.

Before explaining the modifications that are necessary in order to find the shortcut $(3 \to 4, 8 \to 9)$, we briefly examine why this shortcut is not found by a naive combination of the ULTRA preprocessing and the Trip-Based preprocessing. For this, we consider the candidate journey $J^c = \langle\langle w\rangle, \langle 3 \to 4\rangle, \langle x, y\rangle, \langle 8 \to 9\rangle, \langle t\rangle\rangle$, which contains the missing shortcut. During the ULTRA preprocessing, this journey is dominated by the witness journey $J = \langle\langle w\rangle, \langle 5 \to 6\rangle, \langle x, y\rangle, \langle 8 \to 9\rangle, \langle t\rangle\rangle$, hence no shortcut is added. Note that this problem only arises when ULTRA and Trip-Based Routing are combined. When using ULTRA on its own, shortcuts connect pairs

of stops instead of stop events. This means that the two shortcuts ($3 \to 4, 8 \to 9$) and ($5 \to 6, 8 \to 9$) between stop events are both represented with the single shortcut (x, y) between stops. Therefore, finding only one of them is sufficient. On the other hand, when using Trip-Based Routing on its own, the problem does not arise, as the Trip-Based preprocessing does not consider journeys with initial transfers. This means that the candidate journey J^c is not dominated by the witness journey J, since J requires waiting at w, which is considered to be an initial transfer. Therefore, the shortcut ($5 \to 6, 8 \to 9$) is found by the standard Trip-Based preprocessing.

We observe that the problem of missing shortcuts only occurs if a candidate journey and the corresponding witness journey are equivalent with respect to their arrival time and their number of used trips. Thus, the problem can be solved by only dismissing candidate journeys that are strictly dominated by a witness (instead of being weakly dominated as in standard ULTRA). We now continue with describing how this change can be implemented within our preprocessing algorithm. Using strict dominance instead of weak dominance affects all parts of the algorithm where a new arrival time at a vertex v is discovered (i.e., during the relaxation of edges and during route scanning). Normally the label of v is only updated if the newly discovered arrival time is strictly better (earlier) than the previously found arrival time. Instead, we now also update the label of the vertex v if the following three conditions hold: First, the new arrival time at the vertex v is equivalent to the previous arrival time. Secondly, the current label of the vertex v does not correspond to a candidate journey. Thirdly, the journey that corresponds to the new arrival time is a candidate journey. These new rules for updating a label ensure that a newly found candidate journey is not implicitly dominated by a previously found journey with the same arrival time. In the case of equal arrival times, we allow that candidate journeys replace non-candidate journeys, but not vice versa. This is quite important, as it prevents cyclic label updates, which would otherwise lead to infinite loops.

6.3.2 Improved Query

We use the shortcuts computed by the combined ULTRA-Trip-Based preprocessing within a modified version of the Trip-Based query algorithm. As before, with the normal ULTRA query, this requires a special treatment of initial and final transfers, since the shortcuts only cover intermediate transfers. We handle these transfers by performing two Bucket-CH queries, just like we do in the general ULTRA query. However, in contrast to the general ULTRA query, efficiently integrating the results of the Bucket-CH queries into the Trip-Based query is more involved. We provide an overview that shows how initial and final transfers are processed in our ULTRA-Trip-Based query in Algorithm 6.3. In the following, we describe this algorithm in detail.

Bucket-CH Query. The first step of the algorithm (lines 1 - 4) is the execution of the Bucket-CH queries. As in the case of the generic ULTRA query, we split the Bucket-CH queries into three parts, in order to improve efficiency. First, a standard CH query from s to t with departure time τ_{dep} is performed. As result of this query, we obtain the minimal arrival time τ_{min} at the target via a direct path in the transfer graph, the forward CH search space $\mathcal{V}_s$ originating from s, and the backward CH search space $\mathcal{V}_t$ originating from t. The minimal arrival time τ_{min} is ∞ if no path from the source s to the target t exists in the transfer graph. If, on the other hand, $\tau_{\text{min}} < \infty$ holds, then we have found an s-t-journey with arrival time τ_{min} that uses zero trips, which we add to the result set in line 2. Afterwards, we evaluate the buckets containing vertex-to-stop transfer times for all vertices in $\mathcal{V}_s$, which provides us with the arrival time $\tau_{\text{arr}}(s, v)$ for each stop v with $\tau_{\text{arr}}(s, v) \leq \tau_{\text{min}}$. Similarly, we evaluate the buckets containing stop-to-vertex transfer times for all vertices in $\mathcal{V}_t$, in order to obtain transfer times $\tau_{\text{tra}}(v, t)$ for all stops v with $\tau_{\text{tra}}(v, t) \leq \tau_{\text{min}} - \tau_{\text{dep}}$.

Initial Transfer Evaluation. In the second step of the algorithm (lines 5 - 19), we evaluate which trips of the public transit network are reachable by an initial transfer. In the original Trip-Based query [Wit15], this is done by iterating over all stops that are reachable via an initial transfer. For each such stop v and each route R visiting v, the algorithm identifies the earliest trip of R that can be entered at v after taking the initial transfer. This approach is efficient as long as the number of stops reachable via an initial transfer is small. However, in a scenario with unlimited transfers, where almost all stops are reachable by initial transfers, consecutive stops of a route often share the same earliest reachable trip. This can cause the same trip to be found multiple times, leading to redundant work. To avoid this, we propose a new approach for evaluating the initial transfers, which is based on two steps of the RAPTOR algorithm: collecting updated routes and scanning routes.

We start by collecting all routes, which contain a stop that is reachable by an initial transfer from the source, in lines 5 and 6. This is analogous to collecting routes that contain updated stops at the beginning of a RAPTOR round. We proceed by scanning the routes we have collected. The goal of this step is to find for each stop v within a route R the first trip T_{min} of the route R that can be boarded at v, given the arrival time $\tau_{\text{arr}}(s, v)$ at v. We achieve this by processing the stops v in the order they appear in R, while gradually updating T_{min} at the same time.

Let v be the next stop to be processed while scanning the route R. If we have not found a reachable trip for any of the previous stops in R (i.e., $T_{\text{min}} = \infty$), then we use a binary search to find the first trip in R that can be boarded at v (line 13). Otherwise, we assume that the earliest reachable trip at v is probably not much earlier than the previously found trip T_{min}. Therefore, we perform a linear search,

starting from T_{min}, to find this trip in lines 15 - 17. Note that in cases where the earliest reachable trip at v departs later than T_{min}, the linear search will not find it. However, this is not a problem, since it is preferable to enter T_{min} at a previous stop, in this case. After we have found the earliest trip reachable at v, we add it to the queue of trips that have to be scanned in line 18. Finally, we can stop searching for earlier trips if T_{min} is already the earliest trip in the route R.

The original Trip-Based query also collects final transfers to the target before continuing with the trip scanning step. These are used in the trip scanning step to efficiently identify the stops in the trip from which the target can be reached. In the presence of unlimited transfers, this is no longer worth the effort, since the target can be reached from almost all stops. Therefore, we skip this step and evaluate final transfers on the fly while scanning trips. Unfortunately, skipping the evaluation of initial transfers is not an option, as we need to evaluate them in order to know which trips have to be scanned.

Trip Scanning. The third and last step of the query algorithm (lines 20 - 33) is the trip scanning phase, which is mostly identical to the original Trip-Based query algorithm. It is organized in rounds, where the n-th round scans the trips that have previously been collected in Q_n, which correspond to journeys that start at s and contain n trips. For each of these trips, the queue also contains indices i and j, which indicate the first and last stop event of the trip that have to be scanned, respectively. While scanning the i-th stop event of the trip T, the algorithm checks whether a final transfer from the i-th stop of the trip T to the target exists in line 24. If such a transfer exists and if this transfer can be used to improve the earliest known arrival time τ_{min} at the target, then the algorithm has found a new Pareto-optimal journey. In this case, τ_{min} is updated and the newly found journey is added to the result set $\mathcal{J}$. If $\mathcal{J}$ already contains a journey with n trips (note that a Pareto-set can only contain one such journey), this journey is replaced.

After the final transfers have been evaluated, we continue with relaxing the precomputed transfer edges in $\mathcal{E}^{\text{t}}$ that start at the stop event $T[i]$. Each of these edges provides us with a new trip T' that can be used to extend the current journey. Thus, the trip T' (together with the index i' of the first stop event in T' that can be reached) is added to the queue Q_{n+1} of trips that have to be scanned in the next round.

Note that we scan the trips in Q_n twice. We only evaluate final transfers during the first scan and use a separate second scan to relax transfer shortcuts. We do this for two reasons: First, separating the two scans improves memory locality, as $\tau_{\text{tra}}(\cdot, t)$ is only accessed by the first scan, and $\mathcal{E}^{\text{t}}$ is only accessed by the second scan. Secondly, we improve τ_{min} throughout the first scan, which enables better pruning of trips that cannot contribute to Pareto-optimal journeys in line 30 of the second scan.

Algorithm 6.3: ULTRA-Trip-Based Query

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{R})$, transfer shortcut graph $G^{\mathrm{t}} = (\mathcal{V}^{\mathrm{t}}, \mathcal{E}^{\mathrm{t}})$
Bucket-CH of the original transfer graph G,
source vertex s, departure time τ_{dep}, and target vertex t
Output: All Pareto-optimal journeys from s to t for departure time τ_{dep}

$(\tau_{\mathsf{min}}, \mathcal{V}_s, \mathcal{V}_t) \leftarrow$ Run a CH query from s to t with departure time τ_{dep}
if $\tau_{\mathsf{min}} < \infty$ **then** $\mathcal{J} \leftarrow \{(\tau_{\mathsf{arr}}(s,t), 0)\}$
$\tau_{\mathsf{arr}}(s, \cdot) \leftarrow$ Evaluate the vertex-to-stop buckets for vertices in $\mathcal{V}_s$
$\tau_{\mathsf{tra}}(\cdot, t) \leftarrow$ Evaluate the stop-to-vertex buckets for vertices in $\mathcal{V}_t$
for each $v \in \mathcal{V}_s$ **do**
  $\mathcal{R}' \leftarrow \mathcal{R}' \cup \{$Routes from $\mathcal{R}$ that contain $v\}$
for each $R \in \mathcal{R}'$ **do**
  $T_{\mathsf{min}} \leftarrow \infty$
  for i from 0 to $|R|$ **do**
    $v \leftarrow i$-th stop of trips in R
    if $\tau_{\mathsf{arr}}(s, v) \geq \tau_{\mathsf{min}}$ **then continue**
    if $T_{\mathsf{min}} = \infty$ **then**
      $T_{\mathsf{min}} \leftarrow$ Binary search: first $T \in R$ departing from v after $\tau_{\mathsf{arr}}(s, v)$
    else
      while the trip before T_{min} in R departs from v after $\tau_{\mathsf{arr}}(s, v)$ **do**
        $T_{\mathsf{min}} \leftarrow$ The trip before T_{min} in R
        if T_{min} is the first trip in R **then break**
    if $T_{\mathsf{min}} \neq \infty$ **and** $\tau_{\mathsf{dep}}(T_{\mathsf{min}}[i]) \geq \tau_{\mathsf{arr}}(s, v)$ **then** Enqueue$(T_{\mathsf{min}}, i, Q_1)$
    if T_{min} is the first trip in R **then break**
$n \leftarrow 1$
while Q_n is not empty **do**
  for each $(T, j, k) \in Q_n$ **do**
    for i from j to k **do**
      if $\tau_{\mathsf{arr}}(T[i]) \geq \tau_{\mathsf{min}}$ **then break**
      if $\tau_{\mathsf{arr}}(T[i]) + \tau_{\mathsf{tra}}(v(T[i]), t) < \tau_{\mathsf{min}}$ **then**
        $\tau_{\mathsf{min}} \leftarrow \tau_{\mathsf{arr}}(T[i]) + \tau_{\mathsf{tra}}(v(T[i]), t)$
        $\mathcal{J} \leftarrow$ Pareto-set of $\mathcal{J} \cup \{(\tau_{\mathsf{min}}, n)\}$
  for each $(T, j, k) \in Q_n$ **do**
    for i from j to k **do**
      if $\tau_{\mathsf{arr}}(T[i]) \geq \tau_{\mathsf{min}}$ **then break**
      for each $(T[i], T'[i']) \in \mathcal{E}^{\mathrm{t}}$ **do**
        Enqueue(T', i', Q_{n+1})
  $n \leftarrow n + 1$

Enqueueing Trips. The enqueue operation, which is used to add trips to the queues in lines 18 and 32, is identical to the enqueue operation proposed for the original Trip-Based query algorithm [Wit15]. Internally, this operation maintains an index k for every trip T in the network. This index marks the last stop event of the trip that has not been scanned and is initialized as $|T|$. When invoking Enqueue(T, i, Q_n), this index is used to add the triple (T, i, k) to the queue Q_n. Afterwards, k is decreased to $i - 1$ for this trip and all later trips in the route of T.

Data Structures and Memory Layout. In order to achieve the optimal performance possible for the query algorithm, it is quite important that a streamlined memory layout is used. To this end, we implement the FIFO queues Q_n using dynamic arrays. This enables an efficient enqueue operation and efficient scanning of the entries in Q_n. The edges $\mathcal{E}^{\mathrm{t}}$ are also stored in an array, such that edges $(T[i], T_a[j])$ and $(T[i], T_b[k])$, which start at the same stop event $T[i]$, occupy consecutive memory locations. Moreover, the section of this array that contains edges starting with the stop event $T[i]$ is directly in front of the section that contains edges starting with the stop event $T[i+1]$. Finally, we observe that we only need access to the arrival time $\tau_{\mathrm{arr}}(T[i])$ and the stop $v(T[i])$ of the stop events $T[i]$ during the trip scanning step. Thus, we store these values separately from the departure time $\tau_{\mathrm{dep}}(T[i])$ of the stop event, which improves memory locality.

6.4 Experiments

We implemented our algorithms in C++17 and compiled them with GCC version 8.2.1 and optimization flag -O3. Experiments were performed on the following machines:

Xeon A machine with two 8-core Intel Xeon Skylake SP Gold 6144 CPUs, which are clocked at 3.50 GHz, with a boost frequency of up to 4.2 GHz, 192 GiB of DDR4-2666 RAM, and 24.75 MiB of L3 cache.

Epyc A machine with two 64-core AMD Epyc Rome 7742 CPUs, which are clocked at 2.25 GHz, with a boost frequency of up to 3.4 GHz, 1024 GiB of DDR4-3200 RAM, and 256 MiB of L3 cache.

6.4.1 Preprocessing

In this section we evaluate the performance of the ULTRA preprocessing phase, which includes the transfer graph contraction and the shortcut computation. We start by focusing on the Switzerland network, where we analyze the effects of the parameters core degree, witness limit, and transfer speed in great detail. Afterwards, we discuss the general results of the preprocessing phase for all four networks.

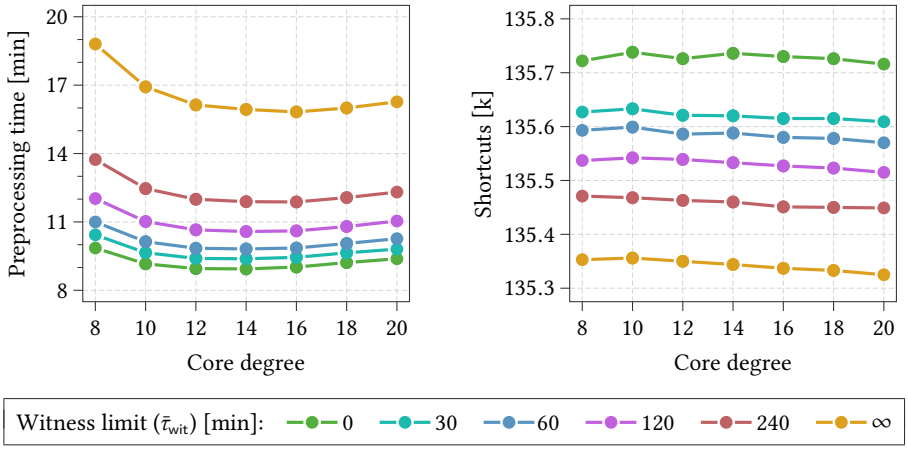


Figure 6.2: Impact of the core degree and the witness limit on the running time of the preprocessing algorithm and the number of computed shortcuts, measured for the Switzerland network on the Xeon machine. Preprocessing time includes both contracting the transfer graph and computing the shortcuts.

Core Degree and Witness Limit. The two main parameters influencing the performance of the ULTRA preprocessing are the average vertex degree of the contracted transfer graph and the witness limit $\bar{\tau}_{\text{wit}}$. Figure 6.2 shows the impact of these two parameters on the Switzerland network. The lowest preprocessing times are achieved with a core degree of 14. While the actual shortcut computation still becomes slightly faster for higher core degrees, this is offset by the increased time required to contract the transfer graph. Contracting up to a core degree of 14 took 8:46 minutes on the Xeon machine and yielded a graph with 32 683 vertices and 466 331 edges.

Overall, the witness limit $\bar{\tau}_{\text{wit}}$ has a much more significant impact on the preprocessing time and the number of computed shortcuts than the core degree. Choosing a witness limit of 0 instead of ∞ approximately cuts the preprocessing time in half. By contrast, the witness limit only has a minor impact on the number of computed shortcuts, with a difference of fewer than 500 shortcuts between $\bar{\tau}_{\text{wit}} = 0$ and $\bar{\tau}_{\text{wit}} = \infty$. For all following experiments, we chose a witness limit of 15 minutes, which yields 135 687 shortcuts for the Switzerland network.

The only network where we use a core degree of 20 instead of 14 is the Germany network. We do this since the share of the core computation in the overall preprocessing time is significantly lower for this network, due to its much larger size. As a result, the contraction took 23:18 minutes and produced a core graph with 313 241 vertices and 6 264 851 edges. Preprocessing results for all four networks are listed in Table 6.2.

Table 6.1: Runtime of the different preprocessing steps required for ULTRA. For the Core-CH and Bucket-CH we use a sequential algorithm. For the ULTRA shortcut computation, which is the slowest preprocessing step, we report sequential and parallel running times. We report preprocessing times for both machines, Xeon and Epyc.

		Stuttgart	London	Switzerland	Germany
Xeon	Core-CH time	2:18	0:22	1:29	23:18
	Bucket-CH time	3:36	0:13	0:49	16:46
	ULTRA time	52:56	3:49:11	1:42:48	102:26:41
	ULTRA time (×2)	28:12	2:00:56	53:09	52:27:19
	ULTRA time (×4)	14:57	1:04:02	28:33	27:12:54
	ULTRA time (×8)	7:59	34:06	15:11	14:18:12
	ULTRA time (×16)	4:19	18:42	8:46	7:33:09
Epyc	Core-CH time	2:42	0:27	1:36	26:02
	Bucket-CH time	4:07	0:14	1:00	18:31
	ULTRA time	1:33:40	5:02:49	2:11:13	127:11:51
	ULTRA time (×2)	49:25	2:47:59	1:09:34	66:47:30
	ULTRA time (×4)	25:00	1:25:32	36:33	34:58:59
	ULTRA time (×8)	12:45	42:48	18:19	17:50:13
	ULTRA time (×16)	6:28	22:03	9:12	9:06:13
	ULTRA time (×32)	3:17	11:03	4:41	4:41:26
	ULTRA time (×64)	1:56	6:19	2:44	2:55:53
	ULTRA time (×128)	1:09	4:16	1:58	2:35:23

Parallelization. In the previous experiment we used all 16 cores of the Xeon machine for the shortcut computation. In order to assess the impact of the parallel execution on the preprocessing time, we repeat the shortcut computation with fewer threads. Additionally, we compare running times of the Epyc machine, which has a lower single core performance but contains more cores than the Xeon machine. An overview of the preprocessing times on both machines is given in Table 6.1. Overall, we observe that the parallelized preprocessing algorithm is quite efficient and achieves good speed-up factors for all networks on both machines. For the Switzerland network the maximal speed-up of the ULTRA shortcut computation is 11.7 on the Xeon machine and 66.7 on the Epyc machine. When including the CH computations, which were not parallelized, the overall speed-up of the preprocessing phase drops to 9.5 and 29.3, respectively. Independent of the network we observe the smallest speed-up when switching from 64 threads to 128 threads on the Epyc machine. In this case the speed-up is most likely limited by the memory bandwidth.

Table 6.2: An overview of the ULTRA preprocessing results. We report the number of computed shortcuts and the size of the underlying core graph.

	Stuttgart	London	Switzerland	Germany
Number of core vertices	30 012	24 838	32 683	313 241
Number of core edges	420 178	347 737	466 331	6 264 851
Number of shortcuts	77 498	164 869	135 687	2 068 544

Transfer Speed. In order to test the impact of the used transfer mode on the shortcut computation, we changed the transfer speed in the Switzerland network from 4.5 km/h to different values between 1 km/h and 140 km/h. We considered two ways of applying the transfer speed: In the first version, we did not allow the transfer speed on an edge to exceed the speed limit given in the road network. This allowed us to model fast transfer modes such as cars fairly realistically. In the second version, we ignored speed limits and assumed a constant speed on every edge. Thus, we can analyze to which extend the effects observed in the first version are caused by the speed limit data. Figure 6.3 reports the preprocessing time and number of computed shortcuts measured for each configuration. In all measurements, the preprocessing time remained below 15 minutes. A peak in the number of shortcuts is reached between 10 and 20 km/h, which roughly corresponds to the speed of a bicycle. The number of shortcuts then starts decreasing again for higher transfer speeds and reaches a plateau at around 188 000 shortcuts, if speed limits are ignored. If speed limits are obeyed, the number of shortcuts eventually rises again and reaches the overall peak at 140 km/h, which is the highest speed limit present in the network.

For low to medium transfer speeds, the results conformed with our expectations. As the transfer speed increases, it becomes increasingly feasible to cover large distances in the transfer graph quickly, making it possible to transfer between trips that are further away from each other. Accordingly, new shortcuts appear between these trips. However, once the transfer speed becomes competitive with the public transit vehicles, it eventually becomes preferable to avoid the public transit network altogether and transfer directly from source to target. In this case, all journeys using trips from the public transit network are dominated by the journey corresponding to the direct transfer. Since no shortcuts are required for such pairs of source and target stop, we would expect a sharp decrease in the number of shortcuts. However, the result of our experiment (Figure 6.3, right plot) does not conform this expectation.

The reason why this decrease is not observed in our measurements is that not all stops in our network instances are connected by the transfer graph. Consider what happens in the shortcut computation for journeys between stops s and t that are isolated from each other and the rest of the transfer graph. In this case, a direct

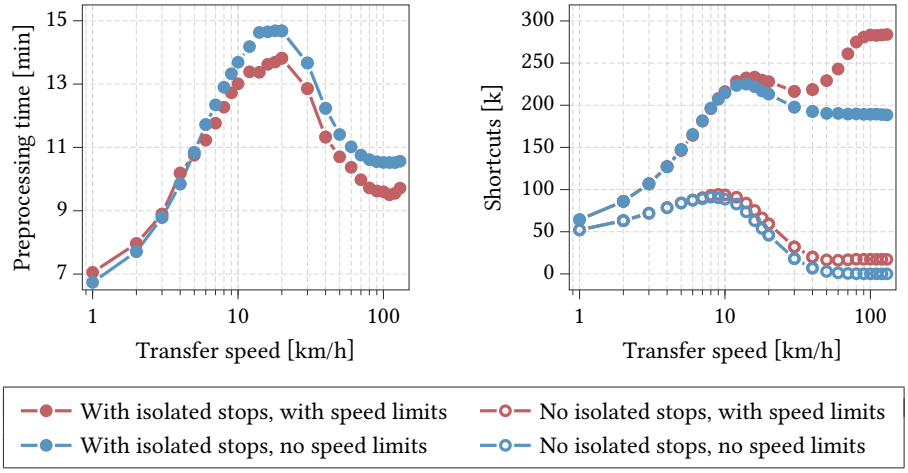


Figure 6.3: Impact of transfer speed on preprocessing time and number of shortcuts, measured on the Switzerland network with a core degree of 14 and a witness limit of 15 min. Speed limits were obeyed for the red lines and ignored for the blue lines. For the two lines at the bottom of the right plot, shortcuts were only added if the source and target of the candidate journey are connected by a path in the transfer graph. This shows that reducing the number of isolated stops can drastically improve the number of required shortcuts. However, this does not impact the preprocessing time.

transfer is not possible, regardless of the transfer speed. In fact, unless there is a route that serves both s and t, any optimal journey from s to t will include at least two trips. If a transfer is necessary between these two trips, then this journey is a non-dominated candidate journey and a shortcut is added for the corresponding transfer. In our Switzerland network, 624 stops are isolated from the transfer graph, usually as a result of incomplete or imperfect data. To assess the impact of these stops on the number of computed shortcuts, we repeated our experiments. However, this time we do not add shortcuts to the result if the source and target stop of the corresponding candidate journey were not connected in the transfer graph. This resulted in much fewer shortcuts, especially for high transfer speeds. If speed limits are ignored, the amount of necessary shortcuts becomes negligible at around 60 km/h and eventually reaches 0. If speed limits are obeyed, the number of shortcuts stagnates at 17 000.

Overall, these experiments show that our shortcut computation remains feasible regardless of the speed of the used transfer mode. Moreover, if the network does not include many stops that are isolated from the transfer graph, transferring between stops is most useful for transfer speeds between 10 and 20 km/h.

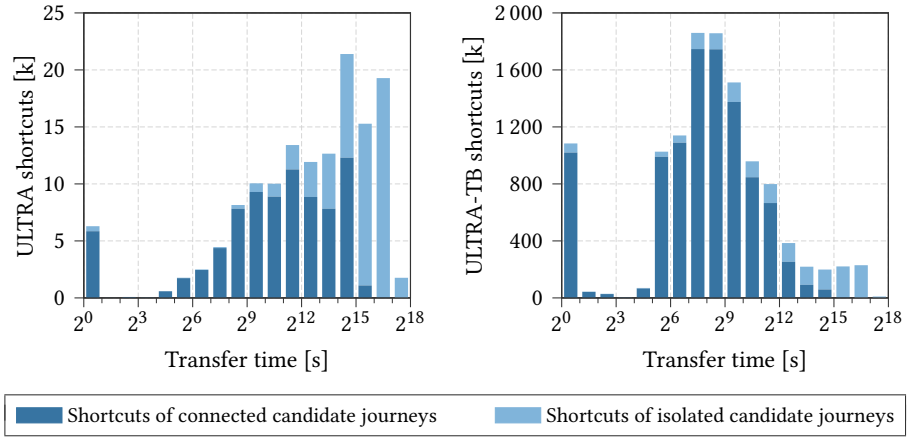


Figure 6.4: Distribution of the precomputed shortcuts with respect to their transfer time for the Switzerland network. The bar between 2^i and 2^{i-1} corresponds to the number of shortcuts with a transfer time in the interval $[2^i, 2^{i-1})$. An exception is the first bar, which also contains shortcuts with a transfer time of less than a second. The dark blue portion of each bar represents shortcuts where the source and the target of the corresponding candidate journey are connected by a path in the transfer graph. *Left:* Shortcuts between stops as computed by the ULTRA preprocessing. *Right:* Shortcuts between stop events as computed by the ULTRA-Trip-Based preprocessing.

Shortcut Graph Structure. The shortcut graph computed by the ULTRA preprocessing phase for Switzerland is structurally very different from the transitively closed transfer graph, which we created in Chapter 4 for public transit algorithms like RAPTOR or CSA. This is already evidenced by the fact that the shortcut graph is much less dense, containing only 3% as many edges as the transitively closed graph. Furthermore, the transitive graph consists of many small fully connected components, with the largest one containing only 1 233 vertices. By contrast, the largest strongly connected component in the shortcut graph contains 10 186 vertices, which corresponds to 40% of all stops. Accordingly, a transitive closure of the shortcut graph would contain more than 100 million edges.

During the construction of the transitively closed transfer graph (Section 4.3) we observed that preserving all transfers with a duration of up to 9 minutes already leads to a transfer graph with a mean vertex degree of more than 100. Thus, we concluded that public transit journey planning algorithms, which require a transitively closed transfer graph, cannot be efficient and at the same time guarantee that long transfers are found. We now compare the travel time of the transfers in the transitive graph

and the shortcuts computed by ULTRA. Looking at the distribution of travel times for the ULTRA shortcuts in Figure 6.4 (left side), we observe that most of the shortcuts have a travel time of more than 9 minutes ($\approx 2^9$ seconds). Thus, most of the shortcuts are not contained in the transitive transfer graph. Only 33 765 edges are shared between the two graphs, which represent 0.7% of all transitive edges and 24.9% of all shortcuts. Altogether, this shows that the transitively closed graph fails to represent most of the relevant intermediate transfers, at the expense of many superfluous ones.

Note that the high number of shortcuts with travel time 0 is caused by cases where several stops model the same physical location. The bars in Figure 6.4 are subdivided into shortcuts that arise from candidate journeys where the source and target stop are connected by a path in the transfer graph (dark blue) and shortcuts where this is not the case (light blue). As before, we make this distinction in order to identify effects caused by imperfect or incomplete data. We observe that stops which are isolated from the rest of the transfer graph not only cause many additional shortcuts, but also that these shortcuts are disproportionately long compared to the other shortcuts. Because of this we suspect that many ULTRA shortcuts are only required by a few special journeys and that they are only relevant at a few times during a day.

We can analyze this effect more thoroughly by looking at the time-dependent ULTRA-Trip-Based shortcuts (which connect stop events instead of stops) in Figure 6.4 (right side). An ULTRA shortcut that is used multiple times throughout a day leads to several ULTRA-Trip-Based shortcuts since they connect stop events, which occur at a fixed point in time. Thus, the number of ULTRA-Trip-Based shortcuts with a certain travel time reflects more accurately how frequently these shortcuts are required. We observe that most ULTRA-Trip-Based shortcuts have a travel time between 2 minutes ($\approx 2^7$s) and 17 minutes ($\approx 2^{10}$s). This is quite different from the original ULTRA, where most shortcuts have a travel time of more than one hour ($\approx 2^{12}$s). Therefore, we conclude that long shortcuts are indeed only rarely required. Furthermore, we observe that the fraction of shortcuts that are added due to candidate journeys between vertices that are not connected in the transfer graph (light blue) is much lower when using the ULTRA-Trip-Based preprocessing instead of the normal ULTRA preprocessing.

Trip-Based Preprocessing. In our final experiment concerning the preprocessing phase we address the ULTRA-Trip-Based preprocessing. An overview of the results obtained by both preprocessing variants is given in Table 6.3. Here, rows labeled with *(integrated)* refer to our new integrated preprocessing approach, while rows labeled with *(sequential)* refer to the naive sequential approach, i.e., using the output of the standard ULTRA preprocessing as input for the Trip-Based preprocessing algorithm. The results show that using our novel integrated preprocessing leads to a significant

Table 6.3: An overview of the ULTRA-Trip-Based preprocessing results. We compare the naive sequential approach for combining ULTRA and Trip-Based Routing with our improved integrated preprocessing variant. Running times were measured on the Xeon machine using all 16 cores and are displayed in the mm:ss format.

	Stuttgart	London	Switzerland	Germany
Shortcuts (sequential)	25 865 892	58 301 120	58 807 528	1 072 750 574
Shortcuts (integrated)	3 900 258	19 856 062	11 646 572	121 676 520
Time (sequential)	4:40	19:15	9:16	7:54:13
Time (integrated)	5:11	22:24	10:04	9:16:15

reduction in the amount of computed shortcuts. This effect is weakest for the London network, where the number of shortcuts decreases only by a factor of 3. For our largest network (i.e., the Germany network) the sequential approach produces over 1 billion shortcuts while the integrated approach only leads to 121 million shortcuts, which corresponds to a reduction factor of almost 9. The cost for this reduction in the number of shortcuts is an increased running time of the preprocessing algorithm. However, in comparison to the significantly decreased number of shortcuts, the running time overhead is only minor. For our four test networks, the increase in preprocessing time ranges from 8% for the Switzerland network to 17% for the Germany network.

Note that all time measurements reported in Table 6.3 were obtained by parallel execution with 16 threads. We have shown before that the ULTRA preprocessing is well suited for parallel execution, and the same holds true for the Trip-Based preprocessing [Wit15]. This also applies to our new integrated preprocessing algorithm. As an example, we have performed the single-threaded preprocessing on the Switzerland network, where we measured running times of 1:48:55 for the sequential approach and 2:11:16 for the integrated approach. This corresponds to a speed-up factor of 11.8 and 13.0 respectively, which matches the speed-ups observed for the ULTRA preprocessing and the Trip-Based preprocessing.

6.4.2 Queries

To evaluate the impact of our ULTRA shortcuts on the query performance, we test them with two public transit algorithms, RAPTOR and CSA. For each algorithm, we compare three variants: one using our ULTRA approach, one using a transitively closed transfer graph, and one using a multimodal variant of the algorithm. Additionally, we compare these algorithms to our integrated ULTRA-Trip-Based approach. Since we only consider sequential query algorithms, we use the Xeon machine (which has a better single core performance) for all following experiments.

Table 6.4: Query performance for CSA, MCSA, and ULTRA-CSA. Query times are divided into two phases: initialization including initial transfers (Init.), and connection scans including intermediate transfers (Scan). All results are averaged over 10 000 random queries. Note that CSA (marked with *) only supports stop-to-stop queries with transitive transfers. In contrast, the other two algorithms support vertex-to-vertex queries on the full graph, and have been evaluated for this query type.

Network	Algorithm	Full graph	Scans [k]		Time [ms]		
			Connection	Edge	Init.	Scan	Total
Stuttgart	CSA*	○	101.6	826.8	0.0	3.9	4.0
	MCSA	●	92.6	1 378.8	0.1	21.9	22.2
	ULTRA-CSA	●	89.2	104.1	1.1	3.0	4.1
London	CSA*	○	48.7	199.3	0.0	1.4	1.4
	MCSA	●	29.0	2 152.5	0.4	109.9	111.6
	ULTRA-CSA	●	28.0	42.5	1.1	1.1	2.2
Switzerland	CSA*	○	126.7	1 307	0.2	5.0	5.2
	MCSA	●	88.0	5 337	12.9	48.4	61.3
	ULTRA-CSA	●	86.8	51	1.8	3.0	4.8
Germany	CSA*	○	2 620.3	6 216	2.9	162.1	165.1
	MCSA	●	1 568.2	118 026	233.6	1462.5	1696.1
	ULTRA-CSA	●	1 553.8	659	25.7	114.6	140.2

CSA Queries. The first query algorithm we evaluate is CSA. We only consider the earliest arrival variant of CSA since the bicriteria variant is outperformed by RAPTOR, which we evaluate in the next section. Since no multimodal variant of CSA has been published thus far, we implemented a naive multimodal version of CSA, which we call MCSA (Multimodal CSA), as a baseline for our comparison. This algorithm alternates connection scans with Dijkstra searches on the contracted core graph, in a similar manner to MCR. Query times for all three CSA variants are reported in Table 6.4. We observe for all networks that ULTRA-CSA has a running time similar to CSA. This is despite the fact that ULTRA-CSA solves a multimodal journey planning problem while CSA solves a much simpler public transit journey planning problem.

Furthermore, we observe that the search space (i.e., the number of scanned connections) of ULTRA-CSA is significantly smaller than the search space of CSA. This is a direct result of the fact that multimodal journeys have usually a shorter travel time. Since CSA scans connections in chronological order, the number of scanned connections correlates directly with the arrival time of the query.

Table 6.5: Query performance for RAPTOR, MR-∞, and ULTRA-RAPTOR. Query times are divided into phases: scanning initial transfers (Init.), collecting routes (Coll.), scanning routes (Scan), and relaxing transfers (Relax). All results are averaged over 10 000 random queries. Note that RAPTOR (marked with *) only supports stop-to-stop queries with transitive transfers, whereas the other two algorithms support vertex-to-vertex queries on the full graph and have been evaluated accordingly.

Network	Algorithm	Full graph	Scans [k]		Time [ms]				
			Route	Edge	Init.	Coll.	Scan	Relax	Total
Stuttgart	RAPTOR*	○	20.2	1 242	0.0	1.7	2.1	3.0	6.8
	MR-∞	●	38.8	817	13.4	5.7	5.3	13.1	38.7
	ULTRA-RAPTOR	●	41.3	103	1.4	3.9	3.8	1.3	10.5
London	RAPTOR*	○	5.2	7 097	0.0	1.6	3.1	12.1	16.9
	MR-∞	●	6.0	556	6.0	2.3	3.4	7.5	19.5
	ULTRA-RAPTOR	●	6.5	181	0.9	1.9	2.8	1.7	7.6
Switzerland	RAPTOR*	○	27.2	3 527	0.0	3.7	6.4	7.8	18.4
	MR-∞	●	34.9	769	11.6	5.9	8.2	12.3	39.3
	ULTRA-RAPTOR	●	35.8	135	1.3	4.6	6.8	1.7	14.5
Germany	RAPTOR*	○	480.4	25 798	0.0	166.9	178.0	85.1	436.5
	MR-∞	●	555.8	12 571	191.1	250.7	202.2	272.2	944.1
	ULTRA-RAPTOR	●	573.4	2 183	25.7	189.2	170.1	29.8	415.2

Compared to MCSA we observe that our new approach is about one magnitude faster on all networks. This is because the performance of CSA mainly stems from the high memory locality of its sequential connection scan. However, MCSA loses this memory locality, as it has to perform a Dijkstra search every time an arrival time is updated after scanning a connection. It is quite likely that this problem is the reason why no multimodal variant of CSA has been published thus far. When using ULTRA-CSA, however, memory locality is restored because only a few shortcut edges have to be relaxed after scanning each connection. As a result, ULTRA-CSA is the first efficient multimodal variant of CSA.

RAPTOR Queries. In the case of RAPTOR, we used the MR-∞ variant of MCR as the multimodal baseline algorithm. The results of our comparison are shown in Table 6.5. Using ULTRA-RAPTOR drastically reduces the time consumption for exploring the transfer graph (the *Relax* phase) compared to MR-∞. While this phase

takes 50%–60% of the overall running time of MR-∞, it only takes 10%–20% of the running time of ULTRA-RAPTOR. The reason for this is that both phases, scanning the initial/final transfers and relaxing the intermediate transfers, are an order of magnitude faster in ULTRA-RAPTOR compared to MR-∞. For the initial and final transfers, the Core-CH search of MR-∞ is replaced by a much more efficient Bucket-CH query in ULTRA-RAPTOR. Similarly, ULTRA-RAPTOR uses the precomputed shortcuts for relaxing the intermediate transfers whereas MR-∞ performs an inefficient Dijkstra search on the core graph. Overall, ULTRA-RAPTOR is twice as fast as MR-∞ and has a similar running time to RAPTOR with transitive transfers. Note that comparing the running times of RAPTOR and ULTRA-RAPTOR has to be done with caution, as they were measured for different sets of queries. Nonetheless, our experiments clearly demonstrate that our novel shortcut technique enables RAPTOR to use unrestricted transfers without incurring the performance loss that is associated with MCR.

If we compare CSA-based queries and RAPTOR based queries, we see that ULTRA-CSA is about 3 to 4 times faster than ULTRA-RAPTOR. This difference is of cause due to that fact that CSA only computes earliest arrival times, while RAPTOR computes a full Pareto-set with respect to the criteria arrival time and number of used trips. However, we still observe that on most networks MCSA is slower than MR-∞. As mentioned before, this is due to the fact that CSA looses its efficiency if the scanning of the connections is interleaved with other tasks.

Impact of Transfer Speed. In addition to overall query performance, we also measured how query times of RAPTOR and ULTRA-RAPTOR are impacted by the transfer speed. Results are shown in Figure 6.5 (left side). The performance gains for ULTRA-RAPTOR compared to MR-∞ are similar for all transfer speeds, and in fact slightly better for higher speeds. To explain this, observe that the time required for the route scanning phase decreases as the transfer speed increases. This is because the total number of rounds and thus the number of scanned routes decreases for higher transfer speeds. ULTRA-RAPTOR benefits more from this since the share of the route scanning phase in the overall running time is greater for ULTRA-RAPTOR than for MR-∞. In all cases, the entire query time for ULTRA-RAPTOR is similar to or lower than the time that MR-∞ takes for the route scanning phases only.

The impact of the transfer speed on the travel time of the fastest journey is shown in Figure 6.5 (right side). As the transfer speed increases, the overall travel time decreases and the share of the travel time that is spent on an initial or final transfer becomes larger. From around 50 km/h onward, transferring directly from source to target is the best option in most cases. In contrast to initial and final transfers, intermediate transfers have a very small impact on the overall travel time, further demonstrating that long intermediate transfers are rarely needed.

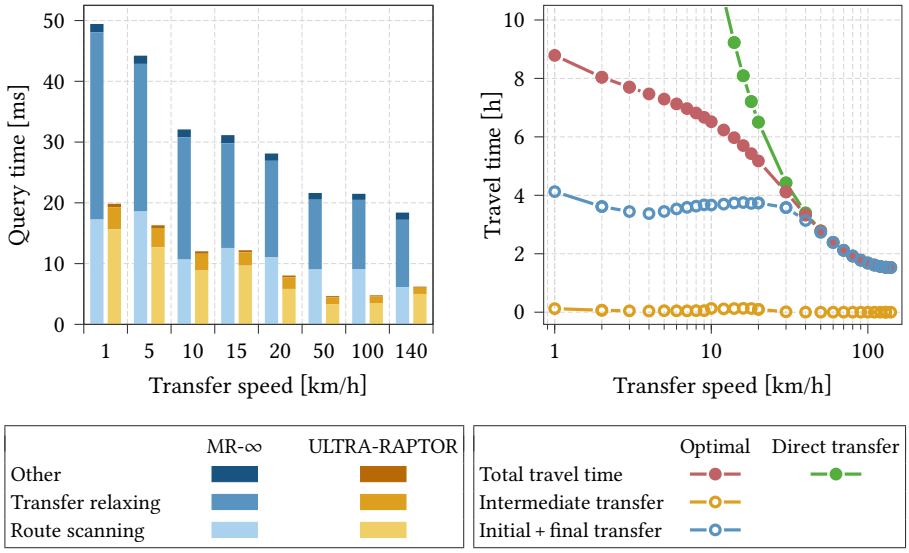


Figure 6.5: Impact of transfer speed on query times and travel times, measured on the Switzerland network with a core degree of 14 and a witness limit of 15 minutes. All results were averaged over 10 000 random queries. *Left:* Query performance of MR-∞ and ULTRA-RAPTOR. Speed limits were obeyed during the construction of the transfer graph. Query times are divided into route collecting/scanning, transfer relaxation, and remaining time. *Right:* Total travel time and time spent on initial/final and intermediate transfers for the journey with minimal arrival time. Additionally, we include a plot of the time required for a direct transfer from source to target as a reference.

Trip-Based Queries. We continue with evaluating our improved ULTRA-Trip-Based query algorithm. Table 6.6 presents the average query performance (based on 10 000 random queries) for all four networks. For comparison, we also include the original Trip-Based algorithm, which cannot solve multimodal queries and thus was evaluated using a different set of random queries. Overall, we see that our improved Trip-Based query combined with the integrated preprocessing yields the lowest query times, independent of the network. For the Germany network, our new algorithm is about 4 times faster than ULTRA-RAPTOR and more than 10 times faster than MR-∞, which previously was the fastest multimodal journey planning algorithm (compare tables 6.6 and 6.5). If the sequential preprocessing is used instead of the integrated version then the running time of the query algorithm increases by a factor of 2. However, this version of ULTRA-Trip-Based is still faster than the other algorithms.

Table 6.6: Query performance for Trip-Based Routing and ULTRA-Trip-Based (ULTRA-TB, sequential and integrated). Query times are divided into phases: the Bucket-CH query (B-CH), the initial transfer evaluation (Initial), and the scanning of trips (Scan). All results are averaged over 10 000 random queries. Note that Trip-Based (marked with *) only supports stop-to-stop queries with transitive transfers, whereas the other two algorithms support vertex-to-vertex queries on the full graph.

Network	Algorithm	Full graph	Scans [k]		Time [ms]			
			Trip	Shortcut	B-CH	Initial	Scan	Total
Stuttgart	Trip-Based*	○	11.5	257.1	0.01	0.03	2.04	2.09
	ULTRA-TB (seq.)	●	25.0	1 528.5	1.41	0.92	5.99	8.33
	ULTRA-TB (int.)	●	17.0	218.4	1.35	0.81	2.38	4.55
London	Trip-Based*	○	22.7	1 376.3	0.01	0.05	6.10	6.16
	ULTRA-TB (seq.)	●	34.1	1 545.1	0.91	0.80	7.47	9.19
	ULTRA-TB (int.)	●	24.7	450.5	0.90	0.70	4.05	5.66
Switzerland	Trip-Based*	○	23.8	757.5	0.01	0.04	5.64	5.70
	ULTRA-TB (seq.)	●	36.5	1 551.1	1.09	1.15	7.18	9.44
	ULTRA-TB (int.)	●	23.5	238.1	1.07	1.03	3.19	5.32
Germany	Trip-Based*	○	337.5	16 116.6	0.01	0.05	116.14	116.21
	ULTRA-TB (seq.)	●	439.4	38 092.3	25.34	18.96	151.35	195.67
	ULTRA-TB (int.)	●	204.2	3 149.9	26.12	19.13	46.38	91.65

For most networks, ULTRA-Trip-Based is even faster than the original Trip-Based algorithm, despite the fact that ULTRA-Trip-Based handles a large, realistic transfer graph while Trip-Based can only consider transitively closed transfer graphs. The reason for this is the reduced size of the search space due to better pruning of the shortcuts and the existence of faster journeys in a network with unlimited transfers. The only exception to this is the Stuttgart network, which has the fewest trips, but the second-largest transfer graph out of our four networks. Thus, the comparison with an algorithm that cannot handle unlimited transfer graphs, such as Trip-Based, is particularly unfair for the Stuttgart network.

In addition to the total query time, we also report time measurements for the three phases of the Trip-Based query algorithm in Table 6.6. Analyzing these measurements, we see that the Bucket-CH query and the initial transfers evaluation take a non-negligible fraction of the total running time for both ULTRA-Trip-Based variants. Furthermore, we observe that using the integrated preprocessing mainly affects the

trip scanning phase of the algorithm. This was expected, as the preprocessing does not affect initial transfers, but only intermediate transfers, which are handled in the trip scanning phase. Moreover, we observe that the integrated preprocessing not only reduces the number of shortcuts that are scanned during the query, but also the number of trips. For the largest network (Germany) the query algorithm scans less than half as many trips when the integrated preprocessing is used instead of the sequential variant.

Impact of the Query Distance.. We conclude the experimental evaluation with a comparison of running times depending on the query distance. In Figure 6.6 we compare the running times of the three fastest bicriteria algorithms (MR-∞, ULTRA-RAPTOR, and ULTRA-Trip-Based) depending on the geo-rank of the query. In order to generate geo-rank queries, a source vertex is picked uniformly at random among all vertices in the network. Afterwards, all vertices are sorted by their geographical distance from the source vertex. The vertex with index i in this order is then the target of the geo-rank query for rank i. For our comparison in Figure 6.6 we generated and evaluated 10 000 of these queries for the Germany network. We observe that independently of the geo-rank the three algorithms have a clear order with respect to running time. For all geo-ranks, our new ULTRA-Trip-Based algorithm is an order of magnitude faster than MR-∞. The running time of ULTRA-RAPTOR lies between these two algorithms and is closer to the running time of the ULTRA-Trip-Based algorithm for local queries, while it is closer to MR-∞ for long range queries. Furthermore, we observe that many short range queries can be solved in less than one millisecond by ULTRA-Trip-Based algorithm with integrated preprocessing.

A comparable geo-rank-based evaluation on the Germany network has also been performed for the original Trip-Based algorithm in [Wit15]. While the results from [Wit15] are mostly similar to our results, they contain significantly more outliers. Across all geo-ranks the evaluation for the original Trip-Based algorithm shows a considerable number of queries that take more than 10 milliseconds. The reason that we found fewer outliers in our evaluation is most likely caused by a better correlation between geo-rank and query complexity in multimodal networks (compared to public transit networks that were evaluated in [Wit15]).

The extreme outliers for low geo-ranks in Figure 6.6 can be attributed to queries where the source vertex is located in particularly sparse parts of the network. The reason for this is a poor correlation between geo-rank and actual distance in sparse parts of the network. Thus, a query can be a long-range query despite having a low geo-rank. An example for this are the queries with geo-rank 2^7, which corresponds to a distance of less than 1 km for most source-target pairs. However, the source of the query that took about 50 ms with ULTRA-Trip-Based (int.) is located in Prague, while its target is located in Germany, which is more than 80 km away.

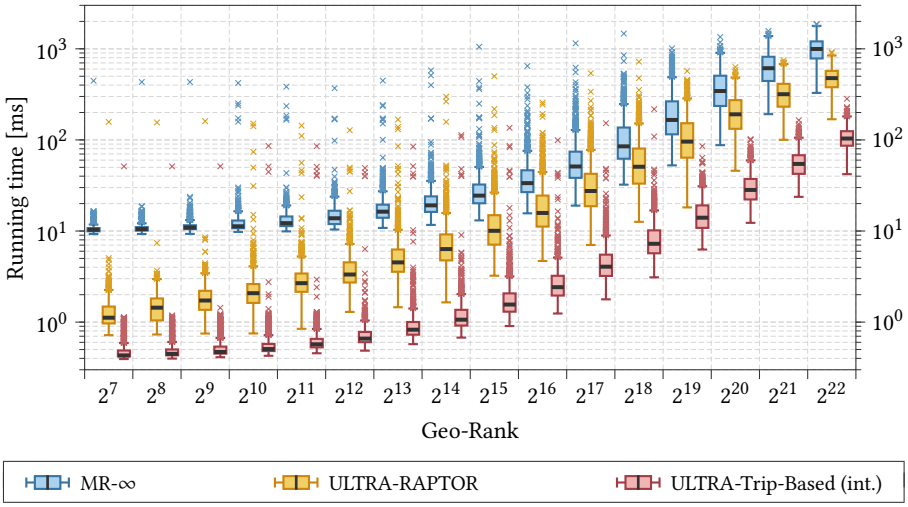


Figure 6.6: Comparison of query times depending on the geo-rank for the Germany network. We evaluated 10 000 random vertex-to-vertex queries for each geo-rank. We compare the previously fastest multimodal journey planning algorithm (MR-∞) to our two bicriteria ULTRA query algorithms: ULTRA-RAPTOR and ULTRA-Trip-Based.

Comparison with HL-based Routing. Most recently, a new speedup-technique that is based on HL has been proposed for multimodal journey planning [PV19]. Similar to ULTRA, this approach can be combined with different public transit algorithms and was evaluated for the variants HLCSA and HLRAPTOR. While these two algorithms are faster than MR-∞, they are outperformed by ULTRA. The preprocessing of HLCSA and HLRAPTOR takes between one and two hours for the networks of London and Switzerland, whereas the ULTRA can process these networks in less than 5 minutes. Regarding query performance we find that ULTRA-CSA is 11.1 times faster than HLCSA and ULTRA-RAPTOR is 3.6 times faster than HLRAPTOR.

6.5 Final Remarks

In this chapter we developed ULTRA, a technique that significantly accelerates the computation of Pareto-optimal journeys in a public transit network with an unrestricted transfer graph. We achieved this by computing shortcuts that provably represent all necessary intermediate transfers. Parallelization enables fast preprocessing, taking only a few minutes for smaller networks and about 3 hours for Germany.

Our evaluation showed that the number of computed shortcuts is low, regardless of the underlying transfer mode. The shortcuts can be used without adjustments by any public transit algorithm that previously required a transitively closed transfer graph.

For RAPTOR and CSA, we showed that using shortcuts leads to similar query times as using a transitively closed transfer graph. Consequently, shortcuts enable the computation of unrestricted multimodal journeys without incurring the performance losses of existing multimodal algorithms. In particular, combining shortcuts with CSA yields the first efficient multi-modal variant of CSA.

For Trip-Based Routing we presented a tailored variant of the ULTRA preprocessing. Our integrated preprocessing variant produces up to 9 times fewer shortcuts than a naive sequential combination of ULTRA and Trip-Based Routing, at only a slight increase in preprocessing time. Furthermore, we presented an improved ULTRA-Trip-Based query algorithm, which is an order of magnitude faster than the fastest previously known multimodal algorithm for bicriteria optimization, MR-∞.

Future Work. For future work, we would like to develop a variant of the ULTRA preprocessing that can handle additional Pareto criteria, such as walking distance or cost. Furthermore, it would be interesting to adapt our shortcut computation to scenarios where public transit vehicles can be delayed. It is of course possible to continue using our ULTRA shortcuts if some public transit vehicles are delayed, just like current public transit approaches do not change the transitive transfer graph in such a case. However, such an approach cannot guarantee to find all optimal multimodal journeys, since journeys with delayed vehicles might require additional shortcuts. We suspect, however, that the underlying principle of ULTRA (i.e., the fact that the set of all intermediate transfers is small) is still valid in a scenario with delays.

7 Bike Sharing

In the previous chapters we have developed algorithms that can handle public transit networks with one additional transfer mode. For multimodal journey planning we of course want to be able to handle more than two transportation modes. While it is possible for ULTRA to handle additional transfer modes, this is not meaningful in the context of the problems that we have addressed thus far. For example, we could combine the public transit networks with two transfer graphs, one for walking and one for cycling. However, in this case no optimal journey would contain walking transfers since cycling is strictly better than walking in all situations.

In order to obtain a meaningful multimodal journey planning problem, we either have to consider additional optimization criteria, such as cycling duration, or we have to incorporate other constraints, such that walking is preferable to cycling in some cases. Journey planning problems with more than two optimization criteria have already been considered in several works [MS07, DMS08, Del+13]. Most commonly these problems are solved by maintaining bags of non-dominated labels for each vertex, as first proposed for the multicriteria variant of Dijkstra's algorithm [Mar84].

In this chapter we explore the second approach, i.e., as before, we only optimize two criteria, but we add additional constraints that lead to an overall more realistic problem formulation. In particular, we consider a scenario with bike sharing, where bicycles have to be rented and returned at bike sharing stations. Furthermore, we improve the model of the public transit network by taking into account that bicycle transport may not be permitted for every trip in the network.

This chapter is based on joint work with Jonas Sauer and Dorothea Wagner, which has previously been published in [SWZ20a].

7.1 Preliminaries

In this section we introduce special notation regarding bike sharing. Additionally, we provide a precise definition of the problem that we address in this chapter.

Bike Sharing. In addition to walking, we consider cycling as a second transfer mode. For each edge $e = (v, w)$ in the transfer graph, we define a *walking time* $\tau_{\text{walk}}(e)$ and a *biking time* $\tau_{\text{bike}}(e)$, which represent the time required to travel from v to w by walking or cycling, respectively. Note that $\tau_{\text{walk}}(e)$ or $\tau_{\text{bike}}(e)$ may be ∞ to signify that e is not usable in the respective transfer mode. Some trips in the public transit network may allow passengers to carry along a bicycle with them, while others may not. The *bicycle transport function* $b\colon \mathcal{T} \to \{\texttt{true}, \texttt{false}\}$ maps to each trip $T \in \mathcal{T}$ a boolean value $b(T)$ that indicates whether T allows bicycle transport or not.

Bikes can be rented from a number of different *bike sharing operators*. We denote the number of bike sharing operators by σ and associate each operator with a number from $\{1, \ldots, \sigma\}$. For simplicity, we will use the number 0 to denote that a passenger is currently not renting a bike. Each operator i operates a set $\mathcal{BS}_i \subseteq \mathcal{V}$ of *bike sharing stations* where passengers can pick up or drop off a bike. Note that a vertex may act as a bike sharing station for more than one operator. Each bike sharing station v has an associated *pick up time* $\tau_{\text{pick}}(v)$ that is required to pick up a bike, and a *drop off time* $\tau_{\text{drop}}(v)$ that is required to drop off a bike. A passenger may only carry one bike at a time.

Journey. A *journey* J defines the movement of a passenger through the public transit network when traveling from a source vertex $s \in \mathcal{V}$ to a target vertex $t \in \mathcal{V}$. As before, it is an alternating sequence of *trip legs* and *transfers*, where a trip leg is a subsequence of a trip that represents the passenger using that portion of the trip, and a transfer is a path in the transfer graph that connects the final stop of one trip leg (or s for the initial transfer) with the first stop of the following trip leg (or t for the final transfer). Note that some or all of the transfers may be empty.

To define which parts of the journey use bike sharing, the journey is augmented with a sequence $\langle (v_1, w_1), \ldots, (v_n, w_n) \rangle$, with $(v_i, w_i) \in \mathcal{BS}_j$ for some $j \in \{1, \ldots, \sigma\}$, that contains one tuple of bike sharing stations for each bike that is rented during the journey. For the i-th rented bike, v_i is the station where the bike is picked up and w_i is the station where it is dropped off. It is required that there is a bike sharing operator j for which $v_i, w_i \in \mathcal{BS}_j$ holds. This ensures that multiple bikes are not rented at the same time and that the journey starts and ends with no bike rented.

The bike sharing stations in the aforementioned sequence must be visited by the transfers of the journey in the same order in which they appear in the sequence. If v_i and w_i are visited during different transfers, all trip legs that lie between these two

transfers must allow bicycle transport. For every transfer edge $e \in \mathcal{E}$ used between v_i and w_i, the travel time for using the edge is $\tau_{\text{bike}}(e)$. For every edge $e \in \mathcal{E}$ that is traversed without a bike, the travel time is $\tau_{\text{walk}}(e)$.

Problem Statement. The criteria we use to evaluate a journey J are its arrival time at the target vertex t and the number of used public transit vehicles, i.e., the number of trip legs in J. A journey J *dominates* another journey J' if J arrives no later than J' and does not use more trips than J'. We call a journey J *Pareto-optimal* if it is not dominated by any other journey. A *Pareto-set* is a set containing a minimal number of journeys such that every valid journey is dominated by a journey in the set. Naturally, all journeys in a Pareto-set are Pareto-optimal. Given source and target vertices $s, t \in \mathcal{V}$ and an earliest departure time τ_{dep}, our objective is to find a Pareto-set among all journeys from s to t that depart no later than τ_{dep}.

7.2 Models for the Bike Sharing Problem

We continue with presenting two approaches for solving the journey planning problem with multiple bike sharing operators. First, we show how MCR can be adapted to handle renting and returning of bicycles explicitly within the algorithm. We call this approach the *operator-dependent* model. Secondly, we introduce the *operator-expanded* model, where all relevant information regarding bike sharing is encoded directly in the network. This allows any existing multimodal journey planning algorithm to handle bike sharing without modifications. We demonstrate this by using ULTRA as an example. Finally, we introduce a preprocessing-based speed-up technique called *operator pruning*, which can be incorporated into MCR and ULTRA.

7.2.1 Operator-Dependent Model

Most public transit algorithms, including MCR, work by propagating vertex labels through the network. For the two criteria arrival time and number of trips, a label at a vertex v can be represented as a tuple (τ_{arr}, k), where τ_{arr} is the arrival time at v, and k is the number of trips used so far. Bike sharing can be incorporated by extending the labels to triples $(\tau_{\text{arr}}, k, i)$, where i is the operator of the currently rented bike (or 0 if no bike is rented). Since rented bikes have to be dropped off before the end of the journey, only labels at the target vertex t with operator 0 represent complete journeys. Whenever a label $(\tau_{\text{arr}}, k, 0)$ reaches a bike sharing station v, a new label $(\tau_{\text{arr}} + \tau_{\text{pick}}(v), k, i)$ must be created for each operator i with $v \in \mathcal{BS}_i$, to represent the passenger picking up a bike of operator i. Similarly, when a label $(\tau_{\text{arr}}, k, i)$ with $i \neq 0$ reaches a bike sharing station $v \in \mathcal{BS}_i$, a new label $(\tau_{\text{arr}} + \tau_{\text{drop}}(v), k, 0)$ must

be created to represent the passenger dropping off the bike. The time required to traverse an edge $e \in \mathcal{E}$ is $\tau_{\text{bike}}(e)$ for labels with a rented bike and $\tau_{\text{walk}}(e)$ otherwise. When propagating a label with operator $i \neq 0$ through a route, any trip T that does not permit bicycle transport (i.e., $b(T) = \texttt{false}$) must be ignored.

Without bike sharing, a label (τ_{arr}, k) may dominate another label (τ'_{arr}, k') if $\tau_{\text{arr}} \leq \tau'_{\text{arr}}$ and $k \leq k'$ hold. The same dominance rule still applies to labels with the same bike sharing operator: A label $(\tau_{\text{arr}}, k, i)$ dominates a label $(\tau'_{\text{arr}}, k', i)$ if $\tau_{\text{arr}} \leq \tau'_{\text{arr}}$ and $k \leq k'$ hold. However, as the following lemma shows, it is not possible to establish dominance rules for labels with different operators:

Lemma 7.1. Let $A = (\tau_{\text{arr}}, k, i)$ and $B = (\tau'_{\text{arr}}, k', j)$ be two labels at some vertex v with $\tau_{\text{arr}} \leq \tau'_{\text{arr}}$, $k \leq k'$, and $i \neq j$. Then, A may not dominate B.

Proof. If $i = 0$, label B has access to a bike and A does not. Using the bike may allow the journey represented by B to overtake the journey represented by A and reach the target faster. If $i \neq 0$, then the passenger represented by A is carrying a bike, which must be returned before reaching the target. This may require a detour that may not be required for B, possibly allowing the journey represented by B to overtake and reach the target faster. □

MCR with Bike Sharing. Following lemma 7.1 we see that bike sharing can be incorporated into the Pareto-optimization by treating the bike sharing operator as a third criterion whose values are all incomparable with each other. In MCR, the number of trips is not stored directly in the labels but unrolled into the round data structure. For the MR-∞ variant of MCR, which only optimizes arrival time and number of trips, it is sufficient to store a single arrival time per vertex and round. Variants with additional criteria replace the single arrival time with a bag of non-dominated labels. When a new label is added, it must be compared to all other labels in the bag to eliminate dominated labels. A naive approach to incorporating bike sharing would be to store bags of labels with two criteria: arrival time and bike sharing operator. However, like the number of transfers, the operator criterion is discrete and only permits a few possible values. Thus, it is more efficient to unroll it into the data structure as well: For each vertex and round, we store an array $(\tau^0_{\text{arr}}, \ldots, \tau^\sigma_{\text{arr}})$, where τ^i_{arr} is the best arrival time achieved so far with operator i. As shown by Lemma 7.1, a new label with operator i only needs to be compared with τ^i_{arr}, since it is incomparable to the other entries. Thus, we can handle all operators independently of each other and do not need bags at all. In each round, we perform $\sigma+1$ independent route scanning phases, where phase i only considers labels with operator i. The resulting algorithm is a variant of MR-∞ whose worst-case running time is proportional to that of the original MR-∞ and $\sigma+1$.

7.2.2 Operator-Expanded Network

One drawback of the operator-dependent model is that it requires algorithms to be explicitly adapted for bike sharing. An alternative is to unroll the bike sharing information into an enlarged public transit network. Any existing multimodal public transit algorithm can then handle bike sharing without modification by operating on this *operator-expanded* network.

Network Construction. Given an original public transit network N and transfer graph G, the operator-expanded network $(N^{\mathrm{e}}, G^{\mathrm{e}})$ structurally consists of $\sigma + 1$ copies $N^0, \dots, N^\sigma$ of N and $\sigma + 1$ copies $G^0, \dots, G^\sigma$ of G. The idea is that N^i and G^i are used by passengers who are currently renting a bike from operator i, with N^0 and G^0 representing walking. Accordingly, the travel time of an edge $e \in \mathcal{E}^i$ is $\tau_{\mathrm{walk}}(e)$ if $i = 0$ holds and $\tau_{\mathrm{bike}}(e)$ otherwise. In all copies N^i with $i \neq 0$, trips that do not allow bicycle transport are removed. The network copies are connected as follows: For a vertex $v \in \mathcal{V}$ in the original network, we denote its copy in N^i and G^i by v^i. For each operator i and each bike sharing station $v \in \mathcal{BS}_i$, the expanded network includes the edges (v^0, v^i) with weight $\tau_{\mathrm{pick}}(v)$ and (v^i, v^0) with weight $\tau_{\mathrm{drop}}(v)$. These edges represent picking up and dropping off a bike, respectively.

Let $\mathcal{OP} = \{1, \dots, \sigma\}$ be the set of all bike sharing operators. We then use the elements of $\mathcal{OP}_0 := \mathcal{OP} \cup \{0\}$ to represent the operator of a rented bike, where 0 indicates that no bike is currently rented. Based on this, we then define the operator-expanded network formally as $N^{\mathrm{e}} = (\mathcal{S}^{\mathrm{e}}, \mathcal{T}^{\mathrm{e}}, \mathcal{R}^{\mathrm{e}})$ and $G^{\mathrm{e}} = (\mathcal{V}^{\mathrm{e}}, \mathcal{E}^{\mathrm{e}})$, with

$$
\begin{aligned}
\mathcal{S}^{\mathrm{e}} &= \{v^i \mid i \in \mathcal{OP}_0 \wedge v \in \mathcal{S}\},\\
\mathcal{T}^{\mathrm{e}} &= \{T^i = \langle v_0^i, \dots, v_k^i \rangle \mid i \in \mathcal{OP}_0 \wedge T = \langle v_0, \dots v_k \rangle \in \mathcal{T} \wedge (b(T) \vee i = 0)\},\\
\mathcal{R}^{\mathrm{e}} &= \{\{T^i \mid T \in R \wedge (b(T) \vee i = 0)\} \mid i \in \mathcal{OP}_0 \wedge R \in \mathcal{R}\},\\
\mathcal{V}^{\mathrm{e}} &= \{v^i \mid i \in \mathcal{OP}_0 \wedge v \in \mathcal{V}\},\\
\mathcal{E}^{\mathrm{e}} &= \{(v^i, w^i) \mid i \in \mathcal{OP}_0 \wedge (v, w) \in \mathcal{E}\} \cup \{(v^0, v^i), (v^i, v^0) \mid i \in \mathcal{OP} \wedge v \in \mathcal{BS}_i\}.
\end{aligned}
$$

An example of a normal public transit network and the operator-expanded network constructed from it is shown in Figure 7.1 on the next page.

An s-t-query on the original network can be solved with an s^0-t^0-query on the operator-expanded network, using an unmodified multimodal public transit algorithm (such as MCR) that no longer needs to handle bike sharing explicitly. For ULTRA, both the preprocessing and the query algorithm can be run on the operator-expanded network. As with MCR, ULTRA contracts the transfer graph before the preprocessing is performed. A naive approach would be to create the operator-expanded network first and then contract the expanded transfer graph. However,

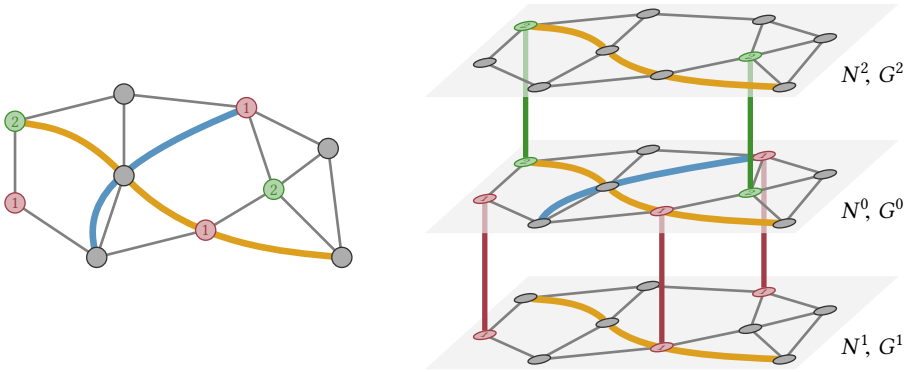


Figure 7.1: A public transit network (left) and the corresponding operator-expanded network (right). The network features two public transit routes, with the trips of the yellow route allowing bicycle transport and the trips of the blue route disallowing it. Bikes can be rented and returned at the three bike sharing stations of operator 1 (red) or at the two stations of operator 2 (green).

since the transfer graphs G^i of all networks with $i \neq 0$ are identical, this would lead to redundant work. Instead, it is more efficient to contract two copies of the original transfer graph: one with walking weights and one with biking weights. Bike sharing stations are left uncontracted at this point. These contracted copies can then be inserted into the operator-expanded network in place of the original graph. If desired, an additional contraction can then be performed on the resulting transfer graph of the operator-expanded network.

The ULTRA query consists of two phases: The Bucket-CH search for the initial and final transfers is done on the transfer graph of the operator-expanded network, taking care of bike renting automatically. The main public transit algorithm (e.g., RAPTOR) uses the operator-expanded network with the shortcuts computed by the preprocessing phase. A shortcut (v^i, w^j) corresponds to a shortcut (v, w) in the original network that requires a bike from operator i to access and ends with the passenger having rented a bike from operator j.

7.3 Operator Pruning

A crucial observation that can be used to speed up bike sharing queries is that bike sharing operators typically only serve a limited region (e.g., a single city). Taking a rented bike far outside that region is typically not useful, since the passenger would

eventually need to travel back in order to return the bike. Consider a journey that involves taking a train from region A served by operator i to region B served by operator j. If the passenger has rented a bike from operator i, it is usually preferable to return it before taking the train, and then rent a bike from operator j in region B if necessary. However, because labels with different operators may not dominate each other, MCR will also continue exploring the option where the passenger takes the bike from operator i into region B, even though it cannot be returned there. Without additional information, the algorithm will not be aware that the only way to turn this into a valid journey is to travel back to region A, return the bike and then come back to region B, creating an unnecessary detour. To prevent this, we compute an *operator hull* for each operator i, which is a region of the network outside of which it is never useful to travel with a bike from operator i. This allows algorithms to prune journeys once they leave the hull for operator i while carrying a bike from operator i.

Preprocessing. For the operator hull computation, we use the *cycling network* N^{c}, which we define as $N^{\mathsf{c}} := (\mathcal{S}, \mathcal{T}^{\mathsf{c}}, \mathcal{R}^{\mathsf{c}})$ with trips $\mathcal{T}^{\mathsf{c}} := \{T \in \mathcal{T} \mid b(T) = \texttt{true}\}$ and routes $\mathcal{R}^{\mathsf{c}} := \{R \in \mathcal{R} \mid \exists T \in R\colon b(T) = \texttt{true}\}$. The cycling network N^{c} together with the transfer graph G and the travel time function τ_{bike} is the network as it appears to a passenger who is using a bike for the entirety of the journey. The *operator hull* $H^i := (\mathcal{V}^i, \mathcal{T}^i)$ for an operator i consists of a set of vertices $\mathcal{V}^i \subseteq \mathcal{V}$ and a set of trips $\mathcal{T}^i \subseteq \mathcal{T}$ such that every journey in N^{c} between bike sharing stations $s, t \in \mathcal{BS}_i$ is dominated by a journey in N^{c} that only uses vertices in $\mathcal{V}^i$ and trips in $\mathcal{T}^i$. It can be computed by running a profile variant of MCR (without bike sharing) on N^{c} from each station $s \in \mathcal{BS}_i$ and unpacking all found journeys ending at another station $t \in \mathcal{BS}_i$. The individual profile searches can be sped up with a simple pruning rule: A profile search from a source station s starts by exploring the initial transfers, computing for each vertex $v \in \mathcal{V}$ the biking time $\tau_{\mathsf{bike}}(s, v)$ from s to v. From this we can compute $\tau^{\mathsf{max}}_{\mathsf{bike}} := \max_{t \in \mathcal{BS}_i} \tau_{\mathsf{bike}}(s, t)$, which is the maximum biking time to any other bike sharing station of the same operator. Since no optimal journey in N^{c} that ends at a station in $\mathcal{BS}_i$ may be longer than this, the profile search can prune labels whose travel time exceeds $\tau^{\mathsf{max}}_{\mathsf{bike}}$.

Note that the operator hull is an overapproximation of the region outside of which it is never useful to travel with a bike: Journeys that are optimal in N^{c} and therefore contribute to the operator hull may be dominated by journeys that require switching between different bike operators or dropping off a bike to take a trip without bicycle transport. These journeys could be excluded from the hull by using MCR with bike sharing for the hull computation. While this might lead to smaller hulls, it would require significantly higher computation times. Moreover, because the hull computation for one operator would no longer be independent of the other operators,

any change in the set of bike sharing stations for one operator would necessitate recomputing all hulls, rather than just the one for the changed operator.

Combination with MCR. Incorporating operator pruning into MCR is straightforward: During the Dijkstra searches, labels with operator i are not propagated to vertices that are not in $\mathcal{V}^i$. Similarly, during the route scanning phase for operator i, routes with no trips in $\mathcal{T}^i$ are ignored and arrival times at stops not in $\mathcal{V}^i$ are not updated. While it would be possible to skip trips that are not in $\mathcal{T}^i$ during the individual route scans, this has no performance benefit since RAPTOR always scans routes until the last stop. Thus, if a trip is skipped, the route scan will simply continue with the next reachable trip.

Combination with Operator-Expanded Network/ULTRA. Given an operator hull $H^i = (\mathcal{V}^i, \mathcal{T}^i)$, we define the *hull network* $N_H^i := (\mathcal{S}^i, \mathcal{T}^i, \mathcal{R}^i)$ with $\mathcal{S}^i = \mathcal{S} \cap \mathcal{V}^i$ and $\mathcal{R}^i \subseteq \mathcal{R}$ being the set of routes for which at least one trip is contained in $\mathcal{T}^i$. The hull network is accompanied by a *hull graph* G_H^i, which is the subgraph of G that is induced by $\mathcal{V}^i$. In order to incorporate operator pruning into the operator-expanded network, we first compute operator hulls on the original, non-expanded network. For each operator $i \neq 0$, we then replace the network copy N^i with the hull network N_H^i. The network copy N^0 that represents walking is left unchanged. In the resulting network, leaving the operator hull for the currently rented bike is no longer possible because the corresponding parts of the network have been deleted. Accordingly, any algorithm that runs on this network (including ULTRA) will automatically benefit from operator pruning.

7.4 Extended Scenarios

Free-Floating Bike Sharing. Some bike sharing operators use a *free-floating* (or *dockless*) sharing system without fixed stations, where bikes can be picked up or dropped off at any location within the served region. This can be handled by considering every vertex in the region as a bike sharing station. Unlike with fixed bike sharing stations, this scenario is inherently dynamic: Bikes are not available at every vertex in the region, and it is not known in advance where bikes will be located. Therefore, precomputation techniques such as ULTRA are not applicable. MCR can handle this by checking explicitly whether a bike is available when arriving at a vertex. Operator pruning can also be adapted by running the profile variant of MCR from each boundary vertex of the region, in addition to including all vertices and trips within the region in the hull.

Fixed Pick up Stations with Free-Floating Drop off. We also consider a hybrid system where pick up is restricted to fixed stations but drop off is allowed at any location. As with the fully station-based system, we assume that a bike is always available at every station. This makes ULTRA feasible again. Under the reasonable assumption that $\tau_{\text{bike}}(e) \leq \tau_{\text{walk}}(e)$ holds for every edge $e \in \mathcal{E}$, it only makes sense to drop off a bike at specific vertices: A bike from operator i may be dropped off at a stop (in order to enter a trip without bicycle transport), a pick up station from a different operator j (in order to switch operators), a boundary vertex of the served region (when leaving the region), or the target vertex. Dropping off the bike at any other vertex would cause unnecessary walking costs. For each such vertex v, the edge (v^i, v^0) is added to the operator-expanded network. Since the target vertex t changes with each query, the edge (t^i, t^0) is only inserted temporarily at query time.

Biking as Additional Trip. In the original MCR publication [Del+13], which considered bike sharing with only one operator, each bike that was used in a journey was counted as an additional trip. Incorporating this into our adapted version of MCR is straightforward. The operator pruning technique can be applied without changes, since its preprocessing only considers journeys that use a single bike for the entire journey. Adapting ULTRA, however, would require fundamental changes because it is based on enumerating all journeys with exactly two trips. If bike usage is counted as an additional trip, two trips are no longer sufficient for finding all relevant transfers.

7.5 Experiments

All algorithms presented in this chapter were implemented in C++17 and compiled with GCC 8.2.1 and optimization flag -O3. All experiments were conducted on a machine with two 8-core Intel Xeon Skylake SP Gold 6144 CPUs clocked at 3.5 GHz, with a turbo frequency of 4.2 GHz, 192 GiB of DDR4-2666 RAM, and 24.75 MiB of L3 cache.

Benchmark Data. We evaluated our algorithms on all four networks that were introduced in Chapter 4. However, we had to make minor adjustments to these networks, in order to account for the extended scenario covered in this chapter. First, we need two different travel times for all edges in the transfer graphs, one for walking and one for cycling. In order to obtain these travel times, we assumed an average walking speed of 4.5 km/h and an average cycling speed of 20 km/h. However, the cycling speed was reduced to the posted speed limit for streets where the speed limit is below 20 km/h. Secondly, we need to know the locations of bike sharing stations. We extracted this information for all four networks from OpenStreetMap[7]. We assigned each bike sharing station to an operator based on the information

[7] https://download.geofabrik.de/

Table 7.1: Sizes of the used public transit networks and their bike sharing systems.

	Stuttgart	London	Switzerland	Germany
Stops	13 583	20 595	25 125	244 055
Routes	12 350	2 107	13 785	231 084
Trips	91 298	125 436	350 006	2 387 292
Transfer graph vertices	1 166 593	183 025	603 691	6 872 105
Transfer graph edges	3 680 930	579 888	1 853 260	21 372 360
Bike Sharing Stations	326	823	534	2 682
Bike Sharing Operators	6	4	11	22

available (e.g., identifying different spellings of the same company). For stations that were not annotated with any information, we chose an operator of other nearby stations at random. Operators with only one bike sharing station were dropped from the dataset, as they are irrelevant for journey planning. Finally, we specify a pick up time of 20 seconds and a drop off time of 10 seconds for all bike sharing stations. An overview of the networks is given in Table 7.1.

7.5.1 Preprocessing

All algorithms discussed in this chapter require some form of preprocessing. The most elaborate preprocessing steps are our novel operator hull computation as well as the ULTRA shortcut computation. In addition to these two steps, several CH computations are required. An overview of all required preprocessing steps and their results is given in Tables 7.2 and 7.3.

Regarding the operator hull computation, we observe that the number of vertices contained in the union of all hulls is significantly smaller than the size of the corresponding core graph. This means that some parts of the network will never be visited with a rented bike. The small hulls also have a direct impact on the expanded networks, as their size is also significantly reduced when operator hulls are applied. Using 16 cores, hulls for the small networks can be computed in a few minutes, while Germany requires less than 9 hours. For the networks of Stuttgart, London, and Switzerland, this corresponds to a speed-up factor of 12 compared to a sequential computation. For the Germany network, we only achieve a speed-up of 9.5. In order to explain this effect, we measured the average number of instructions executed per CPU cycle. For the sequential hull computation on the Germany network, we recorded a value of 1.1, while it was only 0.9 for the parallel computation. These measurements suggest that the reduced speed-up observed for the Germany network is due to an overloaded memory system.

Table 7.2: Overview of all preprocessing steps for the operator-expanded approach *without* operator pruning. We report the sizes of the precomputed data structures as well as the required computation time. Entries marked with ? are not reported as their computation would take several weeks. We report single core running times for all preprocessing steps except for the ULTRA shortcut computation, which was performed in parallel using 16 cores. The *Total* row corresponds to the combined preprocessing time of all required preprocessing steps, using the parallel variant of the ULTRA shortcut computation and the single core version of the other steps.

		Stuttgart	London	Switzerland	Germany
Results	Walking core vertices	42 380	26 814	38 075	443 081
	Cycling core vertices	42 653	26 742	37 779	435 751
	Expanded stops	95 081	102 975	301 500	5 613 265
	Expanded vertices	1 422 818	290 791	1 019 779	16 461 221
	Expanded edges	3 146 277	2 081 218	7 673 596	155 594 242
	ULTRA shortcuts	929 575	1 831 779	3 389 309	?
Time [h:m:s]	Walking Core-CH	1:05	0:06	0:28	6:36
	Cycling Core-CH	1:05	0:06	0:28	6:43
	Expanded Core-CH	1:22	0:08	0:34	8:38
	ULTRA shortcuts (16)	3:27:53	14:14:51	9:59:43	?
	Expanded Bucket-CH	2:25	0:14	1:09	?
	Total	3:33:50	14:15:25	10:02:22	?

The impact of the operator hulls on the ULTRA shortcuts is also quite strong. On the London and Switzerland networks, operator hulls reduce the preprocessing time by a factor of 14 and 20, and the number of shortcuts by a factor of about 4 and 8, respectively. For the Germany network, ULTRA is only viable with operator hulls. Without operator hulls, only 4.7% of the shortcut computation was finished after one week. We therefore estimate that the complete shortcut computation would take about 21 weeks.

7.5.2 Queries

To evaluate the impact of operator pruning and the differences between the operator-dependent and operator-expanded models, we evaluated all algorithms presented in this chapter on 10 000 random queries. An overview of the results is given in Table 7.4.

We use MCR, or more specifically its MR-∞ variant, to compare the operator-dependent and operator-expanded model, as MR-∞ can be used with both models. Without operator pruning, both models perform similarly. This is to be expected, as

Table 7.3: Overview of all preprocessing steps for the operator-expanded approach *with* operator pruning. We report the sizes of the precomputed data structures and the required computation time. We report single core running times with the exception of two rows marked with (16), which correspond to parallel running times with 16 cores. The *Total* row corresponds to the combined preprocessing time of all required preprocessing steps, using the parallelized variant if it is available.

		Stuttgart	London	Switzerland	Germany
Results	Walking core vertices	42 380	26 814	38 075	443 081
	Cycling core vertices	42 653	26 742	37 779	435 751
	Operator hull vertices	27 996	13 735	19 018	351 224
	Expanded stops	25 875	31 216	36 892	411 980
	Expanded vertices	1 194 896	197 558	623 228	7 225 923
	Expanded edges	3 897 676	748 938	2 002 615	24 236 935
	ULTRA shortcuts	430 456	521 882	435 514	7 873 379
Time [h:m:s]	Walking Core-CH	1:05	0:06	0:28	6:36
	Cycling Core-CH	1:05	0:06	0:28	6:43
	Operator hulls	21:15	3:01:21	50:20	83:38:15
	Operator hulls (16)	1:43	15:34	4:15	8:45:22
	Expanded Core-CH	1:08	0:07	0:29	7:07
	ULTRA shortcuts (16)	23:05	43:31	21:50	30:50:13
	Expanded Bucket-CH	1:33	0:09	0:33	17:47
	Total	29:39	59:33	28:03	40:13:48

the operator-dependent algorithm more or less simulates what the standard algorithm does on the operator-expanded model. The number of rounds is exactly the same for both approaches, and the deviations in the number of settled vertices and scanned rounds can be explained by differences in the respective core graphs and better target pruning in the dependent model. Still, the operator-dependent model is slightly faster, as it has less memory usage (due to the compact network representation).

Operator pruning improves query times significantly, achieving speed-up factors that range from 2.2 on the operator-dependent London network to 8.0 on the operator-expanded Germany network. Naturally, the speed-up is greater on larger networks with more bike sharing operators. Approximately one fewer round is performed on average as a result of a reduced search space. Moreover, the operator-expanded model benefits more strongly from operator pruning than the operator-dependent model, being faster by a factor of 1.5 to 1.7. This is because vertices and trips, which are not part of the operator hull, are removed entirely from the network instead of being skipped at query time.

Table 7.4: Performance overview of all approaches for solving the multimodal journey planning problem with bike sharing that we described in this chapter. All algorithms are evaluated with and without the use of operator pruning (OP) on all networks, except for the Germany network where the preprocessing without OP is not feasible. We evaluate the MR-∞ algorithm for both, the operator-dependent (OD) and the operator-expanded (OE) model, while ULTRA can only be applied to the operator-expanded model. The query results are averages over 10 000 random queries. The *Vertices* column reports the average number of vertices settled by the algorithm. Similarly, the *Routes* column reports the average number of routes that were scanned by the algorithm.

Network	Algorithm	Preprocessing	Query			
		Time [h:m:s]	Rounds	Vertices	Routes	Time [ms]
Stuttgart	MR-∞-OD	2:10	9.50	789 756	182 439	310.2
	MR-∞-OD-OP	3:53	8.60	247 158	51 286	112.9
	MR-∞-OE	3:32	9.50	561 959	182 446	304.3
	MR-∞-OE-OP	5:01	8.49	150 961	49 914	58.0
	ULTRA-OE	3:33:50	9.69	79 691	187 668	89.3
	ULTRA-OE-OP	29:39	8.64	27 382	51 397	18.0
London	MR-∞-OD	0:12	9.59	342 361	25 037	112.2
	MR-∞-OD-OP	15:46	8.90	135 765	10 884	51.1
	MR-∞-OE	0:20	9.59	320 286	25 045	119.1
	MR-∞-OE-OP	15:53	8.64	117 188	9 152	34.2
	ULTRA-OE	14:15:25	9.70	78 486	25 922	52.8
	ULTRA-OE-OP	59:33	8.75	23 534	9 532	17.1
Switzerland	MR-∞-OD	0:56	9.55	840 396	171 361	286.8
	MR-∞-OD-OP	5:11	8.49	176 364	54 173	85.0
	MR-∞-OE	1:30	9.55	782 572	171 410	345.0
	MR-∞-OE-OP	5:40	8.35	144 522	43 980	52.8
	ULTRA-OE	10:02:22	9.70	107 627	180 064	117.2
	ULTRA-OE-OP	28:03	8.48	29 394	44 970	21.0
Germany	MR-∞-OD	13:19	11.99	17 421 659	2 888 893	9 830.1
	MR-∞-OD-OP	8:58:41	10.62	2 689 029	706 307	2 183.9
	MR-∞-OE	21:57	11.99	16 120 342	2 889 313	10 599.3
	MR-∞-OE-OP	9:05:48	10.24	2 091 814	679 898	1 322.7
	ULTRA-OE-OP	40:13:48	10.38	301 832	688 525	649.3

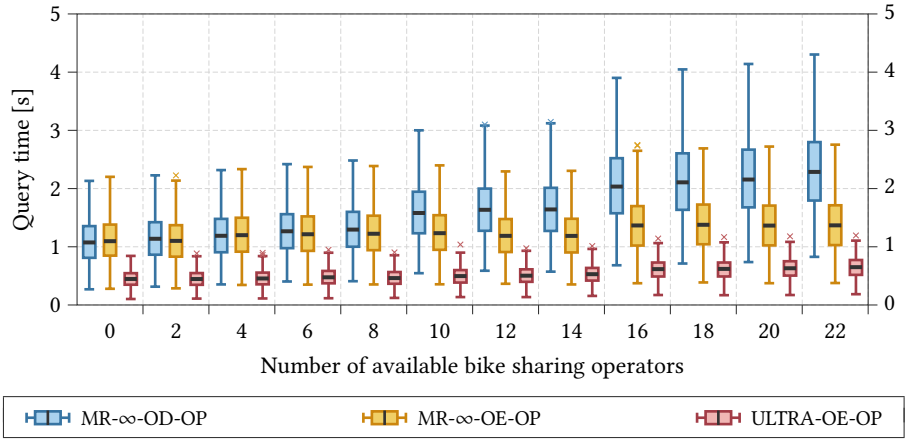


Figure 7.2: Running time of all query algorithms with operator pruning depending on the number of bike sharing operators available. We used the Germany network without bike sharing and gradually added the bike sharing operators in sets of two (in order to compensate for differences in the number of stations). We evaluated the same 1 000 random queries for each number of bike sharing operators.

We achieve the fastest query times by combining the operator-expanded model with ULTRA. For this, we use the RAPTOR-based ULTRA query algorithm, which facilitates the comparison with our baseline approach MR-∞ that is also based on RAPTOR. Independently of the network, we observe significant speed-ups when using ULTRA instead of MR-∞, ranging from a factor of 6.6 for the London network to 15.1 for the Germany network. Furthermore, we observe that ULTRA-OE-OP is the first algorithm that enables query times below a second for all networks.

Impact of Bike Sharing Operators. We evaluated how the number of bike sharing operators influences the query time of the algorithms. We do this for the Germany network, as it is the largest network and has the most bike sharing operators. We started without any bike sharing operators and created partial instances by successively adding operators. To compensate for differences in the number of bike sharing stations, we added operators in pairs of two, pairing large operators with smaller ones. Finally, we evaluated a single set of random queries for all instances. The results are shown in Figure 7.2. We observe that the number of operators is correlated linearly to the query time for all algorithms. The impact on the query time is the strongest for MR-∞-OD-OP, further confirming that the operator-expanded model benefits more from operator pruning than the operator-dependent model.

7.6 Final Remarks

In this chapter we presented two novel approaches for modeling multimodal transportation networks with various competing bike sharing operators: the operator-dependent and operator-expanded model. We showed that both models result in similar query performance, with the operator-dependent model being more memory-efficient and the operator-expanded model being compatible with existing query algorithms without modifications. Given its compatibility, we were able to combine the operator-expanded model with ULTRA, a known speedup technique for multimodal networks, in order to reduce query times. Additionally, we developed a fast preprocessing step called operator pruning, which can be used to accelerate queries in both models. Our experimental evaluation shows that combining operator pruning with ULTRA-RAPTOR enables queries that are more than an order of magnitude faster than the operator-dependent variant of MCR. Beyond that, we showed that using operator pruning also reduces the preprocessing time of ULTRA by more than an order of magnitude.

8 Assignments

In previous chapters we have developed multimodal journey planning algorithms with the goal of providing good journey recommendations for individual travelers. However, this is not the only application for efficient journey planning algorithms. Another important class of problems that require fast journey planning are traffic assignment problems, which arise during the planning of public transit networks. In this chapter we demonstrate how state-of-the-art journey planning algorithms for public transit networks as well as our novel ULTRA approach can be used to efficiently compute traffic assignments. This chapter is based on joint work with Lars Briem, Sebastian Buck, Holger Ebhart, Nicolai Mallig, Jonas Sauer, Ben Strasser, Peter Vortisch, and Dorothea Wagner [Bri+17, SWZ19a, SWZ19b].

Problem Setting. Traffic assignments are an important tool for planning and analyzing transportation networks. Efficient assignment algorithms allow to predict how new infrastructure could improve traffic flows, or to test the limits of existing networks, based on historic, empiric, or expected passenger demand data. For this, the demand is given as a list of origin-destination pairs, where each pair is associated with a desired departure time. A basic variant of the assignment problem then asks for the expected utilization of each vehicle (i.e., the number of passengers using the vehicle) in the public transit network at each point in time. A more intricate second variant additionally asks for a mapping from the origin-destination pairs onto actual journeys through the network that constitute the overall utilization of the vehicles. Finding a high quality solution for either of these problems in an efficient manner generally requires two steps. First, a fast journey planning algorithm is required, in order to

compute possible journeys for every origin-destination pair. Secondly, a sophisticated decision model has to be applied to the journeys found by the first step, in order to adequately reflect which journeys would be chosen by passengers in the real world.

An example of an application that utilizes traffic assignments is the planning of public transportation networks. When implementing new public transit routes (or redirecting existing ones), it is often desired that the capacity of all the vehicles serving the route is well utilized. If vehicles are overcrowded, then more or larger vehicles have to be deployed. However, if vehicles are only sparsely used, they may be dropped from the schedule, thereby reducing the cost of operating the public transit network. Using traffic assignments, the utilization of the vehicles can be estimated ahead of time, allowing for the design of an efficient public transportation service.

Chapter Overview. We begin with a formal introduction of the traffic assignment problem in Section 8.1. For this purpose we first introduce some additional notation, which we will use throughout this chapter. Afterwards, we define the decision models and the assignment problem that we address in this chapter.

Since results on efficient journey planning have not been applied to assignment problems before, we initially only focus on computing assignments for public transit networks with transitive transfer graphs in Section 8.2. Within this section we show in detail how the well-known CSA algorithm can be adapted for solving assignment problems. As a result, we obtain a very efficient assignment algorithm that is significantly faster than the commercial algorithm we use as a baseline.

We continue with describing how our new assignment algorithm can be combined with ULTRA in Section 8.3. To this end, we present some additional steps that are required since ULTRA is inherently a one-to-one algorithm, while the assignment problem is a many-to-many problem. Furthermore, we introduce some minor improvements that can be used independently of the network type.

Finally, we evaluate the performance of both assignment algorithms, which we developed in this chapter, in Section 8.4. In particular, we demonstrate the validity of our approach and show that our algorithm can be combined with several different decision models. Furthermore, we show that both assignment algorithms developed in this chapter are very efficient and outperform the baseline. This is despite the fact that we can solve a multimodal assignment problem, which was previously not possible.

8.1 Preliminaries

In this section we introduce the notation and the concepts used in the context of assignment problems. Additionally, we present a short introduction of discrete choice models. Finally, we define the assignment problem that we solve in this chapter.

8.1.1 Perceived Arrival Time

Within this chapter we use the connection-based representation of public transit networks, i.e., $N = (\mathcal{C}, \mathcal{S}, \mathcal{T})$, since our assignment algorithms will be based on CSA. Furthermore, we use the terms *origin* and *destination* when we refer to the start point or end point of a journey, as these terms are most commonly used in the traffic assignment literature. This is in contrast to the terms source and target, which we used in previous chapters and which are commonly used in the journey planning literature.

The core problem of computing a public transit traffic assignment is to decide for each passenger which journey (or, more specifically, which connections) he or she uses in order to reach his or her destination. The quality of the resulting assignment highly depends on these choices. Thus it is important to model the behavior and preferences of the passengers in a realistic way. This means that we cannot assign connections to the passengers solely based on the travel time of the resulting journey. For example, a passenger might prefer a journey with a slightly longer travel time, if this reduces the number of changes between vehicles.

As a means of reflecting the passengers' preferences, we introduce the notion of perceived arrival time (PAT). Given a connection $c \in \mathcal{C}$ and a destination $d \in \mathcal{V}$, the perceived arrival time $\tau^{\mathrm{p}}(c, d)$ is a measurement for how useful c is in order to reach d. The PAT $\tau^{\mathrm{p}}(c, d)$ depends on the possible journeys that end at d and contain c. We consider five properties of these journeys that influence the perceived arrival time: the actual arrival time at d, the number of transfers, the time spent walking, the time spent waiting, and the delay robustness. In order to control how these properties affect the PAT we introduce some parameters. We account for walking and waiting time by weighting the corresponding times with factors $\lambda_{\mathrm{walk}}, \lambda_{\mathrm{wait}} \in \mathbb{R}_0^+$. For every transfer during the journey we add an additional cost of $\lambda_{\mathrm{trans}} \in \mathbb{R}_0^+$. Finally, we incorporate delay robustness by computing the expected arrival time under the assumption that each connection has a random delay of at most $\lambda_{\mathrm{delay}} \in \mathbb{R}_0^+$.

Delay Model. We adapt the concept of minimum expected arrival time (MEAT), which was introduced in [DSW14], in order to model delay robustness. For this, we define the *waiting time* τ_{wait}, which is the amount of time a passenger has to spend waiting if he arrives at a stop before the departure of the connection he wants to use. If a passenger starts at vertex v at time τ and wants to use a connection c, then the amount of time he has to wait is $\tau_{\mathrm{wait}}(v, \tau, c) \coloneqq \tau_{\mathrm{dep}}(c) - \tau - \tau_{\mathrm{tra}}(v, v_{\mathrm{dep}}(c))$, where $\tau_{\mathrm{tra}}(v, v_{\mathrm{dep}}(c))$ is the time required to transfer from the vertex v to the departure stop of the connection c. Of course, the passenger can only use the connection c if the waiting time is not negative. The notion of waiting time is extended to pairs of connections $c, c' \in \mathcal{C}$ that are part of different trips by defining $\tau_{\mathrm{wait}}(c, c') \coloneqq \tau_{\mathrm{wait}}(v_{\mathrm{arr}}(c), \tau_{\mathrm{arr}}(c), c')$. Thus, a transfer between connections c and c' is valid if and only if $\tau_{\mathrm{wait}}(c, c') \geq 0$ holds.

Following the approach used for the definition of MEAT in [DSW14], we introduce a random variable $D_c \in \mathbb{R}_0^+$ for every connection $c \in \mathcal{C}$, which represents the delay of the connection. This means that the arrival stop $v_{\text{arr}}(c)$ will be reached at $\tau_{\text{arr}}(c) + D_c$. Thus, transferring to another connection can become invalid, if the delay exceeds the waiting time of the transfer. We define the probability $\mathrm{P}[D_c \leq \tau]$ that the delay is at most τ as follows: $\mathrm{P}[D_c \leq \tau] \coloneqq 0$ for $\tau \leq 0$, $\mathrm{P}[D_c \leq \tau] \coloneqq 1$ for $\tau \geq \lambda_{\text{delay}}$, and $\mathrm{P}[D_c \leq \tau] \coloneqq 31/30 - (11\lambda_{\text{delay}})/(300\tau + 30\lambda_{\text{delay}})$ for $0 < \tau < 1$, where λ_{delay} is the maximal delay that can occur. Based on this, the probability that a transfer between two connections $c, c' \in \mathcal{C}$ is valid is given by $\mathrm{P}[D_c \leq \tau_{\text{wait}}(c, c')]$. Additionally, we define the probability $\mathrm{P}[\tau' < D_c \leq \tau] \coloneqq \mathrm{P}[D_c \leq \tau] - \mathrm{P}[D_c \leq \tau']$ that the delay of c is between τ' and τ. For more details on the delay model see [DSW14].

Formal PAT Definition. We now proceed with defining the perceived arrival time $\tau^{\mathrm{P}}(c, d)$ in a recursive way, which allows us to take all journeys beginning with c into account. There exist three distinct cases for continuing a journey after using the connection c. If it is possible to use a transfer from the arrival stop of c to the destination, then the journey can be completed by walking. Otherwise, the journey continues either with the next connection of the same trip as $T(c)$ or the vehicle serving c is left at $v_{\text{arr}}(c)$. Therefore, we define

$$\tau^{\mathrm{P}}_{\text{arr}}(c, d) \coloneqq \min\left\{\tau^{\mathrm{P}}_{\text{arr}}(c, d \mid \mathsf{walk}),\ \tau^{\mathrm{P}}_{\text{arr}}(c, d \mid \mathsf{trip}),\ \tau^{\mathrm{P}}_{\text{arr}}(c, d \mid \mathsf{trans})\right\},$$

where $\tau^{\mathrm{P}}_{\text{arr}}(c, d \mid \mathsf{walk})$ is the PAT under the constraint that the journey is completed by walking from c to d, $\tau^{\mathrm{P}}_{\text{arr}}(c, d \mid \mathsf{trip})$ is the PAT under the constraint that the journey continues with the same trip as $T(c)$, and $\tau^{\mathrm{P}}_{\text{arr}}(c, d \mid \mathsf{trans})$ is the PAT under the constraint that the journey continues with a transfer to another connection after c. The perceived arrival time for walking to the destination is defined as:

$$\tau^{\mathrm{P}}_{\text{arr}}(c, d \mid \mathsf{walk}) \coloneqq \begin{cases} \tau_{\text{arr}}(c) & \text{if } v_{\text{arr}}(c) = d \\ \tau_{\text{arr}}(c) + \lambda_{\mathsf{walk}} \cdot \tau_{\text{tra}}\big(v_{\text{arr}}(c), d\big) & \text{otherwise.} \end{cases}$$

This means that the PAT is the actual arrival time, if the destination is reached directly by using c. If this is not the case, the time needed for walking to the destination is multiplied with the cost factor λ_{walk} and added to the arrival time. For the definition of $\tau^{\mathrm{P}}_{\text{arr}}(c, d \mid \mathsf{trip})$ let $\mathcal{C}_{\text{trip}}(c) \coloneqq \{c' \in \mathcal{C} \mid T(c') = T(c) \wedge \tau_{\text{dep}}(c') \geq \tau_{\text{arr}}(c)\}$ be the set of all connections following after c in the trip of c. We then define the PAT for continuing with the same trip as the minimum over the perceived arrival times of all subsequent connections in the trip:

$$\tau^{\mathrm{P}}_{\text{arr}}(c, d \mid \mathsf{trip}) \coloneqq \begin{cases} \min\{\tau^{\mathrm{P}}(c', d) \mid c' \in \mathcal{C}_{\text{trip}}(c)\} & \text{if } \mathcal{C}_{\text{trip}}(c) \neq \emptyset \\ \infty & \text{otherwise.} \end{cases}$$

Finally, we proceed with defining the PAT $\tau^{\mathsf{p}}_{\mathsf{arr}}(c, d \mid \mathsf{trans})$ for transferring from c to a connection c' of another trip. For this purpose, we first introduce the perceived time $\tau^{\mathsf{p}}_{\mathsf{tra}}(v, w)$ for transferring from v to w as a weighted sum of the walking or waiting time and the transfer costs:

$$\tau^{\mathsf{p}}_{\mathsf{tra}}(v, w) \coloneqq \begin{cases} \lambda_{\mathsf{trans}} + \lambda_{\mathsf{wait}} \cdot \tau_{\mathsf{ch}}(v) & \text{if } v = w \\ \lambda_{\mathsf{trans}} + \lambda_{\mathsf{walk}} \cdot \tau_{\mathsf{tra}}(v, w) & \text{otherwise.} \end{cases}$$

Additionally, we define $\tau^{\mathsf{p}}_{\mathsf{tra}}(c, c') \coloneqq \tau^{\mathsf{p}}_{\mathsf{tra}}(v_{\mathsf{arr}}(c), v_{\mathsf{dep}}(c'))$, in order to reflect the perceived time for transferring from a connection c to another connection c'. Transferring between connections $c, c' \in \mathcal{C}$ may include some additional waiting time $\tau_{\mathsf{wait}}(c, c')$ at the departure stop of c', after the actual transfer took place. We account for this by introducing the perceived waiting times $\tau^{\mathsf{p}}_{\mathsf{wait}}(v, \tau, c) \coloneqq \lambda_{\mathsf{wait}} \cdot \tau_{\mathsf{wait}}(v, \tau, c)$, and $\tau^{\mathsf{p}}_{\mathsf{wait}}(c, c') \coloneqq \lambda_{\mathsf{wait}} \cdot \tau_{\mathsf{wait}}(c, c')$. Using this perceived waiting time we define the perceived arrival time $\tau^{\mathsf{p}}_{\mathsf{arr}}(c, c', d) \coloneqq \tau^{\mathsf{p}}_{\mathsf{tra}}(c, c') + \tau^{\mathsf{p}}_{\mathsf{wait}}(c, c') + \tau^{\mathsf{p}}_{\mathsf{arr}}(c', d)$ of journeys starting with the connection c, followed by a transfer to the connection c', and ending at the destination d. In order to define $\tau^{\mathsf{p}}_{\mathsf{arr}}(c, d \mid \mathsf{trans})$, we only need to specify which connection c' is used after c. Here, we take not only the perceived arrival time $\tau^{\mathsf{p}}(c', d)$ into account, but also the possibility that the transfer from c to c' might become invalid due to a delay of c. We achieve this by considering all connections that are Pareto-optimal with respect to their PAT and their delay robustness as possible candidates. To this end, let $\mathcal{C}_{\mathsf{trans}}(c) \coloneqq \{c' \in \mathcal{C} \mid \tau_{\mathsf{wait}}(c, c') \geq 0\}$ be the set of all connections c' where it is possible to transfer from c to c'. Based on this set, the set $\mathcal{C}^{\mathsf{opt}}_{\mathsf{trans}}(c)$ of all Pareto-optimal connections reachable from c can be defined as:

$$\mathcal{C}^{\mathsf{opt}}_{\mathsf{trans}}(c) \coloneqq \left\{\hat{c} \in \mathcal{C}_{\mathsf{trans}}(c) \,\middle|\, \forall \bar{c} \in \mathcal{C}_{\mathsf{trans}}(c) \colon \tau_{\mathsf{wait}}(c, \bar{c}) \geq \tau_{\mathsf{wait}}(c, \hat{c}) \Rightarrow \tau^{\mathsf{p}}_{\mathsf{arr}}(c, \bar{c}, d) \geq \tau^{\mathsf{p}}_{\mathsf{arr}}(c, \hat{c}, d)\right\}.$$

Let $\langle c_1, \ldots, c_k \rangle$ be the sequence of connections from $\mathcal{C}^{\mathsf{opt}}_{\mathsf{trans}}(c)$ sorted by their waiting time in increasing order, that is $\tau_{\mathsf{wait}}(c, c_i) \geq \tau_{\mathsf{wait}}(c, c_{i-1})$ for $i \in [2, k]$. This means transferring from c to c_1 results in the minimum PAT. If, however, transferring to c_1 is not possible due to delay, c_2 is the next best option, and so on. Based on this, we finally define $\tau^{\mathsf{p}}_{\mathsf{tra}}(c, d)$ as the sum of the perceived arrival times of all c_i, weighted by the probability that the transfer to c_i is valid, while the transfer to c_{i-1} is invalid:

$$\tau^{\mathsf{p}}_{\mathsf{arr}}(c, d \mid \mathsf{trans}) \coloneqq \begin{cases} \displaystyle\sum_{i=1}^{k} \left(\frac{\mathrm{P}[\tau_{\mathsf{wait}}(c, c_{i-1}) < D_c \leq \tau_{\mathsf{wait}}(c, c_i)]}{\mathrm{P}[D_c \leq \tau_{\mathsf{wait}}(c, c_k)]} \cdot \tau^{\mathsf{p}}_{\mathsf{arr}}(c, c_i, d) \right) & \text{if } k > 0 \\ \infty & \text{otherwise.} \end{cases}$$

Note that our recursive definition of $\tau^{\mathsf{p}}(c, d)$ is well-defined, since it only depends on the perceived arrival times of connections c' with $\tau_{\mathsf{dep}}(c') > \tau_{\mathsf{dep}}(c)$.

8.1.2 Decision Models

In general, a passenger who wants to travel to some destination d has more than one option for doing so. An important aspect of the assignment process is to determine for each available option the likelihood of being chosen by the passenger. We do this with the help of a decision model, which estimates the likelihood of each travel option based on the *utility* that the option provides for the traveler. We quantify the utility of a travel option based on the PAT of the option, using the following definition.

Since we want to use PATs to define the utility of a travel option and since the PAT is defined on the level of connections, we also use connections to represent travel options. Assume that there are k possible connections $c_1, \dots, c_k$ that can be used as the first connection in a journey to the destination d. We then define the utility $u(c_i)$ of the i-th travel option, i.e., the utility of using c_i in order to reach d, as

$$u(c_i) := \max\left(0\,,\ \min_{1 \le j \le k} \left(\tau^{\mathsf{p}}_{\mathsf{arr}}(c_j, d)\right) - \tau^{\mathsf{p}}_{\mathsf{arr}}(c_i, d) + \lambda_{\Delta\mathsf{max}}\right),$$

where $\lambda_{\Delta\mathsf{max}} \in \mathbb{R}^+_0$ is a parameter that determines how much worse an option can be compared to the best option before its utility is zero. In other words, the utility of a connection is zero if the PAT of the connection differs by more than $\lambda_{\Delta\mathsf{max}}$ from the best possible PAT. For all other options c_i, the utility correlates linearly to the difference between $\tau^{\mathsf{p}}_{\mathsf{arr}}(c_i, d)$ and the PAT of the best connection (or second best connection in the case that c_i itself is the best connection). The set $\mathcal{CS} := \{c_1, \dots, c_k\}$ of all available options is commonly called the *choice set* in this context.

Given a choice set $\mathcal{CS}$ and a utility function $u\colon \mathcal{CS} \to \mathbb{R}$, a decision model is simply a probability distribution $\mathrm{P}[c \mid \mathcal{CS}] \in [0, 1]$ that establishes the likelihood of a passenger choosing the option c given the choice set $\mathcal{CS}$. This of course implies that $\sum_{c \in \mathcal{CS}} \mathrm{P}[c \mid \mathcal{CS}] = 1$ has to hold , since one of the available options has to be chosen. In the case that $\mathcal{CS}$ is a finite set, which is always the case for a finite transportation network, a discrete choice model can be used as decision model. In the following, we briefly introduce the discrete choice models that we consider in this chapter.

Logit. The Logit model is one of the most widely used discrete choice models for solving traffic assignment problems. It is a Random Utility Model (RUM), which means that it models the utility of a travel option as a random variable U and it assumes that passengers always use the option with the maximum utility. Given the deterministic utility $u(c)$ of a travel option c, the corresponding random utility is defined as $U(c) := u(c) + E(c)$, where $E(c)$ is an independent random variable that captures the error of the deterministic utility (e.g., properties of the travel option that are not covered by the network model). Using this approach, the likelihood of choosing a travel option c is defined as $\mathrm{P}_{\mathsf{logit}}[c \mid \mathcal{CS}] := \mathrm{P}[U(c) > U(\bar{c}) \ \ \forall \bar{c} \in \mathcal{CS} \setminus \{c\}]$.

The Logit model is based on the assumption that the random error $E(c)$ of the utility obeys a Type-I Extreme Value distribution [DM75]. Taking into account the parameter β of this distribution, the probability of choosing the option c from a choice set $\mathcal{CS}$ in the Logit model can be expressed as

$$\mathrm{P}_{\text{logit}}[c \mid \mathcal{CS}] = \frac{e^{\beta \cdot u(c)}}{\sum_{\bar{c} \in \mathcal{CS}} \left(e^{\beta \cdot u(\bar{c})}\right)}.$$

Kirchhoff. In contrast to the Logit model, the Kirchhoff model is not rooted in discrete choice theory, but borrows ideas from Kirchhoff's circuit laws [GN16]. The probability function $\mathrm{P}_{\text{kirchhoff}}$ again depends on a tuning parameter β and is defined as

$$\mathrm{P}_{\text{kirchhoff}}[c \mid \mathcal{CS}] := \frac{u(c)^{\beta}}{\sum_{\bar{c} \in \mathcal{CS}} \left(u(\bar{c})^{\beta}\right)}.$$

Linear. Finally, we propose a new and simple discrete choice model, which is tailored to our PAT-based utility definition. For this purpose, let $\hat{c}$ be the travel option with maximal utility and δ the difference between the utilities of the two best options in $\mathcal{CS}$. Based on this we define the probability function $\mathrm{P}_{\text{linear}}$ as

$$\mathrm{P}_{\text{linear}}[c \mid \mathcal{CS}] := \frac{\max\left(u(c), 2u(c) - u(\hat{c}) + \delta\right)}{\delta + \sum_{\bar{c} \in \mathcal{CS}} u(\bar{c})}.$$

We call our new discrete choice model *linear* since it yields probabilities that depends linearly on the PATs of the travel options. As an example for this, consider the choice set $\mathcal{CS} = \{c_1, c_2\}$ with $|\tau^{\mathrm{p}}_{\text{arr}}(c_1, d) - \tau^{\mathrm{p}}_{\text{arr}}(c_2, d)| \leq \lambda_{\Delta\max}$. Here the probability of the option c_1 is $\mathrm{P}_{\text{linear}}[c_1 \mid \mathcal{CS}] = (\tau^{\mathrm{p}}_{\text{arr}}(c_2, d) - \tau^{\mathrm{p}}_{\text{arr}}(c_1, d) + \lambda_{\Delta\max})/(2\lambda_{\Delta\max})$ and the probability of the option c_2 is $\mathrm{P}_{\text{linear}}[c_2 \mid \mathcal{CS}] = (\tau^{\mathrm{p}}_{\text{arr}}(c_1, d) - \tau^{\mathrm{p}}_{\text{arr}}(c_2, d) + \lambda_{\Delta\max})/(2\lambda_{\Delta\max})$.

8.1.3 Problem Statement

The public transit traffic assignment problem takes as input a public transit network N, a transfer graph G, and some *demand* $\mathcal{D}$, which is a set of origin-destination pairs. Each origin-destination pair $p = (o, d) \in \mathcal{D}$ is also associated with a desired departure time $\tau_{\text{dep}}(p)$. The pair $p = (o, d)$ represents a passenger who wants to travel from the *origin vertex* $o \in \mathcal{V}$ to the *destination vertex* $d \in \mathcal{V}$, starting at $\tau_{\text{dep}}(p)$. The objective of the traffic assignment problem is to assign each origin-destination pair $p = (o, d) \in \mathcal{D}$ to a probability space consisting of journeys that satisfies the demand. The demand

is satisfied if each journey in the probability space departs at o no earlier than $\tau_{\text{dep}}(p)$ and ends at d. The probabilities associated with each journey in the probability space should reflect the likelihood of a real passenger using the journey, as specified by a decision model. Summing the probabilities of all journeys containing a connection c yields the *utilization* $\mu(c)$, which is the expected number of passengers using c.

8.2 Public Transit Assignment

We continue with the description of our assignment algorithms. At first, we only consider the assignment problem for pure public transit networks, i.e., networks with a transitively closed transfer graph, since recent results on efficient public transit journey planning have not yet been applied to this problem. In order to solve this problem we propose the CSA-Based Assignment (CBA) algorithm, which enables a fast assignment algorithm through the use of the state-of-the-art public transit journey planning algorithm CSA.

Algorithm Outline. Our assignment algorithm is based on a microscopic Monte Carlo simulation of individual passengers represented by a unique integer identifier. For each vertex of the network we maintain a list containing all the passengers, who currently reside at the vertex. We then use a sequential route choice model, as proposed in [GP06], to move the passengers gradually from one vertex to the next, until they reach their destination. Within this process we can use an arbitrary discrete choice model in combination with our PAT-based utility in order to determine the vertices visited next by the passengers. In particular, we use the decision model to assign a probability to every possible travel option the passengers could in theory use. Next, we choose one of these options at random for every passenger, following the probability distribution provided by the choice model. In order to increase the accuracy of the simulation, we generate λ_{mul} times as many passengers as specified by the demand. After the simulation finished, the results are divided by λ_{mul}, in order to obtain a stochastic distribution of the passengers specified by the demand.

We observe that passengers with the same destination d and roughly the same time of travel will eventually encounter each other on their journeys (at least at d). If they meet before d at a vertex v, then they have the same options for continuing their journey from v to d. Our algorithm exploits this by evaluating the options and computing the decisions for all passengers at v at once. To this end, we partition the passengers based on their destination. We proceed with showing how the traffic assignment for passengers with a common destination can be computed. A complete traffic assignment can be obtained by doing this for every destination and aggregating the results. For the remainder of this chapter we assume d to be a fixed destination vertex.

We compute the traffic assignment for passengers with destination d in three phases. First, we compute two PATs for every connection, one for taking the connection and one for avoiding it. Next, we simulate the movement of the passengers through the network. Every time a passenger could use a connection c without producing an invalid transfer, we decide based on the previously computed PATs whether the passenger takes the connection c. Connections that are used by a passenger are then added to the passenger's journey. Finally, we simplify the journeys by removing unwanted cycles. Within this procedure we use CSA for the first and the second phase.

8.2.1 Perceived Arrival Time Computation

In the first phase we compute all information required to build the journeys for the passengers one connection at a time. We identified three situations that can occur during the simulation of a passenger's movement and require a decision about the journey's continuation (see Figure 8.1).

Waiting at a Stop. The first of these situations arises when a passenger is waiting at a stop v, while a connection c departs from the stop v. In this case, it has to be decided if the passenger boards the vehicle serving c or keeps waiting at the stop. In order to make this decision, we need the PATs for both alternatives. The PAT for using the connection c (i.e., boarding the vehicle) is given by $\tau^{\mathsf{p}}_{\mathsf{arr}}(c, d)$. On the other hand, skipping the connection c and waiting at the stop implies that some later connection departing from the stop has to be taken. Transferring to another stop is not an option, as we assume that the passenger transferred to his current stop v with the intention of boarding some vehicle at v. The set of all alternative connections departing from the same stop is given by $\mathcal{C}_{\mathsf{alt}}(c) := \{c' \in \mathcal{C} \mid v_{\mathsf{dep}}(c') = v_{\mathsf{dep}}(c),\ \tau_{\mathsf{dep}}(c') > \tau_{\mathsf{dep}}(c)\}$. We use this set of alternative connections to obtain the PAT $\tau^{\mathsf{p}}_{\mathsf{arr}}(c, d \mid \mathsf{skip}\ c)$ for skipping the connection c, which is the sum of the additional perceived waiting time and the perceived arrival time of the best alternative connection. Formally, we define it as: $\tau^{\mathsf{p}}_{\mathsf{arr}}(c, d \mid \mathsf{skip}\ c) := \min\{\tau^{\mathsf{p}}_{\mathsf{wait}}(v_{\mathsf{dep}}(c), \tau_{\mathsf{dep}}(c), c') + \tau^{\mathsf{p}}_{\mathsf{arr}}(c', d) \mid c' \in \mathcal{C}_{\mathsf{alt}}(c)\}$.

Sitting in a Connection. The second situation affects passengers using a connection that is not the last connection of its trip. These passengers again have to make a binary decision. Either they leave the vehicle at the arrival stop of the current connection or they use another connection of the trip. As before, making this decision requires the perceived arrival times of both alternatives. The PAT for continuing with the same trip is given by $\tau^{\mathsf{p}}_{\mathsf{arr}}(c, d \mid \mathsf{trip})$. When disembarking the vehicle, a passenger can continue his journey by either walking to his destination or transferring to another vehicle. Therefore, the PAT for disembarking is given by $\tau^{\mathsf{p}}_{\mathsf{arr}}(c, d \mid \mathsf{disembark}) := \min\{\tau^{\mathsf{p}}_{\mathsf{arr}}(c, d \mid \mathsf{walk}),\ \tau^{\mathsf{p}}_{\mathsf{arr}}(c, d \mid \mathsf{trans})\}$.

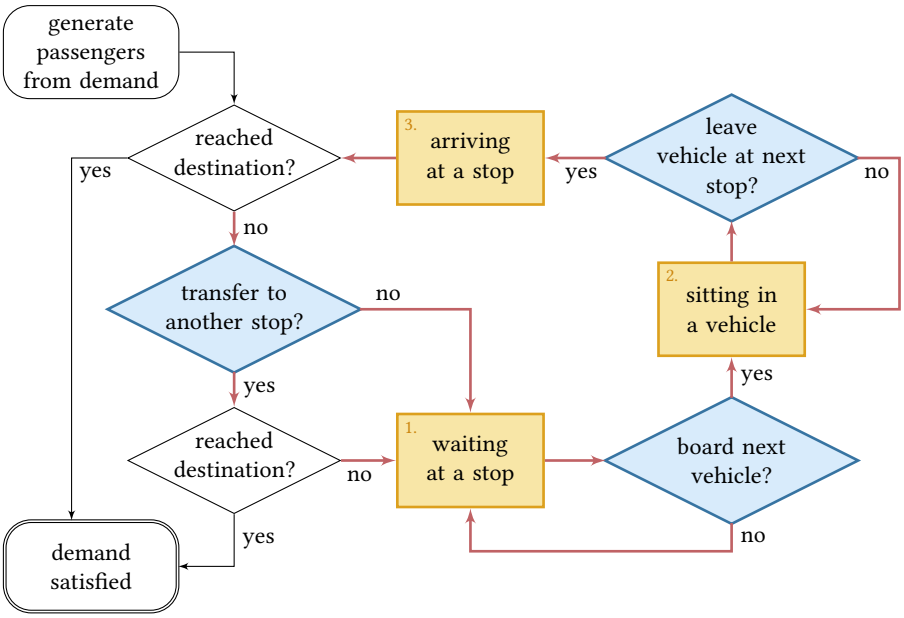


Figure 8.1: A flowchart describing the movement of a passenger through the network. Passengers are generated according to the demand. If their destination differs from their origin, then they enter the main cycle (colored part, with red arrows) of the simulation. Passengers traverse the main cycle until they reach their destination. During this they can be in one of three situations (yellow). The situation a passenger is in changes depending on his decisions (blue).

Arriving at a Stop. The last situation where a decision has to be made occurs when a passenger leaves a vehicle, but has not yet reached his destination. In this case, it has to be decided to which stop the passenger transfers, in order to wait for another connection. This decision requires a perceived arrival time for every stop v that can be reached by a transfer. Similar to the definition of $\tau^{\mathsf{p}}_{\mathsf{arr}}(c, d \mid \mathsf{skip}\ c)$, the PAT for such a stop v is given by the PAT of the best connection c departing from v plus the additional perceived waiting time between the arrival time τ at v and the departure of c. As this value is required for every possible arrival time τ at v, we simply compute a profile function $f^{v,d}_{\mathsf{wait}}$ for every vertex, which we define as:

$$f^{v,d}_{\mathsf{wait}}(\tau) := \min\{\tau^{\mathsf{p}}_{\mathsf{wait}}(v, \tau, c) + \tau^{\mathsf{p}}_{\mathsf{arr}}(c, d) \mid c \in \mathcal{C},\ \tau_{\mathsf{dep}}(c) \geq \tau,\ v_{\mathsf{dep}}(c) = v\}.$$

Actual PAT Computation. In summary, we require three values per connection for decision making: $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d \mid \mathsf{trip})$, $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d \mid \mathsf{skip}\ c)$, and $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d \mid \mathsf{disembark})$. Additionally we need a profile function $f^{v,d}_{\mathrm{wait}}$ for every vertex in the network. We now show how these values can be computed in a single sweep over the connection array. As basis for our algorithm, we use the MEAT algorithm [DSW14], which is a variant of CSA and allows for efficient all-to-one profile queries. Instead of computing minimum *expected* arrival time profiles, as the original MEAT algorithm does, we compute minimum *perceived* arrival time profiles. In addition to the profile $f^{v,d}_{\mathrm{wait}}(\tau)$, we compute a second profile $f^{v,d}_{\mathrm{trans}}(\tau)$, which we use in order to determine the three PAT values needed per connection. The difference between the two profiles is that $f^{v,d}_{\mathrm{trans}}(\tau)$ requires an initial transfer to another stop. Formally, we define:

$$f^{v,d}_{\mathrm{trans}}(\tau) := \min\left\{\tau^{\mathrm{p}}_{\mathrm{tra}}(v, v_{\mathrm{dep}}(c)) + \tau^{\mathrm{p}}_{\mathrm{wait}}(v, \tau, c) + \tau^{\mathrm{p}}_{\mathrm{arr}}(c, d) \;\middle|\; c \in \mathcal{C},\ \tau_{\mathrm{wait}}(v, \tau, c) \geq 0\right\}.$$

Our algorithm maintains for every vertex the two initially incomplete profiles $f^{v,d}_{\mathrm{wait}}$ and $f^{v,d}_{\mathrm{trans}}$. Additionally we store for every trip T a value $\tau^{\mathrm{p}}_{\mathrm{arr}}(T)$ that keeps track of the current value of $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d)$ for the connection c with $T(c) = T$ that was most recently processed by the algorithm. Initially, we set $\tau^{\mathrm{p}}_{\mathrm{arr}}(T) \leftarrow \infty$ for all trips T in the network. We scan the connection array in decreasing order of their departure time. For every connection c we can then directly determine the three required values as follows. Since we store the arrival time for continuing with the same trip separately, we can set $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d) \leftarrow \tau^{\mathrm{p}}_{\mathrm{arr}}(T(c))$. The PAT for ignoring the connection c is given by the profile that describes waiting at the departure stop of c. Thus, we set $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d) \leftarrow f^{v_{\mathrm{dep}}(c),d}_{\mathrm{wait}}(\tau_{\mathrm{dep}}(c))$. Similarly, the PAT for disembarking at the arrival stop of c is given by the profile that requires an initial transfer. Accordingly, we set $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d \mid \mathsf{disembark}) \leftarrow f^{v_{\mathrm{arr}}(c),d}_{\mathrm{trans}}(\tau_{\mathrm{arr}}(c))$. For the special case that a transfer edge from the arrival stop of c to the destination exists, we have to consider the possibility of walking to the destination. Therefore, we set $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d \mid \mathsf{disembark}) \leftarrow \tau_{\mathrm{arr}}(c) + \tau_{\mathrm{tra}}(v_{\mathrm{arr}}(c), d)$, if this is smaller than the previous value of $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d \mid \mathsf{disembark})$. Afterwards, we temporarily compute the PAT of the connection c: $\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d) \leftarrow \min(\tau^{\mathrm{p}}_{\mathrm{arr}}(c, d \mid \mathsf{trip}), \tau^{\mathrm{p}}_{\mathrm{arr}}(c, d \mid \mathsf{disembark}))$. We use this value in order to update the profiles and the value $\tau^{\mathrm{p}}_{\mathrm{arr}}(T(c))$. First, we set $\tau^{\mathrm{p}}_{\mathrm{arr}}(T(c)) \leftarrow \tau^{\mathrm{p}}_{\mathrm{arr}}(c, d)$. Next, we add the point $(\tau_{\mathrm{dep}}(c), \tau^{\mathrm{p}}_{\mathrm{arr}}(c, d))$ as a breakpoint to the profile $f^{v_{\mathrm{dep}}(c),d}_{\mathrm{wait}}$, unless this profile already contains a breakpoint with a smaller PAT. Finally, we iterate over all vertices v with $(v, v_{\mathrm{dep}}(c)) \in \mathcal{E}$. For each such vertex v we add the point $(\tau_{\mathrm{dep}}(c) - \tau_{\mathrm{tra}}(v, v_{\mathrm{dep}}(c)), \tau^{\mathrm{p}}_{\mathrm{arr}}(c, d) + \lambda_{\mathrm{walk}} \cdot \tau_{\mathrm{tra}}(v, v_{\mathrm{dep}}(c)))$ as a breakpoint to the profile $f^{v,d}_{\mathrm{trans}}$, unless the profile already contains a breakpoint with a smaller PAT. We repeat this process for every connection. Afterwards, we have computed all values required for decision making and can continue with the actual assignment.

8.2.2 Passenger Assignment

The second phase of our algorithm uses the previously computed perceived arrival times to compute the journeys for all the passengers with destination d. For this purpose, we maintain for every passenger a list of connections used by the passenger. Additionally, we maintain a list of passengers for every vertex and trip, representing the passengers currently waiting at the vertex and sitting in the vehicle serving the trip, respectively. Furthermore, we use a queue sorted by arrival time for every vertex, containing the passengers that are currently transferring to the stop. The transfer queue of each vertex v is initialized with passengers created from the demand with origin v, using their desired departure time as keys.

Passenger Movement Simulation. We now describe how the passengers' movement through the network is simulated, using a single scan over the connection array in ascending order of departure time, similar to CSA. During this scan, we decide for each connection, which passengers use the connection. When scanning a connection c, we first determine the set of passengers that could enter the vehicle. We do this by removing all the passengers from the transfer queue of $v_{\text{dep}}(c)$ that arrive at $v_{\text{dep}}(c)$ before $\tau_{\text{dep}}(c)$. These passengers are then added to the list of passengers waiting at $v_{\text{dep}}(c)$. Afterwards, the list of passengers waiting at $v_{\text{dep}}(c)$ comprises exactly the passengers that could enter c. These passengers then have exactly two travel options: Option a is to use the connection c and option b is to not use the connection. The PATs of these two options are given by $\tau^{\text{p}}_{\text{arr}}(a) = \min(\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{trip}), \tau^{\text{p}}_{\text{arr}}(c, d \mid \text{disembark}))$ and $\tau^{\text{p}}_{\text{arr}}(b) = \tau^{\text{p}}_{\text{arr}}(c, d \mid \text{skip } c)$, respectively.

Using our linear decision model, the probabilities of the two options are given by $\text{P}[a \mid \{a, b\}] = (\tau^{\text{p}}_{\text{arr}}(b) - \tau^{\text{p}}_{\text{arr}}(1) + \lambda_{\Delta\max})/2\lambda_{\Delta\max}$ and $\text{P}[b \mid \{a, b\}] = 1 - \text{P}[a \mid \{a, b\}]$. Based on these probabilities, we make a random decision for every passenger waiting at the departure stop of the connection c. If a passenger happens to enter the connection, then he is first removed from the list of passengers waiting at $v_{\text{dep}}(c)$ and secondly added to the list of passengers sitting in the trip $T(c)$. Furthermore, the connection c is added to the journey of the passenger.

Next, we decide for every passenger sitting in the trip (i.e., sitting in the connection c that is currently being processed), if he disembarks at the arrival stop of the connection. As before, this is a discrete choice problem with two options. Option a is to leave the vehicle at the stop $v_{\text{arr}}(c)$, which has a PAT of $\tau^{\text{p}}_{\text{arr}}(a) = \tau^{\text{p}}_{\text{arr}}(c, d \mid \text{disembark})$. Option b is to continue with the same trip, which has a PAT of $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{trip})$. Given these two options, we once again use the discrete choice model to compute the probability of each option. Afterwards, we make a random decision for every passenger sitting in the trip, based on these probabilities. We collect all passengers that disembarked from the vehicle in a temporary list. If the arrival stop

of the connection happens to be the destination vertex, then the journeys of all passengers in the temporary list are complete and we simply continue with the next connection. Otherwise, we have to decide for all passengers in the temporary list, to which vertex they transfer (or if they simply wait at the current stop). Let $v_1, \dots, v_k$ be all the vertices for which $\tau_{\text{tra}}(v_{\text{arr}}(c), v_i) < \infty$ holds. In this case we have to decide between k options for all passengers in the temporary list. The perceived arrival time of the i-th option (i.e., transferring to v_i) is given by $\tau^{\text{p}}_{\text{arr}}(v_i) := \tau^{\text{p}}_{\text{tra}}(v_{\text{arr}}(c), v_i) + f^{v_i,d}_{\text{wait}}(\tau_{\text{arr}}(c) + \tau_{\text{tra}}(v_{\text{arr}}(c), v_i))$. Based on these PATs, we compute the probability of a passenger transferring to vertex v_i, using the discrete choice model. As before, we determine randomly for every passenger, which option he chooses. Finally, passengers transferring to vertex v_i are added to the queue of transferring passengers of the vertex v_i, with an arrival time of $\tau_{\text{arr}}(c) + \tau_{\text{tra}}(v_{\text{arr}}(c), v_i)$.

We repeat this process for every connection $c \in \mathcal{C}$. After processing every connection, we have assigned journeys to all passengers except the ones where no valid journey exists. Note that our algorithm can be used with an arbitrary decision model. We only used our linear discrete choice model as an example in the algorithm description.

8.2.3 Cycle Elimination

During the second phase, we assigned a journey to every passenger which might not necessarily be an optimal journey. Therefore, it is possible that the assigned journey contains cycles. In fact, it is even possible that a journey that is optimal with respect to perceived arrival time can contain cycles. This could be the case if the waiting cost λ_{wait} is very high, such that driving in a circle instead of waiting reduces the perceived arrival time. However, for some applications it might be undesirable or inadmissible to assign journeys containing cycles. Thus, we now describe an optional third phase of our algorithm, which removes all cycles from the assigned journeys.

In order to detect and remove cycles from a journey $J = \langle P_0, T_0^{ij}, \dots, T_{k-1}^{mn}, P_k \rangle$, which satisfies some demand $p = (o, d)$, we first convert the journey J into a sequence $\langle (v_1, \tau_1), \dots, (v_\ell, \tau_\ell) \rangle$ of (vertex, time)-pairs. We do this by adding for every trip leg T_i^{mn} and for every $j \in \{m, \dots, n\}$ the two pairs $(v(T[j]), \tau_{\text{arr}}(T[j]))$ and $(v(T[j]), \tau_{\text{dep}}(T[j]))$ to the sequence. Furthermore, we add $(o, \tau_{\text{dep}}(p))$ as the first element of the sequence and $(d, \tau_{\text{arr}}(J))$ as the last element of the sequence. Given the sequence $\langle (v_1, \tau_1), \dots, (v_\ell, \tau_\ell) \rangle$, we say that the journey contains a cycle, if indices i and $j > i + 1$ exist, such that the part of the journey between vertices v_i and v_j can be replaced by a transfer. This is the case if $\tau_i + \tau_{\text{tra}}(v_i, v_j) \leq \tau_j$ holds. Since we assume that the transfer graph is transitively closed, it consists of disjoint cliques. Thus, a journey can only contain a cycle if it contains two vertices v_i, v_j of the same clique. We can check this efficiently while iterating through the sequence of (vertex, time)-pairs. For every $i \in \{1, \dots, \ell\}$, we add i to a set, which we associated with the

clique that contains v_i. Afterwards, we check for each of these sets, if it contains indices i, j such that $\tau_i + \tau_{\text{tra}}(v_i, v_j) \leq \tau_j$ holds. If we find such indices i, j, then we have also found a cycle, which we then remove from the journey.

8.2.4 Implementation Details

We conclude the description of the CBA algorithm with some remarks on implementation details that are essential for the efficiency of the algorithm. We start with a description of the data structures that we use in our implementation. Afterwards, we show how the algorithm can be parallelized. Finally, we describe how our network model can be adapted for zone-based input data.

Data Structures. In order to represent the two profiles $f_{\text{trans}}^{v,d}$ and $f_{\text{wait}}^{v,d}$, which we need for every vertex v, we use dynamic arrays. Within these arrays we keep the breakpoints of the profile functions sorted in descending order with respect to the departure time. Since the profiles are created in the first phase of our algorithm, which scans connections in reverse chronological order, this assures the new breakpoints only have to be added at the end of the dynamic array.

Our algorithm maintains two queues for every stop of the network: one for passengers that are currently transferring to the stop and one for passengers that are already waiting at the stop. For the queue of transferring passengers it is important that we can efficiently extract all passengers that arrive before a certain point in time. Therefore, we use a min-heap (which we implemented as a binary heap) to represent this queue. In contrast, all passengers in the waiting queue are processed at the same time. Thus, a simple dynamic array can be used for this queue.

Finally, we note that we keep two copies of the transfer graph. During the PAT computation phase we have to iterate over all edges $(v, w) \in \mathcal{E}$ for a fixed vertex w. For this purpose we use an array that stores the incoming edges for every vertex. In contrast, we have to iterate over all outgoing edges of a vertex in the assignment phase. Thus, we use a second array that stores the outgoing edges for every vertex.

Parallelization. CBA begins with a short setup phase, during which the connections are sorted and the passengers are divided by their destination. Afterwards, a separate assignment is computed for every destination. Finally, the results are aggregated and the algorithm terminates. The assignment computation for the different destinations is by far the most complex part of the algorithm and can be performed independently for every destination. Therefore, it is quite easy to parallelize this part of the algorithm. First, the destinations are distributed among the available cores. Afterwards each core computes an independent assignment for the corresponding destinations.

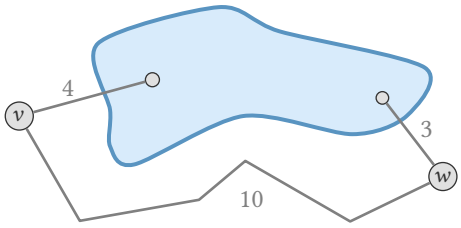


Figure 8.2: An example of a zone violating the triangle inequality. The vertices v and w are connected to the zone drawn in blue, but the distances are calculated based on different endpoints within the zone. Adding the zone directly as a vertex would create an v-w-path of length 7, whereas the actual distance between v and w is 10.

Preprocessing for Zone Based Demand. The input to the assignment algorithm consists of a public transit network with a transfer graph and some demand. Often, the demand data has a lower spatial resolution than the network. Origins and destinations are not supplied as precise locations, but rather as zones, which represent larger areas such as city districts. In this case, distances between zones and nearby stops are supplied in addition to the demand data. Before the assignment algorithm can be executed, the zones have to be integrated into the public transit network. However, simply adding vertices and edges for these zones to the transfer graph may create new paths between stops that are too short and violate the triangle inequality. This is because the zones represent regions with a non-zero expanse and the distances to nearby stops may be measured from any point within the region, not necessarily the center. Two edges whose distances are measured from different endpoints within the region may form a path between stops that is too short, since it does not include the time needed to travel between the two endpoints. An example of this is shown in Figure 8.2. To prevent this, we create two vertices for each zone: a source vertex that only has outgoing edges to stops of the zone and a sink vertex that only has incoming edges from stops of the zone. By not connecting the source and sink vertex, we prevent unwanted paths through zones from forming.

8.3 Multimodal Assignment

In this section we show how our assignment algorithm from the previous chapter can be combined with ULTRA, such that we can compute multimodal assignments. In particular, we explain how departure buffer times and unrestricted walking can be integrated into CBA. Furthermore, we demonstrate how passengers representing the same origin-destination pair can be grouped to improve the running time and the accuracy of the results. We present pseudo-code for our approach in Algorithm 8.1.

8.3.1 Departure Buffer Times

A consequence of using ULTRA is that we must switch from the minimum transfer time model to the departure buffer time model. Accordingly, passengers must observe the departure buffer time whenever a connection is entered, regardless of how it was reached. In the original CSA-based assignment algorithm, the minimum transfer time was considered part of the waiting time and therefore the waiting penalty λ_{wait} was applied to it. However, since the departure buffer time must always be observed, it may not be desirable to penalize it to the same degree as waiting, or at all. Therefore, we introduce a new buffer time penalty $\lambda_{\text{buf}} \in \mathbb{R}_0^+$ that may be different from the waiting penalty λ_{wait}. Whenever a connection is entered, the departure buffer time is multiplied by λ_{buf} and added to the PAT. Any time spent waiting before that (excluding the departure buffer time itself) is multiplied by λ_{wait} and added to the PAT.

8.3.2 ULTRA-Based Passenger Assignment

Our basic assignment algorithm uses CSA variants with restricted walking in both the PAT computation phase and the assignment phase. We extend the algorithm to unrestricted walking by replacing CSA with ULTRA-CSA in both phases. This requires several changes. The original PAT computation phase computed two PAT profiles at every stop with a backward CSA search. We replace this with a backward ULTRA-CSA search (line 11 of Algorithm 8.1), using the shortcut graph for intermediate transfers and a backward Bucket-CH search from the destination d for the final transfers. However, these profiles exclude the initial transfers from the origin vertices, which have to be evaluated at the start of the assignment phase, when the passengers are generated and choose which stop they transfer to.

Initial Transfer Challenges. Evaluating the initial transfers constitutes the main algorithmic challenge when integrating ULTRA into our assignment algorithm. The ULTRA technique was designed for one-to-one queries, where initial transfers can be computed with a single Bucket-CH query from the origin vertex. However, when solving an assignment problem, there typically exists demand from multiple origins for a single destination. Thus, we have to perform one Bucket-CH search for each of these origin vertices. Once the initial transfers are computed, a further challenge is the efficient evaluation of the resulting transfer options. In the restricted walking scenario, the choice set for initial and intermediate transfers was fairly small because passengers could only transfer to stops which were reachable via a direct edge. This made it feasible to simply iterate over all outgoing edges, compute the PAT, utility, and probability for each reached stop, and then make a decision. For the intermediate transfers, we can retain this approach in the unrestricted case by using the shortcut

Algorithm 8.1: ULTRA-CBA.

Input: Public transit network $N = (\mathcal{C}, \mathcal{S}, \mathcal{T})$ with transfer graph $G = (\mathcal{V}, \mathcal{E})$, ULTRA shortcut graph $G^{\mathsf{s}} = (\mathcal{S}, \mathcal{E}^{\mathsf{s}})$, and demand $\mathcal{D}$
Output: Utilization $\mu \colon \mathcal{C} \to \mathbb{R}_0^+$ of every connection and a set of journeys $\mathcal{J}$ for each origin-destination pair

Let $\mathcal{V}_{\mathsf{orig}}$ be the set of all origins with demand in $\mathcal{D}$
Let $\mathcal{V}_{\mathsf{dest}}$ be the set of all destinations with demand in $\mathcal{D}$
for each $o \in \mathcal{V}_{\mathsf{orig}}$ **do**
    $\mathcal{N}(o) \leftarrow \{(v, \tau_{\mathsf{tra}}(o, v)) \mid v \in \mathcal{S}\}$ `// Using Bucket-CH`
    Sort $\mathcal{N}(o)$ in ascending order of distance $\tau_{\mathsf{tra}}(o, \cdot)$
Sort $\mathcal{D}$ by destination
Sort $\mathcal{C}$ ascending by departure time
for each $d \in \mathcal{V}_{\mathsf{dest}}$ **do**
    Let $\mathcal{D}_d$ be the subset of $\mathcal{D}$ with destination d
    Sort $\mathcal{D}_d$ by origin
    Compute PAT profiles from every stop to d
    for each $p = (o, d) \in \mathcal{D}_d$ **do**
        Generate passenger group g of size λ_{mul} for p
        $\mathcal{J} \leftarrow \mathcal{J} \cup \{J_g = \{\}\}$
        Let $\mathcal{CS}$ be an empty choice set for p
        for each $(v, \tau_{\mathsf{tra}}(o, v)) \in \mathcal{N}(o)$ **do**
            $\tau_{\mathsf{dep}} \leftarrow \tau_{\mathsf{dep}}(p) + \tau_{\mathsf{tra}}(o, v)$
            $\tau^{\mathsf{p}}_{\mathsf{walk}} \leftarrow \lambda_{\mathsf{walk}} \cdot \tau_{\mathsf{tra}}(o, v)$
            $\bar{\tau}^{\mathsf{p}}_{\mathsf{arr}} \leftarrow \min\{\tau^{\mathsf{p}}_{\mathsf{arr}} \mid (\tau^{\mathsf{p}}_{\mathsf{arr}}, \cdot, \cdot) \in \mathcal{CS}\} + \lambda_{\Delta\mathsf{max}}$
            if $\tau_{\mathsf{dep}} + \tau^{\mathsf{p}}_{\mathsf{walk}} > \bar{\tau}^{\mathsf{p}}_{\mathsf{arr}}$ **then break**
            $\tau^{\mathsf{p}}_{\mathsf{arr}} \leftarrow f^{v,d}_{\mathsf{wait}}(\tau_{\mathsf{dep}} + \tau_{\mathsf{buf}}(v)) + \tau^{\mathsf{p}}_{\mathsf{walk}} + \lambda_{\mathsf{buf}} \cdot \tau_{\mathsf{buf}}(v)$
            $\mathcal{CS} \leftarrow \mathcal{CS} \cup \{(\tau^{\mathsf{p}}_{\mathsf{arr}}, v, \tau_{\mathsf{dep}})\}$
        Evaluate which choice from $\mathcal{CS}$ the passengers use
    for each $c \in \mathcal{C}$ **do** `// In chronological order`
        Evaluate if passengers waiting at $v_{\mathsf{dep}}(c)$ enter c
        $\mu(c) \leftarrow$ Number of passengers in c
        Add c to journeys J_g of groups g in c
        Evaluate if passengers using c disembark at $v_{\mathsf{arr}}(c)$
        Evaluate if passengers at $v_{\mathsf{arr}}(c)$ can transfer to d
        Evaluate to which stop passengers at $v_{\mathsf{arr}}(c)$ transfer
for each $J_g \in \mathcal{J}$ **do**
    Remove cycles from J_g
return $\mu, \mathcal{J}$

edges computed by ULTRA. However, the shortcuts do not cover initial transfers. When walking is unrestricted, almost all stops are reachable by initial walking from most origins. Therefore, it is no longer practical to explicitly collect all choices and compute their probabilities before making a decision. In practice, however, the probability for the vast majority of options will be 0 because the required footpath is so long that the resulting PAT will exceed the delay tolerance $\lambda_{\Delta\max}$.

Initial Transfer Choice Set. For an efficient evaluation of the initial transfers, we precompute the distances between the origin vertices and stops. For each origin o that occurs in the demand, we perform a Bucket-CH search from o to all stops (line 4). We store the distances from o to all reachable stops in a list of stop-distance tuples $(v, \tau_{\text{tra}}(o, v))$, sorted in ascending order of distance (line 5). After generating the passengers for an origin-destination pair $p = (o, d)$ (line 13), we iterate over the stop-distance tuples (line 16), compute the corresponding PATs and add them to the choice set $\mathcal{CS}$. For each tuple $(v, \tau_{\text{tra}}(o, v))$, we can compute the corresponding PAT by evaluating the profile $f_{\text{wait}}^{v,d}$ via binary search and adding the penalties for walking and the buffer time (line 21). Note that an option only has a non-zero gain and probability if its PAT does not exceed the PAT of the best option by more than $\lambda_{\Delta\max}$. To avoid iterating through the entire list of stop-distance tuples, we compute a lower bound for the PAT that increases monotonically with each tuple. The lower bound consists of the earliest possible departure time τ_{dep} at v (line 17) plus the perceived walking time $\tau_{\text{walk}}^{\text{p}}$ (line 18). Once this lower bound exceeds the best PAT found so far by more than $\lambda_{\Delta\max}$, we can stop iterating through the list since all further options will have a probability of 0 (line 20).

Assignment. After we have collected all relevant options, we evaluate the choice set $\mathcal{CS}$. This involves computing the utility of each option, using a decision model to compute the probabilities, and distributing the passengers to the stops according to the probabilities (line 23). The rest of the assignment phase (lines 24–30) then continues as usual, except that we use the shortcut graph for intermediate transfers and a Bucket-CH search from d for the final transfers. For each connection c, four decisions are made: First it is decided which passengers waiting at $v_{\text{dep}}(c)$ enter c (line 25). The utilization of c is then calculated as the number of passengers using c, including those that entered at $v_{\text{dep}}(c)$ and those that used a previous connection on the trip and remained seated (line 26). Then, c is added to the journey of each passenger using it (line 27). Passengers in c either decide to disembark at $v_{\text{arr}}(c)$ or remain in $T(c)$ (line 28). Those that disembark evaluate if they transfer directly to d (line 29). If they do not, they choose a stop to which they transfer next (line 30). Each decision is made by using a decision model to compute the probabilities and distributing the passengers accordingly. The cycle elimination phase (lines 31 and 32) remains unchanged.

8.3.3 Improved Passenger Grouping

The CBA algorithm only approximates the solution defined by the decision model, as the algorithm is based on a Monte Carlo method. The accuracy of this approximation is primarily influenced by the number of journeys that are sampled per origin-destination pair, which is controlled by the passenger multiplier λ_{mul}. The original assignment algorithm generates λ_{mul} copies of each passenger in the demand and then simulates the movement of all these copies independently. While the gain computation and evaluation of the decision model can be shared among all passengers in the same location, each passenger still has to make an individual random decision and move accordingly. This approach leads to redundant work because different copies of the same passenger will often make the same choices. We solve this problem by grouping passengers that make the same choices together.

Groups and Decision Making. Instead of individual passengers, we now route *passenger groups* g through the public transit network. The number of passengers in a group is indicated by the *group size* $|g|$. At the start of the assignment phase, we generate one group of size λ_{mul} for each origin-destination pair. Previously, whenever we had to make a choice between options with probabilities $\mathrm{P}_1, \ldots, \mathrm{P}_k$, we made an individual decision for each passenger by randomly picking an option according to the probabilities. Now, when making a decision for a group g of size $|g|$, we split it into k smaller groups of sizes $\lfloor \mathrm{P}_1 \cdot |g| \rfloor, \ldots, \lfloor \mathrm{P}_k \cdot |g| \rfloor$ and route each group according to the corresponding option. Because the group sizes are rounded down, some of the original $|g|$ passengers may still be left over afterwards. These passengers are still handled individually. As before, we randomly choose an option for each passenger according to the probabilities and add the passenger to the corresponding group. If the probability of an option is lower than $1/|g|$ and none of the leftover passengers are assigned to it, the corresponding group has a size of 0 and will be removed.

When groups that represent different origin-destination pairs encounter each other at a stop, we do not merge them into a single large group. While doing so would further reduce the computational effort, it would not allow us to reconstruct the journey that is assigned to each group in a straightforward manner.

The expected value for the share of passengers that are assigned to a travel option i is exactly P_i, regardless of whether the passengers are grouped or not. However, by grouping the passengers and splitting the groups according to the probabilities, a large portion of the assignment becomes deterministic. Only the assignment of the leftover passengers created by rounding errors is still randomized. Therefore, the computed utilization will not vary as strongly between different executions of the algorithm.

Instead of interpreting each unit in the simulation as a group of $|g|$ passengers, we can also view it as a single passenger and interpret the group size as a fixed-point

representation of the probability that the passenger will reach the current location, with λ_{mul} representing a probability of 1. In this view, λ_{mul} is a parameter controlling the precision of the fixed-point representation and thereby the accuracy of the results. If the precision was unlimited, rounding errors would no longer occur and the computed group sizes would conform exactly to the probabilities. In this case, our algorithm would no longer be a Monte Carlo simulation but rather compute an exact solution of the assignment problem.

When decisions are made for each passenger individually, the computational effort of the algorithm is proportional to λ_{mul}. With the grouped approach, the effort does not depend directly on λ_{mul}, but only on the number of group splits that are performed during the simulation. This is limited by the number of feasible options. Once the precision becomes high enough that each option with a non-zero probability is represented by a non-empty group, increasing the precision further may still improve the accuracy of the results, but it will not impact the running time.

8.4 Experiments

All algorithms presented in this chapter were implemented in C++17 and compiled with GCC version 8.2.1 and optimization flag -O3. All experiments were conducted on a machine with two 8-core Intel Xeon Skylake SP Gold 6144 CPUs, which are clocked at 3.50 GHz, with a boost frequency of up to 4.2 GHz, 192 GiB of DDR4-2666 RAM, and 24.75 MiB of L3 cache.

Network and Demand Data. Unfortunately, demand data is only available for the Stuttgart network. Thus, we only consider this network for the evaluation of our assignment algorithms. The demand model for the Stuttgart network, which we use for our experiments, was introduced in [SHP11]. Within this model the greater region of Stuttgart is partitioned into 1 174 zones. These zones are smallest in the center of Stuttgart, where a zone covers a few blocks at most. Zones become larger with increasing distance to the city. Finally, surrounding major cities, such as Frankfurt, Munich, or Zürich, are each represented with a single zone. The demand data contains 1 249 910 origin-destination pairs, which represents a typical amount of travel for one business day.

Tuning Parameters. We have introduced several tuning parameters that influence the accuracy of the assignment and the performance of our algorithms. For our experiments, we chose the following values: the walking cost is set to $\lambda_{\text{walk}} = 2.0$, the waiting cost is set to $\lambda_{\text{wait}} = \lambda_{\text{buf}} = 0.5$, the transfer cost as well as the delay tolerance are set to $\lambda_{\text{trans}} = \lambda_{\Delta\text{max}} = 300$ sec, and the maximum delay is set to $\lambda_{\text{delay}} = 60$ sec.

Table 8.1: Running time of CBA depending on the maximum delay λ_{delay}.

λ_{delay} [min]	1	2	4	8	16	32	64
Time [sec]	101.9	103.3	105.2	110.1	118.3	129.5	141.7

8.4.1 Public Transit Assignment

We begin with the evaluation of the CBA algorithm on the original version of the Stuttgart network, i.e., the network without additional transfers from OSM. We perform our experiments on the original network since it is also supported by VISUM[8], which we use as baseline for the evaluation of our approach.

Delay Tolerance. In our first experiments we evaluate the performance of CBA, depending on the various tuning parameters. Most of the parameters are primarily intended to model different passenger preferences. As such they do not directly influence the computational complexity of the algorithm. In fact, when we vary the parameters λ_{walk}, λ_{wait}, and λ_{trans} the runtime does not change more than in does between repeated passes with unchanged parameters. However, increasing the maximum delay λ_{delay} of the connections slightly increases the running time, as stated in Table 8.1. For every column in the table we used a passenger multiplier of $\lambda_{\text{mul}} = 10$, repeated the assignment computation ten times, and report the mean of the resulting running times. The increase in running time is caused by the computation of $\tau^{\text{p}}_{\text{arr}}(c, d \mid \text{trans})$, since more connections have a non-zero probability of being the successor connection for c.

Passenger Multiplier. Another important tuning parameter is the passenger multiplier λ_{mul}. Changing λ_{mul} directly influences the amount of work that has to be done, since more passengers have to be simulated. Figure 8.3 shows the running time of our algorithm dependent on λ_{mul}. In addition to the total running time we also report the time required for the individual phases of our algorithm. As expected, the running time increases with an increasing passenger multiplier. The additional running time is mostly due to the assignment phase, which is the phase that handles the simulated passengers. However, the running time of the assignment phase is not doubled when the number of passengers is doubled. The reason for this is that an increased number of passengers also leads to more passengers making the same decisions. Thus, the assignment computation can benefit from synergy effects. The PAT computation is completely independent from the number of passengers, which leads to a constant running time as indicated by the red curve. The time required for the cycle elimination and the setup phase (i.e., sorting the connections and distributing the passengers by destination) increases only slightly with an increased passenger multiplier.

[8] A commercial software from PTV (`https://www.ptvgroup.com/en/solutions/products/ptv-visum`)

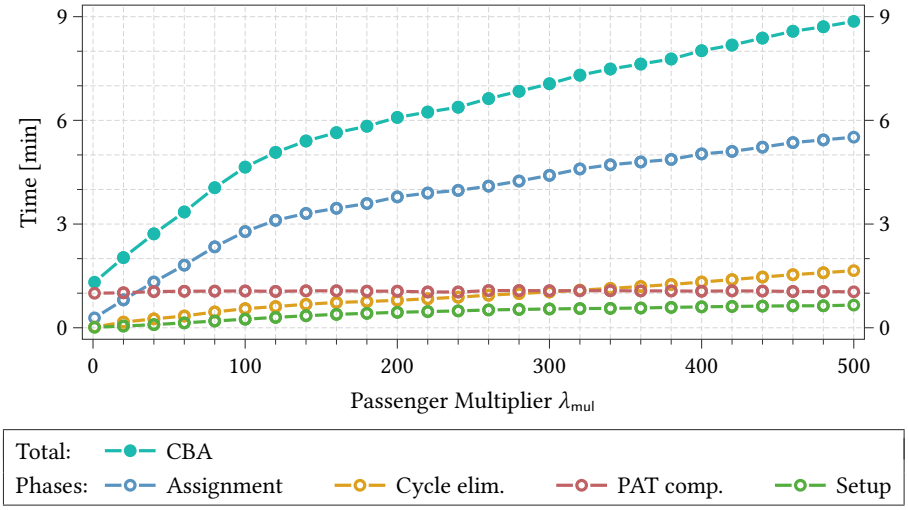


Figure 8.3: The running time of our algorithm depending on the passenger multiplier, differentiated by the phases of the algorithm. Changing the passenger multiplier primarily affects the assignment phase. Every measurement is the mean over the running times of ten repetitions of our algorithm.

Parallelization. Next, we evaluate the performance of the parallelized version of our algorithm. For the following experiment we use a passenger multiplier of $\lambda_{mul} = 10$, since this is in most cases sufficient for an accurate result. As before, we report the mean running time of executing our algorithm ten times. The serial version of the algorithm has a running time of 101.94 sec. Using the parallelized version with only one thread results in a slightly increased running time of 102.31 sec. Using two threads, we achieve a running time of 61.04 sec, four threads achieve 35.16 sec, eight threads achieve 18.37 sec, and 16 threads achieve 10.09 sec. For the case that we use all 16 available cores of our machine, this corresponds to a speed-up of 10.1. We observe that we do not achieve a perfect speed-up, despite the fact that the computations performed by the different threads are complete independent of each other. A likely reason for this is the fact that our algorithm primarily consists of scans through arrays. Thus, memory bandwidth could be a limiting factor.

We also compared the running time of our algorithm to VISUM, which is a commercial tool from PTV AG. For the Stuttgart network the VISUM computation took just above 30 minutes (in parallel with 8 threads). The VISUM assignment was computed on an Intel Core i7-6700 clocked at 3.4 GHz with 64 GiB of RAM, running Windows 10. Thus, CBA outperforms VISUM by a factor of almost 100 (for 8 parallel threads).

Table 8.2: Comparison between an assignment computed by VISUM and our algorithm. We report for every quantity the minimum (min), mean, standard deviation (sd), and maximum (max) over all journeys. The figures for both assignments are quite similar. However, our assignment slightly favors journeys with fewer trips (transfers), at the disadvantage of marginally increased travel time.

	VISUM				CBA			
	min	mean	sd	max	min	mean	sd	max
Total travel time [min]	2.98	46.89	23.75	429.00	2.98	47.20	23.44	429.00
Time spent in vehicle [min]	0.02	21.06	18.80	380.00	0.02	21.23	18.75	323.97
Time spent walking [min]	2.00	22.39	5.20	149.00	2.00	22.48	5.26	149.00
Time spent waiting [min]	0.00	3.43	5.72	217.02	0.00	3.49	5.68	217.02
Trips per passenger	1.00	1.77	0.83	6.00	1.00	1.75	0.84	8.00
Connections per passenger	1.00	9.40	7.44	109.00	1.00	9.47	7.33	97.00
Passengers per connection	0.00	12.74	37.79	1 290.10	0.00	12.85	37.58	1 233.60

Qualitative Evaluation. Finally, we compare the quality of the assignment computed by our algorithm to the one computed by VISUM. Table 8.2 summarizes the results. Overall, the assignments computed by our algorithm and VISUM are quite similar. Our algorithm assigns journeys with slightly longer mean travel time, in favor of a slightly decreased number of transfers. At the same time, our algorithm assigns journeys with a higher maximum number of trips. The reason for this is that VISUM prunes all journeys with more than 6 trips, while our algorithm has no hard limit on the number of transfers. It is noticeable, that both techniques assign about 1200 passengers to a single vehicle, since both are not capable of handling vehicle capacities.

8.4.2 Multimodal Assignment

We continue with the experimental evaluation of the ULTRA-CBA algorithm. For this we used the multimodal variant of the Stuttgart network, i.e., we added the unrestricted transfer graph that we extracted from OSM (assuming a walking speed of 4.5 km/h). We compare our results to the plain version of CBA with the same configuration as in the previous section. We use the same demand data that was already used in the previous section for both algorithms.

Preprocessing. Before the ULTRA-CBA algorithm can be executed, we have to compute the required data structures in a preprocessing step. As our query algorithm utilizes Bucket-CH queries, we have to compute a CH. Additionally we need the

Table 8.3: Comparison of assignments computed with CBA on the original network and assignments computed with ULTRA-CBA on the multimodal network. We report sequential and parallel execution times, where we used 16 threads for the parallel variant. Figures concerning the quality of the computed assignment are averaged over all origin-destination pairs that were assigned to valid journeys in both network variants, with and without unrestricted transfers.

	Real demand		Random demand	
	ULTRA-CBA	CBA	ULTRA-CBA	CBA
Execution time (seq.) [sec]	181.9	278.8	258.8	494.8
Execution time (par.) [sec]	16.8	34.1	19.7	52.9
Travel time [min]	46.8	49.1	82.2	91.2
Walking time [min]	22.2	22.2	24.0	23.7
Time in vehicle [min]	20.7	21.8	48.5	53.5
Connections per passenger	10.19	10.76	19.16	22.02
Trips per passenger	1.85	1.88	2.88	3.06
Journeys per passenger	12.79	9.27	17.85	13.43

ULTRA transfer shortcuts, which in turn require a core graph for their computation. The CH was computed in 2:44 min using a single thread and introduced 5 469 298 shortcuts. The core graph was computed using a Core-CH approach with a limit of 16 for the average vertex degree within the core. This resulted in a preprocessing time of 2:30 min (using one thread) and a core graph that contains 25 477 vertices and 407 664 edges. The core graph was then used for the ULTRA preprocessing, which took another 2:03 min (using 16 threads) and resulted in 74 038 transfer shortcuts.

Comparison of CBA and ULTRA-CBA. The preprocessing, i.e., computing the ULTRA shortcuts and CH, has to be repeated every time the network changes. However, the precomputed data structures can be reused for different demands. In our evaluation of the two algorithms we used two demand sets: the real demand data from [SHP11] and a demand of the same size but with origin and destination zones picked uniformly at random. An overview of the results is given in Table 8.3.

In order to achieve reasonable precision in the result, we use a passenger multiplier of 100 for all experiments, unless stated otherwise. Recall that this means that we record 100 journeys for every origin-destination pair in the demand and thus compute the probability space of possible journeys for the demand pairs with two decimal places. Furthermore, when comparing qualitative figures of the assignment (such as average travel time or number of used vehicles), we only consider origin-destination

pairs that could be assigned to at least one journey in both networks, with and without the multimodal transfer graph. Using the original network, only 1 209 761 of the 1 249 910 origin-destination pairs could be assigned. For all other pairs no feasible journey exists. Using the unrestricted transfer graph, the number of assignable pairs increases to 1 246 337. In both cases some origin-destination pairs that could not be assigned to any journey exist, because the desired departure time is too late for the last possible journey contained in our network data.

Comparing the efficiency of ULTRA-CBA with CBA, we observe that despite solving a more difficult problem, ULTRA-CBA outperforms CBA by a factor of about two regarding execution time (compare row 1 and 2 of Table 8.3). For both demand sets, the parallel version of ULTRA-CBA finishes in below 20 seconds. We also observe that assigning a random demand takes significantly longer than assigning the real demand. A possible reason for this is that picking origin and destination zones uniformly at random tends to produce long-distance demand pairs, while real demand contains more short-distance origin-destination pairs. This assumption is backed by the different average travel times for both demands. While a journey for a real origin-destination pair takes about 48 minutes, a journey for the random demand takes almost twice as long at about 80 to 90 minutes. Additionally, a longer distance between origin and destination vertices tends to lead to a larger set of possible journeys. Our measurements also confirm this correlation, with the number of assigned journeys increasing by about 50% when switching from real demand to random demand. A larger number of assigned journeys implies that more groups have to be split during the assignment phase, which directly affects the execution time of the algorithm and thus explains why the assignment for random demand takes longer. For the real demand our algorithm assigns 12.79 journeys on average to each origin-destination pair. Since the demand contains 1 209 761 pairs, this means that our algorithm computes over 15 million individual journeys in less than 17 seconds.

Impact of Unrestricted Walking. In contrast to the results that we presented in Chapter 5, which were already confirmed in [PV19], we observe that adding unrestricted transfers to a network only has a small effect on the average travel times in the assignment. For the real demand, average travel times are only reduced by 4.6% when switching from the original network to the unrestricted network. However, results in Chapter 5 and in [PV19] were obtained by using random queries between vertices of the network. In contrast, we used a zone-based demand for the assignment computation. Furthermore, it can be assumed that the network was designed to match a demand that is similar to our real demand data. When considering random demand, the effect of unrestricted transfers already becomes more pronounced, as it reduces travel times by 9.8% in this case.

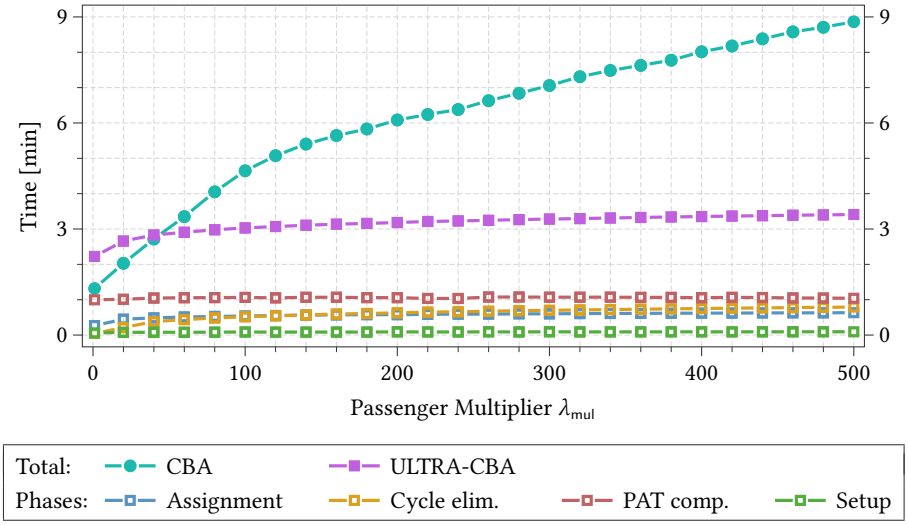


Figure 8.4: Sequential running times of our assignment algorithms depending on the passenger multiplier. We compare ULTRA-CBA to the results for CBA from Figure 8.3. For ULTRA-CBA we also report the running times of its sub-phases. The measured running times are averaged over ten executions of the algorithm.

Passenger Multiplier. The most important tuning parameter of the ULTRA-CBA algorithm is the passenger multiplier λ_{mul}, which influences both the execution time of the algorithm and the accuracy of the computed assignment. As stated before, the effect of λ_{mul} on the result is quite direct, as the logarithm of λ_{mul} corresponds to the number of decimal places in the probability space that are computed. However, the effect of λ_{mul} on the execution time is not as clear. Therefore, we evaluate the performance of our algorithm depending on the passenger multiplier in Figure 8.3. The plot shows that the total execution time increases with increasing λ_{mul}. However, the impact of λ_{mul} on the total execution time decreases notably for high λ_{mul}. The reason for this flattening of the curve is that the execution time of our algorithm does not depend directly on the passenger multiplier, but only on the number of group splits. As more passengers are added, new groups are created less frequently since they represent options with increasingly small probabilities. This result demonstrates the usefulness of our new grouping approach during the assignment phase.

For comparison we also include the execution time of CBA on the original network in Figure 8.4. The direct comparison of our two algorithms shows that CBA outper-

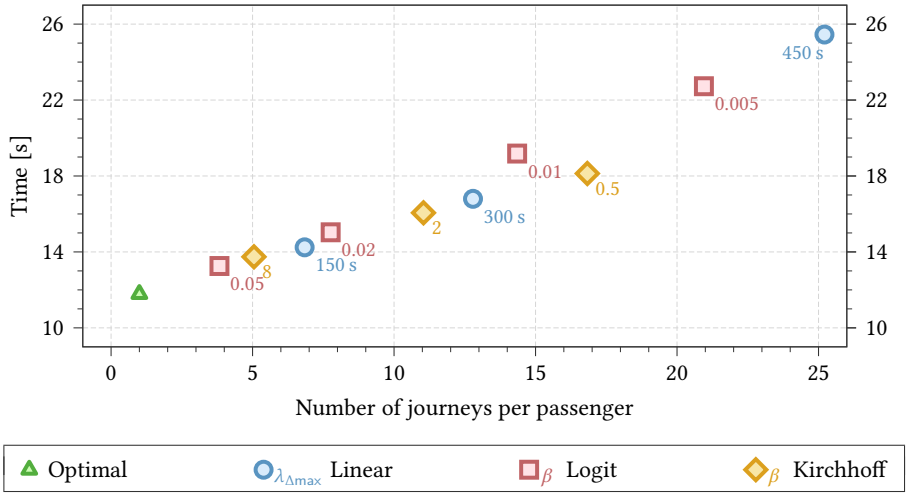


Figure 8.5: Variation in the number of journeys per passenger and the execution time for different discrete choice models. We compare the Linear, Logit, and Kirchhoff models, each with different parameter settings. The used parameter values are annotated at the corresponding marker. For comparison we include the results of an *optimal* decision model, where the journey with optimal utility is chosen deterministically. Running times are averaged over ten executions.

forms ULTRA-CBA for passenger multipliers below 50. However, this is expected since our algorithm operates on a more complex network with unrestricted transfers. On the other hand, for higher passenger multipliers ULTRA-CBA outperforms the previous approach, due to the grouping of the passengers.

We also report running times for the four sub-phases of ULTRA-CBA in Figure 8.3. Note that the colors of the four sub-phase curves correspond to the colors of the line numbers in Algorithm 8.1. The plot shows that the most costly phase of our algorithm is the PAT computation phase. This observation matches our expectations, as the PAT computation phase has to scan the complete multimodal transportation network, including final transfers.

Decision Models. With our last experiment we demonstrate the versatility of our assignment algorithm by showing that our approach is compatible with a multitude of decision models. Therefore, we evaluate the performance of the ULTRA-CBA algorithm combined with the Logit, Kirchhoff, and Linear decision model. Furthermore,

we test each decision model with different parameter settings. The resulting running times (using the parallelized algorithm with 16 threads) are shown in Figure 8.5. Depending on the used decision model and its parameter settings, the execution time varies between 11 seconds and 26 seconds. As before, the reason for this is primarily the number of different journeys that are assigned to each origin-destination pair, which correspond to the number of times that groups have to be split during the assignment phase. To demonstrate this, we plotted the running time of the different models against the number of paths per passenger. This plot clearly shows the linear correlation between the number of journeys per passenger and the execution time, confirming that our algorithm achieves the same efficiency for all tested decision models.

Besides execution time and number of assigned journeys, all decision models yield similar assignments. The average travel time, for example, ranges from 46:28 minutes for the optimal assignment to 47:10 minutes for the Kirchhoff model with $\beta = 0.5$. The reason for this is that additional suboptimal paths found by the algorithm are either only slight variations of the optimal path or have only a small probability and thus do not contribute much to the average. Similar observations can be made for other quality metrics such as average number of vehicles used or average walking time.

8.5 Final Remarks

In this chapter we developed efficient traffic assignment algorithms for both, public transportation networks and multimodal networks. We achieved this by adapting CSA and the MEAT technique for efficient evaluation of a sequential route choice model. As a result we presented the highly efficient CBA algorithm, which is compatible with a wide range of discrete choice models that are commonly used in traffic assignments. In an experimental evaluation we demonstrated that our approach computes assignments that are comparable to those found by a state-of-the-art commercial traffic assignment software. However, our approach is about two orders of magnitude faster than the commercial tool.

We proceeded by combining our assignment algorithm with ULTRA, in order to enable multimodal assignments. Furthermore, we presented an improved representation of passenger groups, which resulted in a significantly reduced running time of our algorithm. Because of this, the running time of ULTRA-CBA is comparable to the running time of CBA, despite the fact that ULTRA-CBA operates on a much larger network. Moreover, for large passenger multipliers (i.e., increased result accuracy) ULTRA-CBA is even faster than CBA. Overall, the parallelized version of our multimodal assignment algorithm is capable of computing an assignment for over 1.2 million origin-destination pairs in less than 17 seconds.

Future Work. For future work, we would like to improve the overall quality of the computed assignments by integrating more complex journey choice models. More sophisticated models could for example consider vehicle capacities and reduce the likelihood of assigning passengers to overcrowded vehicles. Furthermore, it would be interesting to correlate the probabilities of journeys that overlap partially, for example if both use the same vehicle as a leg of the journey. Finally, we would like to incorporate the cycle elimination phase into the assignment phase, such that journeys containing cycles are not assigned in the first place.

9 Conclusion

In this thesis, we developed novel algorithms for efficient multimodal journey planning as well as the computation of multimodal traffic assignments. During the design of these algorithms we focused especially on the combination of schedule-based public transit with non-schedule-based private modes of transportation, which has been a big challenge for previous journey planning algorithms. In order to find practicable solutions for this and other challenges that arise in the context of multimodal journey planning we followed the principles of Algorithm Engineering methodology. Thus, we placed special emphasis on the performance of our algorithms on real world data. In the following, we present a summary of our main contributions. Afterwards, we conclude this work with an outlook on problems that are still open and propose some interesting topics for future work.

9.1 Summary

The first problem that we considered in this work was the efficient computation of multimodal profiles, i.e., finding all optimal journeys in multimodal networks that depart within a given interval. In order to solve this problem we proposed an algorithm that iteratively reduces the departure time interval by cleverly applying preexisting journey planning algorithms for fixed departure times. We demonstrated the feasibility of our approach through an experimental evaluation on four real world networks. Since no multimodal profile algorithms existed prior to our work, we compared our approach to the profile variant of CSA, which can only handle transitively closed

graphs. While our new algorithm is not as fast as CSA, which is not surprising as the multimodal network is significantly larger than the transitive network, running times are still comparable. Overall, our algorithm only needs a few minutes to compute full 24 hour profiles. Furthermore, our efficient profile algorithm enabled us to compare journeys with and without unrestricted walking in depth. As a result, we found that considering multimodal journeys can improve travel times significantly. Moreover, we observed that walking is particularly important at the begin and the end of a journey, while walking between two public transit trips is less often required.

Based on this observation we developed ULTRA, a preprocessing technique that can be combined with various preexisting public transit journey planning algorithms in order to enable efficient multimodal journey planning. Our approach is based on the idea of finding a small set of shortcuts that is sufficient to represent all necessary transfers of Pareto-optimal journeys. For the computation of these shortcuts, we presented an efficient preprocessing algorithm that can easily be parallelized and only takes a few minutes to process smaller networks. Even for the largest network available to us the shortcuts could be computed in two and a half hours. As query algorithms for our ULTRA approach we implemented an evaluated RAPTOR, CSA, and Trip-Based Routing. Notably, this means that we have developed the first multimodal variant of CSA and Trip-Based Routing. In an extensive experimental evaluation we have demonstrated that all three query algorithms have running times that are comparable to their basic variants for public transit networks, despite the fact that the multimodal network is significantly larger. The fastest approach of the three is ULTRA-Trip-Based, which is an order of magnitude faster than MR-∞, the fastest previously known multi-modal algorithm for bi-criteria optimization. Furthermore, we demonstrated that the ULTRA approach does not only work if walking is used as transfer mode, but is in fact feasible independently of the speed of the transfer mode.

We continued with adapting ULTRA for a more complex scenario that considers bike sharing as an additional transportation mode, besides public transit and walking. In particular, we developed two novel approaches for modeling networks that contain bike sharing stations of competing bike sharing operators: the operator-dependent and operator-expanded model. Additionally, we developed a fast preprocessing step called operator pruning, which can be used to accelerate queries in both models. We demonstrated with an experimental evaluation that ULTRA-RAPTOR in combination with our operator pruning approach can compute journeys more than an order of magnitude faster than a variant of MCR, which we used as baseline. Moreover, we showed that the running time improvement achieved by operator running has an even greater effect on the preprocessing phase of ULTRA. For metropolitan networks operator pruning yields a speed-up factor of 10 for the shortcut computation and for the country sized networks the shortcut computation is more than 20 times faster than without operator pruning.

The last problem that we addressed in this work is the assignment problem. We presented the CBA algorithm that efficiently solves the assignment problem for public transit networks by evaluating a sequential route choice model using algorithmic approaches that are based on CSA. This algorithm is capable of computing high quality assignments 100 times faster than state-of-the-art commercial tools. We further improved our algorithm by combining it with ULTRA, which enabled us to compute assignments for multimodal networks. In combination with some general improvements to our assignment algorithm, ULTRA-CBA is capable of computing an assignment for over 1.2 million origin-destination pairs in less than 17 seconds, which is orders of magnitude faster than existing solutions. Moreover, we demonstrated that our assignment algorithm is compatible with several well-known decision models.

In summary, the main contribution of this thesis is ULTRA, a versatile framework for multimodal journey planning. We demonstrated the efficiency and feasibility of our approach for multimodal networks with various transfer modes. Furthermore, we adapted our approach to more complex applications, such as journey planning with bike sharing or the computation of traffic assignments.

9.2 Outlook

For future work, it would be interesting to adapt ULTRA to some advanced scenarios. One such scenario is the multicriteria optimization. Currently ULTRA guarantees that all Pareto-optimal journeys with respect to travel time and number of used trips can be found. However, there exist many other important properties of a journey that a passenger might want to optimize, such as the price of the journey or the total length of all walking transfers. Thus, it would be interesting to compute a small set of transfer shortcuts that cover all Pareto-optimal journeys with respect to these criteria.

Another important scenario is journey planning in networks where some of the trips are delayed. Most public transit journey planning algorithms handle delays by repeating the preprocessing steps once the delay is known. However, for ULTRA this approach is only feasible on small networks, since the preprocessing phase takes several hours for large networks like Germany. Thus, it would be interesting to compute an extended set of transfer shortcuts that are sufficient for the computation of optimal journeys if small or anticipated delays occur.

Regarding the traffic assignment problem it would be interesting to consider vehicle capacities. A common approach for this within the literature are iterative techniques, where the utility of a journey depends on the utilization of the vehicles from previous iterations. However, these techniques are often not practicable, as they take several hours or even days to converge. Since our assignment algorithm is significantly faster than preexisting algorithms it could be used to make iterative approaches viable.

Bibliography

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. **A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks**. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Volume 6630 of Lecture Notes in Computer Science (LNCS), pages 230–241. Springer, 2011.
Cited on pages 1, 14.

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. **Hierarchical Hub Labelings for Shortest Paths**. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*. Volume 7501 of Lecture Notes in Computer Science, pages 24–35. Springer, 2012.
Cited on page 14.

[ALS13] Julian Arz, Dennis Luxen, and Peter Sanders. **Transit Node Routing Reconsidered**. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Volume 7933 of Lecture Notes in Computer Science, pages 55–66. Springer, 2013.
Cited on page 14.

[AW12] Leonid Antsfeld and Toby Walsh. **Finding Multi-criteria Optimal Paths in Multi-modal Public Transportation Networks using the Transit Algorithm**. In *Proceedings of the 19th Intelligent Transport Systems World Congress*, pages 25–34, 2012.
Cited on page 14.

[Bas+10] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. **Fast Routing in Very Large Public Transportation Networks using Transfer Patterns**. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*. Volume 6346 of Lecture Notes in Computer Science, pages 290–301. Springer, 2010.
Cited on pages 2, 16, 48, 60.

[Bas+16] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. **Route Planning in Transportation Networks**. In *Algorithm Engineering - Selected Results and Surveys*. Volume 9220. Lecture Notes in Computer Science. Springer, 2016, pages 19–80.
Cited on pages 1, 11.

[Bas09] Hannah Bast. **Car or Public Transport - Two Worlds**. In *Efficient Algorithms*. Volume 5760 of Lecture Notes in Computer Science, pages 355–367. Springer, 2009.
Cited on page 3.

[Bau+10a] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. **Preprocessing Speed-Up Techniques is Hard**. In *Proceedings of the 7th International Conference on Algorithms and Complexity (CIAC'10)*. Volume 6078 of Lecture Notes in Computer Science, pages 359–370. Springer, 2010.
Cited on page 38.

[Bau+10b] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. **Combining Hierarchical and Goal-directed Speed-up Techniques for Dijkstra's Algorithm**. In *Journal of Experimental Algorithmics (JEA)* volume 15, pages 2.3:1–2.3:31, Association for Computing Machinery (ACM), 2010.
Cited on pages 13, 39, 84.

[Bau+15] Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. **Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles**. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 44:1–44:10. Association for Computing Machinery, 2015.
Cited on pages 13, 84.

[Bau+19a] Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. **UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution**. In *ArXiv e-prints* 1906.04832, URL: http://arxiv.org/abs/1906.04832. Technical report, 2019.
Cited on pages 7, 8, 79.

[Bau+19b] Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. **UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution**. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA'19)*. Volume 144 of Leibniz International Proceedings in Informatics (LIPIcs), pages 14:1–14:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019.
Cited on pages 7, 8, 79.

[BBDW16] Moritz Baum, Valentin Buchhold, Julian Dibbelt, and Dorothea Wagner. **Fast Exact Computation of Isochrones in Road Networks**. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*. Volume 9685 of Lecture Notes in Computer Science, pages 17–32. Springer, 2016.
Cited on page 84.

[BBS13] Hannah Bast, Mirko Brodesser, and Sabine Storandt. **Result Diversity for Multi-Modal Route Planning**. In *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13)*. Volume 33 of OpenAccess Series in Informatics, pages 123–136. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013.
Cited on page 19.

[BDGM09] Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller-Hannemann. **Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected**. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'09)*. Volume 12 of OpenAccess Series in Informatics, pages 2:1–2:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2009.
Cited on pages 3, 12.

[BDSV09] Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. **Time-Dependent Contraction Hierarchies**. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, pages 97–105. SIAM, 2009.
Cited on page 13.

[BDW11] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. **Experimental Study of Speed Up Techniques for Timetable Information Systems**. In *Networks* volume 57, pages 38–52, 2011.
Cited on pages 1, 15, 16.

[Ben75] Jon L. Bentley. **Multidimensional Binary Search Trees Used for Associative Searching**. In *Communications of the ACM* volume 18:9, pages 509–517, ACM New York, 1975.
Cited on page 51.

[BFM09] Hannah Bast, Stefan Funke, and Domagoj Matijevic. **Ultrafast Shortest-Path Queries via Transit Nodes**. In *The Shortest Path Problem: 9th DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 175–192. American Mathematical Society, 2009.
Cited on page 14.

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. **Fast Routing in Road Networks with Transit Nodes**. In *Science* volume 316, pages 566–566, American Association for the Advancement of Science, 2007.
Cited on page 14.

[BGM10] Annabell Berger, Martin Grimmer, and Matthias Müller-Hannemann. **Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks**. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Volume 6049 of Lecture Notes in Computer Science, pages 35–46. Springer, 2010.
Cited on pages 16, 60.

[BGNS10] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. **Time-Dependent Contraction Hierarchies and Approximation**. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Volume 6049 of Lecture Notes in Computer Science, pages 166–177. Springer, 2010.
Cited on page 13.

[BHS16] Hannah Bast, Matthias Hertel, and Sabine Storandt. **Scalable Transfer Patterns**. In *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments (ALENEX'16)*, pages 15–29. SIAM, 2016.
Cited on pages 16, 54, 60.

[BJ04] Gerth S. Brodal and Riko Jacob. **Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries**. In *Proceedings of the 3rd Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*. Volume 92 of Electronic Notes in Theoretical Computer Science, pages 3–15. Elsevier, 2004.
Cited on page 15.

[BJR16] Dominik Bucher, David Jonietz, and Martin Raubal. **A Heuristic for Multi-modal Route Planning**. In *Progress in Location-Based Services*. Lecture Notes in Geoinformation and Cartography, pages 211–229. Springer, 2016.
Cited on page 18.

[Bri+17] Lars Briem, Sebastian Buck, Holger Ebhart, Nicolai Mallig, Ben Strasser, Peter Vortisch, Dorothea Wagner, and Tobias Zündorf. **Efficient traffic assignment for public transit networks**. In *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17)*. Volume 75 of Leibniz International Proceedings in Informatics, pages 20:1–20:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.
Cited on pages 7, 9, 129.

[BS14] Hannah Bast and Sabine Storandt. **Frequency-Based Search for Public Transit**. In *Proceedings of the 22nd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 13–22. Association for Computing Machinery, 2014.
Cited on pages 17, 54, 60.

[CGR96] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. **Shortest Paths Algorithms: Theory and Experimental Evaluation**. In *Mathematical Programming* volume 73, pages 129–174, Springer, 1996.
Cited on pages 36, 85.

[Cio+17] Alessio Cionini, Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos D. Zaroliagis. **Engineering Graph-Based Models for Dynamic Timetable Information Systems**. In *Journal of Discrete Algorithms* volume 46-47, pages 40–58, 2017.
Cited on page 15.

[CNRV96] Ennio Cascetta, Agostino Nuzzolo, Francesco Russo, and Antonino Vitetta. **A Modified Logit Route Choice Model Overcoming Path Overlapping Problems. Specification and some Calibration Results for Interurban Networks**. In *Proceedings of The 13th International Symposium On Transportation And Traffic Theory*, 1996.
Cited on page 20.

[Dal87] Andrew Daly. **Estimating "Tree" Logit Models**. In *Transportation Research Part B* volume 21, pages 251–267, Elsevier, 1987.
Cited on page 21.

[Dan63] George B. Dantzig. **Linear Programming and Extensions**. Princeton University Press, 1963.
Cited on page 12.

[DDP19] Daniel Delling, Julian Dibbelt, and Thomas Pajor. **Fast and Exact Public Transit Routing with Restricted Pareto Sets**. In *Proceedings of the 21st Workshop on Algorithm Engineering and Experiments (ALENEX'19)*, pages 54–65. SIAM, 2019.
Cited on page 19.

[DDPW15] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. **Public Transit Labeling**. In *Proceedings of the 14th International Symposium on Experimental Algorithms*. Volume 9125 of Lecture Notes in Computer Science, pages 273–285. Springer, 2015.
Cited on pages 14, 15, 48, 54, 60.

[DDPZ17] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Tobias Zündorf. **Faster Transit Routing by Hyper Partitioning**. In *Proceedings of the 17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'17)*. Volume 59 of OpenAccess Series in Informatics, pages 8:1–8:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.
Cited on page 16.

[Del+09] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. **High-Performance Multi-Level Routing**. In *The Shortest Path Problem: 9th DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 73–91. American Mathematical Society, 2009.
Cited on page 14.

[Del+13] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. **Computing Multimodal Journeys in Practice**. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Volume 7933 of Lecture Notes in Computer Science, pages 260–271. Springer, 2013.
Cited on pages 2, 13, 18, 43, 47, 61, 84, 113, 121.

[DGPW17] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. **Customizable Route Planning in Road Networks**. In *Transportation Science* volume 51, pages 566–591, INFORMS, 2017.
Cited on page 14.

[DGSW14] Daniel Delling, Andrew V. Goldberg, Ruslan Savchenko, and Renato F. Werneck. **Hub Labels: Theory and Practice**. In *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*. Volume 8504 of Lecture Notes in Computer Science, pages 259–270. Springer, 2014.
Cited on page 14.

[Dij59] Edsger W. Dijkstra. **A Note on Two Problems in Connexion with Graphs**. In *Numerische Mathematik* volume 1, pages 269–271, 1959.
Cited on pages 12, 35, 62, 82.

[DK19] Mattia D'Emidio and Imran Khan. **Dynamic Public Transit Labeling**. In *Proceedings of the 19th International Conference Computational Science and Its Applications (ICCSA'19)*. Volume 11619 of Lecture Notes in Computer Science, pages 103–117. Springer, 2019.
Cited on page 16.

[DKP12] Daniel Delling, Bastian Katz, and Thomas Pajor. **Parallel Computation of Best Connections in Public Transportation Networks**. In *Journal of Experimental Algorithmics (JEA)* volume 17, pages 4.1–4.26, Association for Computing Machinery (ACM), 2012.
Cited on pages 59, 60.

[DM75] Thomas A. Domencich and Daniel McFadden. **Urban travel demand - A Behavioral Analysis**. North-Holland Publishing co. Ltd., 1975.
Cited on pages 20, 135.

[DMS08] Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. **Multi-criteria Shortest Paths in Time-Dependent Train Networks**. In *Proceedings of the 7th International Workshop on Experimental and Efficient Algorithms (WEA'08)*. Volume 5038 of Lecture Notes in Computer Science, pages 347–361. Springer, 2008.
Cited on pages 16, 60, 113.

[DPSW13] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. **Intriguingly Simple and Fast Transit Routing**. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Volume 7933 of Lecture Notes in Computer Science (LNCS), pages 43–54. Springer, 2013.
Cited on pages 16, 24, 60, 86.

[DPSW18] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. **Connection Scan Algorithm**. In *Journal of Experimental Algorithmics* volume 23, pages 1.7:1–1.7:56, Association for Computing Machinery (ACM), 2018.
Cited on pages 16, 31, 40, 54, 59, 67, 86.

[DPW09a] Daniel Delling, Thomas Pajor, and Dorothea Wagner. **Accelerating Multi-modal Route Planning by Access-Nodes**. In *Proceedings of the 17th Annual European Symposium on Algorithms (ESA'09)*. Volume 5757 of Lecture Notes in Computer Science, pages 587–598. Springer, 2009.
Cited on pages 14, 18.

[DPW09b] Daniel Delling, Thomas Pajor, and Dorothea Wagner. **Engineering Time-Expanded Graphs for Faster Timetable Information**. In *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*. Volume 5868. Lecture Notes in Computer Science. Springer, 2009, pages 182–206.
Cited on page 15.

[DPW12] Daniel Delling, Thomas Pajor, and Renato F. Werneck. **Round-Based Public Transit Routing**. In *Proceedings of the 14th Workshop on Algorithm Engineering and Experiments (ALENEX'12)*, pages 130–140, 2012.
Cited on pages 16, 25, 31, 47.

[DPW15a] Daniel Delling, Thomas Pajor, and Renato F. Werneck. **Round-Based Public Transit Routing**. In *Transportation Science* volume 49, pages 591–604, INFORMS, 2015.
Cited on pages 2, 16, 41, 54, 59, 60, 64, 81, 86.

[DPW15b] Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. **User-Constrained Multimodal Route Planning**. In *Journal of Experimental Algorithmics (JEA)* volume 19, pages 3.2:1–3.2:19, Association for Computing Machinery (ACM), 2015.
Cited on pages 13, 18, 84.

[DSW14] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. **Delay-Robust Journeys in Timetable Networks with Minimum Expected Arrival Time**. In *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*. Volume 42 of OpenAccess Series in Informatics, pages 2:1–2:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014.
Cited on pages 17, 60, 131, 132, 139.

[DSW15] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. **Fast Exact Shortest Path and Distance Queries on Road Networks with Parametrized Costs**. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 66:1–66:4. Association for Computing Machinery, 2015.
Cited on page 52.

[DSW16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. **Customizable Contraction Hierarchies**. In *Journal of Experimental Algorithmics (JEA)* volume 21, pages 1.5:1–1.5:49, Association for Computing Machinery (ACM), 2016.
Cited on page 14.

[EP13] Alexandros Efentakis and Dieter Pfoser. **Optimizing Landmark-Based Routing and Preprocessing**. In *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, pages 25–30. Association for Computing Machinery, 2013.
Cited on page 12.

[FHW01] Markus Friedrich, Ingmar Hofsaess, and Steffen Wekeck. **Timetable-Based Transit Assignment Using Branch & Bound**. In *Transportation Research Record* volume 1752, pages 100–107, SAGE, 2001.
Cited on page 20.

[FT87] Michael L. Fredman and Robert E. Tarjan. **Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms**. In *Journal of the ACM* volume 34, pages 596–615, Association for Computing Machinery (ACM), 1987.
Cited on page 36.

[Gei10] Robert Geisberger. **Contraction of Timetable Networks with Realistic Transfers**. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*. Volume 6049 of Lecture Notes in Computer Science (LNCS), pages 71–82. Springer, 2010.
Cited on page 13.

[GH05] Andrew V. Goldberg and Chris Harrelson. **Computing the Shortest Path: A* Search Meets Graph Theory**. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.
Cited on page 12.

[GN16] Guido Gentile and Klaus Nökel (editors). **Modelling Public Transport Passenger Flows in the Era of Intelligent Transport Systems**. Volume 10 of Springer Tracts on Transportation and Traffic. Springer, 2016.
Cited on pages 11, 135.

[GP06] Guido Gentile and Andrea Papola. **An Alternative Approach to Route Choice Simulation: The Sequential Models**. In *Proceedings of the European Transport Conference.* Association for European Transport, 2006.
Cited on pages 21, 136.

[GPZ19] Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos D. Zaroliagis. **Multimodal Dynamic Journey-Planning**. In *Algorithms* volume 12, pages 213:1–213:16, Multidisciplinary Digital Publishing Institute, 2019.
Cited on page 17.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. **Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks**. In *Proceedings of the 7th International Workshop on Experimental and Efficient Algorithms (WEA'08).* Volume 5038 of Lecture Notes in Computer Science, pages 319–333. Springer, 2008.
Cited on pages 1, 13.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. **Exact Routing in Large Road Networks Using Contraction Hierarchies**. In *Transportation Science* volume 46, pages 388–404, INFORMS, 2012.
Cited on pages 13, 37, 39.

[HJ13] Jan Hrnčíř and Michal Jakob. **Generalised Time-Dependent Graphs for Fully Multimodal Journey Planning**. In *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, pages 2138–2145. IEEE, 2013.
Cited on page 17.

[HKMS09] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. **Fast Point-to-Point Shortest Path Computations with Arc-Flags**. In *The Shortest Path Problem: 9th DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 41–72. American Mathematical Society, 2009.
Cited on page 12.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. **A Formal Basis for the Heuristic Determination of Minimum Cost Paths**. In *IEEE Transactions on Systems Science and Cybernetics* volume 4, pages 100–107, 1968.
Cited on page 12.

[HSW09] Martin Holzer, Frank Schulz, and Dorothea Wagner. **Engineering Multilevel Overlay Graphs for Shortest-Path Queries**. In *Journal of Experimental Algorithmics (JEA)* volume 13, pages 1.5:1–2.5:26, Association for Computing Machinery (ACM), 2009.
Cited on page 14.

[JP02] Sungwon Jung and Sakti Pramanik. **An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps**. In *IEEE Transactions on Knowledge and Data Engineering* volume 14, pages 1029–1046, 2002.
Cited on page 14.

[Kir13] Dominik Kirchler. **Efficient Routing on Multi-Modal Transportation Networks**. PhD thesis. Ecole Polytechnique X, 2013.
Cited on page 18.

[Kno+07] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. **Computing Many-to-Many Shortest Paths Using Highway Hierarchies**. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. SIAM, 2007.
Cited on pages 13, 39.

[Lau04] Ulrich Lauther. **An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background**. In *Geoinformation und Mobilität-von der Forschung zur praktischen Anwendung*. Volume 22 of IfGI prints, pages 219–230, 2004.
Cited on page 12.

[Mar60] Jacob Marschak. **Binary Choice Constraints on Random Utility Indicators**. In *Mathematical Methods in the Social Sciences*, Stanford University Press, 1960.
Cited on page 20.

[Mar84] Ernesto Q. V. Martins. **On a Multicriteria Shortest Path Problem**. In *European Journal of Operational Research* volume 16, pages 236–245, Elsevier, 1984.
Cited on page 113.

[McF73] Daniel McFadden. **Conditional Logit Analysis of Qualitative Choice Behavior**. In *Frontiers in Econometrics*, pages 105–142, Academic Press, 1973.
Cited on page 20.

[MKV13] Nicolai Mallig, Martin Kagerbauer, and Peter Vortisch. **mobiTopp – A Modular Agent-based Travel Demand Modelling Framework**. In *Procedia Computer Science* volume 19, pages 854–859, Elsevier, 2013.
Cited on page 49.

[Möh+06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. **Partitioning Graphs to Speedup Dijkstra's Algorithm**. In *Journal of Experimental Algorithmics (JEA)* volume 11, pages 2.8:1–2.8:29, Association for Computing Machinery (ACM), 2006.
Cited on page 12.

[Møl99] Jesper Møller-Pedersen. **Assignment Model for Timetable-Based Systems (TPSchedule)**. In *Proceedings of 27th European Transportation Forum*, pages 159–168, 1999.
Cited on page 20.

[MS07] Matthias Müller-Hannemann and Mathias Schnee. **Finding all Attractive Train Connections by Multi-Criteria Pareto Search**. In *Algorithmic Methods for Railway Optimization*. Springer, 2007, pages 246–263.
Cited on pages 15, 19, 113.

[MS10] Matthias Müller-Hannemann and Stefan Schirra (editors). **Algorithm Engineering: Bridging the Gap Between Algorithm Theory and Practice.** In. Volume 5971 of Lecture Notes in Computer Science. Springer, 2010.
Cited on page 4.

[MSWZ07] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. **Timetable Information: Models and Algorithms**. In *Algorithmic Methods for Railway Optimization*. Volume 4359 of Lecture Notes in Computer Science, pages 67–90. Springer, 2007.
Cited on page 15.

[MW01] Matthias Müller-Hannemann and Karsten Weihe. **Pareto Shortest Paths is Often Feasible in Practice**. In *Proceedings of the 5th Workshop on Algorithm Engineering (WAE'01)*. Volume 2141 of Lecture Notes in Computer Science, pages 185–198. Springer, 2001.
Cited on page 19.

[NF06] Otto A. Nielsen and Rasmus D. Frederiksen. **Optimisation of Timetable-Based, Stochastic Transit Assignment Models Based on MSA**. In *Annals of Operations Research* volume 144, pages 263–285, Springer, 2006.
Cited on page 20.

[Nic66] T. A. J. Nicholson. **Finding the Shortest Route Between Two Points in a Network**. In *The computer journal* volume 9, pages 275–280, 1966.
Cited on pages 12, 36.

[PS98] Stefano Pallottino and Maria G. Scutella. **Shortest Path Algorithms in Transportation Models: Classical and Innovative Aspects**. In *Equilibrium and Advanced Transportation Modelling*. Springer, 1998, pages 245–281.
Cited on page 15.

[PSWZ04] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. **Towards Realistic Modeling of Time-Table Information through the Time-Dependent Approach**. In *Proceedings of the 3rd Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*. Volume 92 of, pages 85–103. Elsevier, 2004.
Cited on page 15.

[PSWZ08] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. **Efficient Models for Timetable Information in Public Transportation Systems**. In *Journal of Experimental Algorithmics (JEA)* volume 12, pages 2.4:1–2.4:39, Association for Computing Machinery (ACM), 2008.
Cited on pages 16, 60.

[PV19] Duc-Minh Phan and Laurent Viennot. **Fast Public Transit Routing with Unrestricted Walking Through Hub Labeling**. In *Proceedings of the Special Event on the Analysis of Experimental Algorithms, SEA²*. Volume 11544 of Lecture Notes in Computer Science (LNCS), pages 237–247. Springer, 2019.
Cited on pages 2, 19, 72, 110, 153.

[RV03] Francesco Russo and Antonino Vitetta. **An Assignment Model With Modified Logit, Which Obviates Enumeration and Overlapping Problems**. In *Journal of Transportation* volume 30, pages 177–201, Springer, 2003.
Cited on page 20.

[San09] Peter Sanders. **Algorithm Engineering – An Attempt at a Definition**. In *Efficient Algorithms*. Volume 5760 of Lecture Notes in Computer Science (LNCS), pages 321–340. Springer, 2009.
Cited on page 4.

[Sau18] Jonas Sauer. **Faster Public Transit Routing with Unrestricted Walking**. Master's Thesis. Karlsruhe Institute of Technology, 2018.
Cited on page 72.

[She85] Yosef Sheffi. **Urban Transportation Networks**. Prentice-Hall, Englewood Cliffs, NJ, 1985.
Cited on pages 19, 20.

[SHP11] Johannes Schlaich, Udo Heidl, and Regine Pohlner. **Verkehrsmodellierung für die Region Stuttgart – Schlussbericht**. Unpublished, 2011.
Cited on pages 49, 148, 152.

[SLM07] Hayssam Sbayti, Chung-Cheng Lu, and Hani S. Mahmassani. **Efficient Implementation of Method of Successive Averages in Simulation-Based Dynamic Traffic Assignment Models for Large-Scale Network Applications**. In *Transportation Research Record* volume 2029, pages 22–30, SAGE, 2007.
Cited on page 20.

[Sto12] Sabine Storandt. **Route Planning for Bicycles - Exact Constrained Shortest Paths Made Practical via Contraction Hierarchy**. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, pages 234–242. AAAI, 2012.
Cited on page 14.

[SW14] Ben Strasser and Dorothea Wagner. **Connection Scan Accelerated**. In *Proceedings of the 16th Workshop on Algorithm Engineering and Experiments (ALENEX'14)*, pages 125–137. SIAM, 2014.
Cited on pages 17, 49, 60.

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. **Using Multi-level Graphs for Timetable Information in Railway Systems**. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*. Volume 2409 of Lecture Notes in Computer Science, pages 43–59. Springer, 2002.
Cited on page 14.

[SWZ19a] Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. **Efficient Computation of Multi-Modal Public Transit Traffic Assignments using ULTRA**. In *ArXiv e-prints* 1909.08519. Technical report, 2019.
Cited on pages 7, 9, 129.

[SWZ19b] Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. **Efficient Computation of Multi-Modal Public Transit Traffic Assignments using ULTRA**. In *Proceedings of the 27th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 524–527. Association for Computing Machinery, 2019.
Cited on pages 7, 9, 129.

[SWZ20a] Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. **Faster Multi-Modal Route Planning with Bike Sharing using ULTRA**. In *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA'20)*. Volume 160 of Leibniz International Proceedings in Informatics (LIPIcs), pages 16:1–16:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.
Cited on pages 7, 8, 113.

[SWZ20b] Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. **Integrating ULTRA and Trip-Based Routing**. In *Proceedings of the 20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'20)*. Volume 85 of OpenAccess Series in Informatics, pages 4:1–4:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.
Cited on pages 7, 8.

[SWZ20c] Ben Strasser, Dorothea Wagner, and Tim Zeitz. **Space-efficient, Fast and Exact Routing in Time-dependent Road Networks**. In *Proceedings of the 28th Annual European Symposium on Algorithms (ESA'20)*. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.
Cited on page 3.

[Tra09] Kenneth E. Train. **Discrete Choice Methods with Simulation**. Cambridge university press, 2009.
Cited on page 20.

[TW99] C. O. Tong and S. C. Wong. **A Schedule-Based Time-Dependent Trip Assignment Model for Transit Networks**. In *Journal of Advanced Transportation* volume 33, pages 371–388, 1999.
Cited on page 20.

[Wan+15] Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. **Efficient Route Planning on Public Transportation Networks: A Labelling Approach**. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*, pages 967–982. Association for Computing Machinery (ACM), 2015.
Cited on page 16.

[Wit15] Sascha Witt. **Trip-Based Public Transit Routing**. In *Proceedings of the 23th Annual European Symposium on Algorithms (ESA'15)*. Volume 9294 of Lecture Notes in Computer Science, pages 1025–1036. Springer, 2015.
Cited on pages 2, 17, 31, 44, 54, 59, 60, 61, 86, 89, 93, 96, 103, 109.

[Wit16] Sascha Witt. **Trip-Based Public Transit Routing Using Condensed Search Trees**. In *Proceedings of the 16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*. Volume 54 of OpenAccess Series in Informatics, pages 10:1–10:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016.
Cited on page 17.

[WZ17] Dorothea Wagner and Tobias Zündorf. **Public Transit Routing with Unrestricted Walking**. In *Proceedings of the 17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*. Volume 59 of OpenAccess Series in Informatics, pages 7:1–7:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017.
Cited on pages 7, 8, 49, 54, 59, 72.

List of Figures

1.1 The algorithm engineering methodology. 4

3.1 Example network and Pareto-set. 28
3.2 Representation of a Pareto-set as profile function. 29
3.3 Construction of a network that replicates minimum change times. 32

4.1 The four networks considered in this work. 48
4.2 Sizes of the transitively closed transfer graphs. 55
4.3 Effect of transfers that are not transitively closed. 57

5.1 Example for a single step of the profile algorithm. 63
5.2 Distance rank running time plot for the Switzerland network. 67
5.3 Distance rank running time plots for Germany, Stuttgart, and London. . . 68
5.4 Comparison of 'multimodal' and 'original' travel times for Switzerland. . . 70
5.5 Comparison of 'multimodal' and 'transitive' travel times for Switzerland. . 71
5.6 Comparison of 'multimodal' and 'intermediate' travel times for Switzerland. 73
5.7 Travel time comparisons for the Germany network. 74
5.8 Travel time comparisons for the Stuttgart network. 75
5.9 Travel time comparisons for the London network. 76

6.1 Missing shortcuts due to weak domination. 91
6.2 Impact of core degree and witness limit on the ULTRA preprocessing. . . . 97
6.3 Impact of the transfer speed on the ULTRA preprocessing. 100
6.4 Overview of the distribution of ULTRA shortcuts. 101

6.5 Impact of transfer speed on query times and travel times. 107
6.6 Comparison of ULTRA query times depending on the geo-rank. 110

7.1 Example for the construction of an operator-expanded network. 118
7.2 Query times depending on the number of bike sharing operators. 126

8.1 A flowchart describing a passenger's movement through the network. . . . 138
8.2 An example of a zone violating the triangle inequality. 143
8.3 Running time of CBA depending on the passenger multiplier. 150
8.4 Running time of ULTRA-CBA depending on the passenger multiplier. . . . 154
8.5 Impact of the decision model on the running time of ULTRA-CBA. 155

List of Tables

4.1 Sizes of the public transit networks considered in this work. 53
4.2 Properties of the transitively closed transfer graphs. 56

5.1 Comparison of the transfer graph sizes of the three network variants. . . . 65

6.1 Runtime of the ULTRA preprocessing. 98
6.2 ULTRA preprocessing results. 99
6.3 Overview of ULTRA-Trip-Based preprocessing results. 103
6.4 Query performance of ULTRA-CSA. 104
6.5 Query performance of ULTRA-RAPTOR. 105
6.6 Query performance of ULTRA-Trip-Based. 108

7.1 Sizes of the used public transit networks and their bike sharing systems. . 122
7.2 Preprocessing results for the operator-expanded approach. 123
7.3 Preprocessing results for the operator-expanded approach with OP. 124
7.4 Query performance of journey planning algorithms with bike sharing. . . 125

8.1 Running time of CBA depending on the maximum delay λ_{delay}. 149
8.2 Qualitative comparison of VISUM and CBS assignments. 151
8.3 Comparison of CBA and ULTRA-CBA. 152

List of Acronyms

ALT	A*, Landmarks, Triangle inequality Used on pages 12, 13, 15, 18
BFS	Breadth-First Search Used on page 44
CBA	CSA-Based Assignment Used on pages 7, 136, 142, 143, 145, 147, 149–156, 161, 181
CCH	Customizable Contraction Hierarchies Used on page 14
CH	Contraction Hierarchie Used on pages 1, 13–18, 35–40, 43, 84–88, 93–98, 106, 108, 118, 123, 124, 144–146, 151, 152
CSA	Connection Scan Algorithm Used on pages v, 6, 16–19, 24, 35–42, 49, 59, 60, 67–69, 80–89, 101–111, 130–144, 156–161
DB	Deutsche Bahn AG Used on pages iii, 49, 203
FIFO	First in, first out Used on pages 44, 96

GTFS General Transit Feed Specification
Used on page 48

HL Hub Labeling
Used on pages 1, 2, 14–16, 19, 110

IQR Interquartile range
Used on pages 70, 71, 73–76

MR-∞ multiModal RAPTOR with unlimited walking (variant of MCR)
Used on pages 43, 105–107, 109–111, 116, 123, 125, 126, 160

MCR multiModal multiCriteria RAPTOR
Used on pages 2, 3, 5, 6, 13, 18, 19, 43, 47, 61–64, 69, 77, 82, 84, 104–106, 115–123, 127, 160

MCSA Multimodal CSA
Used on pages 104–106

MEAT Minimum Expected Arrival Time
Used on pages 17, 60, 131, 132, 139, 156

MLD Multilevel Dijkstra
Used on pages 14, 16, 17

OD Operator-Dependent
Used on pages 125, 126

OE Operator-Expanded
Used on pages 125, 126

OP Operator Pruning
Used on pages 125, 126

OSM Open Street Map
Used on pages 49, 50, 52, 53, 65, 149, 151

PAT Perceived Arrival Time
Used on pages 131–142, 144, 146, 150, 154, 155

PTL Public Transit Labeling
Used on pages 15, 16, 60

RAPTOR Round-bAsed Public Transit Optimized Router
Used on pages 2, 16–19, 25, 35, 40–44, 47, 59–63, 82, 86–93, 101–111, 118, 120, 126, 127, 160

rRAPTOR Range RAPTOR (variant of RAPTOR for profile queries)
Used on pages 42, 43, 45, 59–61, 64, 81, 82, 85, 86

RUM Random Utility Model
Used on pages 20, 134

SUBITO SUBstructure In Time-dependent Optimization
Used on page 60

TB Trip-Based public transit routing
Used on pages 101, 108

TfL Transport for London
Used on page 47

TNR Transit Node Routing
Used on pages 14, 18

ULTRA Algorithm family for UnLimited TRAnsfers
Used on pages iv, v, 5–9, 80, 81, 83–93, 95–111, 113, 115–130, 143–146, 151–156, 160, 161

UCCH User-Constrained Contraction Hierarchies
Used on pages 13, 18, 19

ÖV Öffentlicher Verkehr
Used on pages 203, 204

List of Symbols

Number Sets

$\mathbb{N}$ The set of all natural numbers
Used on page 66

$\mathbb{R}$ The set of all real numbers
Used on pages 29, 30, 134

$\mathbb{R}_0^+$ The set of all non-negative (≥ 0) real numbers
Used on pages 26, 30, 54, 131, 132, 134, 144, 145

Sets

$\mathcal{BS}$ A set of bike sharing stations
Used on pages 114, 115, 117, 119

$\mathcal{C}$ The connections of a public transit network
Used on pages 24, 25, 40, 131–133, 137–139, 141, 145

$\mathcal{C}_{\text{alt}}(c)$ The connections that depart after c from the same stop
Used on page 137

$\mathcal{C}_{\text{trans}}(c)$ The set of all connections c', where transferring from c to c' is possible
Used on page 133

$\mathcal{C}_{\text{trans}}^{\text{opt}}(c)$ The Pareto-set of $\mathcal{C}_{\text{trans}}(c)$, with respect to PAT and delay robustness
Used on page 133

$\mathcal{C}_{\text{trip}}(c)$ The set of all connections following after c in the trip of c ($\mathcal{C}_{\text{trip}}(c) \subseteq \mathcal{C}$)
Used on page 132

$\mathcal{CS}$ A choice set
Used on pages 134, 135, 145, 146

$\mathcal{D}$ The travel demand for a public transit network
Used on pages 135, 145

$\mathcal{DT}$ Set of departure times collected during the rRAPTOR/ULTRA preprocessing
Used on pages 43, 83

$\mathcal{E}$ The edges of a graph ($\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$)
Used on pages 26, 27, 40, 43, 51, 53, 54, 65, 83, 115–117, 121, 139, 142, 145

$\mathcal{E}^{\text{c}}$ The edges of a core graph ($\mathcal{E}^{\text{c}} \subseteq \mathcal{V}^{\text{c}} \times \mathcal{V}^{\text{c}}$)
Used on page 27

$\mathcal{E}^{\text{e}}$ The edges of a graph with expanded bike sharing operators
Used on page 117

$\mathcal{E}^{\text{s}}$ Set of shortcut edges ($\mathcal{E}^{\text{s}} \subseteq \mathcal{S} \times \mathcal{S}$)
Used on pages 83, 87, 145

$\mathcal{E}^{\text{t}}$ Set of Trip-Based transfer edges
Used on pages 44, 94–96

$\mathcal{F}$ A set of functions
Used on pages 28, 29

$\mathcal{J}$ A set of journeys
Used on pages 28, 29, 94, 95, 145

$\mathcal{J}_s^t$ The set of all journeys from s to t
Used on pages 28, 31

$\mathcal{N}$ The neighborhood of a vertex
Used on page 145

$\mathcal{OP}$ The set of all bike sharing operators ($\mathcal{OP} \coloneqq \{0, 1, \ldots, \sigma\}$)
Used on page 117

$\mathcal{R}$ The routes of a public transit network
Used on pages 26, 30, 43, 51, 82, 83, 87, 95, 117, 119, 120, 190

$\mathcal{R}^{\text{c}}$ The routes of a public transit network that allow for bicycle transport
Used on page 119

$\mathcal{R}^{\text{e}}$ The routes of a public transit network with expanded bike sharing operators
Used on page 117

$\mathcal{S}$ The stops of a public transit network ($\mathcal{S} \subseteq \mathcal{V}$)
Used on pages 24, 26, 40, 43, 50, 51, 53–55, 82, 83, 87, 95, 117, 119, 120, 131, 145

$\mathcal{S}^{\mathrm{e}}$ The stops of a public transit network with expanded bike sharing operators
Used on page 117

$\mathcal{T}$ The trips of a public transit network
Used on pages 24–26, 40, 43, 44, 50, 51, 83, 87, 95, 114, 117, 119, 120, 131, 145

$\mathcal{T}^{\mathrm{c}}$ The trips of a public transit network that allow for bicycle transport
Used on page 119

$\mathcal{T}^{\mathrm{e}}$ The trips of a public transit network with expanded bike sharing operators
Used on page 117

$\mathcal{UC}$ A set of unwitnessed shortcut candidates
Used on page 83

$\mathcal{V}$ The vertices of the transfer graph
Used on pages 26, 27, 40, 51–54, 61, 65, 83, 114, 115, 117, 119, 120, 131, 135, 145

$\mathcal{V}_v$ The search space of vertex v
Used on pages 93, 95

$\mathcal{V}^{\mathrm{c}}$ The vertices of a core graph ($\mathcal{V}^{\mathrm{c}} \subseteq \mathcal{V}$)
Used on page 27

$\mathcal{V}_{\mathrm{dest}}$ The set of all destinations of the demand $\mathcal{D}$
Used on page 145

$\mathcal{V}_{\mathrm{orig}}$ The set of all origins of the demand $\mathcal{D}$
Used on page 145

$\mathcal{V}^{\mathrm{e}}$ The vertices of a graph with expanded bike sharing operators
Used on page 117

$\mathcal{V}^{\mathrm{t}}$ Set of stop events ($\mathcal{V}^{\mathrm{t}} := \{T[i] \mid T \in \mathcal{T}, i \leq |T|\}$), used as vertices in G^{t}
Used on pages 44, 95

Sequences, Tuples, and Random Variables

B An bounding box ($B \subseteq \mathbb{R}^2$)
Used on pages 50, 51

D_c A random variable for the delay of the connection c
Used on pages 132, 133

E A random variable for error of the deterministic utility of a travel option
Used on pages 134, 135

G A transfer graph ($G = (\mathcal{V}, \mathcal{E})$)
Used on pages 26, 27, 35, 40, 43, 50–54, 83, 87, 95, 117–120, 135, 145

G^{e} A transfer graph with expanded bike sharing operators
Used on page 117

G^{t} The graph of required transfers between trips (Trip-Based transfer graph)
Used on pages 44, 95

G^{c} A core (or overlay) graph
Used on page 27

G^{s} A shortcut graph
Used on pages 83, 87, 88, 145

H The hull of a bike sharing operator
Used on pages 119, 120

I A closed Interval ($I \in [\tau_{\text{min}}, \tau_{\text{max}}] \subseteq \mathbb{R}$)
Used on pages 30, 31, 43, 61–63

J A journey in a multimodal network
Used on pages 27–31, 56, 57, 81, 84, 86, 90–92, 114, 115, 141, 145

N A public transit network
Used on pages 24, 26, 35, 40, 117, 118, 120, 131, 135, 145

N^{c} That part of a public transit network that allows for bicycle transport
Used on page 119

N^{e} A public transit network where bike sharing operators have been expanded
Used on page 117

P A path in a graph, represented as sequence of vertices ($P = \langle v_0, \ldots, v_k \rangle$)
Used on pages 26–28, 54, 86, 141

Q A Queue (a min-heap for Dijkstra, a FIFO queue for Trip-Based)
Used on pages 94–96

R A route ($R \in \mathcal{R}$) of a public transit network
Used on pages 82, 93–95, 117, 119

T A trip in the public transit network
Used on pages 24–27, 31, 44, 50, 86, 94–96, 114, 116, 117, 119, 132, 139–141, 146

T^{ij} A subsequence of a trip T from index i to j
Used on pages 27, 28, 31, 32, 44, 86, 141

T_{min} The earliest reachable trip, used in the Trip-Based query
Used on pages 93–95

U A random variable for the utility of a travel option
Used on page 134

Basic Variables

β A tuning parameter of the Logit and Kirchhoff decision models
Used on pages 135, 155, 156

b Function that determines if a trip allows for bicycle transport
Used on pages 114, 116, 117, 119

c A fundamental connection of a public transit network
Used on pages 24, 25, 32, 131–141, 145, 146, 149

d The target vertex of a demand
Used on pages 131–142, 144–146, 149

δ_τ A minimal travel time or distance
Used on pages 26, 27, 44

e An edge in a graph ($e \in \mathcal{E}$)
Used on pages 26, 31–33, 36, 51, 54, 55, 85, 114–117, 121

ε An infinitesimal small positive number
Used on pages 25, 62, 63

ϵ A stop event (arrival and subsequent departure at a stop)
Used on pages 25, 27, 32, 44

f A function
Used on page 28

f_{arr} An arrival time profile function
Used on pages 29, 30

f_{tra} A travel time profile function
Used on page 30

$f_{\text{trans}}^{v,d}$ A profile of journeys from v to d that begin with a transfer from v
Used on pages 139, 142

f_{arr}^{+} A trip-dependent arrival time profile function
Used on pages 29, 30

f_{tra}^{+} A trip-dependent travel time profile function
Used on pages 29, 30

$f_{\text{wait}}^{v,d}$ A profile of journeys from v to d that depart directly from v
Used on pages 138, 139, 141, 142, 145, 146

g A group of passengers
Used on pages 145, 147

λ_{buf} The perceived cost of waiting for the buffer time to pass
Used on pages 144, 145, 148

λ_{delay} The maximal possible delay of a connection
Used on pages 131, 132, 148, 149, 181

$\lambda_{\Delta\text{max}}$ The maximal delay in the perceived arrival time tolerated by passengers
Used on pages 134, 135, 140, 145, 146, 148, 155

λ_{mul} The passenger multiplier used in the assignment computation
Used on pages 136, 145, 147–150, 154

λ_{trans} The perceived cost of transferring for the PAT computation
Used on pages 131, 133, 148, 149

λ_{wait} The perceived cost of waiting at a stop for the PAT computation
Used on pages 131, 133, 141, 144, 148, 149

λ_{walk} The perceived cost of walking between stops for the PAT computation
Used on pages 131–133, 139, 145, 148, 149

σ Number of bike sharing operators
Used on pages 114, 116, 117

o The origin vertex of a demand
Used on pages 135, 136, 141, 145, 146

p An origin-destination-pair
Used on pages 135, 136, 141, 145, 146

s The source vertex or the source stop
Used on pages 26–31, 43, 56, 57, 61–64, 82, 83, 87, 88, 91, 93–95, 99, 100, 114, 115, 117, 119

t The target vertex or the target stop
Used on pages 26–31, 39, 56, 57, 61–64, 87–91, 93–95, 99, 100, 114–117, 119, 121

τ A Time
Used on pages 29–31, 131–133, 138, 139, 141, 142

τ_{arr} An arrival time of a journey, trip, trip leg, stop event, or connection
Used on pages 24, 25, 27–29, 31, 32, 44, 61–63, 84, 91, 93, 95, 96, 115, 116, 131, 132, 139, 141

τ_{bike} Time of traversing an edge by bike
Used on pages 114–117, 119, 121

$\tau_{\text{bike}}^{\text{max}}$ Maximum time of a shortest path between bike sharing stations
Used on page 119

τ_{buf} The departure buffer time of a stop
Used on pages 32, 51, 145

τ_{ch} The minimum change time of a stop
Used on pages 31–33, 133

τ_{dep} A departure time of a query, trip, trip leg, stop event, or connection
Used on pages 24, 25, 27–32, 43, 44, 62, 63, 82, 83, 87, 88, 91, 93, 95, 96, 115, 131–141, 145, 146

τ_{drop} Time required to drop off a bike at a bike sharing station
Used on pages 114, 115, 117

τ_{max} The maximal possible departure time or the maximum time of an interval
Used on pages 43, 61–63

τ_{min} The minimal possible arrival time or the minimum time of an interval
Used on pages 43, 61–63, 93–95

τ^{P} The time as it is perceived by a passenger
Used on pages 131–133

$\tau^{\text{P}}_{\text{arr}}$ The perceived arrival time at the destination
Used on pages 132–135, 137–141, 145, 149

$\tau^{\text{P}}_{\text{tra}}$ The perceived duration of a transfer
Used on pages 133, 139, 141

$\tau^{\text{P}}_{\text{wait}}$ The perceived time for waiting at a stop
Used on pages 133, 137–139

$\tau^{\text{P}}_{\text{walk}}$ The perceived time for walking along an edge or path
Used on pages 145, 146

τ_{pick} Time required to pick up a bike at a bike sharing station
Used on pages 114, 115, 117

τ_{tra} Travel time (or transfer time) of an edge, path, or journey
Used on pages 26–29, 32, 33, 37, 39, 82, 83, 87, 88, 93–95, 131–133, 139, 141, 142, 145, 146

τ_{mul} Travel time of a multimodal journey
Used on pages 70, 71, 73–76

τ_{ref} Reference travel time of a journey with limited walking
Used on pages 70, 71, 73–76

τ_{wait} The time spent by waiting for the departure of a connection
Used on pages 131–133, 139

τ_{walk} Walking time of an edge or path (τ_{tra} restricted to walking)
Used on pages 54, 56, 114–117, 121

$\bar{\tau}_{\text{walk}}$ Guaranteed walking time of a transitive graph
Used on page 54

$\bar{\tau}_{\text{wit}}$ Witness limit for the ULTRA preprocessing
Used on pages 84, 97

u The utility of a travel option
Used on pages 134, 135

μ The utilization of a connection
Used on pages 136, 145

v A vertex ($v \in \mathcal{V}$), a stop (since $\mathcal{S} \subseteq \mathcal{V}$), or the stop of a stop event ($v(\epsilon) \in \mathcal{S}$)
Used on pages 25–28, 31–39, 41, 42, 44, 51–57, 64, 66, 82, 85–96, 114–119, 121, 131, 133, 136–146

v_{arr} An arrival stop of a trip, trip leg, or connection
Used on pages 24, 131–133, 139–141, 145, 146

v_{dep} A departure stop of a trip, trip leg, or connection
Used on pages 24, 25, 32, 131, 133, 137–140, 145, 146

w A vertex ($w \in \mathcal{V}$)
Used on pages 26–28, 31, 36, 37, 42, 51, 52, 54–57, 85, 88, 90–92, 114, 115, 118, 133, 142, 143

x A vertex ($x \in \mathcal{V}$)
Used on pages 26, 28, 32, 37, 38, 51, 56, 57, 91, 92

y A vertex ($y \in \mathcal{V}$)
Used on pages 28, 32, 57, 91, 92

z A vertex ($z \in \mathcal{V}$)
Used on pages 28, 32, 33

A Curriculum Vitæ

Personal Data

Name	Tobias Zündorf
Place of Birth	Paderborn, Germany
Nationality	German

Education

01/2015 – 12/2020	PhD student at the Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT) Advisors: Prof. Dr. Dorothea Wagner, Prof. Dr. Matthias Müller-Hannemann
11/2012 – 11/2014	Master of Science in Computer Science Karlsruhe Institute of Technology (KIT)
09/2009 – 10/2012	Bachelor of Science in Computer Science Karlsruhe Institute of Technology (KIT)
06/2009	Abitur (final secondary school examinations) Reismann-Gymnasium Paderborn

Professional Experience

01/2015 – 12/2020	Research Assistant at the group of Prof. Dr. Dorothea Wagner, Department of Informatics, Karlsruhe Institute of Technology
05/2016 – 07/2016	Internship at Apple in the maps department In Sunnyvale, California, USA
04/2013 – 03/2014	Student Assistant at the group of Prof. Dr. Dorothea Wagner, Department of Informatics, Karlsruhe Institute of Technology
10/2010 – 03/2013	Tutor at the Department of Informatics, Karlsruhe Institute of Technology

Awards and Scholarships

2016	Award for the best master thesis from the VKSI (Association of software engineers in Karlsruhe)
2015	Award for the best master thesis from the city of Karlsruhe
10/2011 – 09/2013	Deutschlandstipendium (Scholarship from the country of Germany)

Teaching Experience

04/2020 – 07/2020	Lecture *Algorithms for Route Planning*
10/2019 – 03/2020	Practical course *Algorithm Engineering*
04/2019 – 07/2019	Lecture *Algorithms for Route Planning*
10/2018 – 03/2019	Practical course *Algorithm Engineering*
04/2018 – 07/2018	Lecture *Algorithms for Route Planning*
10/2017 – 03/2018	Practical course *Algorithm Engineering*
04/2017 – 07/2017	Lecture *Algorithms for Route Planning*
10/2016 – 03/2017	Practical course *Algorithm Engineering*
04/2016 – 07/2016	Lecture *Algorithms for Route Planning*

Supervised Students

12/2018 – 06/2019 Sebastian Knapp (Master's Thesis)
Efficient Planning of "Nice" Round Trips

11/2018 – 04/2019 Moritz Halm (Bachelor's Thesis)
Algorithms for the Pagination Problem on Public Transit Networks

07/2018 – 01/2019 Oliver Plate (Master's Thesis)
Ridesharing with Multiple Riders

08/2018 – 11/2018 Hamidulah Doust (Bachelor's Thesis)
PoI-Query-Algorithms Compared

07/2018 – 10/2018 Peter Maucher (Bachelor's Thesis)
Dijkstra-Based map Matching

06/2018 – 10/2018 Robin M. Berger (Bachelor's Thesis)
An Efficient Traffic Assignment Algorithm for Public Transit with Vehicle Capacities

11/2017 – 04/2018 Jonas Sauer (Master's Thesis)
Faster Public Transit Routing with Unrestricted Walking

06/2017 – 11/2017 Patrick Niklaus (Master's Thesis)
A Unified Framework for Electric Vehicle Routing

07/2017 – 10/2017 Florian Grötschla (Bachelor's Thesis)
On the Complexity of Public Transit Profile Queries

05/2017 – 10/2017 Huyen Chau Nguyen (Master's Thesis)
Engineering Multi-Modal Transit Route Planning

05/2016 – 09/2016 Holger Ebhart (Bachelor's Thesis)
Traffic Assignment for Public Transportation Networks using Perceived Arrival Times

06/2015 – 10/2015 Jonas Sauer (Bachelor's Thesis)
Energy-Optimal Routes for Electric Vehicles with Charging Stops

03/2015 – 09/2015 David Weiß (Master's Thesis)
Efficient Enumeration of All Reasonable Journeys in Public Transport Networks

01/2015 – 04/2015 Simeon Andreev (Master's Thesis)
Consumption and Travel Time Profiles in Electric Vehicle Routing

B List of Publications

Journal Articles

[1] **Integrating public transport into mobiTopp**. In *Future Generation Computer Systems* volume 107, pages 1089–1096, 2020. Joint work with Lars Briem, H. Sebastian Buck, Nicolai Mallig, Peter Vortisch, Ben Strasser, and Dorothea Wagner.

[2] **Modeling and Engineering Constrained Shortest Path Algorithms for Battery Electric Vehicles**. In *Transportation Science* volume 54, pages 1571–1600, 2020. Joint work with Moritz Baum, Julian Dibbelt, and Dorothea Wagner.

[3] **Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles**. In *Transportation Science* volume 53, pages 1627–1655, 2019. Joint work with Moritz Baum, Julian Dibbelt, Andreas Gemsa, and Dorothea Wagner.

[4] **Energy-Optimal Routes for Battery Electric Vehicles**. In *Algorithmica* volume 82, pages 1490–1546, 2019. Joint work with Moritz Baum, Julian Dibbelt, Thomas Pajor, Jonas Sauer, and Dorothea Wagner.

Articles in Conference Proceedings

[5] **Integrating ULTRA and Trip-Based Routing**. In *Proceedings of the 20th Symposium on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'20)*. Volume 85 of OpenAccess Series in Informatics, pages 4:1–4:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. Joint work with Jonas Sauer and Dorothea Wagner.

[6] **An Efficient Solution for One-to-Many Multi-Modal Journey Planning**. In *Proceedings of the 20th Symposium on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'20)*. Volume 85 of OpenAccess Series in Informatics, pages 1:1–1:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. Joint work with Jonas Sauer and Dorothea Wagner.

[7] **Faster Multi-Modal Route Planning with Bike Sharing Using ULTRA**. In *Proceedings of the 18th International Symposium on Experimental Algorithms (SEA'20)*. Volume 160 of Leibniz International Proceedings in Informatics, pages 16:1–16:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. Joint work with Jonas Sauer and Dorothea Wagner.

[8] **Efficient Computation of Multi-Modal Public Transit Traffic Assignments using ULTRA**. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'19)*, pages 524–527. Association for Computing Machinery, 2019. Joint work with Jonas Sauer and Dorothea Wagner.

[9] **UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution**. In *Proceedings of the 27th Annual European Symposium on Algorithms (ESA'19)*. Volume 144, Leibniz International Proceedings in Informatics, pages 14:1–14:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. Joint work with Moritz Baum, Valentin Buchhold, Jonas Sauer, and Dorothea Wagner.

[10] **Integrating public transport into mobiTopp**. In *Proceedings of the 6th International Workshop on Agent-based Mobility, Traffic and Transportation Models, Methodologies and Applications (ABMTRANS'17)*, pages 855–860. Elsevier, 2017. Joint work with Lars Briem, H. Sebastian Buck, Nicolai Mallig, Peter Vortisch, Ben Strasser, and Dorothea Wagner.

[11] **Modeling and Engineering Constrained Shortest Path Algorithms for Battery Electric Vehicles**. In *Proceedings of the 25th Annual European Symposium on Algorithms (ESA'17)*. Volume 87 of Leibniz International Proceedings in Informatics, pages 11:1–11:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. Joint work with Moritz Baum, Julian Dibbelt, and Dorothea Wagner.

[12] **Efficient Traffic Assignment for Public Transit Networks**. In *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17)*. Volume 75 of Leibniz International Proceedings in Informatics, pages 20:1–20:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. Joint work with Lars Briem, H. Sebastian Buck, Holger Ebhart, Nicolai Mallig, Ben Strasser, Peter Vortisch, and Dorothea Wagner.

[13] **Consumption Profiles in Route Planning for Electric Vehicles: Theory and Applications**. In *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17)*. Volume 75 of Leibniz International Proceedings in Informatics, pages 19:1–19:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. Joint work with Moritz Baum, Jonas Sauer, and Dorothea Wagner.

[14] **Faster Transit Routing by Hyper Partitioning**. In *Proceedings of the 17th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'17)*. Volume 59 of OpenAccess Series in Informatics, pages 8:1–8:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. Joint work with Daniel Delling, Julian Dibbelt, and Thomas Pajor.

[15] **Public Transit Routing with Unrestricted Walking**. In *Proceedings of the 17th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'17)*. Volume 59 of OpenAccess Series in Informatics, pages 7:1–7:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. Joint work with Dorothea Wagner.

[16] **Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles**. In *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL'15)*, pages 44:1–44:10. Association for Computing Machinery, 2015. Joint work with Moritz Baum, Julian Dibbelt, Andreas Gemsa, and Dorothea Wagner.

[17] **Efficient Computation of Jogging Routes**. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Volume 7933 of Lecture Notes in Computer Science, pages 272–283. Springer, 2013. Joint work with Andreas Gemsa, Thomas Pajor, and Dorothea Wagner.

Technical Reports

[18] **Efficient Computation of Multi-Modal Public Transit Traffic Assignments using ULTRA**. In *ArXiv e-prints* 1909.08519. Technical report, 2019. URL: `http://arxiv.org/abs/1909.08519`. Joint work with Jonas Sauer and Dorothea Wagner.

[19] **UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution**. In *ArXiv e-prints* 1906.04832, Technical report, 2019. URL: `http://arxiv.org/abs/1906.04832`. Joint work with Moritz Baum, Valentin Buchhold, Jonas Sauer, and Dorothea Wagner.

[20] **Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles**. In *ArXiv e-prints* 1910.09812, Technical report, 2019. URL: `http://arxiv.org/abs/1910.09812`. Joint work with Moritz Baum, Julian Dibbelt, Andreas Gemsa, and Dorothea Wagner.

C Deutsche Zusammenfassung

Fahrplanauskunftssysteme (wie der DB Navigator) und Routenplanungssysteme für den Straßenverkehr (wie zum Beispiel Google Maps) sind aus unserem Alltag nicht mehr wegzudenken. Die weite Verbreitung derartiger Routenplanungsanwendungen liegt dabei nicht zuletzt an den algorithmischen Entwicklungen der letzten Jahrzehnte. Für Straßennetze lässt sich zum Beispiel eine kürzeste Strecke quer durch Europa in etwa einer Millisekunde berechnen. Und auch in der Fahrplanauskunft können Verbindungen innerhalb Deutschlands in weniger als 50 Millisekunden gefunden werden. Betrachtet man aber ein kombiniertes (*multimodales*) Netzwerk, in dem das Fortbewegungsmittel beliebig gewechselt werden kann, so steigt die zur Berechnung einer optimalen Reiseverbindung benötigte Zeit deutlich an.

In meiner Arbeit betrachte ich verschiedene Variationen des Kürzeste-Wege-Problems in multimodalen Verkehrsnetzwerken. Dabei betrachte ich das Problem nicht nur aus der Sicht eines Fahrgastes, sondern auch aus Sicht der Verkehrsunternehmen. Für einen Fahrgast sind vor allem Algorithmen relevant, welche die beste Verbindung zwischen zwei gegebenen Orten im Netzwerk berechnen. Verkehrsunternehmen interessieren sich im Gegensatz dazu oft für sogenannte *Traffic Assignments*, welche eine Gesamtsicht auf das Netzwerk geben. Dabei ist eine Liste mit der Nachfrage für das gesamte Netzwerk gegeben und der daraus resultierende Passagierfluss gesucht, um zum Beispiel die Auslastung einzelner Züge bestimmen zu können.

Im ersten Teil meiner Arbeit beschäftige ich mich mit multimodalen Routenplanungsalgorithmen für einzelne Paare von Start- und Zielort. Dabei zeigt sich, dass schon die Verbindung von nur zwei Verkehrsmodi (öffentlicher Verkehr (ÖV) und zu Fuß gehen) zu Anfragezeiten im Sekundenbereich führt. Darüber hinaus stellt

sich die Frage, in welchem Umfang zu Fuß laufen überhaupt relevant für optimale Reiseverbindungen ist. Um dieser Frage nachzugehen, entwickle ich einen ersten Algorithmus für multimodale *Profil-Suchen*, bei denen keine konkrete Reisezeit gegeben ist, sondern alle optimalen Verbindungen in einem Zeitintervall gesucht werden. Die Grundidee des Algorithmus basiert darauf, das Zeitintervall, für das noch nicht alle optimalen Verbindungen bekannt sind, sukzessive zu verkleinern, bis das ganze Profil berechnet wurde. Auf diese Weise wird es möglich, Profile, welche einen ganzen Tag umfassen, in wenigen Sekunden zu berechnen.

Die Auswertung einiger Hundert solcher Profile (für zufällige Paare von Start- und Zielort) erlaubt es, die Struktur der gefundenen Lösungen und die Relevanz von langen Laufwegen genauer zu untersuchen. Dabei zeigt sich, dass langes Laufen zwischen zwei Fahrten mit öffentlichen Verkehrsmitteln eher selten vonnöten ist, wohingegen das Erreichen der ersten Haltestelle oder der Weg von der letzten Haltestelle zum Zielort durchaus längere Laufstrecken erfordern kann. Diese Erkenntnisse nutze ich, um eine Beschleunigungstechnik für multimodale Routenplanungsalgorithmen zu entwickeln. Hierbei ist die Idee, alle Laufwege vorzuberechnen, die zwischen Fahrten mit dem ÖV liegen, so dass nur noch der Weg zur ersten Haltestelle und der Weg von der letzten Haltestelle zur Anfragezeit gesucht werden muss. Im Resultat führt dies zu dem bisher schnellsten bekannten Algorithmus für multimodale Routenplanung.

Darüber hinaus zeige ich, dass dieser Ansatz nicht nur für Laufen, sondern auch für andere Verkehrsmodi, wie Fahrradfahren oder die Nutzung von Taxis in Kombination mit dem öffentlichen Verkehr funktioniert. Außerdem zeige ich, wie sich das Verfahren auf Szenarien mit mehr als zwei Verkehrsmodi (zum Beispiel ÖV + Laufen + Bike-Sharing) ausweiten lässt.

Im zweiten Teil der Arbeit widme ich mich dem Umlegungsproblem. Hierbei geht es nicht mehr darum, eine einzelne Anfrage zu beantworten, sondern für Millionen von Start-Ziel Paaren das Verhalten der Passagiere zu prognostizieren. Da Ergebnisse zu Beschleunigungstechniken aus der Routenplanung bisher nicht auf dieses Problem übertragen wurden, betrachte ich es zunächst unimodal. Im Detail zeige ich, wie der Connection Scan Algorithmus, welcher für Einzelanfragen in reinen ÖV Netzwerken entwickelt wurde, für das Umlegungsproblem angepasst werden kann. Durch geschicktes Gruppieren der Daten entsteht so ein neuer Algorithmus, der in unter einer Minute eine Umlegung berechnet, die zuvor etwa eine halbe Stunde benötigte.

Im Anschluss untersuche ich, inwieweit sich die Ansätze und Ergebnisse aus dem ersten Teil der Arbeit auf das Umlegungsproblem übertragen lassen. Hier zeigt sich, dass der Grundgedanke der vorberechneten Transfers zwischen Fahrten mit dem ÖV auch für Umlegungen anwendbar ist. Aufbauend auf dieser Idee zeige ich, welche weiteren Änderungen am Algorithmus nötig sind, um die beiden Techniken kombinieren zu können. Als Resultat ergibt sich ein erster effizienter Algorithmus für die Umlegungsberechnung in multimodalen Verkehrsnetzwerken.