Closing the Performance Gap Between Multimodal and Public Transit Journey Planning

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Jonas Sebastian Sauer

Tag der mündlichen Prüfung:	31. Mai 2024
Erste Referentin:	Prof. Dr. Dr. h.c. Dorothea Wagner
Zweiter Referent:	Prof. Dr. Matthias Müller-Hannemann

Acknowledgements

I am deeply grateful to Dorothea Wagner for opening up many opportunities for me, for giving me great freedom in choosing my topic, and for offering advice whenever I needed it. When I first came in contact with Dorothea's group, the resident "route planners" were Moritz Baum, Valentin Buchhold, Ben Strasser and Tobias Zündorf. They nurtured my interest in this area of research, and I owe much of my knowledge about scientific programming and methodology to them. Over the years, I also enjoyed many fruitful discussions with my other fellow route planners: Adrian Feilhauer, Tim Zeitz and Michael Zündorf. I would also like to thank Sascha Witt for his advice on implementing the Trip-Based Routing algorithm.

Successful research does not happen in a vacuum, and I was very fortunate to have had regular exchanges with people who broadened my perspective. In particular, I thank Lukas Barth for sharing his experience and insights from an application point of view. I am also indebted to my colleagues from the DFG FOR2083 research group, which provided much of the funding for my research. Getting me to understand the difference between the terms "intermodal" and "multimodal" may have been a hopeless endeavor, as the title of this thesis shows, but they taught me countless valuable things about public transportation systems.

This thesis could not have been written without the contributions of my co-authors. Besides those already mentioned, these include my students Dominik Bez and Moritz Potthoff, who went above and beyond what was expected of them. I also extend my gratitude to my collaborators on projects that did not make it into this thesis and those who otherwise offered their input, including Lars Gottesbüren, Moritz Laupichler and Claudius Proissl. In particular, I thank Patrick Steil for his infectious enthusiasm for all things related to route planning, and for his thoughts on custom(e)ization.

I owe my gratitude to Torsten Ueckerdt for leading Dorothea's group in her absence, and our administrative and technical staff members Lilian Beckert, Isabelle Junge, Ralf Kölmel and Tanja Wehrmann, who kept things running from day to day. I am grateful to Guido Brückner, Vera Chekan, Laura Merker, Torsten Ueckerdt and Wendy Yi, who joined me in the sometimes arduous but rewarding task of teaching theoretical computer science to undergraduate students. I also thank all my co-workers, particularly my office mates Guido Brückner and Tim Zeitz, for creating such a fun work environment. I look back fondly on all the coffee breaks, board game nights, table football matches and bouldering sessions.

In finishing this thesis, I am grateful to Matthias Müller-Hannemann for agreeing to referee it (I did my best to keep it concise), Patrick Steil and Paul Jungeblut for proofreading, and Michael Zündorf for offering his expertise on layout and typographical issues. Finally, I thank my friends and family for their continual love and support.

Abstract

This thesis studies the design of journey planning algorithms for multimodal passenger transport networks. In particular, we consider the combination of public transit (e.g., trains, buses, trams) with one or several *transfer modes* that represent road-based individual transport (e.g., walking, cycling, e-scooters). Currently, there is a significant performance gap between multimodal journey planning algorithms and their unimodal counterparts. One major reason for this is that state-of-the-art multimodal algorithms combine existing techniques for exploring the individual network parts, but the fastest available techniques for road networks are not usable within this context. The second major reason is that multimodal journey planning requires the simultaneous optimization of multiple criteria. However, existing approaches can only efficiently handle Pareto optimization for two criteria: the arrival time and the number of used trips. With additional criteria, the number of Pareto-optimal solutions becomes excessively large, which slows down the algorithms and causes an overwhelming amount of different choices to be presented to the user.

This thesis employs the Algorithm Engineering methodology to develop techniques that close this performance gap. The first major contribution is ULTRA (UnLimited TRAnsfers), a speedup technique that allows any public transit algorithm to operate on multimodal networks without incurring a performance loss. It is based on the *shortcut hypothesis*, which states that the number of paths in the transfer graph that are required to bridge the gap between two public transit vehicles in at least one optimal journey is small. ULTRA exploits this by precomputing these paths and condensing them into a set of *shortcut edges*. We first present ULTRA for a basic scenario with one transfer mode and two optimization criteria: arrival time and number of trips. Afterward, we extend it to a variety of extended scenarios to show that the shortcut hypothesis still holds in them. The extensions include queries with multiple target locations, additional criteria, multiple transfer modes, and vehicle delays.

The second focus of this thesis is on designing new, efficient query algorithms for multimodal scenarios. Currently, the fastest public transit algorithm that does not require expensive preprocessing is Trip-Based Routing (TB), which achieves its good performance by operating on the level of individual *events* (i.e., departures and arrivals of public transit vehicles at stations) rather than vehicle routes and stations. Already in a public transit context, TB requires a short preprocessing step that computes relevant transfers between vehicles. We show that ULTRA can replace this preprocessing step in a multimodal context. Starting from there, we extend the event-based concept of TB to scenarios with additional criteria. We show that under certain conditions, Pareto-optimizing a third criterion in addition to the arrival time and the number of trips can be done in polynomial time. In particular, we

demonstrate that considering the time spent in the transfer modes as the third criterion is crucial in order to achieve a good solution quality. Our approach yields the new query algorithms McTB (Multicriteria Trip-Based Routing) and HydRA (Hybrid Routing Algorithm), which can efficiently handle scenarios with three or more criteria, respectively. Furthermore, we integrate our algorithms with *restricted Pareto sets*, a state-of-the-art approach for reducing the size of the Pareto sets in a methodical manner.

The combination of ULTRA with the event-based query algorithms and restricted Pareto sets closes the performance gap in all considered scenarios, which we show through extensive experiments on real-world networks representing metropolitan areas and countries of varying sizes. On all networks, our algorithms are fast enough for interactive applications. Depending on the scenario, they outperform the state of the art by one to three orders of magnitude.

Contents

1 Introduction

One of the greatest challenges facing urban development today is car dependency. Private cars offer fast and convenient travel for the individual passenger, but they also produce many negative externalities. These include noise and air pollution, a high energy consumption per capita, CO_2 emissions, congestion, a demand for parking space that cannot be used otherwise, and an increased risk of traffic accidents [WBN17, Tig+11, Mar07]. For these reasons, there is a broad consensus that a shift toward more sustainable modes of transportation, such as public transit, walking or cycling, is needed [vB04, Tig+11, SGv14, WBN17]. For example, the Greater London Authority is aiming for 80% of all journeys to be traveled on foot, by bicycle or with public transport by 2041, up from 63% in 2015 [Gre18].

These ambitious goals cannot be achieved by treating the different modes as separate entities. This is because, on their own, none of them can compete with the speed and flexibility of car-based travel. Sustainable individual transport modes, such as walking and cycling, are too slow and cumbersome for long journeys. Public transit has a greater spatial reach, but it is not as flexible. Outside of densely populated areas and peak hours, service tends to be sporadic. Because of the resulting waiting times and the time overhead for access and egress, public transit on its own cannot replace a private car for many passengers.

These shortcomings can be addressed by integrating the different modes into a multimodal transport system. This has the potential of combining their strengths while mitigating their weaknesses [SGv14, KBB16, OCC20]. Individual transport modes can help to bridge gaps in public transit service and vastly improve its door-to-door accessibility [vB04, Mar07, KBB16, OCC20]. In addition, recent years have seen the rapid adoption of micromobility services, such as bicycle sharing or e-scooters, which give passengers more flexibility and a greater range of options to choose from [ALD21, OCC20]. Together with more traditional modes, these services can help turn multimodal travel into a serious alternative for all passengers.

While the greater range of options is beneficial, it also means that multimodal journeys have a higher planning overhead for the passenger. Therefore, the shift toward multimodality requires integrated journey planning services that offer access to the different modes within a single user interface [WBN17]. This thesis focuses on the design of multimodal journey planning algorithms for such services. We consider a transport network to be multimodal if it combines public transit with at least one individual transport mode. In particular, we treat the different types of schedule-based public transit (e.g., train, tram, bus) as the same mode. From an algorithmic view, there is no distinction between them because they all serve a fixed sequence of locations according to a fixed timetable. By contrast, individual transport modes allow the passenger to move freely within the road network. We assume that the movement in these modes is largely unrestricted; if a system only accounts for walking between nearby public transit stops, we do not consider it to be truly multimodal. Some modes, such as ridesharing and dial-a-ride services, are hybrids that combine aspects of both schedule-based and individual transport. We consider such modes to lie outside the scope of this thesis.

Some authors (e.g., [WBN17, OCC20]) make a distinction between multimodal and intermodal travel. In this case, "multimodal" merely means that passengers have a choice between multiple transport modes. If the modes can be combined within a single journey, the system is called intermodal. When designing integrated journey planning algorithms, the intermodal case is the default one, so the distinction is unnecessary. Therefore, we follow most existing literature on journey planning algorithms [Bas+16] and use the term "multimodal" exclusively.

1.1 State of the Art

Most journey planning algorithms are designed for the use case of a server application, which responds to queries by many different users on the same network. The application should be interactive, so response times must be sufficiently fast. Ideally, this means that the time for answering a query is lower than the network latency, so that users do not notice a delay. Because Dijkstra's algorithm [Dij59] is too slow for this purpose, state-of-the-art approaches employ *speedup techniques* [Bas+16]. These compute auxiliary data in a *preprocessing phase*, which is then used to speed up the *query phase*. This is possible because the network topology changes only rarely (e.g., once per day).

Unimodal Journey Planning. Road networks exhibit various structural properties that have proven useful in the design of speedup techniques. Most notably, they can be efficiently decomposed into smaller components of roughly equal size [EG08, DSW16] and shortest paths between far-away locations tend to pass through a small subset of the network (e.g., highways) [BFSS07, Abr+16]. Likely the most widely used technique is Contraction Hierarchies (CH) [GSSV12]. Even on the road network of Europe, CH can answer queries in well below a millisecond with a few minutes of preprocessing time and little additional space. Unfortunately, public transit networks do not exhibit these properties to the same degree [Bas09]. As a result, speedup techniques that offer comparable query times to those on

road networks require very expensive preprocessing phases, in terms of both running time and memory consumption [Bas+10, DDPW15].

More recently, research has focused on exploiting a different set of structural features that are unique to public transit networks, which has resulted in faster query algorithms that require little or no preprocessing. Because the timetable can be represented as a directed acyclic graph (DAG), finding shortest paths does not require Dijkstra's algorithm. Instead, the fastest algorithms explore the graph in topological order [DPSW18] or use variants of breadth-first search (BFS) [DPW15a, Wit15]. Another useful feature is that vehicle routes form long uninterrupted paths. State-of-the-art algorithms exploit modern hardware architecture by exploring these paths with array-based scanning operations. These achieve a high degree of memory locality, leading to faster query times [DPW15a, Wit15].

Another aspect that is especially relevant in public transit journey planning is multicriteria optimization. In addition to the arrival time, many algorithms also consider the number of trips (i.e., the number of vehicles used) [PSWZ08, Bas+10, DPW15a, Wit15]. This serves as a measure for the discomfort associated with a journey: because changing vehicles is cumbersome, many passengers will accept a slightly later arrival time if it allows them to save a trip. To reflect this, these algorithms compute the *Pareto set*, which includes all solutions that are not dominated by another solution in both criteria.

The fastest public transit algorithms that do not require preprocessing are the Connection Scan Algorithm (CSA) [DPSW18], which optimizes only the arrival time, and RAPTOR (Round-bAsed Public Transit Optimized Router) [DPW15a], which supports two-criteria optimization. Trip-Based Routing (TB) [Wit15] is even faster but requires a short preprocessing phase that computes transfers between vehicles. These algorithms achieve query times of a few milliseconds on metropolitan networks and 100–300 ms on the network of Germany.

Methodology. The established methodology for journey planning research is Algorithm Engineering [San09, MS10]. This method addresses some of the shortcomings of the classical approach to algorithm design, which is based on theoretical analysis: given (simplified) input and machine models, performance guarantees are proven that hold for all possible inputs. Theoretical analysis yields results that are more robust than merely evaluating an algorithm's performance for some benchmark instances. However, if the models are too heavily simplified, the guarantees may not be tight enough to accurately predict the performance on real inputs. On the other hand, more sophisticated models make the analysis more challenging.

Journey planning is a classic example of an application in which simple models often fail to provide useful results. Most successful journey planning algorithms are tailored toward the structural properties of real-world transportation networks, so analyzing the worst-case performance across all possible input graphs rarely offers interesting insights. For road networks, some progress has been made toward formalizing these structural features and incorporating them into the theoretical analysis [Abr+16, KV17, BFS21], but public transit or multimodal networks are not as well understood. Another issue is that many algorithms exploit features of modern hardware architectures that are not captured by the random-access

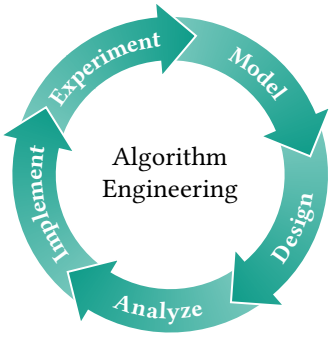


Figure 1.1: The Algorithm Engineering cycle as introduced by Sanders [San09] and Müller-Hannemann and Schirra [MS10]. Figure kindly provided by Tobias Zündorf [Zün20].

machine (RAM) model, which is still commonly used for performance analysis. Factors such as cache locality and branch prediction greatly influence the practical performance of public transit algorithms such as CSA, RAPTOR and TB, but also of algorithms for road networks [DGNW13]. Therefore, analyzing the worst-case running time in the RAM model is not sufficient to determine whether an algorithm meets the goal of offering "interactive" query times on real-world networks.

Algorithm Engineering acknowledges that theoretical analysis is not a substitute for practical implementation and experimental evaluation on real inputs. Instead, the two approaches should complement each other. The central process of Algorithm Engineering (visualized in Figure 1.1) is a cycle consisting of five steps: modeling, design, analysis, implementation, and experiments. The modeling step includes tasks such as choosing representative benchmark inputs and formulating a suitable problem statement. An algorithm for the problem is then designed and analyzed from a theoretical perspective. Afterward, the algorithm is implemented, and finally, experiments are performed on the chosen inputs to evaluate the algorithm's practical performance. A crucial aspect of the Algorithm Engineering cycle is the feedback loop: later phases can influence earlier ones. For example, analysis and experiments can offer insights that lead to new algorithmic ideas or more refined models, prompting a new iteration of the cycle. Each iteration is driven by a falsifiable hypothesis, and the experiments are designed to falsify or support it.

Public Transit with Limited Walking. None of the state-of-the-art algorithms for public transit networks can be easily adapted to multimodal scenarios. At best, they can handle a limited amount of walking. The typical model is that of *one-hop transfers*: the footpaths are modeled as a graph in which every shortest path is represented by a single edge. An unrestricted footpath network cannot be modeled efficiently in this way because the resulting graph would be too dense. Instead, one-hop transfer graphs typically consist of a set of small, disjoint cliques representing nearby stops.

To evaluate how far this model can be stretched, Wagner and Zündorf [WZ17] construct one-hop transfer graphs that are guaranteed to preserve all paths whose walking duration

does not exceed a specified limit τ. They achieve this by connecting all pairs of stops whose walking distance is at most τ and then building the transitive closure of the resulting graph. Their experiments show that if τ is higher than a few minutes, the graph becomes so dense that queries are massively slowed down. For even faster transport modes, such as bicycles or e-scooters, the guaranteed travel time would be so low that the modes become virtually useless. For example, on the network of Greater London, around four minutes of walking can be guaranteed. If we assume that a bicycle is about three times as fast as walking, this corresponds to only 80 seconds of cycling.

On the other hand, Wagner and Zündorf also show that the availability of unrestricted walking significantly improves travel times compared to one-hop transfers with a guaranteed walking duration. To measure the impact, they evaluate random queries between points with a fixed distance, which corresponds to an average travel time of two hours. For each query, they compare the travel times of the fastest journey with unrestricted and with one-hop transfers. On the network of Germany (with a guaranteed walking duration of eight minutes), the travel times differ for almost half of all daytime queries, and even more frequently during the night. In 10–20% of all cases, the travel time difference exceeds one hour. The authors note that if arrival time and number of trips are Pareto-optimized, the effect is even stronger because journeys with fewer transfers require more walking. Phan and Viennot [PV19] perform similar experiments that support the same conclusion, although they note that the impact of unrestricted walking is smaller in dense metropolitan networks. Note that both experiments consider walking as the only additional mode besides public transit. We expect the impact of faster modes to be much greater.

Multimodal Performance Gap. Algorithms that can handle fully multimodal scenarios typically combine existing approaches for each network type. Unfortunately, most speedup techniques for road networks are not applicable in this context because they are designed for one-to-one queries, i.e., queries between a fixed source and target location. However, multimodal journeys may use the road-based modes to transfer between any two points in the public transit network, which requires many-to-many queries. Currently, the fastest multimodal algorithm is MCR (Multimodal Multicriteria RAPTOR) [Del+13], which combines RAPTOR on the public transit network with Dijkstra's algorithm on a contracted version of the road network. For two-criteria optimization, MCR is slower than RAPTOR on its own by a factor of two to three. TB, which is faster than RAPTOR, has not been adapted to multimodal networks so far because its preprocessing phase requires one-hop transfers.

Another challenge is that multicriteria optimization plays an even greater role in multimodal journey planning because the additional transport modes produce more potential tradeoffs. For example, users may want to minimize the walking duration or the costs for rental vehicles. RAPTOR can be extended to an arbitrary number of criteria. However, when adding a third criterion, the running times already increase by an order of magnitude [DPW15a]. For TB, no such extension has been proposed yet. One of the reasons for the slowdown is that Pareto sets tend to become very large for a high number of criteria [Han80, Del+13]. Besides causing

performance issues, the abundance of different options can also be overwhelming for users of a journey planning application. Many Pareto-optimal solutions are extremely similar to each other (e.g., adding a minute of walking to save a minute of travel time), whereas others represent undesirable tradeoffs (e.g., adding three hours of walking to save a minute of travel time). Therefore, a more practical approach is to present the user with a small, diverse selection of "reasonable" Pareto-optimal journeys [Del+13]. Several definitions for a small, representative subset of the Pareto set have been proposed [Del+13, BBS13], but no efficient algorithms are known for computing them exactly, so existing approaches rely on heuristics. Recently, the restricted Pareto set [DDP19] was proposed to address these shortcomings, but it has not been applied to multimodal scenarios so far.

Overall, there is a significant performance gap between algorithms for public transit and multimodal journey planning. There are two main reasons for this: Firstly, the fastest public transit algorithms are not easily adaptable to more complex scenarios. Secondly, the existing approaches for extending public transit algorithms to multimodal networks incur a performance loss.

Shortcut Hypothesis. A promising approach for the problem of efficient multimodal extension was discovered by Sauer [Sau18]. Building on the experiments by Wagner and Zündorf [WZ17], the author shows that the impact of unrestricted footpaths depends on their position in the journey. Transfers can be divided into three categories: *Initial transfers* connect the source location to the stop where the first vehicle is entered, whereas *final transfers* lead from the last vehicle to the target location. Finally, *intermediate transfers* connect two public transit trips. Sauer shows that an algorithm with access to unrestricted initial and final transfers but one-hop intermediate transfers almost always finds the fastest journey.

Based on this finding, the author hypothesizes that the number of intermediate transfers that appear in at least one optimal journey is small. We call this the *shortcut hypothesis* because it implies an algorithmic solution for handling intermediate transfers: precompute all optimal intermediate transfers and condense them into a set of *shortcut edges* between stops. Then, existing approaches for one-to-many searches on road networks can be leveraged for the initial and final transfers. To test the hypothesis, Sauer presents a prototypical shortcut computation algorithm and a variant of RAPTOR that uses the resulting shortcuts. Experimental results offer preliminary support for the hypothesis, but the scope of the experiments is limited: the algorithm is evaluated on only one network, only for the combination of public transit and walking, and only for the two criteria arrival time and number of trips.

1.2 Main Contributions

Recall that there are two reasons for the performance gap between public transit and multimodal journey planning: the performance loss of existing approaches for multimodal extension and the lack of efficient query algorithms that can be applied in more complicated settings.

In this thesis, we close the gap by developing solutions for both problems and showing that they are applicable to a wide range of multimodal scenarios.

Multimodal Extension. To tackle the first issue, we present ULTRA (UnLimited TRAnsfers), a speedup technique that exploits the shortcut hypothesis in a more rigorous fashion than the prototypical algorithm by Sauer [Sau18]. ULTRA decomposes the complexity of multimodal journey planning into modular algorithmic components, each of which can be treated as a black box. The preprocessing phase computes a set of shortcuts that are provably sufficient to reconstruct an optimal set of journeys for every possible query. Initial and final transfers are handled with an existing algorithm for one-to-many search in road networks. Using ULTRA, any public transit algorithm that requires one-hop transfers can be turned into a multimodal algorithm. If the shortcut hypothesis holds, i.e., the number of shortcuts is small, then this algorithm is just as fast as its public transit counterpart.

We do not expect the shortcut hypothesis to hold in all possible multimodal scenarios; there are simply too many of them, with vastly different characteristics. We only consider scenarios that fit the following assumption: Public transit is the "main" mode, which is typically used for all or most of a journey. Individual transport modes serve as *transfer modes* that allow passengers to bridge gaps in areas with poor service or during off-peak hours. We discuss the limits of this assumption throughout the thesis. While many multimodal scenarios conform to it, some do not, such as certain combinations of public transit with car-based modes.

To test the shortcut hypothesis, we follow the iterative approach of Algorithm Engineering. We start with a simple multimodal problem setting that includes only one transfer mode and optimizes the two most common criteria: arrival time and number of trips. Starting from there, we gradually introduce extensions that make the problem more realistic and thereby more challenging. These include one-to-many queries, additional criteria, multiple transfer modes, and delays in the vehicle schedules. For each extended scenario, we adapt ULTRA and examine whether it produces a small set of shortcuts by conducting experiments on four representative real-world networks of varying sizes and types.

Throughout the thesis, we keep to a simple model for the transfer modes: we provide a road network (including pedestrian zones) and assume that it can be explored freely at a constant speed, obeying speed limits where applicable. This model ignores several real-world restrictions. However, it is useful for investigating the shortcut hypothesis because it maximizes the availability of the transfer mode and therefore provides an upper bound on the number of required shortcuts. Modes with rental vehicles, such as bicycles and e-scooters, require additional modeling for pickup and dropoff points, but this is not the focus of this thesis (we refer, for example, to [SWZ20b]).

It may not be immediately clear why it is necessary to support unrestricted intermediate transfers in practice, especially if this requires a potentially expensive preprocessing phase. For walking as the transfer mode, the experiments by Sauer [Sau18] indicate that one-hop intermediate transfers are almost always enough to find the fastest journey. Furthermore, many passengers will not want to walk for long distances even if it improves the travel time.

However, one-hop intermediate transfers are not satisfactory for several reasons. Firstly, the walking duration that can be guaranteed with one-hop transfers is often very low, especially in metropolitan networks. For the network of London, only four minutes of walking can be guaranteed, which is clearly less than what many passengers are willing to accept. Ultimately, the decision on how much walking is reasonable should be made by the user based on the output of the journey planning algorithm, not by the input model. Secondly, as mentioned above, one-hop transfers cannot represent faster transfer modes, such as bicycles and e-scooters, in a useful manner. Finally, even in scenarios with one-hop transfers, a shortcut precomputation algorithm may be useful because it can reduce the number of transfers compared to the transitive closure, thereby speeding up the query algorithm.

Query Algorithms. Our second main contribution is the design of more efficient multimodal query algorithms. We achieve this by carrying recent advances in public transit journey planning over to the multimodal setting and by devising new algorithms to handle additional criteria. TB achieves its performance improvements compared to previous algorithms by operating on the level of individual *events* (i.e., departures and arrivals) rather than stations and vehicle routes. The downside is that in order to do this, it must precompute possible transfers between events. We show that ULTRA can serve as a multimodal version of this precomputation step, allowing TB to operate on multimodal networks for the first time. Because other ULTRA-based algorithms also require this preprocessing phase, TB loses its main disadvantage and becomes the algorithm of choice for multimodal journey planning.

For the more complex problem settings, which include one-to-many queries and additional criteria, we design query algorithms that preserve as many of the advantages of TB as possible. We observe that some advantages are limited to particular contexts, whereas others are more broadly applicable. Additionally, we integrate ULTRA and TB with restricted Pareto sets, which enables the optimization of more criteria without bloating the solution size. The combination of these approaches allows us to close the performance gap to public transit journey planning in all considered scenarios. The resulting algorithms are the first ones that are fast enough for interactive applications even on country-scale networks.

1.3 Thesis Outline

The remainder of this thesis is structured as follows.

Chapter 2 introduces basic terminology and notation.

Chapter 3 reviews the literature on journey planning algorithms in road, public transit and multimodal networks.

Chapter 4 details existing journey planning algorithms that we build upon throughout this thesis. Section 4.1 presents variants of Dijkstra's algorithm and CH for journey planning in road networks. In Section 4.2, we discuss public transit algorithms, including CSA,

RAPTOR and TB. Finally, Section 4.3 outlines the experimental setup that we use throughout the thesis, including the four real-world multimodal networks that we use in our experiments.

Chapter 5 presents ULTRA for a simple multimodal problem setting with one transfer mode and two optimization criteria. Section 5.1 presents the shortcut computation algorithm and proves its correctness. In Section 5.2, we show how ULTRA can be combined with any existing public transit algorithm and re-engineer the TB query algorithm to support multimodal networks more efficiently. In Section 5.3, we evaluate the combination of ULTRA with CSA, RAPTOR and TB. We show that the shortcut hypothesis holds regardless of the speed of the transfer mode and that the ULTRA-based query algorithms fully close the performance gap compared to their public transit counterparts, offering a speedup of an order of magnitude compared to the state of the art. This chapter is based on joint work with Moritz Baum, Valentin Buchhold, Dorothea Wagner and Tobias Zündorf [Bau+19, SWZ20c, Bau+23].

Chapter 6 extends ULTRA to one-to-many and one-to-all queries. This requires us to replace the component that handles final transfers, which relies on a many-to-one search from all public transit stop to a single target vertex. Section 6.1 outlines our approach, which is inspired by the PHAST algorithm [DGNW13]. In Section 6.2, we describe how the TB query algorithm can be adjusted for one-to-many search. Experiments are presented in Section 6.3. This chapter is based on joint work with Dorothea Wagner and Tobias Zündorf [SWZ20a].

Chapter 7 revisits the criteria that are considered for Pareto optimization. We show that in multimodal networks, the two-criteria approach with arrival time and number of trips tends to produce journeys that use the transfer mode excessively, even when it is not necessary. To prevent this, we include the time spent in the transfer mode as a third criterion. In Section 7.1, we investigate the problem of computing the full three-criteria Pareto set. We show that under certain assumptions about the third criterion, this problem can be solved in polynomial time. We present three-criteria extensions of ULTRA and TB. Because the full Pareto set is too large for practical usage, we integrate our approaches with restricted Pareto sets in Section 7.2. Our experiments in Section 7.3 show that the restricted Pareto set with transfer time as a third criterion significantly improves the solution quality compared to the two-criteria Pareto set. We show that the shortcut hypothesis still holds in the three-criteria setting if the transfer mode has a low to moderate speed. Finally, we show that our algorithms offer interactive query times and a speedup of up to two orders of magnitude compared to the state of the art. This chapter is based on joint work with Moritz Potthoff [PS22b].

Chapter 8 extends ULTRA to multiple competing transfer modes. This requires careful modeling decisions, which we discuss in Section 8.1. In particular, we prohibit switching between modes in the middle of a transfer, except to pick up or drop off rental vehicles.

Additionally, we optimize the time spent in each transfer mode as a separate criterion. In Section 8.2, we show that the ULTRA shortcut computation can be run independently for each mode, which means that the preprocessing effort is only linear in the number of modes. Section 8.3 extends existing RAPTOR-based query algorithms to our scenario. Because the number of criteria is variable, the Pareto set can now have exponential size, unlike in Chapter 7. While this means that it is not possible to carry over all of the benefits of TB to this scenario, we propose a hybrid of RAPTOR and TB in Section 8.4 that retains some of them. We evaluate our approach in Section 8.5. When integrated with restricted Pareto sets, the solution set remains small and query times are only slightly higher than with a single transfer mode. This chapter is based on joint work with Moritz Potthoff [PS22a].

Chapter 9 extends ULTRA to handle delays in the vehicle schedules. We observe that the shortcut hypothesis no longer holds if the shortcuts are required to account for every possible combination of delays. Instead, we consider the problem of computing shortcuts for all delays up to a specified limit (e.g., five minutes). We analyze this problem theoretically in Sections 9.2 and 9.3. Based on these findings, we present a delay-robust variant of ULTRA in Section 9.4. To handle larger delays, Section 9.5 proposes a heuristic update phase that adds missing shortcuts based on the current delay information. Our experiments in Section 9.6 show that these approaches, when taken together, offer extremely low error rates while mostly retaining the performance benefits of ULTRA. This chapter is based on joint work with Dominik Bez [BS24].

Chapter 10 recapitulates the main findings of this thesis and gives an outlook on potential future applications and open problems.

2 Preliminaries

This chapter introduces the basic terminology and notation that are used throughout this thesis. First, we cover definitions related to graphs and shortest-path problems. Afterward, we define public transit networks, journeys and journey planning problems.

Graph. A *directed graph* is a tuple $G = (V, E)$ consisting of a set V of *vertices* and a set $E \subseteq V \times V$ of directed *edges* connecting pairs of vertices. An edge $(v, w) \in E$ is called an *incoming* edge of w and an *outgoing* edge of v. It is *incident* to both v and w, and the two vertices are *neighbors* of each other. The *degree* of a vertex is the number of edges incident to it. The graph is called *weighted* if there is a *cost function* $c : E \to \mathbb{R}$ that assigns a cost (also called *weight* or *length*) to each edge. In this work, we only consider integral, non-negative cost functions $c : E \to \mathbb{N}_0$.

A *path* $P = \langle v_1, \ldots, v_k \rangle$ is a sequence of vertices such that each pair v_i, v_{i+1} of consecutive vertices is connected by an edge $(v_i, v_{i+1}) \in E$. If the sequence consists of a single vertex, we call the path *empty*. If $v_1 = v_k$, we call P a *cycle*. Given source and target vertices $v_s, v_t \in V$, we call P an v_s-v_t-path if $v_s = v_1$ and $v_t = v_k$. The cost of P is the sum of its edge costs: $c(P) := \sum_{i=1}^{k-1} c((v_i, v_{i+1}))$. We call an v_s-v_t-path P a *shortest path* if there is no v_s-v_t-path with a lower cost than P. We call G *connected* if for each pair of vertices $v, w \in V$, the graph contains a path from v to w or vice versa, and *strongly connected* if it contains both.

Given a graph $G = (V, E)$, a source vertex $v_s \in V$ and target vertex $v_t \in V$, the *one-to-one* (or *single-pair*) *shortest path problem* asks for a shortest path between v_s and v_t. Several more general variants of this problem are also of interest. The *one-to-all* (or *single-source*) problem only specifies the source vertex and asks for a shortest path to every possible target vertex in V. In the *one-to-many* problem, a set $V_t \subseteq V$ of target vertices is given as part of the input. The *many-to-many* problem asks for pairwise shortest paths between a set $V_s \subseteq V$ of source

vertices and a set $V_t \subseteq V$ of target vertices. Finally, the *all-to-all* (or *all-pairs*) shortest path problem asks for shortest paths between all pairs of vertices in the graph.

The following special classes of graphs are of interest:

- A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. For a subset $V' \subseteq V$ of vertices, we call the graph $(V', E \cap (V' \times V'))$ the subgraph of G *induced* by V'. A *(strongly) connected component* of G is a (strongly) connected subgraph of G that is not part of any larger (strongly) connected subgraph of G.
- A graph $G^o = (V^o, E^o)$ is called an *overlay graph* for another graph $G = (V, E)$ if $V^o \subseteq V$ and G^o preserves shortest path distances in G. This means that for each pair of vertices $v_s, v_t \in V^o$, the cost of the shortest v_s-v_t-path in G is equal to the cost of the shortest v_s-v_t-path in G^o.
- A directed graph $G = (V, E)$ is called a *directed acyclic graph (DAG)* if it contains no cycles. A graph is a DAG if and only if it admits a *topological ordering* of its vertices. This is an ordering such that for every edge $e = (v, w)$, the vertex v comes before w in the ordering.

Network. A public transit network is a 6-tuple $(\mathcal{S}, \Pi, \mathcal{E}, \mathcal{T}, \mathcal{R}, G)$ consisting of a set of *stops* $\mathcal{S}$, a *service period* Π, a set of *stop events* $\mathcal{E}$, a set of *trips* $\mathcal{T}$, a set of *routes* $\mathcal{R}$, and a directed, weighted *transfer graph* $G = (V, E)$. A stop $v \in \mathcal{S}$ is a location in the network where passengers can board or disembark a vehicle (such as buses, trains, ferries, etc.). The service period $\Pi \subseteq \mathbb{N}_0$ defines the times at which vehicles can depart from or arrive at a stop. Throughout this thesis, we represent times by the number of elapsed seconds since midnight of the first day included in the timetable. For example, 5:30:02 AM on the second day is represented as 106 202. A *stop event* $\varepsilon = \big(\tau_{\text{arr}}(\varepsilon), \tau_{\text{dep}}(\varepsilon), v(\varepsilon)\big) \in \mathcal{E}$ represents a visit of a vehicle at the stop $v(\varepsilon) \in \mathcal{S}$, arriving at the *arrival time* $\tau_{\text{arr}}(\varepsilon) \in \Pi$ and departing with the *departure time* $\tau_{\text{dep}}(\varepsilon) \in \Pi$. A trip $T = \langle \varepsilon_0, \ldots, \varepsilon_k \rangle \in \mathcal{T}$ represents the ride of a vehicle as a sequence of stop events. The i-th stop event in T is denoted as $T[i]$. The length $|T| := k + 1$ is the number of stop events in T. The trip of a stop event ε is denoted as $T(\varepsilon)$. A *trip segment* $T[i, j] := \langle \varepsilon_i, \ldots, \varepsilon_j \rangle$ is the contiguous subsequence of T that begins at $T[i]$ and ends at $T[j]$.

The set of routes $\mathcal{R}$ is a partition of $\mathcal{T}$ such that two trips that are part of the same route visit the same sequence of stops and do not *overtake* each other. A trip T_a overtakes another trip T_b with the same stop sequence if there is an index i such that $\tau_{\text{arr}}(T_a[i]) \geq \tau_{\text{arr}}(T_b[i])$ or $\tau_{\text{dep}}(T_a[i]) \geq \tau_{\text{dep}}(T_b[i])$ and another index $j > i$ such that $\tau_{\text{arr}}(T_a[j]) \leq \tau_{\text{arr}}(T_b[j])$ or $\tau_{\text{dep}}(T_a[j]) \leq \tau_{\text{dep}}(T_b[j])$. Note that this implies that two trips of the same route may not have the same arrival or departure time at any stop along the route. Due to the non-overtaking property, we can define a total ordering $\prec$ on the set of trips of a route R: For two trips T_a, T_b of R, we write $T_a \prec T_b$ if $\tau_{\text{arr}}(T_a[0]) < \tau_{\text{arr}}(T_b[0])$. It follows that T_a has a lower arrival and departure time than T_b at every stop along the route. We write $T_a \preceq T_b$ if $T_a \prec T_b$ or $T_a = T_b$.

Note that trips from different routes are not comparable via $\prec$ and $\preceq$. For a given trip T, the route of T is denoted as $R(T)$. The trip that immediately precedes T in R according to $\prec$ is denoted as $\text{pred}(T)$. If T is the earliest trip of R, then $\text{pred}(T) = \bot$. The length $|R|$ of a route R is the length of any trip belonging to the route.

The cost function $\tau_{\text{tra}} : E \to \mathbb{N}_0$ of the transfer graph $G = (V, E)$ represents the time that it takes to traverse an edge (measured in seconds); we call this the *transfer time*. For two vertices $v, w \in V$, we denote by $\tau_{\text{tra}}(v, w)$ the transfer time of a shortest v-w-path, or ∞ if none exists. We distinguish between pure public transit networks and multimodal networks. In a pure public transit network, the transfer graph only represents footpaths between nearby stops, so $V = \mathcal{S}$ must hold. Furthermore, we require that G is transitively closed and fulfills the triangle inequality: for each pair of edges $e_1 = (v, w), e_2 = (w, x) \in E$, an edge $e_3 = (v, x) \in E$ with $\tau_{\text{tra}}(e_3) \leq \tau_{\text{tra}}(e_1) + \tau_{\text{tra}}(e_2)$ must exist. By contrast, multimodal networks impose no restrictions on G. It may contain vertices that are not stops, it does not need to be transitively closed or fulfill the triangle inequality, and it may represent any non-schedule-based mode of travel (e.g., walking, cycling, e-scooter). An example of a public transit network with an unrestricted transfer graph is shown in Figure 2.1.

Journeys. A *journey* describes the movement of a passenger through the network from a source vertex $v_s \in V$ to a target vertex $v_t \in V$. Each ride of the passenger in a public transit vehicle can be described by a trip segment, whereas the transfers between the rides are represented by paths in the transfer graph. An *intermediate transfer* between two trip segments $T_a[i, j]$ and $T_b[m, n]$ is a path P in G with the following properties:

1. The path begins with the last stop of $T_a[i, j]$, i.e., $v(T_a[j])$.
2. The path ends with the first stop of $T_b[m, n]$, i.e., $v(T_b[m])$.
3. The transfer time of the path is sufficient to reach $T_b[m, n]$ after vacating $T_a[i, j]$.

This can be expressed 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 segment $T[i, j]$ is a path in G from the source v_s to the first stop of $T[i, j]$. Correspondingly, a *final transfer* after a trip segment $T[i, j]$ is a path in G from the last stop of $T[i, j]$ to the target v_t.

A *journey* $J = \langle P_0, T_0[i, j], \ldots, T_{k-1}[m, n], P_k \rangle$ is an alternating sequence of transfers and trip segments. Note that some or all of the transfers may be empty. Given source and target vertices $v_s, v_t \in V$, we call journey J an v_s-v_t-journey if P_0 begins with v_s and P_k ends with v_t. The departure time of the journey is defined as $\tau_{\text{dep}}(J) := \tau_{\text{dep}}(T_0[i]) - \tau_{\text{tra}}(P_0)$ and the arrival time as $\tau_{\text{arr}}(J) := \tau_{\text{arr}}(T_{k-1}[n]) + \tau_{\text{tra}}(P_k)$. The transfer time $\tau_{\text{tra}}(J)$ of the journey is the total time spent traversing the transfer graph, i.e., $\tau_{\text{tra}}(J) = \sum_{i=0}^{k} \tau_{\text{tra}}(P_i)$. The number of trips used by the journey is denoted 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. Because such a journey does not use any trips, it can be traversed at any time. Thus, its departure time $\tau_{\text{dep}}(J)$ has to be stated separately, and its arrival time is then given by $\tau_{\text{arr}}(J) := \tau_{\text{dep}}(J) + \tau_{\text{tra}}(P_0)$. The *vertex*

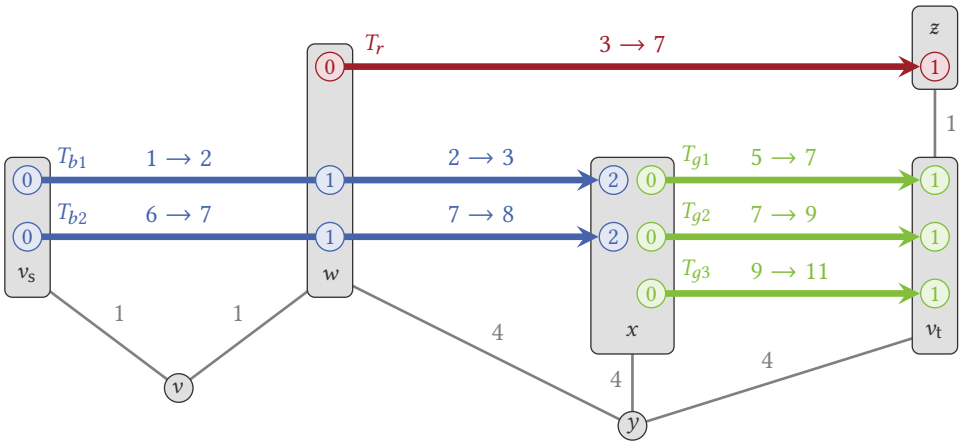


Figure 2.1: An example of a public transit network with an unrestricted transfer graph. Stop events are shown as colored nodes and trips as colored lines between them. Trips and stop events of the same color belong to the same route. Each stop event is labeled with its index along the respective trip. Each connection between two consecutive stop events ε_a and ε_b is labeled with their departure and arrival times, in the format $\tau_{\text{dep}}(\varepsilon_a) \to \tau_{\text{arr}}(\varepsilon_b)$. Stops are displayed as gray boxes enclosing the associated stop events, other vertices as gray nodes. Edges in the transfer graph (gray) are labeled with their transfer time. Throughout this thesis, we only indicate the direction of a transfer edge with an arrow if the reverse edge does not exist or has a different weight. For a query from v_s to v_t with departure time 0, a Pareto set with respect to arrival time and number of trips consists of the journeys $J_0 = \langle\langle v_s, v, w, y, v_t\rangle\rangle$ with arrival time 10, $J_1 = \langle\langle v_s, v, w\rangle, T_r[0,1], \langle z, v_t\rangle\rangle$ with arrival time 8, and $J_2 = \langle\langle v_s\rangle, T_{b1}[0,2], \langle x\rangle, T_{g1}[0,1], \langle v_t\rangle\rangle$ with arrival time 7.

sequence of J is the concatenation of its transfers: $V(J) = P_0 \circ P_1 \circ \cdots \circ P_k$. A *subjourney* of J is a journey $J_s = \langle P'_x, T_x[g,h], \ldots, T_{y-1}[p,q], P'_y\rangle$ such that $\langle T_x[g,h], \ldots, T_{y-1}[p,q]\rangle$ is a contiguous subsequence of J, P'_x is a suffix of P_x and P'_y is a prefix of P_y. If $x = 0$ and $P'_x = P_0$, we call J_s a *prefix* of J. Conversely, if $y = k$ and $P'_y = P_k$, we call J_s a *suffix* of J. Note that a subjourney may start or end in the middle of a transfer but never in the middle of a trip segment. Given two vertices $v, w \in V(J)$, the subjourney of J from v to w is denoted as $J[v, w]$.

Journey Planning Problems. Depending on the studied problem, different criteria are used to evaluate the usefulness of a journey J. In this thesis, we consider four criteria: the arrival time $\tau_{\text{arr}}(J)$, the departure time $\tau_{\text{dep}}(J)$, the transfer time $\tau_{\text{tra}}(J)$, and the number of trips $|J|$. With all criteria except for departure time, the objective is to minimize them, whereas departure time should be maximized. Given a set of criteria, a journey J *weakly dominates*

another journey J' if J is not worse than J' in any criterion. Moreover, J *strongly dominates* J' if J is strictly better than J' in at least one criterion, and not worse in the others. Given a query consisting of source and target vertices $v_s, v_t \in V$ and an earliest departure time τ_{dep}, a journey is called *feasible* if it is an v_s-v_t-journey that does not depart earlier than τ_{dep}. A feasible journey J is called *Pareto-optimal* if no other feasible journey exists that strongly dominates J. A *Pareto set* is a set $\mathcal{J}$ containing a minimal number of Pareto-optimal journeys such that every feasible journey is weakly dominated by a journey in $\mathcal{J}$.

A journey planning problem for a public transit network $(\mathcal{S}, \Pi, \mathcal{E}, \mathcal{T}, \mathcal{R}, G)$ depends on three parameters: the sets of source and target locations, the considered departure times, and the optimized criteria. The simplest problem is the *one-to-one, fixed departure time, earliest arrival* query: given a source vertex $v_s \in V$, a target vertex $v_t \in V$ and an earliest departure time τ_{dep}, the objective is to find a v_s-v_t-journey that is feasible for τ_{dep} and minimizes the arrival time among all such journeys. A *Pareto optimization* query instead requests a Pareto set of feasible journeys for the given set of criteria. In most parts of this thesis, we focus on Pareto optimization queries for the two criteria arrival time and number of trips. Figure 2.1 depicts a Pareto set for these two criteria in the shown example network. In Chapters 7 and 8, we add transfer time as a third criterion.

As with the shortest path problem on graphs, we also consider *one-to-many* and *one-to-all* queries, which replace the single target vertex with a set $V_t \subseteq V$ of target vertices or all of V, respectively. For each target vertex $v_t \in V_t$, the corresponding one-to-one query must be answered. We study these types of queries in Chapter 6. Finally, a *profile* (or *range*) query replaces the earliest departure time with an interval $[\tau_{dep}^{min}, \tau_{dep}^{max}]$ of possible departure times. It asks for a set $\mathcal{J}$ of journeys such that for each fixed departure time query with departure time $\tau_{dep} \in [\tau_{dep}^{min}, \tau_{dep}^{max}]$, $\mathcal{J}$ contains a subset that answers the query. This type of query occurs in the preprocessing phases of many speedup techniques, including ULTRA. In this case, the interval of departure times spans the entire service period Π of the network.

Departure Buffer Times. Many works on public transit journey planning (e.g., [PSWZ08, DDPW15]) include a *minimum change time* in their model, which is specified for each stop. This is the minimum amount of time that must elapse between exiting a trip and entering a different trip at the same stop. However, it does not have to be observed when taking an intermediate transfer to a different stop, nor when entering the first trip at the start of the journey. The minimum change time is useful when a larger station with multiple platforms is modeled as a single stop. In this case, it represents the time that is needed to change between platforms. Note that if the minimum change times differ between pairs of platforms belonging to the same station, this can be represented by modeling each platform as its own stop. For settings with one-hop transfers, minimum change times are a reasonable modeling choice. However, with an unrestricted transfer graph, they can lead to inconsistencies. Consider a stop v with minimum change time τ. If there is a cycle P in G that starts and ends at v and whose transfer time is less than τ, then taking P allows passengers to circumvent the minimum change time.

To prevent this, we use *departure buffer times* as an alternative modeling approach. Each stop $v \in \mathcal{S}$ has a departure buffer time $\tau_{\text{buf}}(v) \geq 0$, which is the minimum amount of time that has to pass after arriving at v before a trip can be boarded. Unlike the minimum change time, the departure buffer time always has to be observed when a trip is entered, even if the stop was reached via a transfer or if the trip is the first one in the journey. Departure buffer times can be integrated into the network implicitly by reducing the departure times of the stop events accordingly. For each stop event $\varepsilon = (\tau_{\text{arr}}(\varepsilon), \tau_{\text{dep}}(\varepsilon), v(\varepsilon)) \in \mathcal{E}$, this yields a modified stop event $\varepsilon' = (\tau_{\text{arr}}(\varepsilon), \tau_{\text{dep}}(\varepsilon) - \tau_{\text{buf}}(v(\varepsilon)), v(\varepsilon))$. Note that this may cause stop events to appear as if they depart before they arrive. However, because the departure time is only relevant when entering the trip at the current stop and not when remaining seated in the trip, this does not lead to trips that travel backward in time. In the following, we will not discuss departure buffer times explicitly and instead assume that they are integrated into the departure times as described here.

3 Literature Overview

This section gives an overview of prior work on journey planning in transportation networks. We cover road networks in Section 3.1, public transit networks in Section 3.2 and multimodal combinations of the two in Section 3.3. For a more comprehensive overview of literature published until January 2015, we refer to a survey article by Bast et al. [Bas+16].

3.1 Road Networks

A road network can be modeled as a graph $G = (V, E)$, in which intersections are modeled as vertices and roads as edges. The edge cost function $c : E \to \mathbb{R}_0^+$ represents a metric that should be optimized along a path, such as travel time or geographical distance. Finding an optimal path between two locations then reduces to solving the one-to-one shortest path problem on G. The two classical algorithms for this problem are Dijkstra's algorithm [Dij59] and the Bellman-Ford algorithm [Bel58, For56]. For non-negative edge costs, Dijkstra's algorithm has a running time of $\mathcal{O}(|V| \log |V| + |E|)$. This relies on the *label-setting* property: every vertex is scanned at most once. With negative edge costs, Dijkstra's algorithm loses this property and becomes *label-correcting*. Consequently, the worst-case running time becomes exponential [Joh73]. By contrast, the running time of Bellman-Ford [Bel58, For56] is always in $\mathcal{O}(|V||E|)$. Negative edge costs occur in some practical settings, for example when optimizing the energy consumption of electric vehicles [EFS11]. However, in the context of this thesis, edge costs always represent travel times, which are non-negative.

3.1.1 Speedup Techniques

For the purpose of answering a single query, Dijkstra's algorithm is the fastest known algorithm. On large networks, however, it is too slow for interactive applications: on a commonly used benchmark instance representing Western Europe, a query takes two seconds on average [Bas+16]. To answer many queries on the same network quickly, speedup techniques precompute auxiliary data in a preprocessing phase. This data is then used to speed up individual queries. Because the network topology rarely changes, this phase is allowed to take up to a few hours. The three main criteria to evaluate a speedup technique are the preprocessing time, the size of the precomputed data, and the query speed. Over the last two decades, many speedup techniques for road networks have been proposed. For this overview, we focus on those with applications in public transit or multimodal journey planning.

A simple improvement for Dijkstra's algorithm that does not require preprocessing is bidirectional search [Dan63, Nic66]. Instead of running one search from the source vertex that is stopped once the target is reached, searches are run from the source and target vertex simultaneously and they are stopped once they meet. This cuts the search space and therefore the query time roughly in half. While this is not enough on its own to achieve interactive query times, bidirectional search is used as a component in more sophisticated techniques.

Goal-Directed Search. Speedup techniques can be broadly divided into *goal-directed* and *hierarchical* approaches. Goal-directed techniques guide the search space of Dijkstra's algorithm towards the target. The oldest example is A* search [HNR68]. It assigns a *potential* to each vertex, which is a lower bound for its distance to the target. This is used to change the order in which Dijkstra's algorithm explores the vertices of the graph. Instead of advancing in increasing order of distance from the source, A* adds the potential to this distance, which ensures that vertices that are closer to the target are explored earlier. On road networks, a successful method for computing the potentials is ALT (A*, landmarks, triangle inequality) [GH05]. A small number of vertices are chosen as *landmarks*, and shortest path distances are computed between each landmark and all other vertices in the graph. Given a target vertex, these are combined into a vertex potential based on the triangle inequality. This yields a speedup of up to two orders of magnitude over Dijkstra's algorithm.

A goal-directed technique that invests more preprocessing effort to achieve faster query times is Arc-Flags [HKMS09, Lau09, Möh+06]. Here, the graph is partitioned into k cells, and each edge is labeled with k bits, called *flags*. The flag for a cell i indicates whether the edge lies on at least one shortest path whose target is located in cell i. Dijkstra's algorithm can then skip edges whose flag for the target cell is not set. The preprocessing step is performed by computing a reverse shortest path tree from every vertex that lies on a cell boundary. Arc-Flags achieves a speedup of three to four orders of magnitude over Dijkstra's algorithm.

Hierarchical Techniques. A second class of speedup techniques is based on the observation that road networks are hierarchical: shortest paths between points that are sufficiently far away from each other tend to pass through a small subset of important roads. This

hierarchy exists at multiple levels, from local arterial roads to highways spanning entire countries. An early approach for exploiting this hierarchy is the overlay graph [SWW00], as defined in Chapter 2. It can be constructed by selecting a subset of very important vertices and inserting *shortcut edges* between them to preserve shortest path distances. This concept was later extended to *multi-level overlays (MLO)* to capture multiple levels of the hierarchy. Overlays can be explored with a bidirectional variant of Dijkstra's algorithm that only explores edges to more or equally important vertices. In these early techniques, the importance of vertices was given as part of the input [SWW00] or based on rough estimates, such as the vertex degree [SWZ02]. Later approaches, such as Highway Hierarchies (HH) [SS12] and Highway-Node Routing (HNR) [SS07], employed sophisticated preprocessing algorithms to compute the hierarchy in a systematic fashion.

These techniques were later subsumed by Contraction Hierarchies (CH) [GSSV12], which achieve better performance with a simpler preprocessing algorithm based on *vertex contraction*. The idea is to rank the vertices according to a simple importance estimate (e.g., vertex degree) and then contract them in order of least to most important. A vertex is contracted by removing it from the graph, inserting shortcut edges between its neighbors and updating their importance estimate. After each contraction step, the remaining graph forms an overlay, so CH can be seen as an extreme version of MLO. The output of the CH preprocessing phase is an *augmented graph*, which only contains edges from less to more important vertices. Queries are answered with a bidirectional search on the augmented graph, thereby quickly guiding the search upwards in the hierarchy. CH outperforms older techniques, such as ALT, Arc-Flags, MLO, HH and HNR, in all respects: on the network of Western Europe, it requires a few minutes of preprocessing time, a few hundred megabytes of space, and achieves query times of around 100 microseconds. Perhaps more crucially, because it is built on simple ingredients, it has proven to be easily adaptable to extended scenarios, as discussed below.

Separator-Based Techniques. An alternative method of obtaining a network hierarchy is via *separators*. A *vertex separator* is a subset of the vertices that, when removed, divides the graph into two or more disconnected components. If the components are of roughly equal size, the separator is called *balanced*. An *edge separator*, also called a *cut*, uses a set of edges instead of vertices to divide the graph. Planar graphs are known to have small, balanced vertex separators that can be computed efficiently [LT79]. By recursively separating the leftover components, one can obtain a *separator decomposition* of the graph. Although road networks are not planar due to over- and underpasses, it has been demonstrated both experimentally [DSW16] and theoretically [EG08] that they also admit efficient separator decompositions. Alternatively, the set of vertices can be partitioned into cells of roughly equal size by repeatedly computing small, balanced cuts. It has been shown experimentally that road networks exhibit such cuts due to natural obstacles such as rivers and mountain ranges [DGRW11]. Furthermore, because the vertices in a road network have low degrees (rarely more than four), any small, balanced vertex separator can be transformed into a small, balanced cut by choosing the incident edges for the cut.

If a vertex belongs to a vertex separator or is incident to an edge separator, it is likely to lie on many shortest paths between different components. For this reason, separator decompositions and partitions are well suited as a basis for MLO [Del+09, HSW09]. Compared to other hierarchy-based approaches, they have the advantage that they are based solely on the graph topology, not on the edge costs. This observation has given rise to a class of *customizable* techniques. These are able to handle frequently changing edge costs (e.g., to incorporate traffic information or different user preferences) by dividing the algorithm into three phases: The first preprocessing phase, which may be slow, builds data structures that depend only on the graph topology, such as a separator decomposition. Edge costs are incorporated in a second *customization* phase. To allow for frequent updates, this phase should take no more than a few seconds. Finally, queries are answered in a third phase. The earliest example of such a technique is Customizable Route Planning (CRP) [DGPW17], which constructs a multi-level partition of the graph and builds a clique between the boundary vertices of each cell. The customization phase is responsible for computing the edge costs for these cliques. Customizable Contraction Hierarchies (CCH) [DSW16] extend CH to the customizable setting by using separator decompositions instead of node contraction to construct the vertex hierarchy. In general, separator-based hierarchies are slightly weaker than their metric-dependent counterparts, so these techniques have slightly longer query times.

Bounded-Hop Techniques. The versatility of CH is showcased by the fact that it can be used as a building block in speedup techniques that invest even more preprocessing effort to achieve query times of a few microseconds or less. One such technique is Hub Labeling (HL) [CHKZ03], also known as *two-hop labeling*. The basic idea is that each vertex stores shortest path distances to a small set of *hubs*. These hubs must fulfill the *cover property*: given any shortest path P, at least one vertex on P must be a hub for both endpoints of P. Using this property, a query can be answered by intersecting the hub sets of the source and target vertices to find the shared hub. The upward CH search space of a vertex forms a valid set of hubs for HL, which can be thinned out by removing redundant entries [ADGW11]. Various other HL precomputation algorithms have been proposed, some of which also perform well on graph classes other than road networks [ADGW12, DGPW14]. HL yields extremely fast queries that only require a few memory accesses on average, albeit at the cost of a very high memory consumption.

A similar approach that requires less memory but can still answer queries in a few microseconds is Transit-Node Routing (TNR) [BFSS07, BFM09, SS09]. It exploits the inherent hierarchy of road networks by selecting a small set of *transit nodes* that cover all long-distance shortest paths. A full distance table is computed between the transit nodes. For each vertex in the graph, the algorithm identifies its *access nodes*, i.e., the transit nodes that occur in long-distance shortest paths from or to that vertex. A long-distance query can then be answered by identifying the correct pair of access nodes for the source and target vertex and combining the distances to the access nodes with the precomputed distance between them.

Note that a fallback routine is required for short-range queries that do not pass through a transit node. Again, CH enable an elegant implementation of TNR [ALS13] by choosing the most important vertices in the hierarchy as the set of transit nodes. In this case, CH also serves as the fallback routine for short-range queries.

Theoretical Models. To explain the good performance of techniques such as CH, TNR and HL from a theoretical perspective, several models have been proposed that capture the hierarchical nature of road networks. Graph parameters such as highway dimension [Abr+16] and skeleton dimension [KV17] are measures for the number of vertices that are needed to cover all shortest paths between vertices that are sufficiently far away from each other. In graphs with low highway and/or skeleton dimension, CH, TNR and HL have significantly better worst-case running times than Dijkstra's algorithm. Recently, bounded-growth graphs [BFS21] have been proposed as an alternative model that is better suited for graphs with grid-like substructures (which occur in road networks). It can be shown that the search spaces of CH, TNR and HL are sublinear in bounded-growth graphs.

Combinations. Finally, goal-directed and hierarchical techniques can be combined for even faster queries. An early example of such a combination is SHARC (Shortcuts + Arc-Flags) [BD10], which achieves an additional order of magnitude of speedup over Arc-Flags by combining it with ideas from MLO. A crucial advantage of this approach is that it does not require bidirectional search. This makes it easier to use in time-dependent networks, where the arrival time at the target is not known in advance. Bauer et al. [Bau+10] evaluate several other combinations of existing speedup techniques, the most notable of which are Core-ALT and CHASE (CH + Arc-Flags + HNR). Core-ALT uses vertex contraction to compute an overlay and then applies ALT only within the overlay, which reduces the preprocessing effort and improves query times. CHASE is a combination of CH and Arc-Flags, which yields query times of only a few microseconds on Western Europe.

3.1.2 Extended Scenarios

The techniques discussed so far are designed for one-to-one queries in graphs with a single, scalar edge cost function. However, many of them have also been extended to more complex scenarios. We give a brief overview in the following.

One-to-Many Search. Several journey planning applications require one-to-many or many-to-many searches. Notable examples include building distance tables for vehicle routing and traveling salesman problems [Kno+07, DW15], point-of-interest (POI) queries (e.g., finding the k nearest stores) [DW15], and computing isochrones, which are regions of the network that are reachable from a given point within a specified time limit [EP14, Bau+16, BBDW19]. Furthermore, several speedup techniques for one-to-one queries, such as Arc-Flags, employ one-to-many searches in their preprocessing step.

PHAST (PHAST Hardware-Accelerated Shortest Path Trees) [DGNW13] is an extension of CH for one-to-all searches that exploits the fact that the augmented graph is a DAG. After an upward search from the source vertex, distances to all vertices are computed by a downward sweep through the augmented graph in a topological order. This sweep can be made highly cache-efficient by reordering the vertices in memory. RPHAST (Restricted PHAST) [DGW11] extends this approach to one-to-many searches by adding a *target selection* phase that discards irrelevant parts of the augmented graph. GRASP (Graph Separators, Range, Shortest Paths) [EP14] applies similar ideas to CRP instead of CH. For one-to-many searches, an alternative to the sweep-based algorithms is a bucket-based approach, which was originally proposed for HH [Kno+07] and later adapted for CH [GSSV12]. For each vertex, it stores a *bucket* consisting of reachable targets and the distances to them. A one-to-many query can then be answered by performing an upward search in the augmented graph and combining the distances to the reached vertices with the distances stored in the buckets.

Time-Dependent Edge Costs. In reality, travel times are not static but vary throughout the day due to predictable traffic patterns. These can be modeled by replacing the scalar edge costs with functions that map departure time to travel time. Several speedup techniques have been adapted to this scenario, including ALT [NDSL12, DN12], CRP [BDPW16], CH [BGSV13], SHARC [Del11] and CCH [SWZ21]. An issue with shortcut-based techniques is that the complexity of the edge cost functions grows dramatically for shortcuts that represent longer paths. As a result, many exact approaches suffer from expensive preprocessing and slow queries. CATCHUp (Customizable Approximated Time-Dependent Contraction Hierarchies Through Unpacking) [SWZ21] is an extension of CCH that remedies this shortcoming. It is based on the observation that while the cost of the shortest path tends to change frequently throughout the day, the path itself does not. To exploit this, CATCHUp does not compute entire travel time functions for each shortcut but rather pointers to the edges from which it was constructed. By unpacking these pointers on the fly during the query phase, the exact travel time can be reconstructed.

Multiple Criteria. The standard approach for handling scenarios with multiple competing criteria (e.g., travel time and distance) is Pareto optimization. A variant of Dijkstra's algorithm that computes Pareto sets was first proposed for two criteria [Han80] and later generalized to an arbitrary number of criteria [Mar84]. Already for two criteria, the number of Pareto-optimal solutions can be exponential in the size of the network [Han80], so the algorithm no longer runs in polynomial time. However, if the values of the criteria are highly correlated, the Pareto set may be small enough in practice that computing it remains tractable. One speedup technique that has been successfully adapted to Pareto optimization is SHARC, achieving a speedup of up to four orders of magnitude over Dijkstra's algorithm [DW09].

A related problem setting is that of *personalized routing* [FS15]. Here, the objective is to minimize a convex combination of multiple criteria, i.e., the value of every criterion is assigned a non-negative weight such that all weights sum up to 1. The weights are individual to each

user and are only revealed at query time, so the preprocessing phase must remain oblivious to them. Several speedup techniques have been adapted to this setting, including CH [GKS10, FS13], CCH and CRP [FS15]. In the original personalized routing model, users must specify the weights directly, which may be challenging to do [BFP21]. An alternative approach is to compute a set of solutions that are optimal for a wide range of possible preferences and allow users to choose from this set [BFS19]. This has the advantage of producing fewer journeys than the full Pareto set: for a fixed number of criteria, Barth et al. [BFP22] show that the number of paths that are optimal for at least one convex combination is subexponential.

3.2 Public Transit Networks

Public transit networks differ from road networks in that they are inherently time-dependent: vehicles depart and arrive according to a fixed schedule. This time dependency can be handled in different ways. Early approaches modeled the network as a graph and applied Dijkstra's algorithm or speedup techniques that were originally developed for road networks. However, because public transit networks do not exhibit some of the structural properties that these techniques rely on, their success has been limited [Bas09]. Instead, more recent approaches have moved away from modeling the network as a graph and instead employ tailor-made data structures that can be explored with cache-efficient scanning operations.

3.2.1 Graph-Based Models

For a thorough overview of graph-based modeling techniques, we refer to an article by Müller-Hannemann et al. [MSWZ07]. The two main modeling approaches are *time-dependent* and *time-expanded* graphs. In a time-dependent graph, vertices represent stops. Two stops are connected by an edge if they are visited consecutively by at least one trip. The time dependency is modeled via the edge costs, which are functions that map departure time to travel time or arrival time. The function associated with an edge incorporates all trips that travel along the edge. This approach is similar to the way that time-dependent edge costs are handled for road networks. However, the functions are quite different. In road networks, travel time functions are typically continuous and the slope varies throughout the day. In public transit networks, if departure time is mapped to arrival time, the functions are piecewise constant. Each constant segment represents one departing trip. After the trip has departed, the fastest remaining option is to wait for the next one, which causes a discontinuity in the function. The time-dependent graph model was first introduced by Brodal and Jacob [BJ04]. Later versions incorporate additional features such as footpaths [DMS08] and minimum change times [PSWZ08]. To model the latter, each stop is split into multiple vertices, one for each route. Recently, a more compact model called REX (REalistic eXchange times) [KMPZ22] was proposed that does not require these additional vertices. In order to properly handle trips from different routes that overtake each other, the authors present a new query algorithm called TRIPLA.

In time-expanded graphs, the time dependency is encoded directly in the graph topology. Each stop event is represented by three vertices: a *departure node*, an *arrival node* and a *transfer node*. Trips are modeled as paths that alternate between arrival and departure nodes, whereas transfers are represented from arrival to departure nodes that pass through transfer nodes. Early versions of the time-expanded model [PS98, MW01] did not include the transfer nodes, which were introduced to handle minimum change times [MS07, PSWZ08]. Extensions that incorporate footpaths have also been proposed [MSWZ07]. Because the time-expanded graph includes several nodes per stop event, it is much larger than the time-dependent model. However, its size can be reduced by contracting some types of nodes [DPW09b].

3.2.2 Basic Algorithms

In graph-based models, the earliest arrival problem can be solved with Dijkstra's algorithm. On time-expanded graphs, it can be applied directly, whereas time-dependent graphs require a variant that can handle functions as edge costs [PSWZ08]. The time-dependent variant of Dijkstra's algorithm is about four times faster than the time-expanded one [Bas+16]. For the latter, an additional pruning rule called node blocking [DPW09b] can be applied to ensure that only the earliest reachable trip of each route is explored.

In public transit journey planning, multiple optimization criteria arise naturally. In addition to arrival or travel time, most algorithms also consider the number of used trips or, equivalently, the number of transfers between trips. Pyrga et al. [PSWZ08] observe that for this particular pair of criteria, a Pareto set can be found in polynomial time: Because each node in the time-expanded graph is associated with a fixed arrival time, there may be at most one Pareto-optimal solution per node. Hence, it suffices to run Dijkstra's algorithm with the number of transfers as the optimization criterion and then collect the Pareto-optimal solutions at the transfer nodes representing the target stop. In time-dependent graphs, the number of transfers can be Pareto-optimized implicitly by creating multiple copies of the network and moving to a new copy every time a trip is exited. In practice, it is faster not to create the copies explicitly but to maintain multiple labels at each node, one for each number of trips [PSWZ08].

Pareto optimization with more than these two criteria has also been considered in a number of works. Multicriteria variants of Dijkstra's algorithm have been applied to both time-expanded [MS07] and time-dependent graphs [DMS08]. For the three criteria travel time, number of transfers and fare, Müller-Hannemann and Weihe [MW01] observe that the Pareto sets are small in practice, with fewer than ten journeys on average. Other considered criteria include transfer reliability [DMS08], walking duration, and the number of used buses [DDP19].

Also of practical interest are profile queries, for which the departure time may lie in a specified interval. This gives users more flexibility in choosing their departure time, which may allow them to select a better (e.g., faster) journey overall [DKP12]. Furthermore, profile searches are used as an ingredient in speedup techniques [Bas+10]. In time-dependent graphs, profile queries can be answered with a variant of Dijkstra's algorithm that maintains travel time functions instead of scalar distances per vertex [Nac95]. Because travel time functions do not admit a total ordering, this variant loses the label-setting property and may scan vertices

more than once. The Self-Pruning Connection-Setting (SPCS) algorithm [DKP12] improves upon this by exploiting the fact that the only possible departure times of a valid journey are those of trips departing at the source stop. SPCS therefore performs one *run* of Dijkstra's algorithm for each such departure time in descending order. Distance labels are not reset between runs, which allows journeys from previous runs to dominate suboptimal ones in the current run. This method, which is known as *self-pruning*, restores the label-setting property for each run. Another approach called frequency-based search [BS14] exploits the periodicity of timetables to compress the data structures and thereby speed up the profile search.

3.2.3 Speedup Techniques

Various speedup techniques for Dijkstra's algorithm that are successful on road networks have also been applied to public transit. The earliest example are multi-level overlays [SWZ02], which were in fact originally developed for public transit networks. Other examples include Arc-Flags [BDGM09, DPW09b], SHARC [BDGM09, Del11] and CH [Gei10]. A commonly used technique is to obtain a potential for A* by constructing a *lower-bound graph* that assumes that every trip can be taken immediately without a waiting time. Since the resulting travel times are lower bounds for the actual travel time, they yield valid potentials. Compared to Dijkstra's algorithm, this version of A* achieves a speedup of about two, which indicates that the lower bounds are not particularly tight. Nevertheless, this technique has been employed in various settings, including two-criteria Pareto optimization in both time-expanded [MS07] and time-dependent graphs [DMS08]. It has also been combined with other techniques, notably Arc-Flags [DPW09b, BDGM09].

Even when combining multiple speedup techniques, none of these approaches manage to achieve a speedup of more than 30 compared to Dijkstra's algorithm. A number of techniques were evaluated in simplified, unrealistic settings. These include Core-ALT, CH and CHASE in [Bau+10] as well as ALT and Arc-Flags in [BDW11]. Even in these simplified scenarios, the achieved speedups are much lower than for road networks. This discrepancy has been explained by the fact that public transit networks do not exhibit some of the favorable properties that make the techniques successful on road networks [Bas09]. Some of the known issues include the following:

- An issue that already occurs in time-dependent road networks is that the arrival time at the target vertex is not known in advance. Thus, bidirectional search cannot be applied in a straightforward manner.
- Techniques based on overlays suffer from the fact that public transit networks contain many high-degree vertices (e.g., large train stations). Accordingly, the overlays quickly become very dense.
- The strong multi-level hierarchy exhibited by road networks does not exist to the same degree in public transit networks. Especially on the local scale (e.g., bus services), almost no hierarchy is apparent.

- Local searches (e.g., in order to construct shortcut edges) tend to have larger search spaces due to waiting times caused by the vehicle schedules: a vertex that is geographically nearby may still take a long time to reach if the service frequency is low, whereas far-away vertices connected to high-frequency lines may be reachable much faster.

Thus far, speedups of two orders of magnitude and more have only been achieved by techniques that employ very expensive preprocessing phases. Public Transit Labeling [DDPW15] is an adaptation of HL for time-expanded graphs that supports Pareto optimization of arrival time and number of trips. On metropolitan and mid-sized country networks, it offers query times in the microsecond range, albeit at the expense of over ten gigabytes of memory consumption. Furthermore, this does not include the significant additional space that would be required to store and retrieve descriptions of the computed journeys.

A technique that was specifically designed for public transit networks and also supports two-criteria Pareto optimization is Transfer Patterns (TP) [Bas+10]. Its preprocessing phase performs a one-to-all profile query spanning the entire service duration of the network from each possible source stop. To keep the memory consumption reasonable, journeys are condensed into their *transfer patterns*, which are the sequences of stops where trips are entered or exited. For each source stop, the transfer patterns of all computed journeys are merged into a DAG. Because many journeys share parts of the same transfer patterns, this DAG is comparatively small. During the query phase, a search graph is extracted from the DAG and explored with a variant of Dijkstra's algorithm. For each edge in the search graph, the used trip is reconstructed on the fly. Unfortunately, no comparisons of TP to other algorithms on the same benchmark instances are available. On the network of Germany, query times below one millisecond are reported [BS14]. Scalable Transfer Patterns [BHS16] reduce the preprocessing time and memory consumption of TP by grouping the stops into clusters. The preprocessing phase is divided into computing local (intra-cluster) and long-distance (inter-cluster) transfer patterns, the latter of which only require searches from the border stops of each cluster. Query times are significantly slower at 30 ms on the Germany network.

3.2.4 Timetable-Based Approaches

A recent group of algorithms improves upon the performance of the graph-based approaches by employing tailor-made data structures. These resemble the time-expanded graph but are designed to be explored with operations that scan large blocks of data in a single sweep, which reduces the rate of cache misses. The Connection Scan Algorithm (CSA) [DPSW18] is based on the observation that the time-expanded graph is a DAG and thus shortest paths can be found by scanning the stop events in topological order. This yields a very cache-efficient algorithm that is up to an order of magnitude faster than Dijkstra's algorithm on the time-dependent graph. The original CSA only optimizes the arrival time, although an extension for profile queries adds the number of trips as a second criterion.

RAPTOR (Round-bAsed Public Transit Optimized Router) [DPW15a] Pareto-optimizes arrival time and the number of trips by exploring the network in rounds. In each round,

RAPTOR scans all reachable routes in order to extend the previously found journeys by another trip. The resulting algorithm is similar to a BFS in which the routes correspond to vertices and the transfers between them to edges. Compared to the fastest Dijkstra-based algorithm for the same two criteria, RAPTOR offers a speedup of five. Trip-Based Routing (TB) [Wit15] improves upon RAPTOR by computing transfers between pairs of stop events in a preprocessing phase. This allows the query phase to skip the step of identifying the earliest reachable trip of a route. Accordingly, the TB query scans trip segments instead of entire routes. Thus far, RAPTOR and TB have not been evaluated on the same machine; a comparison across different machines but with nearly identical networks suggests that TB is faster by a factor of four to five [Wit15]. Both RAPTOR and TB have been extended for profile queries by incorporating the self-pruning method.

McRAPTOR (Multicriteria RAPTOR) [DPW15a] extends RAPTOR to support Pareto optimization for an arbitrary number of criteria in addition to arrival time and number of trips. Thus far, no such extension has been proposed for TB. With three or more criteria, the Pareto set may become excessively large, which in turn slows down the algorithm. A possible solution to this problem is the restricted Pareto set [DDP19], which only contains journeys that make improvements in the additional criteria if they do not arrive much later or use significantly more trips. The restricted Pareto set can be computed with an extension of McRAPTOR called BM-RAPTOR (Bounded McRAPTOR). For walking duration and number of buses as the two additional criteria, BM-RAPTOR is twice as slow as two-criteria RAPTOR and up to 65 times faster than McRAPTOR.

Several speedup techniques have been proposed for the timetable-based algorithms. Connection Scan Accelerated [DPSW18] applies the concept of multi-level overlays to CSA. On country-scale networks, it achieves a speedup of up to seven over CSA, although it offers no improvement on dense metropolitan networks. HypRAPTOR [DDPZ17] is a speedup technique for RAPTOR that partitions the set of routes by representing it as a hypergraph in which the routes are vertices and the visited stops are hyperedges. The preprocessing phase computes a *fill-in*, which is the set of routes that are needed to travel between different cells. The query phase then only explores routes from the source and target cells as well as the fill-in. Because the fill-in is fairly large, HypRAPTOR only achieves a speedup of two over RAPTOR. Finally, Trip-Based Routing with Condensed Search Trees (TB-CST) [Wit16] applies the ideas behind TP to TB [Wit16]. As with TP, the preprocessing phase performs all-to-all profile searches. The computed journeys are stored as *search trees* in which the used route segments are vertices and the used transfers are edges. To reduce the memory consumption, shared suffixes are extracted from the search trees and re-attached during the query phase.

3.3 Multimodal Networks

Many public transit algorithms support walking in a limited capacity and are therefore sometimes referred to as multimodal algorithms. Graph-based models support footpaths in the form of direct edges between pairs of stops [MSWZ07, DMS08]. If a full footpath network

were to be encoded in this manner, the number of edges would be quadratic in the number of stops. To ensure that the graph size remains reasonable, footpaths are typically restricted to small connected components of nearby stops [DKP12], for example by limiting the maximal duration (e.g., five minutes of walking) or distance (e.g., 400 m) [BS14, BHS16, GPZ19] of the footpaths. Timetable-based algorithms, including CSA, RAPTOR and TB, also allow footpaths in the form of direct edges between stops. Additionally, they require that the set of transfer edges is transitively closed and fulfills the triangle inequality. We refer to transfers with this restriction as *one-hop transfers* because it is never necessary to use more than one edge in succession. This removes the need to explore the transfer graph with Dijkstra's algorithm; instead, footpaths are handled by exploring the outgoing edges of each visited stop.

A modification of RAPTOR has been proposed that uses one-hop transfers without requiring a transitive closure [DDP19]. In this case, journeys with multiple consecutive transfer edges are prohibited and the algorithm finds optimal journeys among those that remain. This can lead to counterintuitive journeys that take detours to avoid using two transfer edges in succession. On the other hand, computing the transitive closure significantly increases the size of the transfer graph. As shown by Wagner and Zündorf [WZ17], limiting the maximal walking duration to 20 minutes before computing the transitive closure already leads to a graph that is too large for practical applications. Therefore, we do not classify algorithms with one-hop transfers as proper multimodal algorithms.

The most straightforward approach to multimodal journey planning is to model each network individually as a graph and then combine the graphs with link edges. Note that the combined graph includes both time-dependent (public transit) and time-independent (road) components. Therefore, the time-dependent graph model is more suitable for this use case than the time-expanded model because it makes linking between the components easier [Paj09]. For flight networks, special time-dependent graph models have been designed that more closely reflect the structure of these networks [DPWZ09].

3.3.1 Label-Constrained Shortest Path Problem

Merely computing the shortest path in a combined multimodal network can lead to unrealistic journeys with impossible mode combinations (e.g., using a private car in between two public transit trips). The *label-constrained shortest path problem (LCSPP)* [BJM00] overcomes this issue by allowing mode restrictions to be specified as part of the input. Each edge is labeled with the used mode; concatenating these labels along a path yields the *mode sequence* of the path. The set of valid mode sequences can be described as a formal language. The objective of LCSPP is to find the shortest path among those whose mode sequence is valid. If the language is regular, it can be described with a finite automaton. Then the problem can be solved by applying Dijkstra's algorithm to the product graph of the network and the automaton. Alternatively, the product graph can be represented implicitly by allowing the algorithm to maintain multiple labels at each vertex, up to one per state in the automaton.

Various speedup techniques have been adapted to LCSPP, including bidirectional search and A* [Bar+09]. Access Node Routing (ANR) [DPW09a] adapts TNR to a special case of

LCSPP in which the road network may only be used for initial and final transfers and the public transit network in-between. ANR has only been evaluated on instances in which the public transit network is much smaller than the road network. On these instances, it offers a speedup of more than four orders of magnitude. State-Dependent ALT [KLPC11, KLC12] is an adaptation of ALT that computes a potential for each combination of vertex and state in the mode sequence automaton. Note that this requires the mode sequence constraints to be known at preprocessing time. A technique that allows the constraints to be specified at query time is User-Constrained Contraction Hierarchies (UCCH) [DPW15b]. It performs a partial CH precomputation but leaves vertices that are incident to link edges uncontracted. The query algorithm performs a CH upward search until it reaches the uncontracted core graph, which is explored with LCSPP-Dijkstra.

3.3.2 Multicriteria Optimization

The downside of LCSPP is that it optimizes only a single criterion. MCR (Multimodal Multicriteria RAPTOR) [Del+13] is a multimodal extension of McRAPTOR that allows for the Pareto optimization of an arbitrary number of criteria. It operates similarly to UCCH but explores the public transit network with McRAPTOR instead of Dijkstra's algorithm. MCR has been evaluated on multimodal networks combining public transit, walking, cycling and taxis. Up to four criteria are optimized: arrival time, number of trips, walking duration and taxi cost. With four criteria, the full Pareto set becomes so large that a single MCR query takes over half an hour to answer. To filter the Pareto set, the authors use a weakened definition of Pareto dominance called *fuzzy dominance*. Unfortunately, there seems to be no way of obtaining the exact filtered set without computing the full Pareto set first. The authors therefore propose several heuristics to approximate the Pareto set. However, in the four-criteria setting, these heuristics still require several seconds to answer a single query. Another approach for filtering the Pareto set is TNT (Types aNd Thresholds) [BBS13], which prohibits journeys with unreasonable mode combinations (e.g., long car drives followed by long footpaths). As with MCR, an exact algorithm is too slow for practical use, so the authors apply several heuristics. Restricted Pareto sets [DDP19] may offer a more efficient solution for pruning the Pareto set, but so far they have not been applied to multimodal networks.

A few algorithms have been proposed for more restricted multimodal problem settings. For a scenario with walking as the only transfer mode, HL-RAPTOR and HL-CSA [PV19] interleave RAPTOR and CSA, respectively, with two-hop searches based on HL. The authors report a speedup of 1.7 over the two-criteria variant of MCR and 3.4 over the three-criteria variant with walking duration as the third criterion. Giannakoupoulou et al. [GPZ19] study a variant of the time-expanded graph model that allows for fast updates, for example to handle delays or trip cancellations. They Pareto-optimize arrival time and number of trips but only allow journeys that do not exceed the travel time of the fastest journey by a specified factor. However, in a scenario with unlimited walking, query times are not competitive with the two-criteria variant of MCR.

4 Basic Algorithms and Experimental Setup

As we saw in the previous chapter, approaches for multimodal journey planning typically use existing algorithms for public transit and road networks as building blocks. These are then modified and combined to handle a more complex problem setting. The algorithms presented in this thesis are no exception. In this chapter, we therefore give in-depth explanations of these basic building blocks and discuss crucial implementation details. Section 4.1 focuses on algorithms for road networks, including variants of Dijkstra's algorithm and CH. Public transit and multimodal algorithms are discussed in Section 4.2. In addition to presenting the algorithms, we identify concepts that recur in multiple algorithms or are further developed from one algorithm to the next. Finally, Section 4.3 gives an overview of our experimental setup, including the real-world networks on which we evaluate our algorithms.

4.1 Road Networks

4.1.1 Dijkstra's Algorithm

Given a graph $G = (V, E)$ with an edge cost function $c : E \to \mathbb{R}_0^+$ and a source vertex $v_s \in V$, Dijkstra's algorithm [Dij59] solves the one-to-all shortest path problem. For each vertex $v \in V$, it maintains a *tentative distance* $\texttt{dist}[v]$, which is initialized with ∞, and a *parent pointer* $\texttt{p}[v]$, which is initialized with a dummy value $\perp$. At each point during the execution of the algorithm, $\texttt{dist}[v]$ is the length of the shortest v_s-v-path found so far, and $\texttt{p}[v]$ is the predecessor of v on that path. Additionally, the algorithm maintains a priority queue Q of vertices ordered by their *key*, which is the tentative distance. Initially, v_s is inserted into Q with key $\texttt{dist}[v_s] = 0$ and $\texttt{p}[v_s]$ is set to v_s. Then vertices are extracted from Q in increasing order of key. Each extracted vertex v is *settled* by *relaxing* its outgoing edges. An edge $e = (v, w) \in E$ is relaxed

by comparing the tentative distance $\texttt{dist}[w]$ to the distance $\texttt{dist}[v] + c(e)$ that is achieved by traversing e. If the latter is smaller, $\texttt{dist}[w]$ is updated, $\texttt{p}[w]$ is set to v, and Q is updated: If it already contains w, its key is changed to $\texttt{dist}[w]$. Otherwise, w is inserted with key $\texttt{dist}[w]$. Once the queue is empty, $\texttt{dist}[v]$ is the length of the shortest v_s-v-path for each vertex v. The path can be reconstructed by iteratively following the parent pointer $\texttt{p}[v]$ until v_s is reached.

The correctness of Dijkstra's algorithm follows from the *label-setting* property: Because the edge costs are non-negative, the smallest key in Q does not decrease throughout the execution of the algorithm. Thus, when a vertex v is settled, we know that $\texttt{dist}[v]$ is equal to the length of the shortest v_s-v-path. Dijkstra's algorithm can be adapted to solve the single-pair shortest path problem for a target vertex $v_t \in V$ by adding a *stopping criterion*: once v_t is settled, the search is stopped. Once again, the correctness follows from the label-setting property.

Dijkstra's algorithm has a running time in $\mathcal{O}(|V| \log |V| + |E|)$ if the priority queue is implemented with a Fibonacci heap [FT87]. In practice, implementations using k-ary heaps are faster [CGR96], although their worst-case running time is higher at $\mathcal{O}((|V| + |E|) \log |V|)$. In this thesis, we use an implementation with 4-ary heaps.

A speedup of approximately a factor of two can be achieved with bidirectional search [Dan63, Nic66]. In addition to the forward Dijkstra search from v_s with queue $\overrightarrow{Q}$ and tentative distances $\overrightarrow{\texttt{dist}}[\cdot]$, a backward search is run from v_t with queue $\overleftarrow{Q}$ and distances $\overleftarrow{\texttt{dist}}[\cdot]$. Different strategies have been proposed for alternating between the two searches, such as always continuing with the queue that currently has the smaller minimum key [Nic66]. In our implementation, they alternate after each settling step. To find the shortest path, the algorithm maintains a tentative overall distance μ, which is initialized with ∞. Whenever the forward or backward distance of a vertex v is updated, the tentative distance μ is set to the minimum of itself and $\overrightarrow{\texttt{dist}}[v] + \overleftarrow{\texttt{dist}}[v]$. Let $\underline{\kappa}(Q)$ denote the smallest key in a queue Q. Then the search can be stopped once $\underline{\kappa}(\overrightarrow{Q}) + \underline{\kappa}(\overleftarrow{Q}) \geq \mu$ holds.

Optimizing Multiple Criteria. Dijkstra's algorithm can be extended to Pareto-optimize multiple criteria [Han80, Mar84]. With k criteria, the edge cost function $c : E \to (\mathbb{R}_0^+)^k$ maps each edge e to a k-dimensional cost vector $c(e) = (c_1, \ldots, c_k)$. Given source and target vertices $v_s, v_t \in V$, the objective is to compute a Pareto set of v_s-v_t-paths.

Instead of a single tentative distance, each vertex v now stores a *bag* $B(v)$ of labels. Each label $\ell \in B(v)$ represents a path $P = \langle v_s, \ldots, w, v \rangle$ with the cost vector $c(\ell) = c(P)$. For path unpacking, the label also stores a pointer to the label representing the prefix $\langle v_s, \ldots, w \rangle$ in the bag $B(w)$ of the predecessor vertex w. Within a bag, no label may weakly dominate another label. When a new label is added to the bag, this invariant is upheld by comparing the label to all other labels in the bag and removing dominated ones.

The priority queue Q now operates on individual labels instead of vertices. The labels are ordered according to a key function $\kappa(\cdot)$ that assigns a scalar value to each label. The total ordering induced by the key function must be consistent with the partial ordering induced by strong Pareto dominance, i.e., if a label ℓ_1 is strongly dominated by another label ℓ_2, then $\kappa(\ell_1) > \kappa(\ell_2)$ must hold. Orderings with this property include lexicographical ordering

(i.e., initially comparing according to the first criterion, then according to the second one in case of equality, then the third one and so forth) or a linear combination of the criteria. When a label ℓ belonging to a vertex v is extracted from Q, it is settled in a similar manner to the single-criterion algorithm. Each outgoing edge $e = (v, w)$ is relaxed by creating a new label ℓ' with cost $c(\ell) + c(e)$. If ℓ' is not weakly dominated by any labels in the bag $B(w)$, it is added to $B(w)$, weakly dominated labels are removed and ℓ' is added to Q.

If the key function is consistent with Pareto dominance, then the algorithm remains label-setting in the sense that once a label is settled, it is known to be Pareto-optimal. However, it is no longer label-setting in the sense that each vertex is only settled once, since there may be multiple Pareto-optimal labels per vertex. As a result, the stopping criterion no longer works: once the first label at v_t is settled, the search cannot be stopped because the full Pareto set at v_t may not have been found yet. Instead, *target pruning* [DMS08] can be applied: When an edge is relaxed and a new label is created, the algorithm checks whether it is dominated by any of the labels in $B(v_t)$. If so, the label is discarded. This ensures that once a Pareto set at v_t has been found, no more labels will be added to Q.

To reduce the number of elements in the priority queue, our implementation uses the following optimization [Bau17]: Instead of individual labels, the global priority queue Q operates on vertices. The key of a vertex v is the smallest key among the unsettled labels in $B(v)$. To keep track of this, each vertex maintains a local priority queue of unsettled labels. In each step of the algorithm, the vertex v with the smallest key is settled by extracting the label with the smallest key from the local queue of v and settling it. Afterward, if $B(v)$ still contains unsettled labels, then the key of v is recalculated and v is reinserted into Q.

4.1.2 Contraction Hierarchies

We now present several variants of CH [GSSV12] that are used throughout this work.

Basic Algorithm. The basic building block of CH is *vertex contraction*: A vertex v is contracted by removing v and its incident edges from the graph and building an overlay graph for the vertex set $V \setminus \{v\}$. This requires inserting *shortcut edges* between the (former) neighbors of v in order to preserve shortest path distances. The simplest approach is to iterate over every pair of incoming edge $e_1 = (w, v)$ and outgoing edge $e_2 = (v, x)$ and insert the shortcut (w, x) with cost $c(e_1) + c(e_2)$. If such an edge already exists, its cost is set to the minimum of the shortcut cost and its previous cost. To avoid inserting superfluous shortcuts, a *witness search* can be performed, which is a bidirectional Dijkstra search from w to x. If this search finds a shorter w-x-path than the one via v, the shortcut is not inserted.

The CH preprocessing phase for a graph $G = (V, E)$ iteratively contracts the vertices in a heuristically determined order. This condenses the original graph into progressively smaller overlay graphs until the overlay finally becomes empty. The position of a vertex in this contraction order is called its *rank*. The final output is an *augmented graph* $G^+ = (V, E^+)$ containing E and all inserted shortcuts. The augmented graph can be split into an *upward graph* $G^\uparrow = (V, E^\uparrow)$ containing only edges from lower-ranked to higher-ranked vertices, and

a corresponding *downward graph* $G^{\downarrow} = (V, E^{\downarrow})$ with edges from higher- to lower-ranked vertices. Note that $G^{\uparrow}$ and $G^{\downarrow}$ are DAGs, with the contraction order (or the inverse contraction order for $G^{\downarrow}$) as one possible topological ordering.

For every pair v_s, v_t of source and target vertices, it can be shown that G^+ contains a shortest v_s-v_t-path that is an *up-down path*. This is a path that can be split into a prefix with all edges in $G^{\uparrow}$ and a suffix with all edges in $G^{\downarrow}$. To find an up-down path, the query algorithm performs a bidirectional Dijkstra search, in which the forward search explores $G^{\uparrow}$ and the backward search explores $G^{\downarrow}$. The stopping criterion is slightly different from a regular bidirectional search: the search is stopped once $\min(\underline{\kappa}(\overrightarrow{Q}), \underline{\kappa}(\overleftarrow{Q})) \geq \mu$ holds. To obtain a corresponding path in the original graph G, the shortcuts in the up-down path are *unpacked*. Each shortcut stores its *via vertex*, i.e., the vertex whose contraction created it. From this, the two edges that the shortcut represents can be reconstructed. The original path can then be obtained by recursively unpacking all shortcuts until only original edges remain.

Implementation Details. Our implementation of CH uses a greedy heuristic to determine the contraction order. It maintains a priority queue of the vertices in which the key represents the estimated importance. In each step, the vertex with minimal key is contracted and the keys of its neighbors are recalculated. Vertices with a degree of at most two are assigned the lowest possible importance, since contracting them decreases the number of edges in the graph. Otherwise, the key is a linear combination of the *edge difference* (weighted with factor 4) and the *level* (weighted with factor 1). The edge difference of a vertex v is the number of shortcuts that would be inserted if v were contracted next, divided by the number of edges that are currently incident to v. This value is determined by simulating the contraction of v. The level of a vertex is initially set to 0. When a vertex with level ℓ is contracted, the levels of its neighbors are set to the maximum of their current value and $\ell + 1$. This ensures that the height of $G^{\uparrow}$ and $G^{\downarrow}$ (i.e., the length of a longest path) remains low. To limit the time that is spent on witness searches, each search is terminated once 200 vertices have been settled. This may cause the search to miss shorter paths and insert superfluous shortcuts, but this does not affect the correctness of the query algorithm.

Core-CH. Multimodal algorithms, such as UCCH and MCR, employ a special variant of the CH precomputation that we call *Core-CH* [Bau+10, DPW15b, Del+13]. Here, the precomputation is not allowed to contract vertices that coincide with stops. As a consequence, the iterative contraction is interrupted at some point, leaving an uncontracted overlay graph $G^{\circ} = (V^{\circ}, E^{\circ})$ with $\mathcal{S} \subseteq V^{\circ} \subseteq V$, which is called the *core graph*. The augmented graph $G^+ = (V, E^+ \cup E^{\circ})$ is then formed by the edges in the core graph as well as the inserted shortcuts. The core edges E° are included in both the upward and downward graphs.

If we enforce $V^{\circ} = \mathcal{S}$, then $|E^{\circ}|$ becomes quadratic in $|\mathcal{S}|$. This slows down the precomputation and query algorithms to the point of being impractical. In practice, the contraction process is therefore stopped once the average vertex degree in the core graph surpasses a specified threshold. As a result, the core graph may contain vertices that are not stops.

Bucket-CH. Bucket-CH [Kno+07, GSSV12] is an extension of CH for one-to-many queries. It operates in three phases. First, given the graph $G = (V, E)$, the CH precomputation is performed. Second, given the set $V_t \subseteq V$ of targets, a *bucket* containing distances to the targets is computed for every vertex. This is done by performing a backward search in $G^{\downarrow}$ from every target vertex $v_t \in V_t$. For each vertex v settled by this search with distance $\texttt{dist}(v, v_t)$, the entry $(v_t, \texttt{dist}(v, v_t))$ is added to the bucket of v. Finally, given a query with source vertex v_s, the algorithm performs a forward search on $G^{\uparrow}$. For each vertex v settled by this search with distance $\texttt{dist}(v_s, v)$, the bucket of v is evaluated. For each bucket entry $(v_t, \texttt{dist}(v, v_t))$, the shortest distance to v_t found so far is compared with $\texttt{dist}(v_s, v) + \texttt{dist}(v, v_t)$ and updated if it is improved.

(R)PHAST. PHAST [DGNW13] extends CH for one-to-all queries. A PHAST query begins with an upward search from v_s in $G^{\uparrow}$. This is followed by a *downward sweep* that scans the vertices of $G^{\downarrow}$ in some topological order (e.g., the inverse contraction order). For each scanned vertex v and each incoming downward edge $e = (w, v) \in E^{\downarrow}$, the distance $\texttt{dist}(v)$ is set to the minimum of itself $\texttt{dist}(w) + \tau_{\text{tra}}(e)$. To make the downward sweep cache-efficient, the vertices of $G^{\downarrow}$ are stored in memory in the same order in which they are scanned. In a many-to-all scenario with multiple source vertices, the memory locality of PHAST can be further improved by combining k one-to-all searches into a single sweep (for a fixed k). Instead of one distance value per vertex, the algorithm then stores an array of k values, one for each of the k sources, which are updated consecutively during each edge relaxation.

For the one-to-many problem with a target set $V_t \subseteq V$ that does not change between queries, RPHAST improves on PHAST by performing a *target selection* phase before queries are run. This involves running a backward BFS on $G^{\downarrow}$, initializing the queue with all target vertices at once. The downward sweep is then run on the subgraph of $G^{\downarrow}$ induced by the vertices that were visited by the BFS.

4.2 Public Transit Networks

Unless otherwise noted, the algorithms in this section are designed for pure public transit networks with one-hop transfers, i.e., the transfer graph must be transitively closed and fulfill the triangle inequality.

4.2.1 Time-Expanded Graph

Although none of the algorithms in this thesis operate directly on the time-expanded graph, we describe its construction because it aids in understanding the structural features of public transit networks and how the timetable-based algorithms exploit them. There are multiple different definitions of the time-expanded graph in the literature. Ours is based on the *realistic time-expanded digraph* proposed by Pyrga et al. [PSWZ08], which was designed to incorporate minimum change times. The version presented here uses departure buffer times instead.

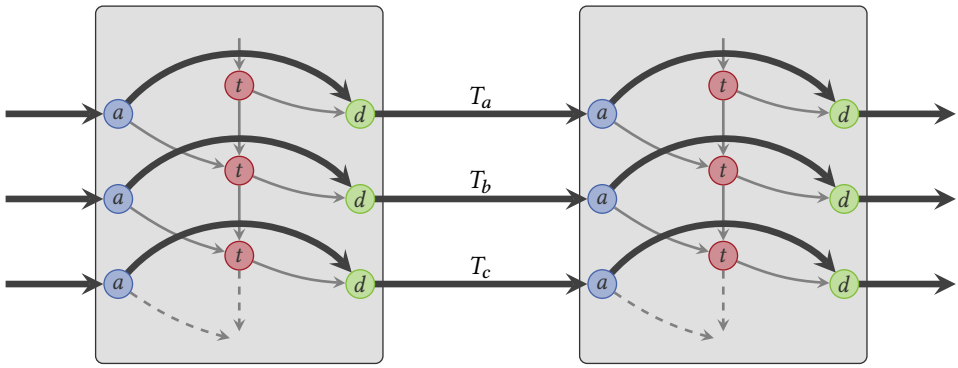


Figure 4.1: Modeling of a time-expanded graph for two example stops. Arrival nodes in blue, transfer nodes in red, and departure nodes in green. Stops are depicted as gray boxes enclosing the associated nodes. Trip edges are thick and in black, transfer edges are thin and in gray. Edge weights have been omitted for clarity.

An example of a time-expanded graph is shown in Figure 4.1. Each stop event is represented by three nodes: an *arrival node*, a *departure node*, and a *transfer node*. Associated with each node is its *event time*. For the arrival and departure node, this is the arrival and departure time of the stop event, respectively. For the transfer node, it is the departure time minus the departure buffer time of the stop. A trip is represented by a path of *trip edges* that alternate between the arrival and departure nodes of the corresponding stop events. Entering and exiting vehicles is modeled via *transfer edges*. Each transfer node has a transfer edge to the corresponding arrival node, which represents entering the vehicle. The transfer nodes of a stop are sorted in increasing order of event time and connected by a chain of transfer edges; these represent waiting at the stop for later trips. Finally, from each arrival node, a transfer edge leads to the earliest transfer node at the stop that can be reached while respecting the departure buffer time; this represents exiting the trip. The cost of an edge is the difference in the event times of the nodes that it connects.

An earliest arrival query with source stop v_s, target stop v_t and departure time τ_{dep} can be solved with Dijkstra's algorithm as follows. The source vertex is the earliest transfer node of v_s whose event time is not before τ_{dep}. There is no single target vertex; rather, the objective is to find the earliest reachable transfer node of v_t. This is achieved with a modified stopping criterion: as soon as the first transfer node of v_t is settled, the search is stopped.

Discussion. A major downside of the time-expanded graph model is that it obscures some structural features of public transit networks that can be used to prune the search. For example, when optimizing the two criteria arrival time and number of trips, it is sufficient to only explore the earliest reachable trip of each route, as using a later trip will not yield an improvement in either criterion. We refer to this insight as *route-based pruning*. Because it is not encoded in

the time-expanded graph, Dijkstra's algorithm also explores these redundant later trips. This is in contrast to the time-dependent model, which handles different trips of the same route via the time-dependent edge cost function. When a trip edge is explored and the function is evaluated, the earliest reachable trip is chosen automatically. Delling et al. [DPW09b] propose two approaches for employing route-based pruning in the time-expanded graph. The first is to identify and remove redundant arcs to later trips in a preprocessing step. The other approach, *node blocking*, is applied dynamically at query time. Whenever a departure node is settled, later departure nodes of the same trip are blocked, which means that their outgoing edges are no longer relaxed.

Another disadvantage of the time-expanded model is that footpaths between stops cannot be integrated in a straightforward manner. Müller-Hannemann et al. [MSWZ07] discuss two solutions for modeling a footpath from stop v to w: The naive approach is to create a copy of the footpath for each transfer node of v, which leads to the earliest reachable transfer node of w. The alternative is to find the first reachable transfer node at w on the fly whenever a transfer node of v is settled. In both cases, the footpath is explored anew every time a transfer node at v is settled, even though it would be sufficient to explore it only once when the first transfer node of v is reached. On the other hand, an advantage of the time-expanded graph is that it does not require one-hop transfers: because it is explored with Dijkstra's algorithm, footpaths consisting of multiple edges can be found without issues.

4.2.2 Connection Scan Algorithm

CSA [DPSW18] solves the one-to-one (or one-to-all), fixed departure time, earliest arrival problem. It exploits the fact that the time-expanded graph is a DAG and that the event time yields a topological ordering of the nodes. Hence, shortest paths can be found with a single sweep across the nodes in increasing order of event time, relaxing the outgoing edges of each node. This has two advantages over Dijkstra's algorithm: Firstly, it removes the need for a priority queue and the running time (excluding the sorting by event time, which is a preprocessing step) therefore becomes linear. More importantly, by reordering the nodes in memory according to the topological ordering, the sweep becomes highly cache-efficient.

CSA does not operate directly on the time-expanded graph. Rather, it uses its own data structures based on the notion of the *connection*, which is a 5-tuple

$$c := \big(v_{\text{dep}}(c), v_{\text{arr}}(c), \tau_{\text{dep}}(c), \tau_{\text{arr}}(c), T(c)\big).$$

It represents the trip $T(c)$ departing from stop $v_{\text{dep}}(c)$ at time $\tau_{\text{dep}}(c)$ and traveling to the next stop $v_{\text{arr}}(c)$, which is reached at time $\tau_{\text{arr}}(c)$. A connection is equivalent to a trip edge from a departure node to an arrival node in the time-expanded graph, and to a trip segment of length two. For a trip $T := \langle \varepsilon_0, \ldots, \varepsilon_k \rangle$, the associated set of connections is given by

$$\mathcal{C}(T) := \{\big(v(\varepsilon_i), v(\varepsilon_{i+1}), \tau_{\text{dep}}(\varepsilon_i), \tau_{\text{arr}}(\varepsilon_{i+1}), T\big) \mid 0 \leq i < k\}.$$

The set of all connections in the network is given by $\mathcal{C} := \bigcup_{T \in \mathcal{T}} \mathcal{C}(T)$. In a preprocessing step, the connections are sorted in ascending order of departure time.

The query algorithm uses two data structures: For each stop v, the tentative arrival time is given by $\tau_{\text{arr}}(v)$, which is initialized with ∞. Additionally, the algorithm maintains a boolean flag $r(T)$ (initialized with `false`) for each trip T, which indicates whether the search has reached T yet. A query with source stop v_s and departure time τ_{dep} starts by setting $\tau_{\text{arr}}(v_s)$ to τ_{dep} and relaxing all outgoing footpaths: for each edge $e = (v_s, v) \in E$, the arrival time $\tau_{\text{arr}}(v)$ is set to $\tau_{\text{dep}} + \tau_{\text{tra}}(e)$. Then, the earliest connection c_0 with $\tau_{\text{dep}}(c_0) \geq \tau_{\text{dep}}$ is found with a binary search. Starting from c_0, the connections in $\mathcal{C}$ are scanned in increasing order of departure time. A connection c is scanned as follows: First, the algorithm tests whether c is reachable. This is the case if its trip has already been reached, i.e., $r(T(c)) = \texttt{true}$, or if the search has reached its departure stop in time to enter it, i.e., $\tau_{\text{arr}}(v_{\text{dep}}(c)) \leq \tau_{\text{dep}}(c)$. If c is not reachable, it is skipped. Otherwise, the flag $r(T(c))$ is set to `true` and the arrival time $\tau_{\text{arr}}(c)$ is compared to the tentative arrival time $\tau_{\text{arr}}(v_{\text{arr}}(c))$ at the arrival stop. If the former is lower, then the latter is updated and all outgoing footpaths of $v_{\text{arr}}(c)$ are relaxed. In the one-to-one problem setting with a target stop v_t, the sweep is stopped once it reaches a connection c with $\tau_{\text{dep}}(c) \geq \tau_{\text{arr}}(v_t)$, since none of the following connections can improve the best found solution.

Discussion. The main advantage of CSA is that it is extremely simple and highly cache-efficient. This makes up for the fact that its search space is unoptimized, especially for local queries. If τ_{dep} is the departure time of the query and τ_{arr} is the arrival time of the fastest journey, then CSA scans all connections in the network that depart within $[\tau_{\text{dep}}, \tau_{\text{arr}}]$, even if they are far away from the source stop and therefore unreachable. Moreover, CSA does not apply any route-based pruning rules; a connection is scanned even if a corresponding connection of an earlier trip has already been reached. This lack of pruning rules becomes an issue in more complex problem settings. While it is possible to design a variant of CSA that Pareto-optimizes arrival time and number of trips, preliminary experiments have shown that it is not competitive with RAPTOR. This is mainly because the stopping criterion becomes weaker and therefore the search space becomes even larger.

Note that CSA only relaxes the outgoing footpaths of a stop v if its tentative arrival time was improved by the incoming connection. This solves the issue with time-expanded graphs, in which the same footpath is explored anew for each transfer node. However, this optimization requires one-hop transfers. Consider an outgoing edge (v, w). If the shortest path to v ends with an edge (x, v), then CSA may not relax (v, w). Therefore, in order to reach w, the algorithm requires the transitive edge (x, w).

4.2.3 RAPTOR

RAPTOR addresses the main shortcoming of CSA by only exploring parts of the network that are reachable from the source stop. It resembles a BFS in which the routes correspond to vertices and the transfers between them to edges. Because the depth of a node in the BFS tree equals the number of trips used to reach it, this approach naturally supports Pareto optimization for the two criteria arrival time and number of trips.

Algorithm 4.1: RAPTOR query algorithm.

Input: Public transit network $(\mathcal{S}, \Pi, \mathcal{E}, \mathcal{T}, \mathcal{R}, G)$, source stop v_s, departure time τ_{dep}, target stop v_t
Output: Earliest arrival time $\tau_{arr}(v_t, n)$ at v_t for each number of trips n

1 **for each** $v \in \mathcal{S}$ **do**
2 $\tau_{arr}(v, 0) \leftarrow \infty$
3 $\tau^*_{arr}(v) \leftarrow \infty$
4 $\tau_{arr}(v_s, 0) \leftarrow \tau_{dep}$
5 $\tau^*_{arr}(v_s) \leftarrow \tau_{dep}$
6 $\mathcal{S}' \leftarrow$ `RelaxTransfers`$(\{v_s\}, 0)$
7 **for** $n \leftarrow 1, 2, \ldots$ **do**
8 **if** $\mathcal{S}' = \emptyset$ **then break**
9 **for each** $v \in \mathcal{S}$ **do** $\tau_{arr}(v, n) \leftarrow \infty$
10 $\mathcal{R}' \leftarrow$ `CollectRoutes`$(\mathcal{S}')$
11 $\mathcal{S}' \leftarrow$ `ScanRoutes`$(\mathcal{R}', n)$
12 $\mathcal{S}' \leftarrow$ `RelaxTransfers`$(\mathcal{S}', n)$

Algorithm 4.2: RAPTOR transfer relaxation procedure.

1 **Procedure** `RelaxTransfers`$(\mathcal{S}', n)$
2 **for each** $v \in \mathcal{S}'$ **do**
3 **for each** $e = (v, w)$ **do**
4 $\tau \leftarrow \tau_{arr}(v, n) + \tau_{tra}(e)$
5 **if** $\tau < \min(\tau^*_{arr}(w), \tau^*_{arr}(v_t))$ **then**
6 $\tau_{arr}(w, n) \leftarrow \tau$
7 $\tau^*_{arr}(w) \leftarrow \tau$
8 $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{w\}$
9 **return** $\mathcal{S}'$

Basic Algorithm. The basic RAPTOR algorithm [DPW15a] solves the one-to-one (or one-to-all), fixed departure time, Pareto optimization problem. It operates in *rounds*, in which the n-th round finds journeys with n trips by appending an additional trip to journeys found in the previous round. For each stop $v \in \mathcal{S}$ and each round n, the algorithm maintains a *tentative arrival time* $\tau_{arr}(v, n)$, which is the earliest arrival time among all journeys to v with n trips found so far. Additionally, it maintains an *earliest arrival time* $\tau^*_{arr}(v)$ for each stop v, which is the minimum of the tentative arrival times across all rounds.

Pseudocode for a RAPTOR query with source stop v_s, target stop v_t and departure time τ_{dep} is given in Algorithm 4.1. Lines 1–5 initialize the tentative and earliest arrival times with τ_{dep} for v_s and ∞ otherwise. Afterward, the `RelaxTransfers` procedure (detailed in Algorithm 4.2)

is called in line 6 to relax the outgoing transfer edges of v_s. This returns a set $\mathcal{S}'$ of stops that were *marked* because their tentative arrival time was improved. The remainder of the algorithm (lines 7–12) performs rounds until $\mathcal{S}'$ is empty. At the start of round n, line 9 initializes the tentative arrival time $\tau_{\text{arr}}(v, n)$ of each stop v with ∞. Afterward, the algorithm performs a *route scanning phase*, consisting of the procedures `CollectRoutes` and `ScanRoutes` (detailed in Algorithm 4.3), followed by a *transfer relaxation phase* consisting of `RelaxTransfers`.

The procedure `CollectRoutes` collects the set $\mathcal{R}'$ of routes that visit a marked stop. For each route R, it also identifies the index of the first marked stop along R. For each collected tuple $(R, j) \in \mathcal{R}'$ with route R and index j, the procedure `ScanRoutes` *scans* R by iterating across its stops from index j onwards. While doing so, it rebuilds the set $\mathcal{S}'$ of marked stops for the next round. During the scan, the algorithm maintains an *active trip* T_{min}, which is the earliest trip of R that can be entered at any of the already processed stops. For each stop v along the route, two steps are performed. First, line 17–20 test whether exiting T_{min} improves the arrival time at v with n trips. Two pruning rules are applied here: *Local pruning* compares the arrival time $\tau_{\text{arr}}(T_{\text{min}}[i])$ to the earliest arrival time $\tau^*_{\text{arr}}(v)$ at v found so far. For one-to-one queries, target pruning additionally compares it to the earliest arrival time at v_t. If it is earlier than both, $\tau^*_{\text{arr}}(v)$ and $\tau_{\text{arr}}(v, n)$ are updated accordingly and v is marked. In the second step, lines 21–22 test whether an earlier trip than T_{min} can be entered at v when arriving with $n - 1$ trips. Line 21 checks whether the predecessor trip $\text{pred}(T_{\text{min}})$ can be entered at v when arriving at $\tau_{\text{arr}}(v, n - 1)$. If so, the procedure `FindEarliestTripFrom` finds the earliest reachable trip with a backward linear search through the trips of R, starting with $\text{pred}(T_{\text{min}})$. Once all collected routes have been scanned, the procedure `RelaxTransfers` is called. For every marked stop $v \in \mathcal{S}'$, each outgoing transfer edge $e = (v, w) \in E$ is relaxed. Once again, local and target pruning are applied to test whether the arrival time $\tau_{\text{arr}}(v, n) + \tau_{\text{tra}}(e)$ is an improvement. If so, then $\tau^*_{\text{arr}}(w)$ and $\tau_{\text{arr}}(w, n)$ are updated and w is marked.

As outlined thus far, RAPTOR only computes the earliest arrival time $\tau_{\text{arr}}(v, n)$ for each stop v and number of trips n. To retrieve the corresponding journeys, this is augmented with a parent pointer $\mathtt{p}(v, n)$. If v was reached via a transfer (w, v), this points to w. If v was reached via a trip segment $T[i, j]$, it points to the stop $v(T[i])$ at which the trip segment was entered. Additionally, the algorithm stores a boolean flag that indicates whether v was reached via a transfer or a trip segment. The journey can then be retrieved by repeatedly unwinding the parent pointers, decreasing the number of trips if the stop was reached via a trip segment.

Discussion. Unlike CSA, RAPTOR implements a route-based pruning rule by maintaining the active trip T_{min} during the route scan and updating it via `FindEarliestTripFrom`. To make the route scanning procedure as cache-efficient as possible, a special memory layout is used for the arrival and departure times of the stop events. For each route R with k trips and $|R|$ stops, they are stored in an array $\mathtt{StopTimes}[\cdot]$ of size $k \cdot |R|$, where index $\mathtt{StopTimes}[i \cdot |R| + j]$ contains the arrival and departure times of the j-th stop event along the i-th trip of R. Advancing to the next stop along the route is then done by stepping one index to the right, whereas switching to the next-earliest trip is done by stepping back $|R|$ entries to the left.

Algorithm 4.3: RAPTOR route collection and scanning procedures.

```
1  Procedure CollectRoutes(S')
2      R' ← ∅
3      for each v ∈ S' do
4          for each route R ∈ R visiting v do
5              i ← index of v in R
6              if R' contains (R, j) then
7                  R' ← R' \ {(R, j)} ∪ {(R, min(i, j))}
8              else
9                  R' ← R' ∪ {(R, i)}
10     return R'
11 Procedure ScanRoutes(R', n)
12     S' ← ∅
13     for each (R, j) ∈ R' do
14         T_min ← ⊥
15         for i from j to |R| − 1 do
16             v ← i-th stop of R
17             if T_min ≠ ⊥ and τ_arr(T_min[i]) < min(τ*_arr(v), τ*_arr(v_t)) then
18                 τ_arr(v, n) ← τ_arr(T_min[i])
19                 τ*_arr(v) ← τ_arr(T_min[i])
20                 S' ← S' ∪ {v}
21             if pred(T_min[i]) ≠ ⊥ and τ_arr(v, n − 1) ≤ τ_dep(pred(T_min)[i]) then
22                 T_min ← FindEarliestTripFrom(pred(T_min), i, τ_arr(v, n − 1))
23     return S'
```

Additional Criteria. McRAPTOR [DPW15a] extends RAPTOR to Pareto-optimize an arbitrary number of additional criteria besides arrival time and number of trips. For each stop v and round n, the tentative arrival time $\tau_{\text{arr}}(v, n)$ is replaced with a *bag* $B^n(v)$ of labels that represent Pareto-optimal journeys to v with n trips. Likewise, the earliest arrival time $\tau^*_{\text{arr}}(v)$ is replaced with a *best bag* $B^*(v)$, which contains all labels at v that are Pareto-optimal if the number of trips ignored as a criterion. When a new label is found at a stop v in round n, it is compared with the labels in $B^*(v)$. If it is not dominated, it is merged into $B^*(v)$ and $B^n(v)$, removing labels that are dominated by it.

During the scan of a route R, the algorithm maintains a *route bag* B_{route}, which contains labels representing journeys that end with a trip of R. Instead of one active trip for the entire route, each label in B_{route} maintains its own active trip, which is the trip of R used by the corresponding journey. When the route scan visits a stop v, journeys exiting the route at v are found by merging B_{route} into $B^n(v)$. Then, for each label in $B^{n-1}(v)$, the algorithm finds the earliest trip T that can be entered at v, creates a label with active trip T and merges it

into B_{route}. Note that this step still employs route-based pruning. This is only correct if the additional criteria conform to the condition that it is never useful to enter a later trip than the earliest reachable one. Criteria for which this is not the case (e.g., vehicle occupancy) are not handled correctly by McRAPTOR.

Multimodal Networks. In turn, McRAPTOR can be extended to support multimodal scenarios with unlimited transfers. The resulting algorithm, MCR [Del+13], replaces the transfer relaxation phase of (Mc)RAPTOR with a Dijkstra search on a core graph $G^{\text{o}} = (V^{\text{o}}, E^{\text{o}})$ computed with Core-CH. The two-criteria variant of MCR was originally proposed under the name MR-∞. We refer to it as MR (Multimodal RAPTOR) for the sake of simplicity. MR/MCR maintains arrival times or bags for every vertex in V^{o}, not just for stops. In MR, the transfer relaxation phase for round n runs Dijkstra's algorithm on the core graph, using $\tau_{\text{arr}}(\cdot, n)$ as the tentative distances. MCR does the same with the multicriteria variant of Dijkstra's algorithm and the bags $B(\cdot, n)$. The priority queue is initialized with all marked stops, and all stops that are settled by the search are themselves marked. Note that the Dijkstra search on the core graph can only guarantee finding shortest paths between pairs of stops. Because the source and target vertices are not necessarily stops themselves, initial and final transfers are explored with searches on the upward and downward graph produced by Core-CH, respectively.

Profile Search. Profile queries can be answered with another RAPTOR extension, rRAPTOR (Range RAPTOR) [DPW15a]. rRAPTOR exploits the observation that every Pareto-optimal journey (except for a direct transfer from v_{s} to v_{t}) starts by entering a trip at v_{s} or a stop reachable via a transfer from v_{s}. This limits the number of possible departure times to a small set of discrete values. For each of these departure times, rRAPTOR performs a *run* of the basic RAPTOR algorithm. The departure times are processed in descending order, and the arrival times $\tau_{\text{arr}}(\cdot, \cdot)$ are not reset between runs. As a result, journeys found during the current run are implicitly pruned by journeys that depart later and neither arrive later nor have more trips. This property is called *self-pruning*. The output of rRAPTOR can be interpreted as a Pareto set with departure time as a third criterion that is maximized.

4.2.4 Trip-Based Routing

TB is a faster alternative to RAPTOR for one-to-one queries [Wit15]. It features a preprocessing phase that computes a set $E^{\text{s}} \subseteq \mathcal{E} \times \mathcal{E}$ of relevant intermediate transfers between pairs of stop events. These transfers are used to speed up the query phase. Unlike in RAPTOR, it is no longer necessary to identify the earliest reachable trip when entering a route, since the precomputed transfers already provide that information. TB exploits this by scanning individual trip segments instead of entire routes.

Transfer Precomputation. Initially, the preprocessing phase generates all transfers $e = (T_a[i], T_b[j])$ with $i > 0$ and $j < |T_b| - 1$ such that T_b is the earliest trip of its route that can

Algorithm 4.4: TB query algorithm.

Input: Public transit network $(\mathcal{S}, \Pi, \mathcal{E}, \mathcal{T}, \mathcal{R}, G)$, event-to-event transfers E^{s}, source stop v_{s}, departure time τ_{dep}, target stop v_{t}

Output: Labels $\mathcal{L}$ representing Pareto set of v_{s}-v_{t}-journeys for departure time τ_{dep}

```
1  Initialize()
2  RelaxInitialTransfers()
3  τ_min ← τ_dep + τ_tra(v_s, v_t)
4  if τ_min < ∞ then L ← {(τ_min, 0)}
5  Q_1 ← ∅
6  CollectInitialTrips(Q_1)
7  for n ← 1, 2, ... do
8      if Q_n = ∅ then break
9      Q_{n+1} ← ∅
10     ScanTrips(Q_n, Q_{n+1})
```

be entered at $v(T_b[j])$ when arriving at $\tau_{\mathrm{arr}}(T_a[i]) + \tau_{\mathrm{tra}}(v(T_a[i]), v(T_b[j]))$. An exception to this is if T_a and T_b belong to the same route, $T_a \preceq T_b$ and $i \leq j$, In this case, it is preferable to remain seated in T_a, so e is not generated.

Afterward, two reduction rules are applied to discard unnecessary transfers. These are not exhaustive, so E^{s} may still contain transfers that do not occur in any Pareto-optimal journey. The U-turn reduction rule removes transfers $e = (T_a[i], T_b[j])$ with $v(T_a[i-1]) = v(T_b[j+1])$ and $\tau_{\mathrm{arr}}(T_a[i-1]) \leq \tau_{\mathrm{arr}}(T_b[j+1])$. The second reduction rule removes transfers that can be replaced with another outgoing transfer from the same trip. For each trip T, the algorithm scans its stop events in reverse order and maintains an earliest arrival time $\tau_{\mathrm{arr}}(v)$ at each stop v, which is initialized with ∞. For each stop event $T[i]$, the outgoing footpaths are relaxed and the arrival times of the reached stops are updated accordingly. Afterward, the outgoing transfers are relaxed. For each transfer $e = (T[i], T_b[j])$, the algorithm scans the trip T_b starting from index j. At each reached stop v, the arrival time of v is updated and the outgoing footpaths are relaxed. If this improves the arrival time of any stop, the transfer e is kept; otherwise, it is discarded.

Lehoux and Loiodice [LL20] propose an alternative transfer generation step that avoids generating some superfluous transfers, thereby improving the preprocessing time. If a transfer $(T_a[i], T_b[j])$ exists, then all other transfers of the form $(T_a[k], T_c[\ell])$ with $k \leq i$, $\ell \geq j$ and $T_c \succeq T_b$ are not generated.

Query Algorithm. Pseudocode for the TB query algorithm is given in Algorithms 4.4 (outline) as well as 4.5 and 4.6 (details). The procedure `Initialize` (lines 1–7 of Algorithm 4.5) initializes the used data structures. Instead of tentative arrival times at stops, TB maintains a *reached index* $r(T)$ for each trip T. This is the index k of the first stop event $T[k]$ that has been

Algorithm 4.5: TB initialization procedures.

```
1  Procedure Initialize()
2      for each v ∈ S do
3          τ_tra(v_s, v) ← ∞
4          τ_tra(v, v_t) ← ∞
5      for each T ∈ 𝒯 do
6          r(T) ← |T|
7      ℒ ← ∅
8  Procedure RelaxInitialTransfers()
9      τ_tra(v_s, v_s) ← 0
10     τ_tra(v_t, v_t) ← 0
11     for each e = (v_s, v) ∈ E do τ_tra(v_s, v) ← τ_tra(e)
12     for each e = (v, v_t) ∈ E do τ_tra(v, v_t) ← τ_tra(e)
```

reached by the search. Initially, it is set to $|T|$. Additionally, TB maintains a set $\mathcal{L}$ of labels representing Pareto-optimal journeys at the target stop v_t. The procedure `RelaxInitialTransfers` (lines 8–12) relaxes the outgoing transfers of the source stop v_s and the incoming transfers of v_t. For every reached stop v, this yields the minimum transfer times $\tau_{tra}(v_s, v)$ for an initial transfer from v_s and $\tau_{tra}(v, v_t)$ for a final transfer to v_t. Line 3 of Algorithm 4.4 initializes the earliest arrival time τ_{min} at v_t found so far. Initially, this represents the direct transfer from v_s to v_t. If $\tau_{min} < \infty$, a label representing this journey is added to the result set $\mathcal{L}$ in line 4.

Like RAPTOR, TB operates in rounds. Each round scans trip segments collected in a first-in, first-out (FIFO) queue. The procedure `CollectInitialTrips` (lines 1–8 of Algorithm 4.6) fills the queue Q_1 for the first round by processing stops that are reachable from v_s with an initial transfer. For each stop v and each route R visiting v, the algorithm finds the earliest trip T of R that can be entered at v. To find the corresponding trip segment and add it to Q_1, the `Enqueue` procedure is called for the first stop event at which T can be exited. If v is the i-th stop of R, this is $T[i+1]$. Pseudocode for the `Enqueue` procedure of a stop event $T[j]$ is shown in lines 21–26. If $r(T) \leq j$, the trip has already been reached from index j onwards. Otherwise, the trip segment $T[j, r(T)-1]$ is added to the queue for the next round. Then, for each trip T' of the route $R(T)$ that does not depart before T, the reached index $r(T')$ is set to $\min(r(T'), j)$.

Once the first queue has been filled, TB performs rounds until the current queue is empty. For round n, the `ScanTrips` procedure (lines 9–20 of Algorithm 4.6) scans all trip segments in Q_n and adds newly reached ones to Q_{n+1}. A trip segment $T[j,k]$ is scanned by iterating over the stop events from $T[j]$ to $T[k]$. For each stop event $T[i]$, this requires two steps: Lines 12–15 check whether exiting at $T[i]$ and taking a final transfer (if necessary) to v_t improves the best journey with at most n trips found so far. This is the case if the corresponding arrival time is smaller than τ_{min}. If so, τ_{min} is updated and a label representing the newly found journey is added to the result set $\mathcal{L}$. If $\mathcal{L}$ already contains a label with n trips (note that

Algorithm 4.6: TB trip collection, scanning and enqueuing procedures.

```
1  Procedure CollectInitialTrips(Q_1)
2      S' ← {v_s}
3      for each e = (v_s, v) ∈ E do S' ← S' ∪ {v}
4      for each v ∈ S' do
5          for each route R ∈ R visiting v do
6              i ← index of v in R
7              T ← FindEarliestTrip(R, i, τ_dep + τ_tra(v_s, v))
8              if T ≠ ⊥ then Enqueue(T[i + 1], Q_1)
9  Procedure ScanTrips(Q_n, Q_{n+1})
10     for each T[j, k] ∈ Q_n do
11         for i from j to k do
12             if τ_arr(T[i]) ≥ τ_min then break
13             if τ_arr(T[i]) + τ_tra(v(T[i]), v_t) < τ_min then
14                 τ_min ← τ_arr(T[i]) + τ_tra(v(T[i]), v_t)
15                 L ← L ∪ {(τ_min, n)}, removing dominated labels
16     for each T[j, k] ∈ Q_n do
17         for i from j to k do
18             if τ_arr(T[i]) ≥ τ_min then break
19             for each (T[i], T'[i']) ∈ E^s do
20                 Enqueue(T'[i' + 1], Q_{n+1})
21 Procedure Enqueue(T[j], Q)
22     if r(T) ≤ j then return
23     Q ← Q ∪ {T[j, r(T) − 1]}
24     for each T' ≥ T do
25         if r(T') ≤ j then break
26         r(T') ← j
```

a Pareto set can only contain one such label), this label is replaced. The second step is to relax the outgoing intermediate transfers of $T[i]$, which is done in lines 18–20. For each transfer $(T[i], T'[i']) \in E^s$, the `Enqueue` procedure is called for $T'[i' + 1]$, which adds the relevant segment of T' to Q_{n+1}. Target pruning is applied during both trip segment scans (see lines 12 and 18): if the arrival time of $T[i]$ is not earlier than $\tau_{\min}$, then the remainder of the trip segment is skipped since it cannot produce a journey that improves upon $\tau_{\min}$.

Note that the trip segments in Q_n are scanned twice: once to evaluate the final transfers and then again to relax intermediate transfers. This is done with two scans instead of one for two reasons. First, it improves memory locality because $\tau_{\text{tra}}(\cdot, v_t)$ is only accessed by the first scan and E^s is only accessed by the second scan. Second, because $\tau_{\min}$ is improved throughout the first scan, the target pruning check in line 18 becomes stricter.

Memory Layout. The TB query algorithm relies on a streamlined memory layout to achieve its performance. In the network instances used throughout this thesis, all trips have fewer than 2^8 stops. Accordingly, we use 8 bits to store the reached index of a trip. The FIFO queues Q_n are combined into a single array whose size is preallocated to the number $|\mathcal{E}|$ of stop events. This ensures that the array does not need to be reallocated during an `Enqueue` call. The size of $|\mathcal{E}|$ cannot be exceeded because the reached indices ensure that the algorithm does not enqueue multiple trip segments with the same initial stop event. During each round, pointers are used to track the position of the first and last element in the current queue.

The intermediate transfers E^{s} are stored in an array such that all outgoing transfers of a stop event $T[i]$ are consecutive in memory and the outgoing transfers of the next stop event $T[i+1]$ follow directly afterward. As a result, the outgoing transfers of a trip segment form a continuous range. To exploit this, the for-loop in lines 16–20 of Algorithm 4.6 is split into two parts. The first part evaluates the target pruning condition from line 18 for each trip segment $T[j,k]$. The result is a (potentially shorter) trip segment $T[j,k']$ with $k' \leq k$. The second part relaxes all outgoing transfers of $T[j,k']$. Since these are consecutive in memory, this can be done with a single for-loop instead of two nested for-loops.

Finally, note that the trip scanning step only needs access to the arrival time $\tau_{\text{arr}}(T[i])$ and the stop $v(T[i])$ of a stop event $T[i]$. Therefore, these values are stored separately from the departure time $\tau_{\text{dep}}(T[i])$ of the stop event, which improves memory locality.

Journey Retrieval. To retrieve journey descriptions, each trip segment in a queue Q_n stores a pointer to the trip segment in Q_{n-1} from which it was reached. Because the queues are combined into a single array, this pointer can be implemented as an index into the array. Likewise, each target label in $\mathcal{L}$ stores a pointer to the last trip segment in the corresponding journey. The sequence of scanned trip segments can be reconstructed by repeatedly following these pointers. What remains to be done is to identify for each trip segment the index at which it is exited to reach the next trip segment (or the target). This is done by rescanning the segment and searching for a matching outgoing transfer.

Discussion. Unlike RAPTOR, TB does not maintain any arrival times at stops in order to perform local pruning. This is because the reduction in the search space gained by local pruning does not outweigh the additional effort of maintaining these data structures. In RAPTOR, local pruning at a stop v serves two purposes: If v is reached via a route but the arrival time is not improved, it prevents the unnecessary relaxation of its outgoing transfers. If it is reached via a transfer without improving the arrival time, then local pruning ensures that the routes visiting v are not unnecessarily scanned in the next round. In TB, the latter purpose is already achieved by the reached indices. Besides providing a route-based pruning rule, they also ensure that no stop event is scanned more than once, which is not the case in RAPTOR. This only leaves the unnecessary exploration of transfers. However, because most irrelevant transfers are already pruned in the precomputation step, the potential savings from this are small.

Table 4.1: Sizes of the public transit networks and the accompanying transfer graphs. Also reported is the number of edges in the transitively closed transfer graphs, which are used for comparisons with unimodal algorithms.

	Stuttgart	London	Switzerland	Germany
Stops	13 584	19 682	25 125	243 167
Routes	12 351	1 955	13 786	230 255
Trips	91 304	114 508	350 006	2 381 394
Stop events	1 561 972	4 508 644	4 686 865	48 380 936
Transfer graph vertices	1 166 604	181 642	603 691	6 870 496
Transfer graph edges	3 682 232	575 364	1 853 260	21 367 044
Transitive edges (4.5 km/h)	1 369 928	3 212 206	2 639 402	22 571 280
Transitive edges (15 km/h)	1 558 234	2 374 294	2 432 366	20 057 494

4.3 Experimental Setup

To evaluate the algorithms presented in this thesis, we implemented them in C++17 and measured their performance on four real-world multimodal networks representing Stuttgart, London, Switzerland, and Germany. This section gives an overview of the network sources, how the networks were prepared and how the experiments were conducted.

Network Sources. An overview of the four networks is given in Table 4.1. The Stuttgart instance was originally built as part of a macroscopic traffic model [SHP11]; it was previously used to study travel demand modeling [MKV13] and traffic assignment [Bri+17]. Although it has not been used to evaluate journey planning algorithms so far, we include it because it was used in the evaluation of an ULTRA-based traffic assignment algorithm [SWZ19a]. The network contains local transport in the Stuttgart metropolitan area as well as long-distance trains within the greater Stuttgart region, which comprises most of the state of Baden-Württemberg. It covers the schedule of two successive identical business days; the second day is included in order to capture overnight journeys.

The other three networks have all been previously used to evaluate journey planning algorithms; see Table 4.2 for an overview. The public transit timetable of Greater London was obtained from Transport for London[1]. Unlike the other networks, which encompass two days, the London network only covers a single Tuesday in the periodic summer schedule of 2011. We decided not to add a second day to allow for easier comparisons with experiments in other publications. The Switzerland network was extracted from a publicly available GTFS (General Transit Feed Specification) feed[2] and comprises two successive business days (30th and 31st

[1] `https://data.london.gov.uk`
[2] `http://gtfs.geops.ch/`

Table 4.2: List of algorithms that were previously evaluated on our benchmark networks. Additionally, the Switzerland and Germany networks were used for the unrestricted walking experiments by Wagner and Zündorf [WZ17]. Note that the methods of network preparation differ slightly between publications, so the reported network sizes are not always identical.

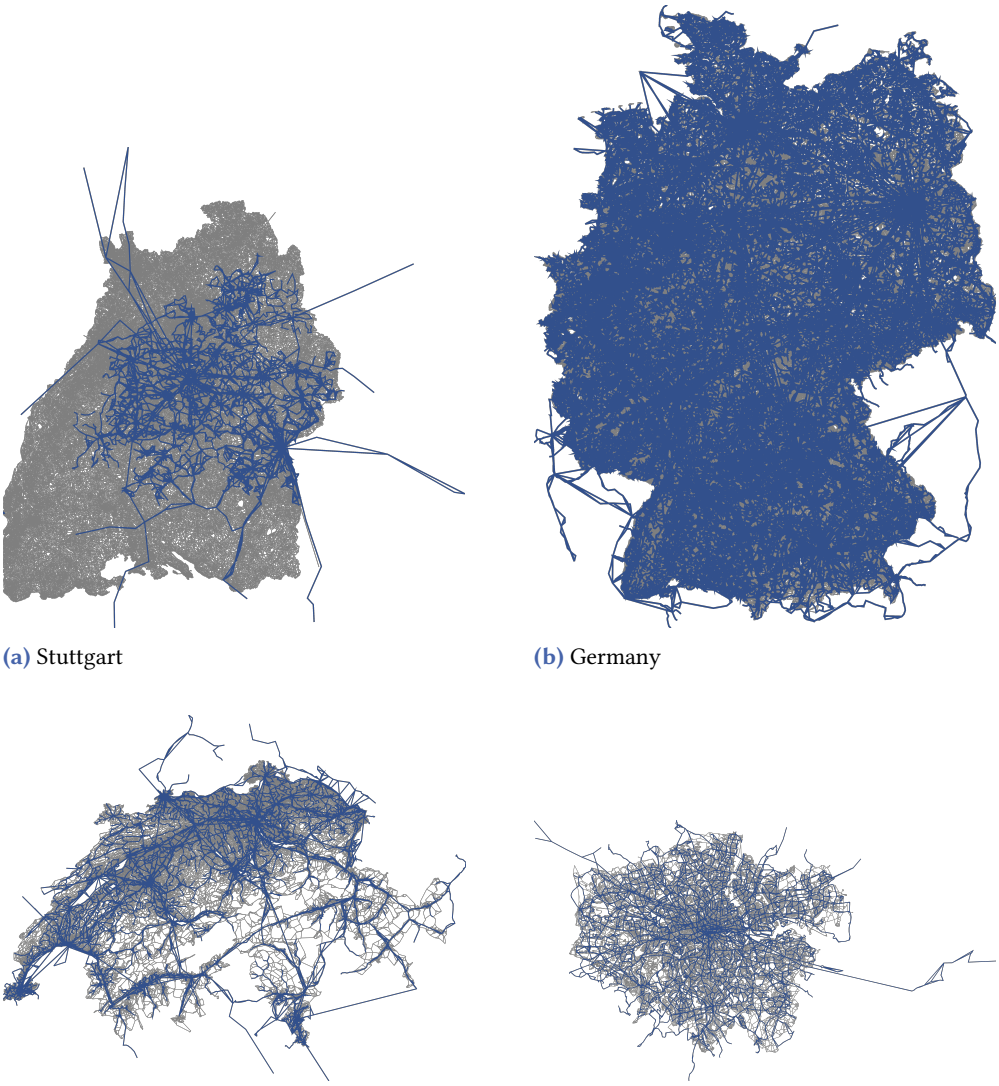
	London	Switzerland	Germany
Time-dependent Dijkstra	[DPW15a, DPSW18]	–	[DPSW18]
Time-expanded Dijkstra	[Bas+16, DPSW18]	–	[DPSW18]
Connection Scan Algorithm	[DPSW18]	–	[DPSW18]
RAPTOR	[DPW15a, DPSW18]	[DDPZ17]	[DPSW18]
McRAPTOR	[DPW15a]	–	–
Trip-Based Routing	[Wit15]	–	[Wit15]
Connection Scan Accelerated	[DPSW18]	–	[DPSW18]
HypRAPTOR	–	[DDPZ17]	–
TB-CST	[Wit16]	[Wit16]	[Wit16]
Transfer Patterns	–	[Bas+10]	[BS14]
Scalable Transfer Patterns	–	–	[BHS16]
Public Transit Labeling	[DDPW15]	[DDPW15]	–
MCR	[Del+13]	–	–
HL-RAPTOR/HL-CSA	[PV19]	[PV19]	–

of May, 2017). Finally, the Germany network was kindly provided to us by Deutsche Bahn for research purposes. It is based on data from `bahn.de` for Winter 2011/2012, comprising two successive identical days. To ensure comparability with experiments on the same network in other works, both days include all trips that are listed in the timetable, regardless of their actual days of operation. Note that some older publications (e.g., [BDGM09, DPW15a]) use much smaller Germany instances that only include trains or even only long-distance trains; our network covers most public transit in Germany, including regional trains and local transport.

Unrestricted transfer graphs for all four networks were obtained by extracting road graphs, including pedestrian zones and staircases, from OpenStreetMap (OSM)[3]. To aid reproducibility, we make the London and Switzerland networks publicly available[4]. Unfortunately, we cannot provide the Germany and Stuttgart networks as they are based on proprietary data.

The combined multimodal networks are depicted in Figure 4.2. Our selection of networks intentionally covers a range of different sizes and structural features. We note that the complexity of a network does not depend only on its geographical size; other important factors include service frequency and how the density of the public transit network compares to that of the road network. For example, the London network represents a dense metropolitan

[3] `https://download.geofabrik.de/`
[4] `https://i11www.iti.kit.edu/PublicTransitData/ULTRA/`



(a) Stuttgart

(b) Germany

(c) Switzerland

(d) London

Figure 4.2: The four networks used in our experiments. Relative sizes of the networks are not to scale. Road networks are depicted in gray, public transit routes in blue.

area with a high service frequency. The Stuttgart network has a smaller metropolitan area in its center but also covers much of the state of Baden-Württemberg, including many rural areas. Furthermore, because the network does not cover local transport outside of the center, many regions are only reachable via the transfer graph. This makes the network a useful example of scenarios in which public transit availability is limited. The Germany network is by far the largest of the four and contains a mixture of densely and sparsely populated areas. Finally, the Switzerland network represents a mid-sized country that is very inhomogeneous, with densely connected areas in the north and very remote Alpine regions in the south.

Network Preparation. To turn the input data into usable multimodal networks, we mostly follow the procedure outlined by Wagner and Zündorf [WZ17]. Minimum change times for each stop are supplied in the input data. We interpret these as departure buffer times by reducing the departure times of the associated stop events, as outlined in Chapter 2. Because the groupings of the trips into routes are not part of the input data, we compute them ourselves with a greedy approach that iterates across the set $\mathcal{T}$ of trips: For each trip T, we check if a route R with the same stop sequence as T has already been generated such that T does not overtake any trips in R and is not overtaken by any of them itself. If so, we add T to R. Otherwise, we generate a new route for T. Note that this approach does not generate a minimal number of routes for all possible inputs. A minimal set of routes can be found in polynomial time [Ste23]. However, because few trips actually overtake each other in practice, the greedy approach produces optimal results on all four benchmark networks.

To represent different transfer modes, we use the same transfer graph but vary the travel speed. For the sake of simplicity, we assume that every transfer mode is able to traverse all edges. While this assumption is not realistic (e.g., cars may not enter pedestrian zones), it is sufficient for our primary objective, which is to test the limits of the shortcut hypothesis. If usage of a mode is restricted to certain areas, this tends to decrease the number of intermediate transfers in which it can be used and thereby the number of required shortcuts. The OSM input data specifies the geographical distance and speed limit associated with each edge. From these, we calculate the travel time by assuming a constant speed along all edges, which is only reduced to obey the speed limit (unless otherwise noted). For roads without a speed limit, we assume a maximum speed of 140 km/h. In most experiments, we assume that the transfer mode represents walking, for which we use a speed of 4.5 km/h. For bicycles or e-scooters, we use a speed of 15 km/h.

We use the following procedure to connect the public transit network with the OSM transfer graph. For each stop $v \in \mathcal{S}$, we locate its (geographically) nearest neighbor $w \in V$ in the transfer graph. If v and w are less than five meters apart and v is also the nearest neighbor of w, we identify v with w. Otherwise, we add a new vertex for v. Additionally, if the two vertices are less than 100 meters apart, we insert edges (v, w) and (w, v) to connect them. Note that the input data for the public transit networks already contains some transfer edges between stops, which we retain. These mostly represent footpaths between stops that model different platforms of the same station or are otherwise very close to each other.

After the two networks have been connected, we contract all vertices with degrees one and two except for those that coincide with stops. These vertices are mainly included for visualization purposes. For routing purposes, they are superfluous, except as source and target locations of queries. Removing them is a common practice for graphs sourced from OSM. This allows for a more accurate comparison to graphs obtained from other sources, which often include fewer of these vertices [DSW15]. Note that in order to obtain accurate travel times, they must be computed before this contraction step. Finally, we remove remote and isolated parts of the network by applying a bounding box and removing everything except the largest connected component.

One-Hop Transfers. To evaluate whether our algorithms achieve the goal of closing the multimodal performance gap, we benchmark them against existing public transit algorithms, which require one-hop transfers. A perfect comparison is not possible because the two problem settings are different. Most obviously, a multimodal network allows for queries between a larger selection of source and target locations. However, even if the source and target vertices are limited to stops, the number of possible transfers between stops impacts both the query performance and the result size. If more transfers are included, the search space and therefore the query time increases, but so does the number of Pareto-optimal journeys. In order to make the comparison as fair as possible, we evaluate the public transit algorithms on one-hop transfer graphs that are as large as possible without sacrificing too much query speed. We follow the methodology of Wagner and Zündorf [WZ17], which is to insert edges between all stops whose distance in the unrestricted transfer graph lies below a certain threshold and then compute the transitive closure. The threshold is chosen so that the resulting graph has an average vertex degree of approximately 100. This represents a good tradeoff between preserving as many transfers as possible and keeping query times reasonable. For walking, this yields limits of 9 minutes for Stuttgart and Switzerland, 8 minutes for Germany, and 4 minutes for London. For bicycles and scooters, the limits are much lower at 180 s for Stuttgart and Switzerland, 150 s for Germany, and 80 s for London. This is hardly enough to allow for realistic usage of these faster transfer modes. The numbers of edges in the resulting transitively closed graphs are listed in Table 4.1.

Performance Measurements. All algorithms were implemented in C++17 and compiled with GCC, using the optimization flag -O3. Parallelization was done with OpenMP. The source code for all algorithms is publicly available at `https://github.com/kit-algo/ULTRA`. Experiments were performed on the following machines:

Xeon: A machine with two 8-core Intel Xeon Skylake SP Gold 6144 CPUs clocked at 3.50 GHz, with a boost frequency of 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 clocked at 2.25 GHz, with a boost frequency of 3.4 GHz, 1024 GiB of DDR4-3200 RAM, and 256 MiB of L3 cache.

Generally, experiments that benefit from heavy parallelization were run on the Epyc machine, whereas the Xeon machine was used for experiments that focus on single-core performance (including all queries) and those that only require light parallelization. Throughout this thesis, we indicate the machine used for each experiment.

Unless otherwise noted, we measure the performance of our algorithms on randomly generated queries. For these, we choose the source and target vertices uniformly at random, and the departure time uniformly at random within the first day covered by the timetable. Note that the resulting queries do not match realistic usage patterns: they are skewed towards sparsely populated areas and off-peak hours. This is a deliberate choice. One of the advantages of multimodal transportation systems is that they can serve rare travel requests more efficiently than any single mode can on its own. However, because today's transportation systems still have limited multimodal capabilities, current usage patterns mostly reflect unimodal transport. Queries for which multimodal transport would be useful are often those that are cumbersome to complete with public transit alone because they fall into areas or times of day for which service is low. Currently, passengers often use their private car for these. Therefore, queries based on real public transit usage patterns tend to be those that benefit the least from the presence of an unrestricted transfer mode. By contrast, uniformly random queries are more challenging to answer and require combinations of different modes more frequently.

5 ULTRA: UnLimited TRAnsfers for Efficient Multimodal Journey Planning

In order to study the shortcut hypothesis and exploit it algorithmically, we start with the simplest problem setting that can still be considered multimodal. In addition to public transit, we allow for one unrestricted transfer mode, which may represent any road-based mode of transport (e.g., walking, e-scooter, car). The objective is to answer one-to-one queries, either minimizing the arrival time or Pareto-optimizing the arrival time and the number of trips.

Our main baseline algorithm for this scenario is MR, the two-criteria variant of MCR. As the experiments in Section 5.3 will show, MR is about two to three times slower on most networks than RAPTOR with the one-hop transfer graphs constructed in Chapter 4.3. The slowdown is due to the Core-CH search that is used to explore the transfer graph. In turn, RAPTOR is slower by another factor of two to four than TB, of which no multimodal variant currently exists. The goal of this chapter is to obtain a multimodal algorithm that matches the performance of TB on public transit networks.

Previous Approaches. Two different methods for replacing the Core-CH search of MR have been proposed in the literature: the HL-based approach by Phan and Viennot [PV19] and the shortcut-based one by Sauer [Sau18]. The method by Phan and Viennot runs the HL preprocessing step on the transfer graph and uses its output to perform two-hop searches. For each vertex, HL computes a set of outgoing and incoming hub vertices. These must fulfill the cover property: for each pair v_s, v_t of vertices, there is a vertex v that lies on a shortest v_s-v_t-path such that v is an outgoing hub of v_s and an incoming hub of v_t. The approach by Phan and Viennot partially inverts the vertex-hub relation: each stop stores edges to its outgoing hubs, and each hub stores edges to the stops for which it is an incoming hub. This allows intermediate transfers to be explored with two hops: For each reached stop, relax the edges to the outgoing hubs. Then, for each reached hub, relax the edges to the outgoing stops.

The authors combine their approach with CSA and RAPTOR, yielding the multimodal variants HL-CSA and HL-RAPTOR. They report preprocessing times between one and two hours for computing the hubs. For HL-RAPTOR in the two-criteria setting, they report a speedup of 1.7 over MR. However, this figure is based on a comparison to the MR query times reported by Delling et al. [Del+13], which were measured on an older machine and likely for a different set of queries. HL-CSA is the only multimodal variant of CSA proposed so far. However, the reported experiments indicate that it is only barely faster than HL-RAPTOR on most networks and slower on London.

The shortcut hypothesis suggests another way to replace the Core-CH search of MR: If the set of intermediate transfers that occur in Pareto-optimal journeys is small and a set of shortcuts representing them can be precomputed efficiently, then intermediate transfers can be explored simply by relaxing the outgoing shortcuts of each reached stop. Initial and final transfers can then be handled with an existing one-to-many algorithm.

Sauer proposes a prototypical speedup technique for RAPTOR that exploits this observation. The shortcuts are computed by running a one-to-all profile search from each stop. This is done with a multimodal variant of rRAPTOR that replaces the RAPTOR search in each run with MR. A crucial aspect that allows the algorithm to achieve practical performance is that each MR search is stopped after the first two rounds. This is based on the observation that it is always possible to construct a Pareto-optimal journey with more than two trips by concatenating Pareto-optimal journeys with exactly two trips. However, the correctness of this approach is not proven by Sauer. In fact, we will see in Section 5.1 that it is possible to construct examples in which it fails to find required shortcuts, although these rarely occur in real-world networks. After each run, all Pareto-optimal journeys with empty initial and final transfers are unpacked and shortcuts are generated for their intermediate transfers. To decrease the preprocessing time at the expense of computing superfluous shortcuts, Sauer proposes to prune the search once the initial transfer exceeds a specified length.

The query algorithm is identical to MR with one exception: instead of running Dijkstra's algorithm on the core graph to explore intermediate transfers, it relaxes the outgoing shortcuts of all marked stops. Sauer conducts an experimental study of this approach on the Switzerland network, using walking as the transfer mode. The number of shortcuts is manageable at around two million. The highest achieved speedup over MR is 1.8, which approximately matches the performance of RAPTOR on the transitively closed transfer graph. The exact shortcut precomputation takes over three hours when parallelized with 16 cores. This can be reduced to 50 minutes by pruning the initial transfers, but this vastly increases the number of shortcuts, reducing the speedup to 1.2.

Chapter Outline. We revisit the idea of a shortcut-based speedup technique but re-engineer it from the ground up. We name the resulting technique ULTRA (UnLimited TRAnsfers). In Section 5.1, we describe the ULTRA shortcut computation algorithm and prove that the computed set of shortcuts is sufficient for answering all possible queries correctly. Section 5.2 explains how the ULTRA shortcuts can be integrated into any query algorithm that

requires one-hop transfers. To explore the initial and final transfers, we replace Core-CH with a more efficient one-to-many algorithm, Bucket-CH. In order to integrate ULTRA with TB, we show that only minor changes are necessary to make ULTRA compute shortcuts between stop events instead of stops. This allows ULTRA to replace the TB preprocessing phase, resulting in the first multimodal variant of TB. We demonstrate that this significantly reduces the number of required shortcuts and the query time compared with a naive approach, i.e., using the output of ULTRA as input for the TB preprocessing. We also present modifications to the TB query algorithm to make it more efficient in a multimodal setting.

We evaluate the performance of our preprocessing and query algorithms on the four benchmark networks in Section 5.3. Compared to the prototypical algorithm by Sauer, ULTRA reduces both the preprocessing time and the number of shortcuts by more than an order of magnitude. We evaluate the combination of ULTRA with RAPTOR, CSA and TB. The resulting multimodal algorithms have roughly the same query performance as the original restricted algorithms, regardless of the speed of the considered transfer mode. The fastest algorithm, ULTRA-TB, outperforms MR, which was previously the fastest multimodal algorithm, by about an order of magnitude. This yields query times of a few milliseconds on metropolitan networks and less than 100 ms on the much larger network of Germany. Finally, we summarize our results in Section 5.4.

5.1 Shortcut Computation

The ULTRA preprocessing phase computes a set E^{s} of shortcut edges that represent intermediate transfers between trips. These shortcuts must be sufficient for answering every point-to-point query correctly. This is achieved if every query can be answered with a Pareto set of journeys in which all intermediate transfers are represented by shortcuts. On the other hand, the number of shortcuts should be as small as possible to allow for fast queries.

We present two variants of the ULTRA preprocessing, which differ in the granularity of the computed shortcuts: In the *stop-to-stop* variant, the shortcuts $E^{\mathrm{s}} \subseteq \mathcal{S} \times \mathcal{S}$ connect pairs of stops. This yields a one-hop transfer graph, which is sufficient for most public transit algorithms, including RAPTOR and CSA. By contrast, TB requires shortcuts $E^{\mathrm{s}} \subseteq \mathcal{E} \times \mathcal{E}$ between stop events, which are computed by the *event-to-event* variant of ULTRA. Both variants are identical except for a few crucial details, which are discussed explicitly as appropriate.

ULTRA works by enumerating a set of journeys $\mathcal{J}^{\mathrm{c}}$ with exactly two trips such that all required shortcuts occur as intermediate transfers in $\mathcal{J}^{\mathrm{c}}$. For each enumerated journey, the intermediate transfer is unpacked and a shortcut is generated for it. Before we describe the algorithm, Section 5.1.1 establishes a definition for $\mathcal{J}^{\mathrm{c}}$ that is sufficient for answering all queries while keeping the number of shortcuts as low as possible. We then provide a high-level overview of the ULTRA shortcut computation and prove that it enumerates $\mathcal{J}^{\mathrm{c}}$ in Section 5.1.2. Afterward, Section 5.1.3 discusses running time optimizations to make the algorithm efficient in practice. Finally, Section 5.1.4 compares event-to-event ULTRA to the TB transfer generation phase and explains why it discards more unnecessary transfers.

5.1.1 Enumerating a Sufficient Set of Journeys

Consider the subproblem in which only queries between fixed source and target vertices v_s, v_t must be answered. Then the following naive algorithm computes a sufficient set of shortcuts: enumerate the set $\mathcal{J}^{\text{opt}}$ of all v_s-v_t-journeys J that are Pareto-optimal for the departure time $\tau_{\text{dep}}(J)$ and generate a shortcut for every intermediate transfer that occurs in $\mathcal{J}^{\text{opt}}$. This produces more shortcuts than necessary: if there are multiple Pareto-optimal journeys that are equivalent in both criteria, only one of them is required to answer a query. The goal is therefore to find a set $\mathcal{J}^{\text{canon}} \subseteq \mathcal{J}^{\text{opt}}$ of journeys that excludes such duplicates but is still sufficient for answering all queries correctly. We observe that every journey in $\mathcal{J}^{\text{canon}}$ with more than two trips can be decomposed into subjourneys with two trips each. Every shortcut that occurs in $\mathcal{J}^{\text{canon}}$ also occurs in the much smaller set containing only these subjourneys. To exploit this algorithmically, we want to define $\mathcal{J}^{\text{canon}}$ such that it is closed under subjourney decomposition, i.e., every subjourney of a journey in $\mathcal{J}^{\text{canon}}$ is itself contained in $\mathcal{J}^{\text{canon}}$.

Tiebreaking Sequences. To achieve closure under subjourney decomposition, we define a consistent set of rules for breaking ties between equivalent journeys. These are modeled after the rules employed by MR, with additional ones for cases in which the choice made by MR is not clearly defined. For this purpose, we define total orderings on the sets of routes and vertices with a *route index* function $\text{id}_{\mathcal{R}} : \mathcal{R} \to \mathbb{N}$ and a *vertex index* function $\text{id}_V : V \to \mathbb{N}$. Then ties between equivalent journeys are broken as follows (see Figure 5.1 for examples): Journeys that end with trip segments are preferred over journeys that end with (non-empty) transfers. For journeys that end with a trip segment $T[i, j]$, the index of the route $R(T)$ and the index i at which the trip segment starts are used as tiebreakers in this order. For journeys that end with an edge (w, v), ties are broken by first considering the arrival time at w and then the vertex index $\text{id}_V(w)$. If two journeys share a non-empty suffix, it is ignored and the respective prefixes of the journeys are compared instead. To formalize these rules, we assign a unique *tiebreaking sequence* to each non-empty journey. For a journey J with vertex sequence $V(J) = \langle v_s = v_1, \ldots, v_k = v_t \rangle$ and $k > 1$, the *route tiebreaking sequence* is given by

$$X_r(J) := \begin{cases} \langle \text{id}_{\mathcal{R}}(R(T)), i \ \rangle & \text{if } J \text{ ends with a trip segment } T[i, j], \\ \langle \infty \qquad\qquad , \infty \rangle & \text{if } J \text{ ends with an edge } (v_{k-1}, v_k), \end{cases}$$

and the *edge tiebreaking sequence* by

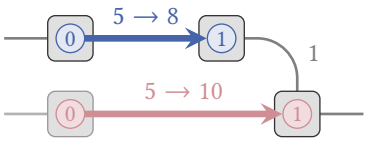
$$X_e(J) := \begin{cases} \langle \infty \qquad\qquad , \infty \qquad\quad \rangle & \text{if } J \text{ ends with a trip segment } T[i, j], \\ \langle \tau_{\text{arr}}(J[v_s, v_{k-1}]), \text{id}_V(v_{k-1}) \rangle & \text{if } J \text{ ends with an edge } (v_{k-1}, v_k). \end{cases}$$

These are combined into the *local tiebreaking sequence*

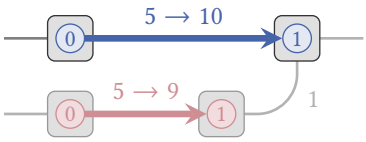
$$X_\ell(J) := \langle \tau_{\text{arr}}(J) \rangle \circ X_r(J) \circ X_e(J).$$

The overall tiebreaking sequence is the concatenation of the local sequences in reverse order:

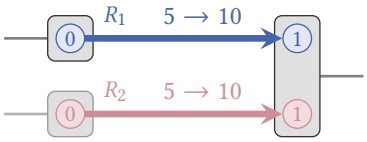
$$X(J) := X_\ell(J[v_s, v_k]) \circ \cdots \circ X_\ell(J[v_s, v_2]).$$



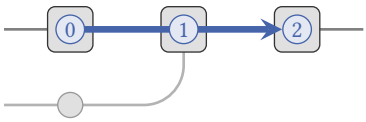
(a) Different arrival time.



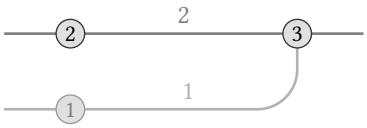
(b) Empty vs. non-empty final transfer.



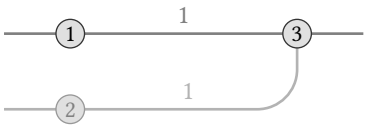
(c) Different final routes.



(d) Same trip.



(e) Final transfer; different parent arrival times.



(f) Final transfer; same parent arrival times.

Figure 5.1: An illustration of the choices between journeys that are enforced by the tiebreaking sequences. Shared suffixes are disregarded for the comparison. In each example, the upper journey is preferred over the lower journey. In Subfigure a, this is due to a lower arrival time. In all other examples, the arrival times are equal. Journeys with an empty final transfer are preferred (b). If both journeys have an empty final transfer (c), the one with the lower route ID according to $\mathrm{id}_{\mathcal{R}}$ is preferred. If they use the same final trip (d), the one that enters it earlier is preferred. In Subfigures e and f, both journeys have a non-empty final transfer and vertices are labeled with their ID according to id_V. If the arrival times at the parent vertex differ (e), the one with the lower arrival time is preferred. Otherwise (f), the one with the lower parent vertex ID is preferred.

This sequence is unique among all v_s-v_t-journeys. In particular, if two journeys J and J' end with trip segments $T_a[i, j] \neq T_b[m, n]$, then their tiebreaking sequences are different. If $\tau_{\text{arr}}(J) = \tau_{\text{arr}}(J')$ and $R(T_a) = R(T_b)$, then $T_a = T_b$ and $j = n$ must hold because the trips cannot overtake each other. Then it follows that $i \neq m$ and the tiebreaking sequences are different. Sequences are ordered lexicographically: for sequences $A = \langle a_1, a_2, \ldots, a_k \rangle$ and $B = \langle b_1, b_2, \ldots, b_k \rangle$ of equal length, $A < B$ if $a_1 < b_1$, or $a_1 = b_1$ and $\langle a_2, \ldots, a_k \rangle < \langle b_2, \ldots, b_k \rangle$. For sequences of different lengths, the shorter one is padded with $-\infty$ on the right side before they are compared.

Canonical Journeys. Because tiebreaking sequences are strictly ordered, ambiguities between equivalent journeys can be resolved by replacing the criterion arrival time with the tiebreaking sequence. We say that an v_s-v_t-journey J *canonically dominates* another v_s-v_t-journey J' if $X(J) < X(J')$ and $|J| \leq |J'|$. Because the two tiebreaking sequences cannot be equal, there is no need to distinguish between strong and weak canonical dominance. A v_s-v_t-journey J is called *canonical* if it is Pareto-optimal with respect to tiebreaking sequence and number of trips for the departure time $\tau_{\text{dep}}(J)$, i.e., if no other v_s-v_t-journey exists that is feasible for $\tau_{\text{dep}}(J)$ and canonically dominates J. Because no two journeys can be equivalent in both criteria, the set that consists of all feasible canonical journeys for a given query is the only Pareto set. We call this the *canonical Pareto set*. The set $\mathcal{J}^{\text{canon}}$ is the union of the canonical Pareto sets for all possible v_s-v_t-queries. It is closed under subjourney decomposition:

Lemma 5.1. *For every canonical v_s-v_t-journey J and every pair $v, w \in V(J)$ of vertices visited by J, the subjourney $J[v, w]$ is canonical.*

Proof. Assume that $J[v, w]$ is not canonical. Then there is another journey $J'[v, w]$ that is feasible for $\tau_{\text{dep}}(J[v, w])$ and for which $X(J'[v, w]) < X(J[v, w])$ and $|J'[v, w]| \leq |J[v, w]|$ hold. Because $J'[v, w]$ does not depart earlier or arrive later than $J[v, w]$, replacing $J[v, w]$ with $J'[v, w]$ in J yields a feasible journey J' with $|J'| \leq |J|$. Adding the prefix $J[v_s, v]$ to $J'[v, w]$ and $J[v, w]$ adds identical suffixes to both tiebreaking sequences. This does not change their relative order, so $X(J'[v_s, w]) < X(J[v_s, w])$. Similarly, adding the suffix $J[w, v_t]$ to $J'[v_s, w]$ and $J[v_s, w]$ adds identical prefixes to both tiebreaking sequences, which does not change their relative order. Therefore, $X(J') < X(J)$ and J is not canonical. □

Candidate Journeys. We exploit the fact that $\mathcal{J}^{\text{canon}}$ is closed under subjourney decomposition by defining a suitable set of subjourneys that need to be enumerated. A *candidate* is a journey that consists of two trips connected by an intermediate transfer but with empty initial and final transfers. For a canonical journey J that uses at least two trips, consider the set of subjourneys of J that are candidates. Every intermediate transfer of J appears in exactly one candidate subjourney. Furthermore, by Lemma 5.1, these subjourneys are themselves canonical. Accordingly, every shortcut that occurs in $\mathcal{J}^{\text{canon}}$ also occurs in the set $\mathcal{J}^{\text{c}} \subseteq \mathcal{J}^{\text{canon}}$ of canonical candidate journeys. A sufficient set of shortcuts can therefore be computed by enumerating $\mathcal{J}^{\text{c}}$.

Canonical MR. Canonical Pareto sets can be computed by making slight modifications to MR in order to ensure proper tiebreaking: First, at the start of each round, the collected routes are sorted according to $\text{id}_{\mathcal{R}}$ before they are scanned. The second change concerns the keys of vertices in the priority queue of Dijkstra's algorithm. In standard MR, the key of a vertex v in round n is the tentative arrival time $\tau_{\text{arr}}(v, n)$ at v with n trips. This is now replaced with $\langle \tau_{\text{arr}}(v, n), \text{id}_V(v) \rangle$. The resulting implementation of MR, which we call *canonical MR*, finds equivalent journeys in increasing order of tiebreaking sequence. Hence, canonical journeys are found first and all other equivalent journeys are discarded because they are weakly dominated by them. This is proven by the following lemma.

Lemma 5.2. *Canonical MR returns the canonical Pareto set for every query.*

Proof. It follows from the correctness of MR that canonical MR returns a valid Pareto set. We show that this is the canonical Pareto set. Consider a query with source and target vertices $v_s, v_t \in V$ and departure time τ_{dep}. Let J be a journey in the canonical Pareto set for this query and J' another journey that is feasible for τ_{dep}, with $\tau_{arr}(J) = \tau_{arr}(J')$, $|J| = |J'|$ and $X(J) < X(J')$. Let v be the vertex at which the longest shared suffix of J and J' starts. We show that canonical MR discards the corresponding prefix $J'[v_s, v]$ in favor of $J[v_s, v]$.

Because $J[v_s, v]$ does not share a suffix with $J'[v_s, v]$, its local tiebreaking sequence must be smaller, i.e.,

$$\langle \tau_{arr}(J[v_s, v]) \rangle \circ X_r(J[v_s, v]) \circ X_e(J[v_s, v]) < \langle \tau_{arr}(J'[v_s, v]) \rangle \circ X_r(J'[v_s, v]) \circ X_e(J'[v_s, v]).$$

If $\tau_{arr}(J[v_s, v]) < \tau_{arr}(J'[v_s, v])$, then it follows from the correctness of MR that canonical MR discards $J'[v_s, v]$ in favor of $J[v_s, v]$. Assume therefore that $\tau_{arr}(J[v_s, v]) = \tau_{arr}(J'[v_s, v])$. Then the comparison depends on whether the journeys end with a trip segment or an edge:

Case 1a: $J[v_s, v]$ ends with a trip segment $T_a[i, j]$ and $J'[v_s, v]$ ends with an edge. Then the route scanning phase of round $|J[v_s, v]|$ finds $J[v_s, v]$, and $J'[v_s, v]$ is discarded when it is found in the subsequent transfer relaxation phase.

Case 1b: $J[v_s, v]$ ends with a trip segment $T_a[i, j]$ and $J'[v_s, v]$ ends with a different trip segment $T_b[m, n]$. Then $X_r(J[v_s, v]) < X_r(J'[v_s, v])$, which is equivalent to

$$\langle \mathrm{id}_{\mathcal{R}}(R(T_a)), i \rangle < \langle \mathrm{id}_{\mathcal{R}}(R(T_b)), m \rangle.$$

If $R(T_a) \neq R(T_b)$, it follows that $\mathrm{id}_{\mathcal{R}}(R(T_a)) < \mathrm{id}_{\mathcal{R}}(R(T_b))$. Then the route scanning phase of round $|J[v_s, v]|$ scans $R(T_a)$ before $R(T_b)$, finds $J[v_s, v]$ first and discards $J'[v_s, v]$. If both journeys use the same route, then it follows from the fact that their arrival times are equal that $T_a = T_b$, $j = n$ and $i < m$. Then the scan of route $R(T_a)$ finds $J[v_s, v]$ when it enters at the i-th stop of $R(T_a)$ and discards $J'[v_s, v]$ because it does not improve the active trip at the m-th stop.

Case 2: $J[v_s, v]$ ends with an edge (w, v). Then $X_r(J[v_s, v]) = \langle \infty, \infty \rangle$, so $J'[v_s, v]$ must also end with an edge (x, v). Because the local tiebreaking sequence of $J[v_s, v]$ is smaller and its first two components are identical, the edge tiebreaking sequence must be smaller as well, i.e.,

$$\langle \tau_{arr}(J[v_s, w]), \mathrm{id}_V(w) \rangle < \langle \tau_{arr}(J'[v_s, x]), \mathrm{id}_V(x) \rangle.$$

These are the keys of w and x in the priority queue when they are extracted during the Dijkstra search in round $|J[v_s, v]|$. Thus, the search extracts w before x, the edge (w, v) is relaxed before (x, v), $J[v_s, v]$ is found first and $J'[v_s, v]$ discarded.

In all cases, canonical MR discards $J'[v_s, v]$ and therefore also J'. Because this is the case for every journey J' that is equivalent to J but has a higher tiebreaking sequence, it follows that canonical MR finds the canonical journey J. □

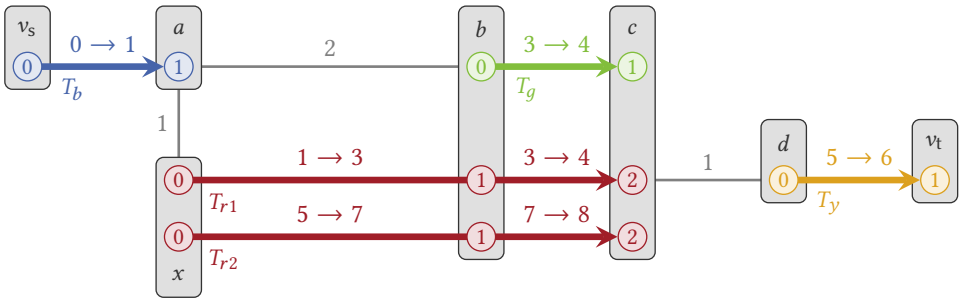


Figure 5.2: An example network in which the set of journeys computed by MR is not closed under subjourney decomposition. There are two equivalent Pareto-optimal v_s-v_t-journeys: $J = \langle\langle v_s\rangle, T_b[0,1], \langle a,b\rangle, T_g[0,1], \langle c,d\rangle, T_y[0,1], \langle v_t\rangle\rangle$ and $J' = \langle\langle v_s\rangle, T_b[0,1], \langle a,b\rangle, T_{r1}[1,2], \langle c,d\rangle, T_y[0,1], \langle v_t\rangle\rangle$. Let R_g denote the route of T_g and R_r the route of T_{r1} and T_{r2}. Assume that $\mathrm{id}_{\mathcal{R}}(R_g) < \mathrm{id}_{\mathcal{R}}(R_r)$. Then an MR query from b for the departure time 3 scans R_g first and consequently finds $J[b, v_t] = \langle\langle b\rangle, T_g[0,1], \langle c,d\rangle, T_y[0,1], \langle v_t\rangle\rangle$. However, a query from v_s reaches x and collects R_r there before it reaches b and collects R_g. Note that $\langle\langle v_s\rangle, T_b[0,1], \langle a,x\rangle, T_{r1}[0,2], \langle c,d\rangle, T_y[0,1], \langle v_t\rangle\rangle$ is not a valid journey because R_r departs too early at x to be entered. However, R_r is still collected at x because there is a later trip departing at 5, which can be entered. Thus, a query from v_s scans R_r before R_g and therefore finds J' and its subjourney $J'[v_s, c] = \langle\langle v_s\rangle, T_b[0,1], \langle a,b\rangle, T_{r1}[1,2], \langle c\rangle\rangle$. Overall, the set of journeys output by MR includes J' and $J[b, v_t]$, but not J and $J'[b, v_t]$, so it is not closed under subjourney decomposition.

Issues with Non-Canonical MR. Note that the journeys returned by a straightforward (non-canonical) implementation of MR are not closed under subjourney decomposition. This is because the order in which two equivalent journeys are explored by MR can flip if the same prefix is added to both journeys. An example of this is shown in Figure 5.2. The two v_s-v_t-journeys J and J' are equivalent and differ only in the route that is used for the second trip. The order in which the journeys are found depends on the order in which these routes are scanned. At the start of each round, RAPTOR iterates over all stops that were marked in the previous round and collects all routes that visit them. The order in which these routes are then scanned is not specified in the original description of RAPTOR [DPW15a], but it is natural to scan them in the order in which they were collected. Routes visiting the same stop are collected in the order defined by $\mathrm{id}_{\mathcal{R}}$. However, if multiple stops were updated, the order in which the routes are collected and scanned depends on the order in which the stops were reached in the previous round.

In Figure 5.2, if the search is started from v_s, the stop x is reached before b. Therefore, the route R_r is collected before the route R_g, which does not visit x, and J' is explored before J. However, if the shared prefix $J[v_s, b]$ is omitted and the search is started at b, then R_g is preferred and $J[b, v_t]$ is found. Thus, if the ULTRA preprocessing used non-canonical MR, it

would generate event-to-event shortcuts for the intermediate transfers of $J'[v_s, c]$ and $J[b, v_t]$, but not for those of $J[v_s, c]$ and $J'[b, v_t]$. As a result, neither J nor J' could be reconstructed from these shortcuts.

5.1.2 Algorithm Overview

We now describe how $\mathcal{J}^c$ can be enumerated efficiently. Directly applying the definition of $\mathcal{J}^c$ yields a simple but wasteful approach: For every possible source stop and every possible departure time, a one-to-all canonical MR search restricted to the first two rounds is performed. A candidate J^c is canonical if there is no feasible journey J^w with at most two trips that canonically dominates J^c (and is therefore found before J^c by the respective canonical MR search). If such a journey J^w exists, we call it a *witness* because its existence proves that J^c is not canonical. Unlike candidates, witnesses may have non-empty initial or final transfers, and they may use fewer than two trips. If there is no witness for a candidate J^c, the corresponding canonical MR search will include J^c in its Pareto set. A shortcut representing the intermediate transfer of J^c is then generated.

Adapting rRAPTOR. The reason this approach is wasteful is that it does not exploit the self-pruning property of rRAPTOR: if journeys with later departure times are explored first, they can be used to dominate worse journeys with an earlier departure time. We therefore adapt rRAPTOR to the ULTRA setting: the RAPTOR search that is performed in each run is replaced with a canonical two-round MR search. This version of rRAPTOR is then invoked for each possible source stop $v_s \in \mathcal{S}$ with a departure time interval that covers the entire service period Π of the timetable.

We can make further improvements by carefully choosing the departure times for which runs are performed. rRAPTOR performs a run for every possible departure time τ_{dep} at v_s. A departure time τ_{dep} is possible if there is a stop v (which may be v_s itself) that is reachable from v_s via an initial transfer of length $\tau_{tra}(v_s, v)$ and a trip that departs from v at $\tau_{dep}+\tau_{tra}(v_s, v)$. If transfers are unrestricted, the number of possible departure times is very high because, typically, most stops in the network will be reachable from v_s. Accordingly, a straightforward multimodal adaptation of rRAPTOR performs many runs and is therefore slow. In the context of ULTRA, however, most possible departure times require a non-empty initial transfer, which means that the corresponding runs would not find any candidates. The prototypical shortcut computation algorithm by Sauer exploits this by imposing a limit τ on the length of initial transfers. As a result, runs in which all journeys have initial transfers longer than τ can be skipped. However, because the corresponding witnesses from these runs are not found, they are not available to dominate suboptimal candidates. As a result, many superfluous shortcuts are generated.

For ULTRA, we employ a different optimization that has the same purpose but does not lead to superfluous shortcuts. Only the runs for departure events that occur directly at v_s are performed, since only these can find candidates. Let $\mathcal{D} = \{\tau^0_{dep}, \ldots, \tau^k_{dep}\}$ be the set of departure

Algorithm 5.1: ULTRA transfer shortcut computation.

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

```
1  for each v_s ∈ S do
2      Clear all arrival labels and priority queues
3      τ_tra(v_s, ·) ← Compute transfer times in G° from v_s to all stops
4      D ← Collect departure times of trips at v_s
5      for each τ^i_dep ∈ D in descending order do                  // Canonical MR run
6          Collect and sort routes reachable within [τ^i_dep, τ^{i+1}_dep)   // first round
7          Scan routes
8          Relax transfers
9          Collect and sort routes serving updated stops            // second round
10         Scan routes
11         E^s_new ← Relax transfers, thereby collecting shortcuts
12         E^s ← E^s ∪ E^s_new
```

times of trips departing directly at v_s, sorted in ascending order. The run for τ^i_{dep} explores candidates departing at τ^i_{dep} and witnesses with departure times in the interval $[\tau^i_{\text{dep}}, \tau^{i+1}_{\text{dep}})$. We define $\tau^{k+1}_{\text{dep}} := \infty$ to ensure that the run for τ^k_{dep} explores all witnesses that depart after τ^k_{dep}. By integrating the witness search into the candidate runs, the algorithm is guaranteed to find the canonical journeys for each possible candidate departure time, regardless of whether these journeys are candidates or witnesses. However, it skips witnesses that are only canonical for other departure times that occur in between, since these are irrelevant for dominating candidates. Thus, the ULTRA preprocessing is much faster than a straightforward multimodal adaptation of rRAPTOR.

Pseudocode. High-level pseudocode for the ULTRA shortcut computation scheme is given by Algorithm 5.1. For each source stop $v_s \in \mathcal{S}$, the algorithm performs the modified multimodal rRAPTOR search described above. To avoid redundant invocations of Dijkstra's algorithm, initial transfers to all other stops are explored only once per source stop (line 3) and the results are then reused for each run in line 6. The departure times at v_s for which runs need to be performed are collected in line 4. The runs are performed in lines 6–12. Each run consists of two canonical MR rounds, which are subdivided into three phases: collecting routes and sorting them according to $\mathrm{id}_{\mathcal{R}}$ (lines 4 and 6), scanning routes (lines 7 and 10), and relaxing transfers with Dijkstra's algorithm (lines 8 and 11). After the final transfer relaxation phase in line 11, the remaining candidates that have not been dominated by witnesses are canonical, so shortcuts representing their intermediate transfers are added to the result set E^{s} in line 12.

Extracting Shortcuts. The final transfer relaxation phase in line 11 identifies canonical candidates and extracts their shortcuts. Whenever a stop is settled during the Dijkstra search, the algorithm checks whether the corresponding journey J is a candidate, i.e., has an empty initial and final transfer. If so, we know that J is canonical because any witness that canonically dominates it would have been found already. Therefore, an edge representing the intermediate transfer of J is added to the shortcut set E^s. In order to extract the intermediate transfer, each vertex v maintains two parent pointers $\mathsf{p}_1[v]$ and $\mathsf{p}_2[v]$, where $\mathsf{p}_k[v]$ is the parent for reaching v using k trips (i.e., within the k-th MR round). If the journey to v ends with a trip, $\mathsf{p}_k[v]$ points to the stop at which this trip was entered. If the journey ends with a transfer, it points to the stop at which the transfer starts. For a candidate ending at a stop v_t, the shortcut representing its intermediate transfer is given by $(\mathsf{p}_1[\mathsf{p}_2[v_t]], \mathsf{p}_2[v_t])$. Because intermediate transfers only need to be extracted for candidates, the parent pointer is set to a dummy value $\perp$ if the corresponding journey has a non-empty initial or final transfer. Then the final Dijkstra search in line 11 can check whether the journey ending at a stop v is a candidate or a witness by inspecting $\mathsf{p}_2[v]$.

The event-to-event variant of ULTRA generates shortcuts not between stops, but between stop events. The parent pointer definitions are changed accordingly: If the journey to a vertex v ends with a trip, $\mathsf{p}_k[v]$ now points to the stop event at which this trip was entered. If the journey ends with a transfer, it points to the stop event at which the preceding trip was exited. Because only candidates have valid parent pointers and candidates have empty initial transfers, this preceding trip always exists. For a candidate that ends at a stop v_t, the corresponding shortcut is now given by $(\mathsf{p}_1[v(\mathsf{p}_2[v_t])], \mathsf{p}_2[v_t])$.

Repairing Self-Pruning. Using a rRAPTOR-based approach with self-pruning allows ULTRA to discard many irrelevant candidates early on. However, self-pruning can also cause the algorithm to discard canonical journeys. By exploring journeys with later departure times first, rRAPTOR implicitly maximizes departure time as a third criterion. With this additional criterion, there may be queries for which all Pareto-optimal journeys include suboptimal subjourneys. An example of this is shown in Figure 5.3. In this case, some canonical candidates are suboptimal for three criteria and therefore not found by the rRAPTOR-based scheme. Moreover, in the depicted network, there is no Pareto set for two criteria that is closed under subjourney decomposition and only includes journeys that are Pareto-optimal for three criteria. To solve this issue, we modify the dominance criterion to ensure that canonical journeys are not discarded.

For a journey J, let $\mathsf{run}(J)$ be the highest i with $\tau^i_{\text{dep}} \in \mathcal{D}$ such that $\tau_{\text{dep}}(J) \geq \tau^i_{\text{dep}}$. This is the run in which our modified rRAPTOR finds J. For each vertex v and round n, the algorithm maintains a label $\ell(v,n) = (\tau_{\text{arr}}(v,n), \mathsf{p}_n[v], \mathsf{run}(v,n))$, in which $\tau_{\text{arr}}(v,n)$ is the tentative arrival time, $\mathsf{p}_n[v]$ is the parent pointer, and $\mathsf{run}(v,n)$ is the run of the journey corresponding to this label, which we denote as $J(v,n)$. Let $\ell = (\tau_{\text{arr}}, \mathsf{p}, j)$ be the label of a new journey J that is found by the algorithm at v in round n. Normally, rRAPTOR discards J if it is weakly dominated by $J(v,n)$, i.e., $\tau_{\text{arr}}(v,n) \leq \tau_{\text{arr}}$. Otherwise, it replaces $\ell(v,n)$ with ℓ. Our modified

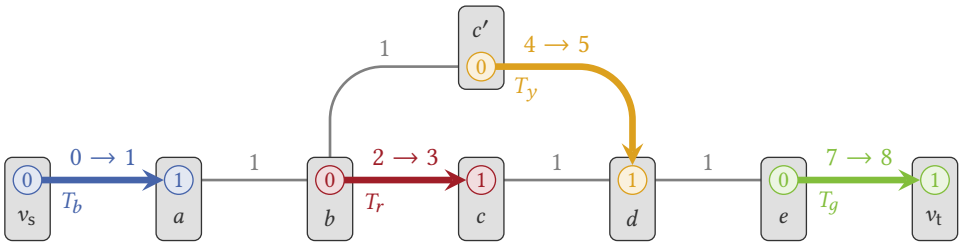


Figure 5.3: An example network in which every v_s-v_t-journey that is Pareto-optimal with respect to the three criteria arrival time, number of trips, and departure time includes a suboptimal subjourney. The two Pareto-optimal journeys are $J = \langle\langle v_s \rangle, T_b[0,1], \langle a, b \rangle, T_r[0,1], \langle c, d, e \rangle, T_g[0,1], \langle v_t \rangle\rangle$ and $J' = \langle\langle v_s \rangle, T_b[0,1], \langle a, b, c' \rangle, T_y[0,1], \langle d, e \rangle, T_g[0,1], \langle v_t \rangle\rangle$. The subjourney $J[b, v_t]$ of J is not Pareto-optimal because it has an earlier departure time than $J'[b, v_t]$ and is otherwise equivalent. Likewise, the subjourney $J'[v_s, d]$ is suboptimal because it has a later arrival time than $J[v_s, d]$.

algorithm discards J if it is weakly dominated by $J(v, n)$ and one of the following conditions is fulfilled:

1. J is not a prefix of a candidate, i.e., $\mathsf{p} = \bot$.
2. J is strongly dominated by $J(v, n)$, i.e., $\tau_{\text{arr}}(v, n) < \tau_{\text{arr}}$ or $\tau_{\text{arr}}(v, n-1) \leq \tau_{\text{arr}}$.
3. $J(v, n)$ was found in the current run, i.e., $\mathsf{run}(v, n) = j$.

With this modified dominance condition, we can prove that the ULTRA preprocessing computes a sufficient set of shortcuts:

Theorem 5.3. *Let $J = \langle P_0, T_0[i, j], \ldots, T_{k-1}[m, n], P_k \rangle$ be a canonical journey. ULTRA generates a shortcut for every intermediate transfer in J.*

Proof. Consider an intermediate transfer P_{x+1} of J and the corresponding candidate subjourney $J^c = \langle T_x[g, h], P_{x+1}, T_{x+1}[p, q] \rangle$. We show that the modified rRAPTOR search for the source stop $v(T_x[g])$ finds this candidate in the run for $\tau_{\text{dep}}(J^c)$ and generates a shortcut for it. Assume J^c is not found. Then some prefix J' of J^c is discarded by the search in favor of a witness J^w. By Lemma 5.1, J' is canonical and therefore not strongly dominated by J^w. Then by our modified dominance criterion, J^w must have been found in the same canonical MR run as J'. However, by Lemma 5.2, canonical MR discards J^w in favor of J', a contradiction. □

5.1.3 Optimizations

We now discuss running time optimizations that are not mentioned in the high-level overview given by Algorithm 5.1 but are crucial for achieving fast preprocessing times.

Initial Route Collection. An rRAPTOR run with departure time τ_{dep}^i explores journeys that depart at the source stop v_s within the interval $[\tau_{\text{dep}}^i, \tau_{\text{dep}}^{i+1})$. Line 6 collects the set $\mathcal{R}(\tau_{\text{dep}}^i)$ of routes that must be scanned in the first round of this run. This set consists of all routes R for which there is a stop v visited by R and a trip T of R such that $\tau_{\text{dep}}(T, v) - \tau_{\text{tra}}(v_s, v) \in [\tau_{\text{dep}}^i, \tau_{\text{dep}}^{i+1})$. In order to speed up this step, the set $\mathcal{R}(\tau_{\text{dep}}^i)$ is precomputed when τ_{dep}^i is added to the set $\mathcal{D}$ of candidate departure times in line 4. This leads to the following procedure for calculating $\mathcal{D}$ and $\mathcal{R}(\cdot)$: First, the algorithm collects all departure triplets $(v, \tau_{\text{dep}}, R)$ of departure stop v, departure time τ_{dep}, and route R that occur in the network. They are then sorted by their departure time at v_s, which is $\tau_{\text{dep}} - \tau_{\text{tra}}(v_s, v)$, and processed in descending order. The algorithm maintains a tentative set $\mathcal{R}'$ of routes for the next candidate departure time that is added to $\mathcal{D}$. For each triplet $(v, \tau_{\text{dep}}, R)$, the algorithm checks whether $v = v_s$. If $v \neq v_s$, R is added to $\mathcal{R}'$. Otherwise, τ_{dep} is a candidate departure time. If τ_{dep} is already contained in $\mathcal{D}$, the algorithm already found another route departing from v_s at τ_{dep}, so R is added to $\mathcal{R}(\tau_{\text{dep}})$. Otherwise, τ_{dep} is added to $\mathcal{D}$, $\mathcal{R}(\tau_{\text{dep}})$ is set to $\mathcal{R}' \cup \{R\}$, and $\mathcal{R}'$ is cleared.

Limited Dijkstra Searches. The algorithm can be sped up by introducing a stopping criterion to the Dijkstra search for final transfers in line 11. For this purpose, the preceding route scanning phase in line 10 counts the number of stops that are marked because their tentative arrival time is improved by a candidate. Whenever such a stop is settled in line 11, the counter is decreased. Once the counter reaches zero, we know that the Dijkstra search has processed all candidates that have been found in this run, so it is stopped.

A similar stopping criterion is applied to the intermediate Dijkstra search in line 8. Here, the first route scanning phase in line 7 counts the stops whose tentative arrival time is improved by a candidate prefix, i.e., a journey with an empty initial transfer. As in line 11, the Dijkstra search is stopped as soon as no such stops are left in the priority queue. This does not affect the correctness of the computed shortcut set E^s because all candidates are still processed. However, some of the witnesses that are pruned might be required to dominate non-canonical candidates. In this case, superfluous shortcuts will be added to E^s. This can be counteracted by continuing the Dijkstra search for some time after the last candidate prefix has been extracted. We introduce a parameter λ^w called the *witness limit* that determines how long the search continues. Let τ_c be the arrival time of the last extracted candidate prefix. Instead of stopping the Dijkstra search immediately, it continues until the smallest element in the queue has an arrival time greater than $\tau_c + \lambda^w$.

Once a Dijkstra search is stopped, the remaining witness labels are kept in the queue because they may dominate candidates in later runs. This requires that the two Dijkstra searches in lines 8 and 11 use separate queues, so that labels from the final Dijkstra search of a previous run do not interfere with the intermediate Dijkstra search of the current run. As a consequence, if a label is discarded because it is dominated, it must be explicitly removed from any queues that still contain it. Moreover, the run in which a label is settled may no longer be the same one in which it was enqueued. Accordingly, the run in which a journey J is found may no longer equal $\texttt{run}(J)$. To ensure that the dominance condition is applied correctly, the

run of a newly created label is carried over from its parent label, rather than setting it to the currently performed run.

With these changes, the only remaining part of the algorithm that performs an unlimited Dijkstra search on the core graph is the initial transfer relaxation in line 3. Unlike the searches for the intermediate and final transfers, this search is only performed once for every source stop instead of once per run, so its impact on the overall running time is small.

Pruning with Found Shortcuts. Once a shortcut is found and added to the shortcut set E^s, it is no longer necessary to find candidates that produce the same shortcut. We exploit this by further restricting the definition of candidates: a journey is only classified as a candidate if its intermediate transfer is not contained in the set of already computed shortcuts. Because this reduces the number of candidates, the stopping criterion for the Dijkstra searches in lines 8 and 11 may be applied earlier, further saving preprocessing time.

Whenever a potential candidate is found during the second route scanning phase in line 10, the stop-to-stop variant of ULTRA checks if the corresponding shortcut is already contained in E^s. If so, the journey is classified as a witness by setting its parent pointer to $\perp$. In the event-to-event variant, this check is more expensive because the number of shortcuts is much larger. Furthermore, because an event-to-event shortcut typically occurs in many fewer candidate journeys than its stop-to-stop counterpart, it is much less likely that the shortcut is already contained in E^s. Our preliminary experiments showed that the benefit of potentially dismissing a candidate no longer outweighs the work required to look up the shortcut. Therefore, the check is skipped in the event-to-event variant.

When a candidate is extracted from the Dijkstra queue in line 11 and a shortcut is inserted for it, there may be other candidates remaining in the queue that use the same intermediate transfer. These must be turned into witnesses by setting the respective parent pointers to $\perp$. This requires keeping track of all candidates belonging to a particular shortcut. Within a single canonical MR run, the search can find at most one intermediate transfer ending at a particular stop or stop event. In stop-to-stop ULTRA, each stop v therefore maintains a list of all non-dominated candidates whose intermediate transfer ends at v. The event-to-event variant does the same for each stop event. When a shortcut is inserted, all candidates in the corresponding list are turned into witnesses.

Transfer Graph Contraction. As with MR, the Dijkstra searches are performed on a core graph, which is constructed by the Core-CH preprocessing step. Because ULTRA only needs to compute journeys between pairs of stops rather than arbitrary vertices in the transfer graph, only transfers that start and end at stops are relevant. Accordingly, the initial and final transfer searches that MR performs on the upward and downward CH graphs can be omitted.

Another type of contraction is performed for cliques of stops that have a pairwise distance of 0 in the transfer graph. These cliques typically occur when different platforms of a larger station are modeled as individual stops. Each such clique is contracted into a single stop. This decreases the number of canonical MR runs that need to be performed: The number of runs

for a source stop v_s is equal to the number of unique departure times at v_s. If a departure time occurs at multiple stops within a clique with transfer distance 0, then the algorithm performs one run for this departure time at each stop. The journeys found by these runs are identical, save for initial transfers of length 0. By contracting the clique into a single stop, these redundant runs are merged into one. This does not affect the correctness of the algorithm because it is conceptually equivalent to allowing candidates to begin with an initial transfer of length 0.

Parallelization. Finally, we observe that ULTRA allows for trivial parallelization. The preprocessing algorithm searches for candidates once for every possible source stop (line 1 of Algorithm 5.1). As these searches are mostly independent of each other, they can be distributed to parallel threads and the results are then combined in a final sequential step. The only aspect of the algorithm that introduces a dependency between the searches for different source stops is the restricted candidate definition: a journey is only considered a candidate if no shortcut has yet been added for its intermediate transfer. If a shortcut was added by a different thread, the algorithm does not notice this. However, because this is merely a performance optimization, the algorithm remains correct if only shortcuts added by the current thread are considered.

5.1.4 Integration with Trip-Based Routing

Unlike other public transit algorithms, TB on its own already requires a preprocessing step, even when used without ULTRA. One possible approach for enabling unlimited transfers in TB is with a *sequential* three-phase algorithm: First, shortcuts between stops are generated with the stop-to-stop variant of the ULTRA preprocessing. These are then used as input for the TB preprocessing, which generates event-to-event transfers that can be used by the ULTRA-TB query. However, we show that an *integrated* two-phase approach is superior. Here, the TB preprocessing is replaced entirely by the event-to-event variant of the ULTRA preprocessing. The resulting shortcuts between stop events are then used as input for the ULTRA-TB query.

The advantage of the integrated approach is that it produces fewer shortcuts because ULTRA applies stricter pruning rules than the TB preprocessing. Both algorithms enumerate journeys with at most two trips in order to find witnesses that prove that a potential shortcut is not necessary. The TB preprocessing does this in a transfer reduction step, after all potential shortcuts have been generated. Because the latter is no longer feasible with unlimited transfers, ULTRA interleaves the generation and pruning of shortcuts. Furthermore, ULTRA examines a larger set of witnesses. In the TB preprocessing, witnesses must start with the same stop event as the candidate, whereas ULTRA also considers witnesses that start with a non-empty initial transfer or a different initial trip. Furthermore, because the TB preprocessing explores intermediate transfers by iterating along the stop sequence of the initial trip in reverse, a candidate cannot be pruned by witnesses that exit the initial trip before the candidate. Overall, this means that ULTRA has more options for pruning candidates and thus produces fewer shortcuts.

5.2 Query Algorithms

ULTRA shortcuts can be combined with any public transit query algorithm that requires one-hop transfers. The idea is to replace the original transfer graph $G = (V, E)$ with a shortcut graph $G^{\mathrm{s}} = (\mathcal{S}, E^{\mathrm{s}})$ consisting of the precomputed stop-to-stop shortcuts. Then the original query algorithm can be run on the resulting network. Note that although the ULTRA shortcut graph is not transitively closed, it is still a valid one-hop transfer graph: Theorem 5.3 proves that journeys with two consecutive shortcut edges are never required to answer a query correctly. Accordingly, if a transitive edge is missing in the shortcut graph, we know that it is never required as part of an optimal journey.

The shortcut graph covers intermediate transfers between two trips, but it 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 Section 5.2.1, we describe how initial and final transfers can be integrated into the query algorithms efficiently. Additionally, Section 5.2.2 outlines optimizations for the TB query algorithm that make it more efficient in a scenario with unlimited transfers.

5.2.1 Query Algorithm Framework

The ULTRA query algorithm exploits the fact that, for initial and final transfers, one endpoint of the transfer is fixed. All initial transfers start at the source vertex v_{s} of the query, whereas all final transfers end at the target vertex v_{t}. Therefore, initial and final transfers can be explored with two additional one-to-many searches on the original transfer graph: a forward search to compute distances from v_{s} to all stops and a backward search for the distances from all stops to v_{t}. Possible algorithms for performing the one-to-many searches include Bucket-CH and PHAST. ULTRA uses Bucket-CH because it allows for additional search space pruning if the source and target are close to each other. Thus, ULTRA requires three preprocessing steps in total: First, a core graph is constructed with the Core-CH precomputation. This is then used as an input for the transfer shortcut computation outlined in Section 5.1. The third step is the Bucket-CH preprocessing for the original transfer graph G. The query algorithm then takes as input the public transit network, the shortcut graph, and the Bucket-CH data. Pseudocode for the query algorithm is shown in Algorithm 5.2.

A query begins with a bidirectional CH search from v_{s} to v_{t} in line 1. This yields the travel time $\tau_{\mathrm{tra}}(v_{\mathrm{s}}, v_{\mathrm{t}})$ for a direct transfer from v_{s} to v_{t} (which may be ∞ if no direct transfer is possible). A naive approach would then perform a forward Bucket-CH search from v_{s} and a reverse Bucket-CH search from v_{t}, yielding for every stop v the initial transfer distance $\tau_{\mathrm{tra}}(v_{\mathrm{s}}, v)$ and the final transfer distance $\tau_{\mathrm{tra}}(v, v_{\mathrm{t}})$. However, not all of these distances are actually needed. An initial transfer to a stop v cannot be part of an optimal journey if $\tau_{\mathrm{tra}}(v_{\mathrm{s}}, v) \geq \tau_{\mathrm{tra}}(v_{\mathrm{s}}, v_{\mathrm{t}})$ because any journey containing the initial transfer is dominated by the direct transfer from v_{s} to v_{t}. Likewise, no optimal journey can include a final transfer to a stop v with $\tau_{\mathrm{tra}}(v, v_{\mathrm{t}}) \geq \tau_{\mathrm{tra}}(v_{\mathrm{s}}, v_{\mathrm{t}})$. The algorithm exploits this by using the forward and backward search spaces V_{s} and V_{t} of the bidirectional CH search. Because the CH search is stopped once the shortest v_{s}-v_{t}-path has been found, these contain no vertices whose

Algorithm 5.2: ULTRA query algorithm framework.

Input: Public transit network $(\mathcal{S}, \Pi, \mathcal{E}, \mathcal{T}, \mathcal{R}, G)$,
shortcut graph $G^s = (\mathcal{S}, E^s)$, Bucket-CH data for G,
source vertex v_s, departure time τ_{dep}, target vertex v_t
Output: Pareto set $\mathcal{J}$ of v_s-v_t-journeys for departure time τ_{dep}

1 $(\tau_{tra}(v_s, v_t), V_s, V_t) \leftarrow$ Run a CH query from v_s to v_t with departure time τ_{dep}
2 $\tau_{tra}(v_s, \cdot) \leftarrow$ Evaluate the vertex-to-stop buckets for vertices in V_s
3 $\tau_{tra}(\cdot, v_t) \leftarrow$ Evaluate the stop-to-vertex buckets for vertices in V_t
4 $\tilde{G} \leftarrow (\mathcal{S} \cup \{v_s, v_t\}, E^s)$
5 Add edge (v_s, v_t) with transfer time $\tau_{tra}(v_s, v_t)$
6 **for each** $v \in \mathcal{S} \setminus \{v_s, v_t\}$ with $\tau_{tra}(v_s, v) < \tau_{tra}(v_s, v_t)$ **do**
7 Add edge (v_s, v) to $\tilde{G}$ with transfer time $\tau_{tra}(v_s, v)$
8 **for each** $v \in \mathcal{S} \setminus \{v_s, v_t\}$ with $\tau_{tra}(v, v_t) < \tau_{tra}(v_s, v_t)$ **do**
9 Add edge (v, v_t) to $\tilde{G}$ with transfer time $\tau_{tra}(v, v_t)$
10 Run black-box public transit algorithm on $(\mathcal{S} \cup \{v_s, v_t\}, \mathcal{T}, \mathcal{R}, \tilde{G})$

distance from v_s and to v_t, respectively, is greater than $\tau_{tra}(v_s, v_t)$. Therefore, it is sufficient to scan the forward buckets of all vertices in V_s (line 2) and the backward buckets of all vertices in V_t (line 3). Additional query time can be saved by sorting the entries of each bucket in ascending order of distance during the preprocessing phase. Then the scan for the forward bucket of a vertex v can be stopped once it reaches a stop w within the bucket with $\tau_{tra}(v_s, v) + \tau_{tra}(v, w) \geq \tau_{tra}(v_s, v_t)$ (and analogously for backward buckets). Doing so drastically improves local queries, as they do not need to evaluate all stops, but only stops that are close to the source or target.

After the distances for the initial and final transfers have been computed, the algorithm creates a temporary copy $\tilde{G}$ of the shortcut graph G^s, which contains v_s and v_t as additional vertices. Lines 5–9 augment $\tilde{G}$ with edges for the initial and final transfers, and the direct transfer from v_s and v_t, using the distances obtained from the Bucket-CH searches. Finally, a public transit algorithm of choice is invoked as a black box on the public transit network with the temporary graph $\tilde{G}$ in line 10. The temporary graph is sufficient for obtaining correct results, as it contains edges for all necessary initial, intermediate and final transfers, and an edge for a direct transfer from source to target.

If the public transit algorithm is not treated as a black box, its performance can be improved further by skipping the construction of $\tilde{G}$. Most algorithms, including RAPTOR and CSA, maintain a tentative arrival time at each stop, which is improved as new journeys are found. For each edge $(v_s, v) \in \tilde{G}$, the tentative arrival time of v can be initialized with $\tau_{dep} + \tau_{tra}(v_s, v)$. To incorporate final transfers, whenever the tentative arrival time at a stop v is set to some value τ, the algorithm can try to improve the tentative arrival time at v_t with $\tau + \tau_{tra}(v, v_t)$.

Algorithm 5.3: Modified procedures for the ULTRA-TB query algorithm.

```
1  Procedure RelaxInitialTransfers()
2      (τ_tra(v_s, v_t), V_s, V_t) ← Run a CH query from v_s to v_t with departure time τ_dep
3      τ_tra(v_s, ·) ← Evaluate the vertex-to-stop buckets for vertices in V_s
4      τ_tra(·, v_t) ← Evaluate the stop-to-vertex buckets for vertices in V_t
5  Procedure CollectInitialTrips(Q_1)
6      R' ← ∅
7      for each v ∈ S with τ_tra(v_s, v) < τ_tra(v_s, v_t) do
8          R' ← R' ∪ {Routes from R that visit v}
9      for each R ∈ R' do
10         T_min ← ⊥
11         for i from 0 to |R| − 1 do
12             v ← i-th stop of R
13             if τ_dep + τ_tra(v_s, v) ≥ τ_min then continue
14             T'_min ← FindEarliestTripFrom(T_min, i, τ_dep + τ_tra(v_s, v))
15             if T'_min ≺ T_min then
16                 T_min ← T'_min
17                 Enqueue(T_min[i + 1], Q_1)
18                 if pred(T_min) = ⊥ then break
```

5.2.2 Improved TB Query

TB already distinguishes between initial/final and intermediate transfers, exploring different graphs for both. The original transfer graph G is only used for the initial and final transfers, whereas intermediate transfers are explored using the precomputed event-to-event transfers. In the context of ULTRA, this requires a modification to the query framework shown in Algorithm 5.2: The temporary graph $\tilde{G}$ now only contains the edges added for the initial and final transfers but not the ULTRA shortcuts. The query then uses $\tilde{G}$ for the initial and final transfers, and the event-to-event shortcuts E^s for the intermediate transfers.

Additionally, we optimize the TB query algorithm for networks with unlimited transfers. The original algorithm is optimized for a use case in which only a few stops are reachable with an initial or final transfer. However, with unlimited transfers, almost all stops are usually reachable. Therefore, we reorganize the algorithm in order to process the huge number of possible initial and final transfers more efficiently. Algorithm 5.3 shows the modified `RelaxInitialTransfers` and `CollectInitialTrips` procedures. The remainder of the query algorithm remains unchanged from Algorithms 4.4, 4.5 and 4.6 in Chapter 4.2.4.

The `RelaxInitialTransfers` procedure (lines 1–4) now performs a Bucket-CH search, in the same manner as the generic ULTRA query framework. For every reached stop v, this yields the minimum transfer times $\tau_{tra}(v_s, v)$ from v_s and $\tau_{tra}(v, v_t)$ to v_t. The procedure `CollectInitialTrips` (lines 5–18) identifies trip segments that are reachable via an

initial transfer. In the original TB algorithm, 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, in which almost all stops are reachable, consecutive stops of a route often share the same earliest reachable trip. In this case, the algorithm performs redundant work by searching for the same trip multiple times. To avoid this, we use a different approach for evaluating the initial transfers, which is based on two steps of the RAPTOR algorithm: collecting updated routes and scanning routes.

Lines 7 and 8 collect all routes visiting a stop that is reachable via an initial transfer. This is analogous to the `CollectRoutes` procedure of RAPTOR, which collects routes that visit marked stops (cf. Algorithm 4.3 in Chapter 4.2.3). Then, all collected routes are scanned in a similar manner to the `ScanRoutes` procedure of RAPTOR. As in RAPTOR, a route R is scanned by processing its stops in the order in which they are visited by R. The algorithm maintains an active trip T_{min}, which is the earliest trip of R that is reachable from any of the already processed stops. Initially, T_{min} is set to a dummy value $\perp$ (line 10). Let v be the next stop to be processed during the scan of R. To check if T_{min} can be improved, line 14 finds the earliest trip T'_{min} of R that can be boarded when arriving at v with the arrival time $\tau_{\text{dep}} + \tau_{\text{tra}}(v_s, v)$. If no reachable trip has been found for any of the previous stops in R (i.e., $T_{\text{min}} = \perp$), then T'_{min} is found with a binary search. Otherwise, the algorithm starts a linear search from T_{min} and looks backward for earlier trips. Because T'_{min} is often not much earlier than T_{min}, this is faster than a binary search in practice. Note that T'_{min} will not be found if it is later than T_{min}, but in this case entering T'_{min} at v does not produce an optimal journey, so it can be discarded. If T'_{min} is earlier than T_{min}, then T_{min} is updated and the `Enqueue` procedure is called for the corresponding stop event in line 17. The `Enqueue` procedure itself is unchanged from the original TB algorithm (see Algorithm 4.6 in Chapter 4.2.4). If T'_{min} is the earliest trip in R, the remainder of the route scan can be skipped.

5.3 Experiments

We evaluate the performance of ULTRA on the four networks presented in Chapter 4.3. Section 5.3.1 evaluates the preprocessing, including the shortcut computation. Our ULTRA-based query algorithms are evaluated in Section 5.3.2. Finally, we compare ULTRA to the HL-based approach of Phan and Viennot [PV19] in Section 5.3.3. Unless otherwise noted, we use walking with a speed of 4.5 km/h as the transfer mode. All queries were evaluated on the Xeon machine. For the preprocessing experiments, we list the used machine individually.

5.3.1 Preprocessing

In this section, we evaluate the performance of the ULTRA preprocessing phase, which includes the Core-CH transfer graph contraction, the shortcut computation, and the Bucket-

Table 5.1: Overview of the stop-to-stop ULTRA preprocessing results. All running times were measured on the Xeon machine and are displayed as (hh:)mm:ss. The Core-CH and Bucket-CH computations were run sequentially. The shortcut computation used all 16 cores.

	Stuttgart	London	Switzerland	Germany
Core-CH time	1:45	0:19	1:09	20:16
Number of core vertices	25 631	23 860	33 219	313 351
Number of core edges	358 842	334 112	465 067	6 267 050
Shortcut computation time	4:27	18:01	8:54	8:01:25
Number of shortcuts	83 086	190 388	170 713	2 907 691
Bucket-CH time	2:13	0:11	0:43	14:49

CH computation. We analyze the effects of the parameters core degree, witness limit, and transfer speed in detail for the Switzerland network, and then discuss more general results for all four networks.

Core Degree and Witness Limit. The two main parameters influencing the performance of the ULTRA preprocessing are the average vertex degree of the contracted core graph and the witness limit λ^w. Figure 5.4 shows the impact of these parameters on the Switzerland network. The lowest preprocessing times are achieved with a core degree of 14. Although the actual shortcut computation is slightly faster for higher core degrees, this is offset by the increased time required to contract the transfer graph. The witness limit λ^w has a larger impact on the preprocessing time. Choosing a witness limit of 0 instead of ∞ nearly cuts the preprocessing time in half. Regardless of core degree or witness limit, the event-to-event variant takes about one minute longer than the stop-to-stop variant. Both parameters have a negligible effect on the number of computed shortcuts. For all following experiments, we therefore choose a core degree of 14 and a witness limit of 0 to minimize the preprocessing time. The only exception is the Germany network, for which we use a core degree of 20. This is because the share of the Core-CH computation in the overall running time is significantly lower for this network, due to its much larger size. Preprocessing results for the stop-to-stop variant on all four networks are listed in Table 5.1.

ULTRA-TB Preprocessing. To evaluate the effectiveness of the event-to-event ULTRA shortcut computation, we compare it to the original TB preprocessing, using the transitively closed transfer graphs presented in Chapter 4.3 as input, and to a naive sequential approach, i.e., using stop-to-stop ULTRA shortcuts as input for the TB preprocessing. The results are shown in Table 5.2. The integrated ULTRA preprocessing drastically reduces the amount of shortcuts compared with the sequential approach. This reduction ranges from a factor of 6 for the London network to over 15 for Germany. Furthermore, the sequential approach using

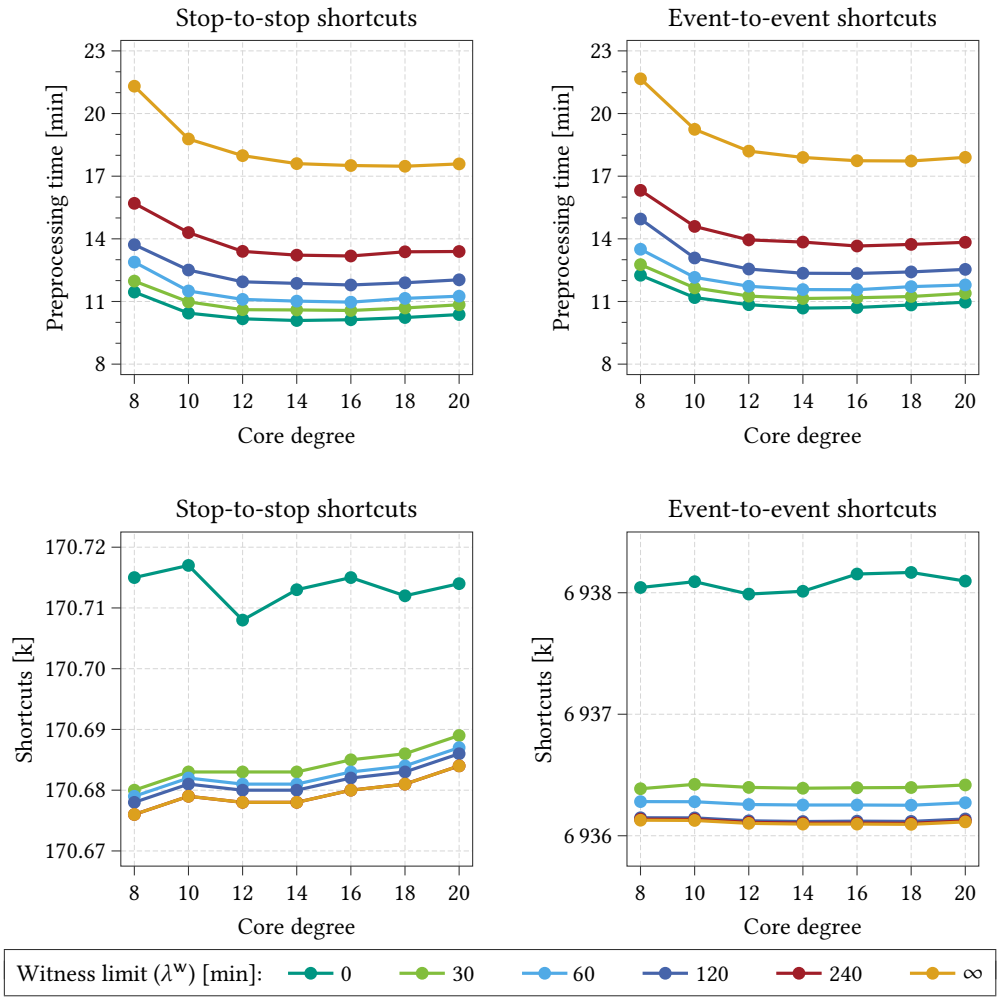


Figure 5.4: Impact of the core degree and the witness limit on the running time of the ULTRA preprocessing and the number of computed shortcuts, measured for the Switzerland network on the Xeon machine. Preprocessing time includes both contracting the transfer graph and computing the shortcuts. The time required for the Bucket-CH computation, which is independent of both parameters, is excluded.

the optimized TB preprocessing proposed by Lehoux and Loiodice [LL20] is only marginally faster than the integrated approach. Overall, the integrated preprocessing is clearly preferable because it produces many fewer shortcuts with only a minor overhead in running time.

Table 5.2: Number of shortcuts and preprocessing times for different TB preprocessing variants. "Transitive" refers to the original TB preprocessing on the transitively closed transfer graph presented in Chapter 4.3. "Sequential" uses stop-to-stop ULTRA shortcuts as input for the TB preprocessing, whereas "integrated" uses event-to-event ULTRA shortcuts directly. "Optimized" refers to the improved TB preprocessing algorithm of Lehoux and Loiodice [LL20]. Running times were measured on the Xeon machine with 16 cores and are displayed as (hh:)mm:ss.

	Stuttgart	London	Switzerland	Germany
Shortcuts (transitive)	7 387 445	50 242 519	31 507 264	458 826 534
Shortcuts (transitive, optimized)	7 387 586	50 240 558	31 507 543	458 763 050
Shortcuts (sequential)	19 361 708	53 179 082	65 485 696	1 195 573 925
Shortcuts (sequential, optimized)	19 361 129	53 181 238	65 484 976	1 195 509 797
Shortcuts (integrated)	1 973 321	8 576 120	6 938 012	77 515 291
Time (transitive)	9:30	1:42:35	1:01:54	73:43:07
Time (transitive, optimized)	0:37	13:12	4:41	2:55:06
Time (sequential)	4:41	18:43	9:40	8:57:46
Time (sequential, optimized)	4:37	18:28	9:24	8:22:37
Time (integrated)	4:42	20:43	9:40	8:37:49

Remarkably, event-to-event ULTRA significantly outperforms the original TB preprocessing in both the number of shortcuts and the computation time, despite operating on an unrestricted transfer graph instead of a transitively closed one. This underscores that the original TB preprocessing was only designed for very small transfer graphs and confirms the findings of Lehoux and Loiodice that it does not scale well for larger graphs. Compared with the optimized TB preprocessing, ULTRA is slower by a factor of two to three on most networks. On the Stuttgart network, it is about eight times slower. The difference is explained by the fact that Stuttgart is the only network for which the transitively closed transfer graph has fewer edges than the full transfer graph. Overall, the preprocessing results show that ULTRA is much more effective than the TB preprocessing at identifying necessary transfers, at the cost of a somewhat higher preprocessing time.

Parallelization. The previous experiments used all 16 cores of the Xeon machine for the shortcut computation. To assess the impact of parallelization on the preprocessing time, we evaluate the running time of the stop-to-stop shortcut computation for different numbers of threads. Additionally, we compare running times of the Epyc machine, which has a worse single-core performance but more cores. Running times on both machines are listed in Table 5.3. Overall, the parallelized shortcut computation achieves good speedups for all networks on both machines. For the Switzerland network, the highest speedup is 13.5 on

Table 5.3: Impact of parallelization on the running time of the stop-to-stop ULTRA shortcut computation. Running times are displayed as (hh:)mm:ss.

Machine	Cores	Stuttgart	London	Switzerland	Germany
Xeon	1	59:28	4:00:31	2:00:29	100:02:46
	2	30:42	2:05:06	1:02:24	54:12:12
	4	15:49	1:06:24	32:17	29:02:18
	8	8:28	34:52	17:13	15:26:13
	16	4:27	18:01	8:54	8:01:25
Epyc	1	1:14:37	4:53:01	2:25:26	122:35:42
	2	40:38	2:43:33	1:21:57	72:42:27
	4	20:10	1:19:21	40:39	37:56:49
	8	10:03	39:54	20:23	19:11:35
	16	5:05	19:54	10:08	9:49:56
	32	2:37	10:08	5:11	4:57:06
	64	1:29	5:52	2:55	2:56:49
	128	0:54	3:44	1:57	2:53:57

the Xeon machine and 74.6 on the Epyc machine. The speedup for the entire preprocessing phase, including the sequential Core-CH and Bucket-CH computation times on the Xeon machine, drops to 11.4 and 38.6, respectively. Independently of the network, we observe the smallest speedup when switching from 64 to 128 threads on the Epyc machine. In this case, the performance is likely limited by the memory bandwidth. The results are similar for the event-to-event variant. On the Switzerland network, the single-threaded performance on the Xeon machine is 2:07:00 for the sequential approach and 2:10:10 for the integrated approach. This corresponds to speedup factors of 13.1 and 13.5, respectively, which matches the speedups observed for the stop-to-stop variant and the TB preprocessing.

Transfer Speed. To test the impact of the transfer mode on the shortcut computation, we change the transfer speed in the Switzerland network from 4.5 km/h to values between 1 km/h and 140 km/h. We consider two ways of applying the transfer speed: In the first version, the speed on an edge is not allowed to exceed the speed limit given in the road network. This models fast transfer modes, such as cars, fairly realistically. In the second version, speed limits are ignored and the same constant speed is assumed for every edge. This allows us to analyze the extent to which the effects observed in the first version are caused by the speed limit data. Figure 5.5 reports the preprocessing times and number of shortcuts (both stop-to-stop and event-to-event) measured for each configuration. In all measurements, the preprocessing time remains below 15 minutes. The number of stop-to-stop shortcuts initially increases with the transfer speed until it peaks at about 300 000 between 10 and 20 km/h (roughly the

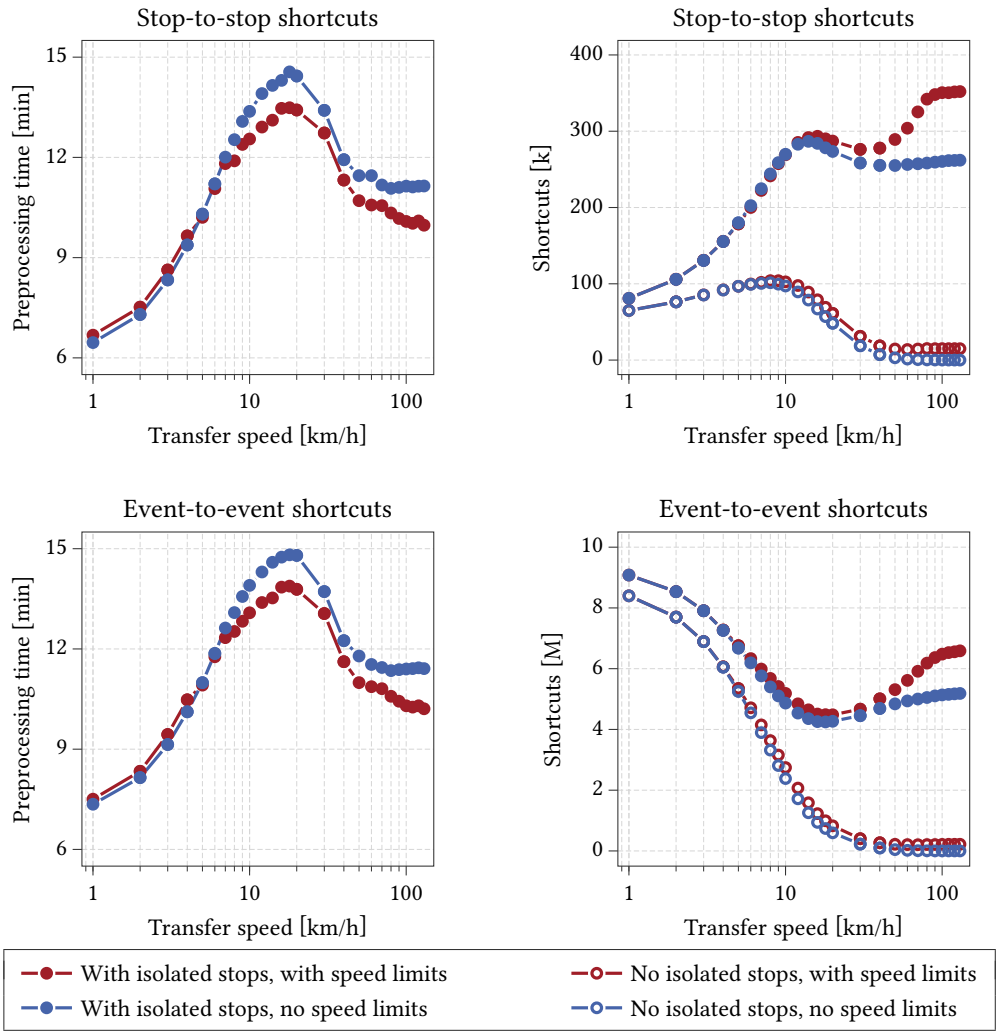


Figure 5.5: Impact of transfer speed on preprocessing time and number of shortcuts, measured on the Switzerland network with a core degree of 14 and a witness limit of 0. Speed limits were obeyed for the red lines and ignored for the blue lines. For the two lines at the bottom of the right plots, shortcuts were only added if the source and target of the candidate journey are connected by a path in the transfer graph.

speed of a bicycle). In the event-to-event variant, the behavior is the opposite: the number of shortcuts is highest for 1 km/h and decreases from there. Above 20 km/h, both variants exhibit a slight increase in the number of shortcuts, which is more pronounced if speed limits are obeyed. Overall, the results show that ULTRA is practical for all transfer speeds both in terms of preprocessing time and the number of shortcuts.

To explain the difference in behavior between the two variants, consider how the transfer speed affects Pareto-optimal journeys. As the transfer mode becomes faster, it becomes increasingly feasible to cover large distances in the transfer graph quickly. This has two effects: On the one hand, more witnesses that require long initial or final transfers become feasible and start dominating slower candidates. Accordingly, the number of canonical candidates decreases, from 409 million for 1 km/h to 114 million for 10 km/h. This explains the decrease in the number of event-to-event shortcuts. On the other hand, longer intermediate transfers between trips also become feasible. This means that, although there are fewer canonical candidates for higher transfer speeds, their shortcuts tend to cover greater distances in the transfer graph. The number of stop pairs within a certain distance of each other grows roughly quadratically with the distance. This explains why the number of stop-to-stop shortcuts rises with the transfer speed even as the number of event-to-event shortcuts declines.

Once the transfer speed becomes faster than public transit, the direct transfer from source to target dominates all other journeys, including all candidates. Accordingly, we should expect the number of shortcuts to eventually reach zero for very high transfer speeds. The reason why this is not observed in our measurements is that not all stops in our network instances are reachable from each other in the transfer graph. For example, in the Switzerland network, 624 stops are isolated in the transfer graph, usually as a result of incomplete or imperfect data. Consider what happens in the shortcut computation for journeys between stops v_s and v_t that are isolated in the transfer graph. In this case, a direct transfer is not possible regardless of the transfer speed. In fact, unless there is a route that serves both v_s and v_t, all v_s-v_t-journeys with at most two trips are candidates and the shortcut computation will add shortcuts for the canonical ones. If we omit shortcuts for candidates whose source and target stop are not connected in the transfer graph, the number of shortcuts behaves as expected: If speed limits are obeyed, a few shortcuts remain even for the highest transfer speed. If they are ignored, a direct transfer is always the fastest option and thus no shortcuts are required.

Shortcut Graph Structure. The stop-to-stop shortcut graph computed by ULTRA for Switzerland is structurally very different from the transitively closed transfer graph that we created for comparison with pure public transit algorithms in Chapter 4.3. This is already evidenced by the fact that the shortcut graph is much less dense, containing only 6% 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 1 004 vertices. By contrast, the largest strongly connected component in the shortcut graph contains 10 891 vertices, which corresponds to 43% of all stops. Accordingly, a transitive closure of the shortcut graph would contain more than 100 million edges.

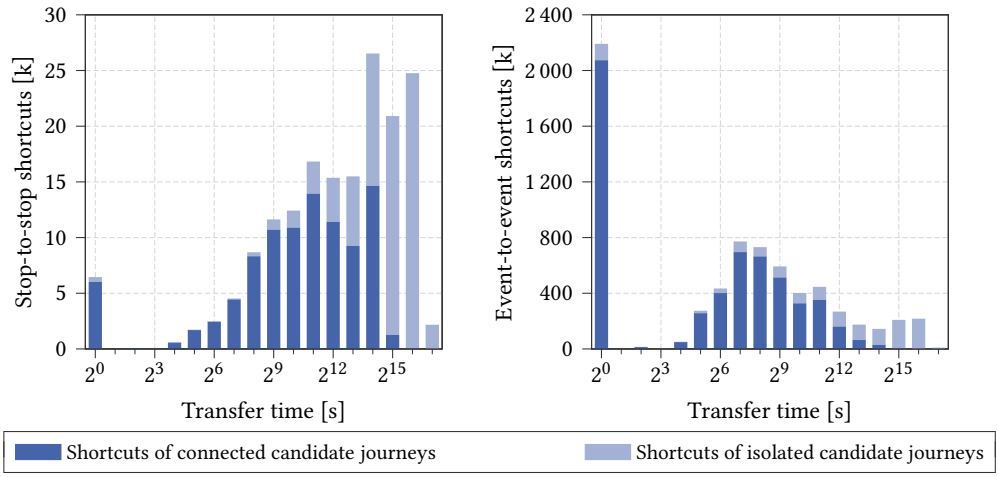


Figure 5.6: Transfer time distribution of the ULTRA shortcuts for the Switzerland network. The bar for 2^i corresponds to the number of shortcuts with a transfer time in the interval $[2^{i-1}, 2^i)$. An exception is the first bar, which represents all shortcuts with a transfer time below one second. The dark blue portion of each bar represents shortcuts for which the source and the target of the corresponding candidate are connected by a path in the transfer graph.

Figure 5.6 (left) shows the distribution of transfer times for the ULTRA shortcuts. Note that the high number of shortcuts with transfer time 0 is caused by cases in which multiple stops model the same physical location. Most of the shortcuts have a transfer time of more than 9 minutes ($\approx 2^9$ seconds), which is the guaranteed walking duration for the transitively closed graph. In fact, only 26 826 edges are shared between the two graphs, which constitute 1.0% of all transitive edges and 15.7% 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.

As with the transfer speed experiment, Figure 5.6 distinguishes between shortcuts generated by candidates whose source and target stop are connected in the transfer graph (dark blue) and shortcuts for which source and target are isolated (light blue). We observe that most of the very long shortcuts are produced by candidates with isolated stops. To analyze how often longer shortcuts are required, we examine the distribution of the event-to-event shortcuts in Figure 5.6 (right). Because stop events occur at a fixed point in time, a stop-to-stop shortcut that is required at several times throughout the day corresponds to multiple event-to-event shortcuts. Thus, the number of event-to-event shortcuts of a certain length reflects more accurately how frequently intermediate transfers of that length are required. Approximately one third of all event-to-event shortcuts have a transfer time of 0. Most of these connect pairs

of trips at the same stop and therefore have no stop-to-stop counterpart. Among the remaining shortcuts, most have a transfer time between 1 minute ($\approx 2^6$ s) and 34 minutes ($\approx 2^{11}$ s). This is in contrast to the stop-to-stop shortcuts, most of which have a transfer time of more than one hour ($\approx 2^{12}$ s). This shows that very long shortcuts are only rarely required. Furthermore, the fraction of shortcuts that are generated by candidates with isolated source and target is much lower in the event-to-event variant than in the stop-to-stop variant.

5.3.2 Queries

To evaluate the impact of ULTRA on the query performance, we test its combination with CSA, RAPTOR, and TB. For CSA and RAPTOR, we compare our new ULTRA variant to the original algorithm on the transitively closed transfer graph and to a multimodal variant with Dijkstra searches. For TB, no multimodal variants have been proposed thus far. We therefore compare the original TB algorithm on the transitively closed transfer graph to ULTRA-TB with sequential and integrated preprocessing.

CSA. Unlike the other algorithms we evaluate, CSA only supports optimizing arrival time as the sole criterion. Although the CSA variant for profile queries also supports optimizing the number of trips as a second criterion, no two-criteria variant for the fixed departure time problem has been published thus far. We conducted preliminary experiments that showed that a two-criteria variant of CSA is in fact outperformed by RAPTOR. Therefore, we only consider single-criterion optimization for CSA. Furthermore, no Dijkstra-based multimodal variant of CSA has been proposed thus far. We therefore implemented a naive multimodal version of CSA, which we call MCSA (Multimodal CSA), as a baseline for our comparison. Normally, after scanning a connection arriving at a stop v, CSA relaxes the outgoing transfer edges of v. MCSA instead performs a Dijkstra search on a contracted transfer graph built with Core-CH, in a similar manner to MR. Once the smallest key in the priority queue exceeds the departure time of the next connection, the Dijkstra search is interrupted and the next connection is scanned. Query times for all three CSA variants are reported in Table 5.4.

On all networks, ULTRA-CSA has a similar running time to CSA with transitively closed transfers. Caution has to be taken when comparing these running times because CSA does not support fully multimodal vertex-to-vertex queries and was therefore evaluated on a different set of stop-to-stop queries. Nonetheless, our experiments demonstrate that ULTRA enables CSA to use unrestricted transfers without performance loss. Compared with MCSA, the ULTRA approach is faster by a factor of three to four on most networks and even more on the Stuttgart network, which has a particularly large transfer graph. By replacing the Core-CH search of MCSA with Bucket-CH, ULTRA speeds up the exploration of initial and final transfers by a factor of six to eight. The time required for the exploration of intermediate transfers is difficult to measure directly because it is interleaved with the individual connection scans. Nevertheless, we observe that using ULTRA shortcuts speeds up the connection scanning phase in its entirety by a factor of two to four compared with MCSA.

Table 5.4: 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. The other two algorithms have been evaluated for vertex-to-vertex queries on the full graph.

Network	Algorithm	Full graph	Scans [k]		Time [ms]		
			Connection	Edge	Init.	Scan	Total
Stuttgart	CSA*	○	52.6	281	0.0	1.4	1.4
	MCSA	●	113.7	238	10.1	6.4	16.5
	ULTRA-CSA	●	113.4	42	1.2	1.7	2.9
London	CSA*	○	83.9	663	0.0	3.0	3.0
	MCSA	●	58.2	182	4.6	4.5	9.1
	ULTRA-CSA	●	57.7	53	0.8	1.9	2.7
Switzerland	CSA*	○	135.2	787	0.1	4.9	4.9
	MCSA	●	88.2	241	8.4	8.1	16.4
	ULTRA-CSA	●	87.6	59	1.1	2.9	4.0
Germany	CSA*	○	2 587.8	6 351	1.3	144.3	145.5
	MCSA	●	1 662.1	3 191	142.8	195.2	338.0
	ULTRA-CSA	●	1 657.3	877	22.4	107.4	129.8

On all networks except Stuttgart, the multimodal variants scan significantly fewer connections than CSA on the transitively closed transfer graph. This is a direct result of the fact that fully multimodal journeys usually have a shorter travel time [WZ17]. Because CSA scans connections in chronological order, the number of scanned connections correlates directly with the earliest arrival time of the query. The Stuttgart network exhibits the opposite behavior because the transfer graph covers a much larger geographical area than the public transit network. Therefore, when the source and target are picked among all vertices instead of only stops, the average query distance increases and the search space becomes larger.

RAPTOR. To evaluate RAPTOR, we use MR as the multimodal baseline algorithm. The results of our comparison are shown in Table 5.5. The share of the overall running time spent exploring the transfer graph (i.e., the *Init* and *Relax* phases) is reduced from 50–75% for MR to 20–40% for ULTRA-RAPTOR. Because RAPTOR explores intermediate transfers in a separate phase, the impact of using ULTRA shortcuts can now be measured directly. Compared with the Dijkstra searches on the core graph performed by MR, exploring the transfer shortcuts is up to an order of magnitude faster. Overall, ULTRA-RAPTOR is two to three times as fast as MR and has a similar running time to RAPTOR with transitive transfers.

Table 5.5: Query performance for RAPTOR, MR, and ULTRA-RAPTOR. Query times are divided into phases: initialization, including 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 three 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*	○	19.8	756	0.2	1.6	2.1	2.1	5.9
	MR	●	35.6	687	12.3	5.2	5.2	11.1	33.5
	ULTRA-RAPTOR	●	37.9	105	1.4	3.5	3.5	1.0	9.6
London	RAPTOR*	○	4.4	2 573	0.3	1.1	2.2	5.4	8.9
	MR	●	5.0	500	6.4	1.9	2.7	7.0	18.0
	ULTRA-RAPTOR	●	5.4	179	1.2	1.5	2.3	1.2	6.2
Switzerland	RAPTOR*	○	26.2	2 115	0.4	2.4	5.0	5.0	12.8
	MR	●	33.0	731	10.6	4.8	7.2	11.7	34.1
	ULTRA-RAPTOR	●	35.9	177	1.6	3.3	6.2	1.4	12.5
Germany	RAPTOR*	○	472.9	26 420	7.0	102.6	120.4	74.2	304.2
	MR	●	541.4	12 359	154.2	187.5	153.5	236.2	731.4
	ULTRA-RAPTOR	●	599.7	3 165	33.0	144.0	151.7	33.3	362.1

TB. We continue with evaluating our improved ULTRA-TB query algorithm. Table 5.6 compares the query performance for ULTRA-TB with sequential and integrated preprocessing, as well as the original TB query algorithm on the transitively closed transfer graph. ULTRA-TB with integrated preprocessing achieves significantly lower query times than the state of the art. Depending on the network, it offers a speedup of 2–5 over ULTRA-RAPTOR and 5–10 over MR, which was previously the fastest multimodal algorithm (cf. Table 5.5). As with RAPTOR and CSA, ULTRA-TB is able to match the query performance of the original TB algorithm despite solving a harder multimodal problem. Furthermore, ULTRA-TB achieves a similar performance to ULTRA-CSA, despite optimizing an additional criterion.

Although ULTRA-TB with sequential preprocessing still outperforms other algorithms, it is slower than the integrated version by a factor of two. This is because the integrated preprocessing reduces the number of relaxed shortcuts by around an order of magnitude. This in turn reduces the overall search space and thereby the number of scanned trips. Overall, the trip scanning phase is sped up by a factor of three to four and only takes up around half of the overall query time. The remaining half is spent performing the Bucket-CH searches and evaluating initial trips, both of which are unaffected by the number of transfer shortcuts.

Table 5.6: Query performance for TB and 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 TB (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	TB*	○	10.9	223	0.0	0.0	1.5	1.6
	ULTRA-TB (seq.)	●	25.1	1 417	1.2	1.0	5.8	7.9
	ULTRA-TB (int.)	●	15.3	112	1.1	0.8	1.7	3.6
London	TB*	○	15.3	830	0.0	0.0	3.7	3.7
	ULTRA-TB (seq.)	●	23.5	1 021	0.8	0.7	5.1	6.6
	ULTRA-TB (int.)	●	14.5	153	0.8	0.6	1.9	3.3
Switzerland	TB*	○	23.4	662	0.0	0.0	4.5	4.5
	ULTRA-TB (seq.)	●	34.9	1 620	1.0	1.2	7.1	9.3
	ULTRA-TB (int.)	●	19.5	138	1.0	1.0	2.2	4.3
Germany	TB*	○	389.1	16 331	0.0	0.0	106.6	106.9
	ULTRA-TB (seq.)	●	467.5	43 219	19.9	19.3	162.6	202.0
	ULTRA-TB (int.)	●	196.5	2 057	19.6	19.3	37.9	77.0

Impact of Transfer Speed. In addition to the overall query performance, we also measure how the query times of MR, ULTRA-RAPTOR and ULTRA-TB are impacted by the transfer speed. The results are shown in Figure 5.7 (left). The performance gains of ULTRA-RAPTOR over MR are similar for all transfer speeds and, in fact, slightly better for higher speeds. To explain this, we observe that the route scanning phase becomes faster 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. Because ULTRA-RAPTOR increases the share of the route scanning phase in the overall running time compared to MR, it benefits more from this effect. For all speeds, the entire query time for ULTRA-RAPTOR is similar to or lower than the time that MR takes for the route scanning phases only. ULTRA-TB achieves its highest speedup over the other two algorithms for medium transfer speeds, which yields the lowest number of event-to-event shortcuts. For high transfer speeds, the Bucket-CH search for the initial and final transfers starts to dominate the running time of both ULTRA-based algorithms. Accordingly, the speedup of ULTRA-TB over ULTRA-RAPTOR decreases.

The impact of the transfer speed on the travel time of the fastest journey is shown in Figure 5.7 (right). As the transfer speed increases, the overall travel time decreases. The time that is spent on an initial or final transfer also decreases at first, but its share in the overall

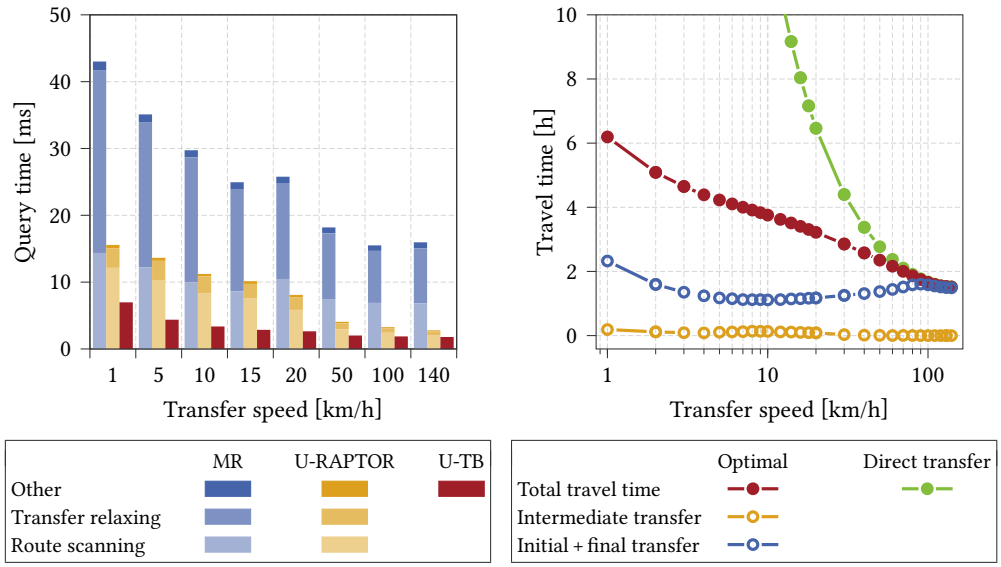


Figure 5.7: Impact of transfer speed on query times and travel times, measured on the Switzerland network with a core degree of 14 and a witness limit of 0. All results were averaged over 10 000 random queries. *Left:* Query performance of MR, ULTRA-RAPTOR and ULTRA-TB. Speed limits were obeyed during the construction of the transfer graph. For MR and ULTRA-RAPTOR, 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. The travel time of a direct transfer from source to target is shown for reference. For this comparison, we only chose random queries for which the source and target vertex are connected in the transfer graph.

travel time becomes larger. From 10 km/h onward, transferring directly from the source to the target starts becoming the best option for more queries, and consequently the time spent on initial and final transfers starts increasing. For very high transfer speeds, a direct transfer is almost always the fastest option. This matches our observation that intermediate transfers become useless for high transfer speeds unless the source and target are isolated from each other in the transfer graph. 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.

Impact of Query Distance. Figure 5.8 compares the running times of the two-criteria algorithms (MR, ULTRA-RAPTOR, and ULTRA-TB) depending on the query distance. We use the geo-rank as a measurement for the distance. Geo-rank queries are generated by picking a

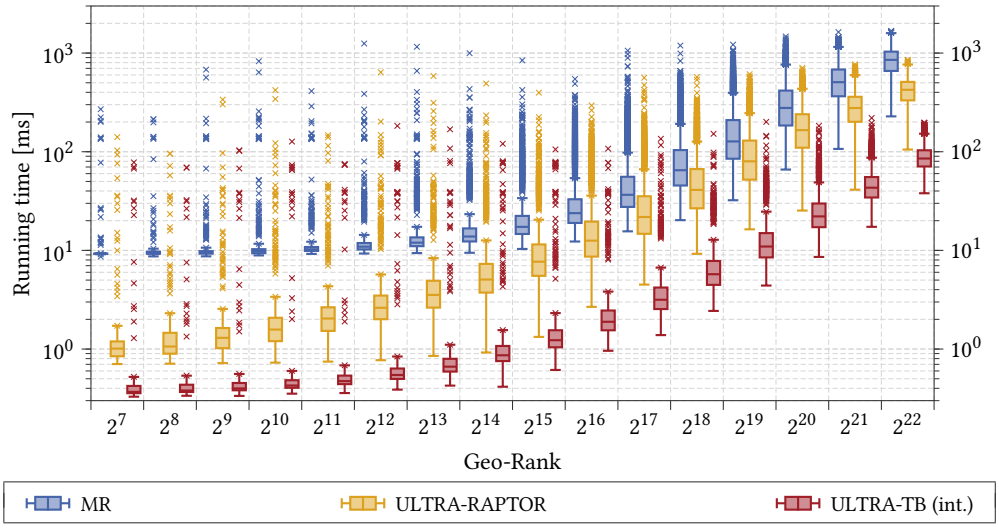


Figure 5.8: Comparison of query times depending on the geo-rank for the Germany network. For each geo-rank, we evaluated 10 000 random vertex-to-vertex queries. We compare the previously fastest multimodal algorithm (MR) to ULTRA-RAPTOR and ULTRA-TB.

source vertex v_s uniformly at random and sorting all vertices by their geographical distance to v_s. The i-th vertex in this order is then the target of the geo-rank query for rank i. For our comparison in Figure 5.8, we generated and evaluated 10 000 of these queries for the Germany network. For all geo-ranks, ULTRA-TB is an order of magnitude faster than MR. ULTRA-RAPTOR lies between these two algorithms; it is closer to ULTRA-TB for local queries and 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-TB with integrated preprocessing.

A geo-rank-based evaluation on the Germany network was also performed for the original TB algorithm by Witt [Wit15]. The results are similar to ours but contain significantly more outliers, especially for low ranks. Across all geo-ranks, the evaluation for the original TB algorithm shows a considerable number of queries that take more than 10 milliseconds. These can be attributed to queries for which the source vertex is located in particularly sparse parts of the public transit network. In these regions, the correlation between geo-rank and actual distance is poor, and thus a query can be a long-range query despite having a low geo-rank. Adding an unrestricted transfer graph improves the correlation between geo-rank and query complexity, which explains why we observe fewer outliers in comparison.

Table 5.7: Overview of the HL preprocessing results. Running times were measured on the Xeon machine with a single core and are displayed as (hh:)mm:ss. Only the outgoing hub edges are reported. Because all evaluated transfer graphs are symmetrical, the number of incoming hub edges is identical.

	Stuttgart	London	Switzerland	Germany
Preprocessing time	1:07:14	3:56	21:14	52:48:23
Outgoing hub edges of vertices	153 323 291	13 314 082	53 744 836	1 320 767 674
Outgoing hub edges of stops	1 898 414	1 363 960	1 952 586	45 075 714
Average vertex out-degree	131.4	73.3	89.0	192.2

5.3.3 Comparison to HL-Based Approach

We conclude with a comparison of ULTRA to the HL-based algorithms proposed by Phan and Viennot [PV19]. Recall that the speedups reported by the authors are based on a comparison between experiments performed on different machines. To allow for a fair comparison, we re-implemented the HL-RAPTOR query, building on the same RAPTOR code that we also used for MR and ULTRA-RAPTOR.

Preprocessing. For the HL preprocessing, we used the HL implementation provided by the authors[1]. Results for the preprocessing phase are reported in Table 5.7. Note that Phan and Viennot only compute hubs between pairs of stops because they only evaluate their algorithms for stop-to-stop queries. In order to support vertex-to-vertex queries on the full transfer graph, we compute hubs between all pairs of vertices. Unlike ULTRA, the HL preprocessing is not easily parallelizable and was therefore run on a single core. Although the HL preprocessing has a lower single-core preprocessing time, ULTRA becomes significantly faster with parallelization. An exception to this is the London network, which has a particularly small transfer graph but a large and complex public transit network. Because the HL preprocessing operates only on the transfer graph whereas ULTRA has to consider both, ULTRA is outperformed here even when parallelized. Regarding memory consumption, ULTRA clearly outperforms HL because the number of shortcuts is much smaller than the number of hub edges.

Queries. Table 5.8 compares the query performance of HL-RAPTOR to MR and ULTRA-RAPTOR. We choose not to evaluate HL-CSA because the results reported by Phan and Viennot indicate that it is only marginally faster than HL-RAPTOR, and therefore slower than MCSA. This was confirmed by our preliminary experiments. We observe that HL-RAPTOR only slightly outperforms MR on the three smaller networks and is slower on Germany. Its speedup comes entirely from the initial transfer exploration. In the intermediate transfer

[1] https://github.com/lviennot/hub-labeling/

Table 5.8: Query performance for MR, HL-RAPTOR, and ULTRA-RAPTOR. Query times are divided into phases: initialization, including scanning initial transfers (*Init.*), collecting routes (*Coll.*), scanning routes (*Scan*), and relaxing transfers (*Relax*). All results are averaged over 10 000 random queries.

Network	Algorithm	Scans [k]		Time [ms]				
		Route	Edge	Init.	Coll.	Scan	Relax	Total
Stuttgart	MR	35.6	687	12.3	5.2	5.2	11.1	33.5
	HL-RAPTOR	39.3	3 068	1.9	5.3	5.1	18.2	30.7
	ULTRA-RAPTOR	37.9	105	1.4	3.5	3.5	1.0	9.6
London	MR	5.0	500	6.4	1.9	2.7	7.0	18.0
	HL-RAPTOR	5.6	1 599	0.9	1.8	2.7	8.0	13.4
	ULTRA-RAPTOR	5.4	179	1.2	1.5	2.3	1.2	6.2
Switzerland	MR	33.0	731	10.6	4.8	7.2	11.7	34.1
	HL-RAPTOR	38.4	2 337	1.6	5.0	7.8	17.8	32.1
	ULTRA-RAPTOR	35.9	177	1.6	3.3	6.2	1.4	12.5
Germany	MR	541.4	12 359	154.2	187.5	153.5	236.2	731.4
	HL-RAPTOR	629.4	41 773	28.8	214.8	183.4	381.4	808.3
	ULTRA-RAPTOR	599.7	3 165	33.0	144.0	151.7	33.3	362.1

phase, using the precomputed hubs is actually slower than the Dijkstra search performed by MR. This is due to the very high number of edges in the hub graph. The first few rounds of a query typically reach most stops in the network, so most hub edges that are incident to a stop need to be relaxed. This causes HL-RAPTOR to relax more than three times as many edges as MR. HL-RAPTOR performs best on the London network, achieving a speedup of 1.3 over MR. By comparison, ULTRA-RAPTOR is faster than the HL-based approach by a factor of more than two on all networks.

5.4 Conclusion

We revisited the shortcut hypothesis for a simple setting with one transfer mode and two-criteria Pareto optimization and presented a more sophisticated speedup technique, ULTRA, to exploit it. The centerpiece of ULTRA is a redesigned shortcut computation algorithm that provably identifies all necessary intermediate transfers. Compared to the prototypical approach by Sauer, ULTRA employs more effective optimizations to reduce the preprocessing time while keeping the number of shortcuts low. With parallelization, the shortcut computation takes only a few minutes for metropolitan and mid-sized country networks. Because the computational effort is quadratic in the number of stops, the preprocessing times are

higher for Germany but still manageable at about three hours. Our experiments offer strong support for the shortcut hypothesis in this problem setting. Regardless of the speed of the transfer mode, the shortcut graph is much smaller than the transitively closed transfer graph constructed in Chapter 4.3, even though it supports unrestricted transfers.

We proposed a generalized query framework that allows ULTRA shortcuts to be employed by any public transit algorithm that requires one-hop transfers. This enables the computation of unrestricted multimodal journeys without incurring the performance losses of existing algorithms. In particular, combining ULTRA with CSA yields the first efficient multimodal variant of CSA. To combine ULTRA with TB, we developed tailored versions of the ULTRA preprocessing and the TB query. The resulting ULTRA-TB algorithm closes the performance gap for the two-criteria setting and outperforms MR, the fastest previously known multimodal algorithm for two-criteria optimization, by an order of magnitude.

6 One-to-Many Search

The ULTRA framework as presented so far does not support one-to-all or one-to-many searches. This is because the final transfers are explored with a backward Bucket-CH search, which requires a single target vertex from which it is run. A naive solution [SWZ19b, SWZ19a] for one-to-many searches would be to perform one Bucket-CH search for each target vertex. However, this is only viable for very small target sets, as the running time is proportional to the number of targets. In this chapter, we therefore present a more sophisticated approach to adapt ULTRA for one-to-many queries.

Related Work and Applications. Although studied extensively for road networks, one-to-all and one-to-many problems have received little attention on multimodal networks. One application that has been studied in a multimodal context is the computation of isochrones, which are regions that are reachable from a given point within a specified time limit. Thus far, algorithmic approaches for multimodal isochrones have been limited to applying Dijkstra's algorithm to a graph representation of the network [GBCI11, GBI12, KSSG17]. Another application is simulation-based traffic assignment for public transit, such as the CSA-based algorithm by Briem et al. [Bri+17]. Here, many-to-many routing is used to simulate the movement of individual agents. An ULTRA-based multimodal variant of this algorithm has already been proposed [SWZ19b, SWZ19a]. However, it relies on a naive adaptation of ULTRA to a many-to-one setting, which is only feasible if the number of origin locations for the passenger demand is fairly small. A scalable multimodal one-to-all algorithm would enable the computation of full door-to-door assignments.

Isochrones, traffic assignment and other potential applications, such as POI queries, often only require optimization of a single criterion. An area that requires one-to-many Pareto optimization is the design of speedup techniques for one-to-one queries. Examples of public

transit algorithms that employ one-to-many searches in a preprocessing phase are Transfer Patterns [Bas+10] and Access Node Routing [DPW09a]. So far, no comparable speedup technique has been developed for multimodal networks, partly due to prohibitively high preprocessing costs. A more efficient algorithm for multimodal one-to-many search could be a first step towards developing such a technique.

Chapter Outline. We present our extension of the ULTRA framework to one-to-many queries in Section 6.1. The resulting algorithm scheme, ULTRA-PHAST, is the first efficient approach for multimodal one-to-many search. In order to explore the final transfers to the target vertices, ULTRA-PHAST adopts ideas from the RPHAST algorithm. Note that this requires us to solve a more challenging problem than a regular one-to-many query because every stop reached via the public transit network is a potential source vertex of a final transfer.

Similarly to ULTRA, ULTRA-PHAST can be combined with any public transit algorithm that supports one-to-all search. We demonstrate this for three such combinations: For queries optimizing only arrival time, we evaluate UP-CSA, the combination of ULTRA-PHAST and CSA. For the Pareto optimization of arrival time and number of trips, we propose an additional optimization that groups the final transfer searches for different numbers of trips. We evaluate this in combination with RAPTOR, yielding UP-RAPTOR. Additionally, in Section 6.2 we propose a one-to-many variant of TB, which previously only supported one-to-one queries, and combine it with ULTRA-PHAST. We present experimental results in Section 6.3. For a small to moderate number of targets, our algorithms are almost as fast as their public transit counterparts. For large target sets, we achieve a speedup of up to an order of magnitude compared with naive baseline algorithms. Finally, we offer concluding remarks and discuss potential applications in Section 6.4.

6.1 ULTRA-PHAST

To handle multiple target vertices, ULTRA-PHAST replaces the Bucket-CH backward search for the final transfers with a forward search inspired by PHAST. We first outline our approach for the earliest arrival problem in Section 6.1.1. Afterward, we extend it to the Pareto optimization problem in Section 6.1.2.

6.1.1 Earliest Arrival Queries

The naive approach of performing one Bucket-CH search per target solves a many-to-many problem, computing the distances between all stops and all targets. This is more information than is required in our case: For each target v_t, we only need the distance from a single stop, namely the stop at which the last used trip is exited in the optimal journey to v_t. The difficulty lies in the fact that we do not know this stop in advance. However, we can reformulate the final transfer search as a one-to-many problem and solve it using PHAST in the following manner: First, we compute the earliest arrival time at each stop $v \in \mathcal{S}$, using a standard ULTRA query

Algorithm 6.1: ULTRA-PHAST query algorithm.

1 Dijkstra search from v_s in $G^\uparrow$, initialized with τ_{dep}
2 Downward sweep in $G^\downarrow[\mathcal{S}]$
3 Initialize the public transit algorithm with the stop arrival times found in line 2
4 Run the public transit algorithm without target pruning
5 Upward sweep in $G^\uparrow[\mathcal{S}]$, initialized with the arrival times found in line 4
6 Downward sweep in $G^\downarrow[V_t]$

without the backward Bucket-CH search and without target pruning. Afterward, we insert a temporary edge (v_s, v) with transfer time $\tau_{arr}(v) - \tau_{dep}$ into the PHAST upward graph $G^\uparrow$. We can then find the earliest arrival time at every target with a single PHAST search on our augmented graph $G^\uparrow$. If we are also interested in the corresponding journey, we can simply substitute the temporary edge (v_s, v) with the journey to v found by the ULTRA query. In practice, we do not actually insert temporary edges into $G^\uparrow$. Instead, we initialize the priority queue used for the search in $G^\uparrow$ by directly inserting each stop v with $\tau_{arr}(v)$ as its distance.

As presented thus far, our approach still has a performance issue: The efficiency of the upward search in $G^\uparrow$, which comprises the first phase of PHAST, relies on the fact that the upward search space of a single source vertex is small. However, we perform an upward search from all reached stops simultaneously. Hence, the search space of our upward search will be the union of the search spaces of all stops, which is a large portion of the graph.

Efficient Upward Search. In order to improve the efficiency of the upward search, we optimize its memory and cache usage. First, we note that only vertices in the upward search space of a stop are relevant for our algorithm. Because the set of stops does not change between queries and is known beforehand, we can perform a *stop selection* analogous to the target selection in RPHAST: We run a forward BFS on $G^\uparrow$ from all stops simultaneously and remove all vertices that are not visited. The resulting stop-selected upward graph is denoted as $G^\uparrow[\mathcal{S}]$. Furthermore, we observe that if the transfer graph is strongly connected, every query will reach every stop, regardless of the source vertex. Thus, every vertex in the stop-selected upward graph will be visited during the upward search. We can therefore replace the Dijkstra search in $G^\uparrow[\mathcal{S}]$, which requires a priority queue, with a more efficient upward sweep that is done analogously to the downward sweep of PHAST. If the transfer graph is not strongly connected, such a sweep might scan many unreachable stops. Thus, we modify the ULTRA query to keep track of the stop with the lowest rank that has been reached and start the upward sweep at this stop.

Algorithm Overview. The algorithmic framework for our one-to-many approach, which we call ULTRA-PHAST, is outlined in Algorithm 6.1. The original ULTRA query explores initial transfers with a Bucket-CH search from v_s, using the results of a backward Bucket-CH

search from the target vertex to prune the search space. Because this pruning technique is no longer applicable in a scenario with multiple target vertices, the initial transfer search will reach all stops that are reachable from v_s. In this case, it is more efficient to explore the initial transfers with an RPHAST search to $\mathcal{S}$ instead of Bucket-CH. The RPHAST search consists of a Dijkstra search from v_s in the CH upward graph $G^\uparrow$ (line 1) and a downward sweep on the stop-selected downward graph $G^\downarrow[\mathcal{S}]$ (line 2). The public transit network is then explored using a black-box public transit algorithm without target pruning. This algorithm is initialized with the arrival times at the stops found by the RPHAST search in line 3 and then run in line 4. It outputs minimal arrival times for all stops in the network, which are then propagated to the target set using a final upward and downward sweep in lines 5 and 6. Because the upward sweep is equivalent to an RPHAST downward sweep in reverse, it has a similar running time. Thus, the total running time of an ULTRA-PHAST query is roughly equal to the combined running time of a public transit query without target pruning, two RPHAST queries to $\mathcal{S}$, and one RPHAST query to V_t.

Optimized Contraction Order. The three sweeps can be further sped up by delaying the contraction of stops and targets during the CH precomputation. Specifically, delaying the contraction of stops will reduce the number of vertices in $G^\downarrow[\mathcal{S}]$ and $G^\uparrow[\mathcal{S}]$, whereas delaying the contraction of targets will reduce the number of vertices in $G^\downarrow[V_t]$. However, this is only useful up to a certain point because eventually the quality of the contraction order will degrade. This will lead to an unreasonable preprocessing time and cause too many shortcuts to be inserted, which will in turn slow down the sweeps. We take this into account by introducing tuning parameters f_s and f_t that determine how much the contraction of stops and targets is delayed, respectively. Initially, only vertices that are neither a stop nor a target may be contracted. Once fewer than $f_t \cdot |\mathcal{S} \cup V_t|$ uncontracted vertices remain, we also allow targets to be contracted. Stops remain uncontractable until fewer than $f_s \cdot |\mathcal{S}|$ vertices remain.

Vertex Reordering. As demonstrated by Delling et al. [DGNW13], the order in which the vertices of a graph are stored in memory can have a significant impact on the performance of a journey planning algorithm. In particular, the order in which vertices are settled by a depth-first search (DFS) has been shown to yield a good memory locality for Dijkstra-like searches. For the PHAST-like sweeps in the upward graph $G^\uparrow[\mathcal{S}]$ as well as the downward graphs $G^\downarrow[\mathcal{S}]$ and $G^\downarrow[V_t]$, the vertices must be scanned in a topological order. We obtain a topological ordering by performing a DFS on $G^\uparrow$ and reorder the vertices according to it. Preliminary experiments have shown that this order performs at least as well as the level order used by PHAST, which was chosen primarily because it allows for easy parallelization.

Implementation Details. Although the topological ordering of the vertices improves the performance of the sweeps, it is inefficient for the public transit part of the query. Many public transit algorithms, including RAPTOR or CSA, achieve a large part of their efficiency by keeping the stop data consecutive in memory. One way to achieve this in multimodal

Algorithm 6.2: Downward sweep to target set V_t.

```
1  timestamp++
2  for v ∈ V↓[V_t] in topological order do
3      if timestamp[v] ≠ timestamp then
4          timestamp[v] ← timestamp
5          τ_arr[v] ← ∞
6      for each e ← (w, v) ∈ E↓[V_t] do
7          τ_arr^new ← τ_arr[w] + τ_tra[e]
8          update ← τ_arr^new < τ_arr[v]
9          if update then τ_arr[v] ← τ_arr^new          // conditional move
10         if update then p[v] ← p[w]                   // conditional move
```

scenarios is to assign vertex IDs between 0 and $|\mathcal{S}| - 1$ to the stops, and IDs between $|\mathcal{S}|$ and $|V| - 1$ to the remaining vertices. However, this conflicts with the topological ordering used for the RPHAST-like sweeps. Thus, we use different vertex orderings and IDs for the public transit data structures and the RPHAST data structures, translating between them whenever we switch between RPHAST and public transit searches. For the public transit data structures, we assign IDs from 0 to $|\mathcal{S}| - 1$ to the stops in such a manner that their relative positions according to the topological order are preserved. This ensures that the two orders are as similar as possible, and that sweeping over one ID range still requires only a single sweep over the other.

Detailed pseudocode for one of the three sweeps (line 6 of Algorithm 6.1) is given in Algorithm 6.2. The outer for-loop iterates over the vertices of the target-selected downward graph $G^{\downarrow}[V_t]$ in topological order. For each vertex v, the inner loop in lines 6–10 relaxes the incoming edges. Within the inner loop, parent pointers $\mathtt{p}[\cdot]$ (which are required for journey retrieval) and arrival times are updated frequently, but only if the arrival time of v is improved. It is crucial for the performance of the sweep that these updates are implemented without branching operations, since this would cause costly pipeline flushes if the branch prediction fails. We therefore use conditional move operations to update the arrival time and parent pointer in lines 9 and 10. To avoid resetting the arrival times of all vertices before each sweep, the algorithm maintains a global timestamp, which is incremented at the start of each sweep, and a local timestamp $\mathtt{timestamp}[v]$ for each vertex v. Whenever a new vertex v is accessed in the outer loop, lines 3–5 compare $\mathtt{timestamp}[v]$ to the global timestamp. If they are not equal, the timestamp of v is updated and its arrival time is reset. Note that because the sweep processes the vertices in topological order, there is no need to check the timestamp of the vertex w in line 7 because it was already processed by the outer loop.

6.1.2 Optimizing Number of Trips

We proceed with describing how our approach can be extended to find a two-criteria Pareto set (optimizing arrival time and number of trips) for every target. Because the maximum number of trips required by any Pareto-optimal journey is usually quite low, it is feasible to simply perform the final upward and downward sweep of our algorithm once for every possible number of trips. Furthermore, we can apply an optimization that was originally proposed for speeding up multiple PHAST searches from different source vertices [DGNW13]: Given a fixed parameter k, we no longer explore the final transfer for journeys using between 0 and $k-1$ trips with k separate upward and downward sweeps. Instead, we perform one upward and downward sweep that update all k arrival time values at once. Note that k must be a fixed value because the sweeps are only efficient if the arrival times are stored consecutively in arrays of a fixed size k.

Journeys using k or more trips are not handled by this grouped sweep. However, we observe that only a few stops are reached by Pareto-optimal journeys that require a high number of trips. Propagating such journeys via a PHAST sweep, which always explores the entire graph, is wasteful because it will not improve the arrival times of most vertices. Thus, for journeys using k or more trips, we switch to Dijkstra searches on a contracted transfer graph that contains all stops and targets, in a similar manner to MR. Similarly to the sweeps, the Dijkstra searches use timestamps to initialize only the labels of visited vertices. However, when the label of a vertex is initialized, we do not set its arrival time to ∞, but to the best arrival time found during the grouped sweeps. This ensures that journeys that are dominated by journeys with fewer trips are pruned early on.

6.2 Integration with Trip-Based Routing

Combining ULTRA-PHAST with TB requires a one-to-all variant of TB. For CSA and RAPTOR, supporting one-to-all queries is trivial because they already maintain optimal solutions at every reached stop. Thus, a one-to-all query can be answered correctly by simply disabling target pruning. This is not the case for TB because it maintains solutions only for the target stop. Although a one-to-all variant of TB was previously used in the preprocessing phase of TB-CST [Wit16], it was not described in detail. We therefore give a thorough description of UP-TB, which combines one-to-all TB with ULTRA-PHAST.

Whereas the original TB query maintained a set of Pareto-optimal labels only at the target stop, UP-TB maintains arrival times at every stop. For each stop v and number of trips n, the earliest arrival time via a trip is given by $\tau_{\text{arr}}(v, n)$. For $n = 0$, it is initialized with ∞. Following the ULTRA-PHAST framework, initial transfers are explored with an RPHAST search (cf. lines 1–3 of Algorithm 6.1). Initial trip segments are then collected using the same approach described for the ULTRA-TB query in Chapter 5.2.2: for each route visited by an initial transfer, the earliest reachable trip segments are identified and collected in a FIFO queue Q_1. Afterward, the algorithm proceeds in rounds: In round n, the arrival time $\tau_{\text{arr}}(v, n)$

Algorithm 6.3: Modified trip scanning procedure for the UP-TB query algorithm.

```
1  Procedure ScanTrips(Q_n, Q_{n+1})
2      for each T[j,k] ∈ Q_n do
3          for i from j to k do
4              v ← v(T[i])
5              if τ_arr(T[i]) ≥ τ_arr(v,n) then continue
6              τ_arr(v,n) ← τ_arr(T[i])
7      for each T[j,k] ∈ Q_n do
8          k' ← j − 1
9          for i from j to k do
10             v ← v(T[i])
11             if τ_arr(T[i]) > τ_arr(v,n) then continue
12             k' ← i
13         k ← k'
14     for each T[j,k] ∈ Q_n do
15         for i from j to k do
16             for each (T[i], T'[i']) ∈ E^s do
17                 Enqueue(T'[i'+1], Q_{n+1})
```

of each stop v is initialized with the arrival time $\tau_{\text{arr}}(v, n-1)$ from the previous round. Then the trip segments in Q_n are scanned by invoking the `ScanTrips` procedure. Newly reached trip segments are collected in the queue Q_{n+1} for the next round. This is repeated until Q_n is empty. Then, final transfers are explored with upward and downward sweeps (cf. lines 5–6 of Algorithm 6.1), using the arrival times $\tau_{\text{arr}}(\cdot, \cdot)$ as input.

The `ScanTrips` procedure must be modified to support one-to-all search; pseudocode is given in Algorithm 6.3. Each enqueued trip segment $T[j,k]$ is scanned three times. The first scan (lines 2–6) updates the arrival times for all stops visited by the trip segment. The second scan (lines 7–13) then uses the updated arrival times to apply local pruning. Consider a stop event $T[i]$ with $j \le i \le k$ and its visited stop $v = v(T[i])$. If the arrival time of $T[i]$ is later than the stop arrival time $\tau_{\text{arr}}(v, n)$, then $T[i]$ is not the fastest way to reach v with n trips, so its outgoing transfers do not need to be relaxed. The second scan identifies the last stop event $T[k']$ that cannot be discarded with local pruning and shortens the trip segment to $T[j,k']$. The third scan (lines 14–17) then relaxes the outgoing intermediate transfers of all stop events $T[i]$ with $j \le i \le k'$ by iterating over the set E^s of ULTRA shortcuts. This step is unchanged from the original TB algorithm.

Note that this approach does not apply local pruning to its full extent. It only removes the longest suffix of irrelevant stop events from $T[j,k]$; the outgoing transfers of irrelevant stop events between $T[j]$ and $T[k']$ are still relaxed. Although this slightly increases the search space, it allows the third scan to be implemented with a single for-loop instead of two nested

ones, as in the original TB algorithm (cf. Section 4.2.4). Because the outgoing shortcuts of consecutive stop events are stored consecutively in memory, the shortcuts that are relaxed in lines 15–17 form a continuous range that can be processed with a single for-loop.

6.3 Experiments

We evaluate our algorithms on the four benchmark networks, using walking with a constant speed of 4.5 km/h as the transfer mode. All experiments were conducted on the Xeon machine. The ULTRA shortcuts were computed using the same settings as in Chapter 5.3: The transfer graph was contracted up to an average vertex degree of 20 for Germany and 14 for the other networks. The shortcut computation was performed in parallel on all 16 cores with a witness limit of 0.

Baseline Algorithms. Because no multimodal algorithms that support one-to-many queries have yet been proposed, we create baseline algorithms for comparison by adapting the ideas of MR and MCSA (the latter of which was introduced in Chapter 5.3.2) to a scenario with multiple target vertices. MR and MCSA handle initial and final transfers by running forward and backward searches on the partial upward and downward graph constructed by Core-CH, followed by Dijkstra searches in the core graph. When adapting MR and MCSA to a one-to-many scenario, the forward search can be run unchanged, but the backward search is no longer feasible. Instead, we modify the Core-CH precomputation such that vertices in $\mathcal{S} \cup V_t$ may not be contracted, rather than just stops. The backward search then becomes unnecessary because the Dijkstra searches in the core graph already reach all targets. For our experiments, we contracted up to an average vertex degree of 14, except for very large target sets with $4|V_t| \geq |V|$, for which we used a vertex degree of 10 instead.

Target Sets. For our experiments, we consider three types of target sets: all vertices, all stops, and randomly generated target sets. For the randomly generated target sets, we follow the approach by Delling et al. [DGW11]: We pick a center vertex $c \in V$ uniformly at random and then run a Dijkstra search from c to find a *ball* $\mathcal{B} \subseteq V$ consisting of the $|\mathcal{B}|$ nearest neighbors of c. From that ball, we then pick target vertices uniformly at random. We evaluate our algorithms for different combinations of ball size $|\mathcal{B}|$ and target set size $|V_t|$, to study the impact of both the number of targets and the distribution of the targets in the graph.

6.3.1 UP-CSA

For the earliest arrival problem, we compare UP-CSA, a combination of ULTRA-PHAST and CSA, to our one-to-many adaptation of MCSA.

Contraction Order. In Figure 6.1 (left), we evaluate the impact of the tuning parameters f_t and f_s on the performance of the three sweeps performed by ULTRA-PHAST: the

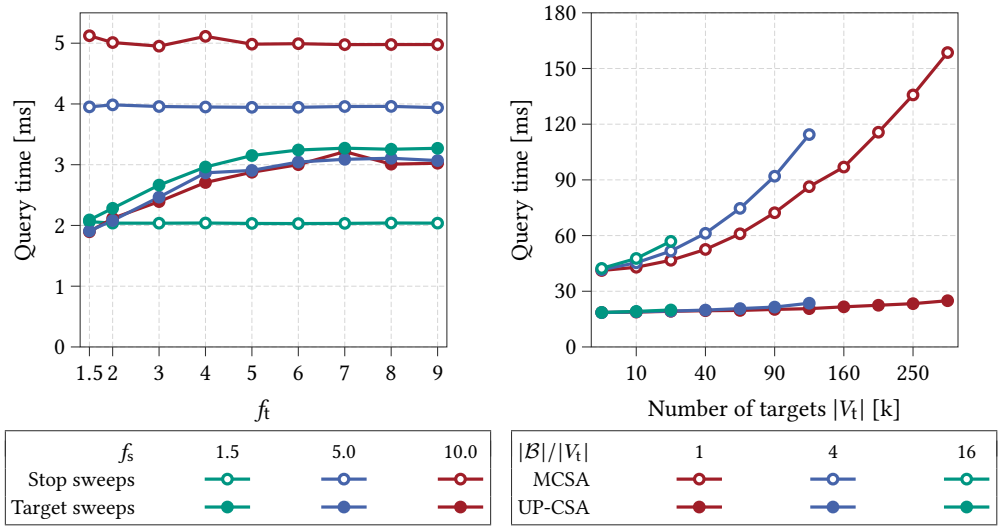


Figure 6.1: Impact of delayed contraction (left) and number of targets (right), measured on the Switzerland network. All running times are averaged over 1 000 queries each on 10 randomly chosen ball target sets. *Left:* Performance of the three ULTRA-PHAST sweeps depending on f_t and f_s, for ball target sets with $|V_t| = 2^{16}$ and $|\mathcal{B}| = 2^{18}$. *Right:* Performance of MCSA and UP-CSA for different values of $|V_t|$ and $|\mathcal{B}|$. Configurations with $|\mathcal{B}| > |V|$ were omitted.

downward sweeps in $G^{\downarrow}[\mathcal{S}]$ and $G^{\downarrow}[V_t]$, and the upward sweep in $G^{\uparrow}[\mathcal{S}]$. The contraction of stops and targets is prohibited until $f_t \cdot |\mathcal{S} \cup V_t|$ vertices are left, whereas stops are further left uncontracted until $f_s \cdot |\mathcal{S}|$ vertices remain. We observe that delaying the target contraction speeds up the target-related sweep by up to a factor of 1.6 without impacting the performance of the stop-related sweeps. Delaying the contraction of stops slightly increases the running time of the sweep in $G^{\downarrow}[V_t]$, but this is offset by the significant performance gains for the stop-related sweeps. For all following experiments involving ball target sets, we set both f_t and f_s to 1.5. The CH precomputation time for this configuration is 2:27 minutes, approximately three times as long as a CH precomputation without delayed contraction.

Target Set Size. Figure 6.1 (right) shows how the size and distribution of the target set impacts the performance of MCSA and UP-CSA. For both algorithms, the exploration of transfers becomes more costly as the target set, and thus the size of the search graphs, increases. The effect is much more pronounced for MCSA, for which the Dijkstra searches quickly start to dominate the overall running time. By contrast, UP-CSA spends a much smaller portion of the running time on exploring transfers. Accordingly, we only observe

Table 6.1: Preprocessing performance of MCSA and ULTRA-PHAST for three types of target sets: all vertices (V), all stops ($\mathcal{S}$), and vertices randomly chosen from a ball ($\mathcal{B}$). For the ball configuration, measurements are averaged over 10 randomly generated target sets with $|V_t| = 2^{17}$ for Germany, $|V_t| = 2^{14}$ for the other networks, and $|\mathcal{B}|/|V_t| = 2$ for all networks. All times are in (m)m:ss format. Preprocessing times for ULTRA-PHAST exclude the ULTRA shortcut computation.

Network	V_t	Core-CH time	ULTRA-PHAST		
			CH time	$\|V^{\downarrow}[V_t]\|$	$\|E^{\downarrow}[V_t]\|$
Stuttgart	V	–	4:10	1 166 604	4 599 303
	$\mathcal{S}$	1:45	4:10	19 376	266 130
	$\mathcal{B}$	2:47	7:35	22 280	242 493
London	V	–	0:17	181 642	697 593
	$\mathcal{S}$	0:19	0:17	29 478	166 170
	$\mathcal{B}$	0:32	0:32	19 940	126 424
Switzerland	V	–	1:28	603 691	2 360 885
	$\mathcal{S}$	1:09	1:28	37 669	284 328
	$\mathcal{B}$	1:39	2:18	19 662	148 182
Germany	V	–	29:04	6 870 496	27 700 776
	$\mathcal{S}$	20:16	29:01	364 689	3 535 769
	$\mathcal{B}$	18:35	47:23	152 845	1 304 285

a 30% increase in the running time between the fastest and slowest configuration. Increasing the ball size causes the stop and target selection to become less effective, as the targets are spread over a wider area of the graph. However, this only has a small effect on the overall performance of UP-CSA.

Detailed Performance. Tables 6.1 and 6.2 give a detailed overview of the performance of MCSA and UP-CSA for three types of target sets: all stops, all vertices, and a ball target set of moderate size. For the ball target sets, we use contraction delay factors of $f_s = f_t = 1.5$. For the other two sets, for which delaying the contraction of targets is pointless, we achieve the best performance with $f_s = 1.5$. Preprocessing times and graph sizes are reported in Table 6.1. Unlike MCSA, UP-CSA requires a target-independent preprocessing phase to compute the ULTRA shortcuts. For most configurations, this phase dominates the overall preprocessing time. Once the target set is chosen, MCSA performs a Core-CH precomputation, whereas UP-CSA performs a stop- and target-delayed CH precomputation, reorders the vertices and performs the stop and target selection. The time required for the latter steps is negligible (about a second on the smaller networks and 15 seconds on Germany). The delayed CH

Table 6.2: Query performance of MCSA and UP-CSA, using the same experimental setup as in Table 6.2. Measurements are averaged over 10 000 random queries, which are distributed evenly among the 10 target sets for the ball configuration. Query times are divided into phases: initialization, including initial transfers (*Init.*), connection scan (*Scan*), final upward sweep (*Up*), and final downward sweep (*Down*).

Network	V_t	Algorithm	Query time [ms]				
			Init.	Scan	Up	Down	Total
Stuttgart	V	MCSA	205.9	269.5	–	–	475.4
		UP-CSA	0.7	4.8	0.7	18.9	25.2
	$\mathcal{S}$	MCSA	7.3	13.4	–	–	20.8
		UP-CSA	0.7	4.3	0.7	0.5	6.3
	$\mathcal{B}$	MCSA	8.5	15.0	–	–	23.5
		UP-CSA	0.9	4.1	0.9	0.7	6.5
London	V	MCSA	18.7	36.2	–	–	55.1
		UP-CSA	0.6	9.2	0.6	2.4	12.8
	$\mathcal{S}$	MCSA	4.0	17.3	–	–	21.3
		UP-CSA	0.6	8.6	0.6	0.4	10.2
	$\mathcal{B}$	MCSA	6.1	19.8	–	–	26.0
		UP-CSA	0.7	8.4	0.7	0.4	10.2
Switzerland	V	MCSA	79.1	161.0	–	–	240.1
		UP-CSA	0.9	17.5	1.0	9.6	29.0
	$\mathcal{S}$	MCSA	7.3	33.2	–	–	40.5
		UP-CSA	0.9	17.2	1.0	0.7	19.7
	$\mathcal{B}$	MCSA	9.5	36.5	–	–	46.1
		UP-CSA	1.1	16.4	1.1	0.5	19.1
Germany	V	MCSA	1 892.0	4 295.2	–	–	6 187.2
		UP-CSA	11.4	343.9	12.9	185.5	553.7
	$\mathcal{S}$	MCSA	117.1	670.7	–	–	787.9
		UP-CSA	11.8	391.5	13.0	8.1	424.4
	$\mathcal{B}$	MCSA	163.9	791.4	–	–	955.3
		UP-CSA	12.6	338.1	15.2	5.5	371.5

precomputation is slower than the Core-CH precomputation in most configurations, but never by more than a factor of three. In terms of space consumption, both algorithms are lightweight: MCSA requires the core graph and the upward and downward graph generated by the Core-CH precomputation, which are similar in size to the original graph. ULTRAPHAST requires the set of shortcuts and the three sweep graphs $G^{\downarrow}[\mathcal{S}]$, $G^{\uparrow}[\mathcal{S}]$ and $G^{\downarrow}[V_t]$.

Table 6.3: Performance of the UP-RAPTOR final transfer exploration depending on the number k of grouped sweeps. Running times are averaged over 10 000 random queries with the set $\mathcal{S}$ of stops as the target set.

Network		Sweep [ms]			Dijkstra [ms]			Total [ms]		
	$k =$	4	6	8	4	6	8	4	6	8
Stuttgart		2.9	4.5	6.3	3.7	1.5	0.5	6.6	6.0	6.8
London		2.1	3.1	4.1	3.0	0.5	0.0	5.0	3.5	4.1
Switzerland		3.5	5.4	7.4	5.2	0.6	0.0	8.7	5.9	7.4
Germany		45.5	69.6	101.4	166.9	29.0	2.2	212.4	98.6	103.6

The size of the latter is listed in Table 6.1, whereas the size of the former two can be inferred from the $V_t = \mathcal{S}$ configuration, in which case all three graphs are of nearly identical size.

Query performance is detailed in Table 6.2. On the smaller target sets, UP-CSA is roughly two to three times faster than MCSA. On all networks except for Stuttgart, the connection scanning phase takes up more than 80% of the running time. This indicates that the performance of UP-CSA is close to the optimum that can be achieved with CSA. If all vertices are chosen as targets, MCSA is particularly inefficient because its main optimization (contracting the transfer graph) is no longer possible. By substituting the Dijkstra searches with memory-efficient sweeps, UP-CSA reduces the share of the transfer graph exploration in the overall query time to 30–40% on most networks. This results in a speedup over MCSA of 4.3 on London, 8.3 on Switzerland and 11.2 on Germany. Again, the Stuttgart network is an outlier because its transfer graph is very large in comparison to the public transit network. Thus, although the transfer graph exploration still takes up 80% of the running time, the speedup over MCSA is particularly high at 18.9.

We also evaluate how the RPHAST downward sweep for the initial transfers compares to a Bucket-CH search, which is used by the original ULTRA algorithm: On Switzerland, a Bucket-CH search takes 1.8 ms compared to 0.9 ms for RPHAST. On Germany, it takes 46.4 ms, which is more than the forward and backward Bucket-CH searches performed by ULTRA combined, whereas RPHAST requires only 11.4 ms. This demonstrates that although Bucket-CH is more efficient in a one-to-one scenario because it allows for target pruning, RPHAST is clearly superior in a one-to-many context.

6.3.2 UP-RAPTOR and UP-TB

For the Pareto optimization problem, we combine ULTRA-PHAST with RAPTOR as well as our implementation of one-to-many TB. We compare the resulting algorithms, UP-RAPTOR and UP-TB, to one-to-many MR. The Dijkstra searches for the non-grouped rounds operate on the same core graph that is also used by MCSA, MR and the ULTRA shortcut computation. To determine the best choice for the number k of grouped sweeps, we evaluate the performance

of UP-RAPTOR for random queries with the target set $\mathcal{S}$. The results are shown in Table 6.3. The best tradeoff is achieved for $k = 6$. For $k = 4$, the Dijkstra searches for the later rounds still take up the majority of the running time. For $k = 8$, the time for the Dijkstra searches becomes negligible, but this is outweighed by the increased cost for the sweeps. For all subsequent experiments, we choose $k = 6$ unless otherwise noted.

Detailed performance measurements for MR, UP-RAPTOR and UP-TB are presented in Table 6.4. For $V_t = V$, UP-RAPTOR achieves a speedup between 3.8 and 7.8 compared with MR. As expected, this is lower than the speedup of UP-CSA over MCSA. As the search space of the Dijkstra searches performed by MR becomes smaller in the later rounds, the performance gains of the RPHAST sweeps decline. Grouping the sweeps helps counteract this effect: although UP-RAPTOR performs eight or more rounds on average, exploring the final transfers only takes three to five times as long compared with UP-CSA. Because the share of the final transfer phase in the overall running time is very high (55–85%), UP-TB is only able to achieve a small speedup over UP-RAPTOR.

For $V_t = \mathcal{S}$, UP-RAPTOR achieves a speedup between 2.7 and 5.7 over MR and reduces the share of the final transfer phase in the running time to 13–29%. UP-TB improves the speedup further to 3.4–9.7 and increases the share to 32–49%. For this target set, the performance gains are much stronger than those observed for UP-CSA. This is mainly for two reasons: Firstly, MR makes no distinction between intermediate and final transfers, whereas UP-RAPTOR only explores intermediate transfers for which there is an ULTRA shortcut. Accordingly, UP-RAPTOR reaches fewer stops in each intermediate transfer phase, which in turn speeds up the route collection phase. This search space reduction has a stronger effect on RAPTOR than on CSA, which iterates over all connections, regardless of whether they are reachable. The second reason is that one-to-all MR has a high overhead due to the initialization of data structures. One-to-one MR maintains one arrival label per vertex, which is overwritten in each round. The only exception is the target vertex, at which one label is maintained for each round in order to store all Pareto-optimal journeys. In one-to-all MR, however, every vertex is a target, so it must maintain one label per vertex and round. At the start of each round, a significant amount of time is spent allocating these labels. By contrast, UP-RAPTOR mostly avoids this overhead. Because intermediate transfers are explored via direct stop-to-stop shortcuts, vertex arrival labels are only required for the initial and final transfers. For the k grouped rounds, these labels are allocated in advance and reset on demand using timestamps.

6.4 Conclusion

We adapted ULTRA for one-to-many and one-to-all queries. Because ULTRA explores initial and final transfers with a bidirectional search, which is not feasible for a large number of target vertices, we developed a new final transfer search that incorporates ideas from RPHAST. We replaced the upward CH search of RPHAST with an efficient upward sweep, taking into account that all stops that are reachable via a trip act as potential source vertices for the final transfer search. We also extended our approach to two-criteria Pareto optimization,

Table 6.4: Query performance of MR, UP-RAPTOR and UP-TB, using the same experimental setup as in Table 6.2. Query times are divided into phases: *Init.* includes the initialization of data structures and the exploration of initial transfers. *Collect* consists of the `CollectRoutes` procedure for the RAPTOR-based algorithms and `CollectInitialTrips` for UP-TB. *Inter.* represents the exploration of intermediate transfers (for the RAPTOR-based algorithms only). *Final* encompasses the exploration of final transfers, which consists of an upward and downward sweep for the grouped rounds and Dijkstra searches for the remainder. The number of grouped sweeps is set to $k = 6$ for all configurations except $V_t = V$ on Germany, for which $k = 8$ performs better.

Network	V_t	Algorithm	Time [ms]					
			Init.	Collect	Scan	Inter.	Final	Total
Stuttgart	V	MR	220.1	16.0	9.9	576.5	–	820.8
		UP-RAPTOR	1.2	8.1	7.2	2.3	102.7	121.4
		UP-TB	0.7	1.9	5.6	–	102.3	110.5
	$\mathcal{S}$	MR	77.1	10.7	8.4	20.5	–	116.7
		UP-RAPTOR	1.0	6.5	5.4	1.7	6.0	20.6
		UP-TB	0.7	1.4	4.0	–	5.9	12.0
London	V	MR	21.8	4.4	4.7	62.3	–	92.9
		UP-RAPTOR	1.1	3.1	4.2	2.9	13.5	24.6
		UP-TB	0.5	1.4	6.2	–	13.0	21.2
	$\mathcal{S}$	MR	18.7	4.0	4.1	12.5	–	38.6
		UP-RAPTOR	1.1	2.9	3.8	2.6	3.5	13.9
		UP-TB	0.5	1.3	5.9	–	3.6	11.3
Switzerland	V	MR	115.4	16.9	16.3	327.4	–	476.0
		UP-RAPTOR	1.6	9.0	14.2	4.5	48.9	78.2
		UP-TB	0.9	2.7	10.5	–	48.2	62.4
	$\mathcal{S}$	MR	48.1	12.0	14.3	24.9	–	99.3
		UP-RAPTOR	1.6	7.4	12.5	4.1	5.9	31.5
		UP-TB	0.8	2.4	8.7	–	5.7	17.8
Germany	V	MR	2 302.9	511.7	275.4	10 610.9	–	13 701.0
		UP-RAPTOR	19.5	313.9	264.9	91.7	1 061.6	1 751.6
		UP-TB	11.1	42.7	142.0	–	1 043.6	1 240.0
	$\mathcal{S}$	MR	653.7	512.5	278.1	656.7	–	2 101.1
		UP-RAPTOR	18.4	309.2	256.3	89.9	98.6	772.4
		UP-TB	10.5	41.3	141.4	–	98.8	292.5

which requires multiple final transfer searches. The resulting algorithmic framework, ULTRA-PHAST, yields the first algorithms specifically designed for one-to-all and one-to-many searches in multimodal networks. We evaluated ULTRA-PHAST versions of CSA, RAPTOR and TB on the networks of Switzerland and Germany. For small and moderately sized target sets, the share of the transfer exploration in the overall running time is reduced to 10–20%, with the rest being equivalent to an unimodal public transit query. For large target sets, we achieved a speedup of up to an order of magnitude over naive adaptations of MR and MCSA.

Regarding possible applications, UP-CSA could serve as an ingredient in algorithms for extended one-to-many problems that require single-criterion optimization. Note that in some of these applications, including POI queries and isochrones, the set of target vertices is not known in advance but discovered during the query. Existing algorithms for these problems on road networks build on hierarchical speedup techniques, such as CRP and CH, in order to restrict the search space compared to a full one-to-all query [EP14, DW15, BBDW19]. Similar techniques could be applied to the final transfer sweep that is performed by ULTRA-PHAST. A potential drawback of our approach is that it requires a full search on the public transit network before the final transfers are explored. In order to restrict the search space in the public transit network as well, integrated approaches would need to be developed.

For problems that require two-criteria Pareto optimization, UP-RAPTOR or UP-TB are potential building blocks. The grouped sweeps used to optimize the number of trips could be sped up further by using SIMD (single instruction, multiple data) instructions, such as SSE or AVX. Finally, ULTRA-PHAST could serve as an ingredient in a multimodal speedup technique that, unlike ULTRA, also accelerates the search in the public transit network. This would likely require ULTRA-PHAST to be combined with profile search.

7 Optimizing Transfer Time

In Chapter 5, we presented ULTRA for the Pareto optimization problem with the two criteria arrival time and number of trips. For pure public transit journey planning, this is the most commonly studied problem setting. The reason for including the number of trips as a criterion is that it serves as a measure of the discomfort associated with a journey: passengers will often accept a longer travel time if it saves a transfer between vehicles. However, especially in multimodal networks, this is not the only source of discomfort. In a scenario with unrestricted walking, Delling et al. [Del+13] have previously considered the walking duration as a proxy for discomfort. To generalize this to arbitrary transfer modes, we argue that the transfer time, i.e., the time spent using the transfer mode, should be considered as a discomfort criterion. In networks with restricted transfers, this is not as important because journeys will generally have a low transfer time anyway. With an unlimited transfer mode, however, journeys that are optimal according to the other two criteria often require an excessive amount of transfer time. As our experiments in Section 7.3 will show, there are often alternatives that reduce the transfer time considerably, at only a slight additional expense in the other two criteria. Such alternatives are missed by existing two-criteria algorithms, regardless of whether they allow unlimited transfers or not. Hence, we argue that in order to compute satisfactory journeys in a multimodal setting, it is necessary to minimize the transfer time as a third criterion. In this chapter, we investigate how this can be done efficiently.

State of the Art. Adding a third criterion poses algorithmic challenges. The fastest two-criteria algorithms, RAPTOR and TB, achieve their low query times in part by avoiding the explicit representation of Pareto sets. Both algorithms handle the number of trips by splitting their data structures per round. Within each round, RAPTOR maintains the earliest arrival time at each stop. Since the data structures required for this have a fixed size, they

can be preallocated at the start of each round. If more criteria are added, however, there may be multiple Pareto-optimal solutions per stop and round. McRAPTOR stores these in a bag, whose size may grow and shrink throughout the execution of the algorithm. In practice, the bags are implemented as dynamic arrays, which may require reallocations once they exceed the available space in memory. These reallocations are one of two reasons why McRAPTOR is much slower than RAPTOR.

The second reason is that Pareto sets become impractically large for three or more criteria [Del+13, BBS13, DDP19]. Many of these journeys are slight variations of each other with unattractive tradeoffs, such as adding an hour of transfer time to save one minute of travel time. Our goal is to present a small selection of journeys with favorable tradeoffs to the user. Several definitions for a filtered Pareto set have been proposed in a multimodal context [Del+13, BBS13], but so far no efficient algorithms are known for computing them exactly. For our use case, restricted Pareto sets [DDP19] are of particular interest because they limit the amount of extra arrival time or additional trips compared to the two-criteria Pareto set. In networks with one-hop transfers, they can be computed quickly and exactly with BM-RAPTOR (Bounded McRAPTOR), a query algorithm based on (Mc)RAPTOR. However, so far restricted Pareto sets have not been applied to multimodal journey planning.

Chapter Outline. To obtain efficient algorithms that optimize transfer time as a third criterion, we take a two-step approach. In Section 7.1, we develop efficient approaches for computing full three-criteria Pareto sets. While we mainly consider transfer time as the third criterion, we also discuss the degree to which these results carry over to other criteria.

In Section 7.1.1, we show that the problem can be solved in polynomial time for this particular combination of criteria. Using this insight, we develop a three-criteria extension of TB called McTB (Multicriteria Trip-Based Routing) in Section 7.1.3. In contrast to RAPTOR, TB does not maintain arrival times for each stop. Instead, it exploits the observation that every stop event is already associated with a particular arrival time. For each round and trip, the reached index keeps track of the earliest reachable stop event along the trip. When a third criterion is added, this is no longer sufficient: it may be possible to reach a later stop event with a better value for the third criterion. However, for each round and stop event, the journey with the lowest value for the third criterion still dominates all others. McTB exploits this observation by tracking the currently best value for the third criterion at each stop event. Combined with the round-based approach that is already part of TB, this makes the third criterion the only one whose value must be tracked explicitly. Consequently, McTB is the first algorithm that optimizes a third criterion without requiring dynamic arrays to represent the Pareto sets, except for a single set of solutions at the target vertex.

TB requires a preprocessing phase that computes transfers between stop events. As shown in Chapter 5, unlimited transfers can be enabled by replacing this phase with the event-to-event variant of ULTRA. To apply this to McTB, we develop McULTRA (Multicriteria ULTRA), a three-criteria extension of ULTRA, in Section 7.1.2. Analogously to ULTRA, McULTRA can be combined with any three-criteria public transit algorithm that requires one-hop transfers.

Combining McULTRA with McTB already yields a fairly efficient algorithm, but its performance is still hampered by the large size of the computed Pareto sets. In Section 7.2, we therefore integrate our approaches with restricted Pareto sets. We define restricted Pareto sets and outline the BM-RAPTOR algorithm for computing them in Section 7.2.1. In Section 7.2.2, we show that only minor changes are necessary to make BM-RAPTOR utilize McULTRA shortcuts, thus enabling fast computation of restricted Pareto sets in a multimodal network. To achieve even faster query times, we re-engineer the pruning scheme of BM-RAPTOR to support (Mc)TB as the underlying query algorithm in Section 7.2.3.

In Section 7.3, we evaluate our algorithms on the four benchmark networks. We show that the shortcut hypothesis holds in the three-criteria setting if the speed of the transfer mode is low to moderate. Combining McULTRA and McTB yields a speedup of five to ten compared with MCR, the fastest previously known multimodal algorithm for three criteria. UBM-TB (Bounded ULTRA-McTB), our new algorithm for restricted Pareto sets, offers interactive query times even on large networks, achieving a speedup of 30–90 compared with MCR.

7.1 Three-Criteria Pareto Optimization

In this section, we investigate the one-to-one, fixed departure time, Pareto optimization problem for a specific combination of three criteria. The first two, as usual, are arrival time and number of trips. The third criterion must fulfill two requirements:

(R1) In the time-expanded graph representation of the network, there must be an edge cost function with scalar values and the following property: the value of a journey according to the third criterion is the sum of the edge costs along the associated path.

(R2) A shortest path in the transfer graph according to arrival time is also a shortest path according to the third criterion.

Transfer time fulfills both requirements because its edge cost is 0 for all trip edges and equal to the travel time for all transfer edges. Requirement (R1) ensures that there is a single optimal value for the third criterion. An example of a criterion for which this is not the case is the set of visited fare zones, which has been considered by Delling et al. [Del+13] as a proxy for fare. Requirement (R2) ensures that if we ignore the public transit network entirely and only consider the transfer graph, the problem reduces to finding the shortest path according to a single criterion (i.e., travel time). Without this requirement, the number of Pareto-optimal solutions with zero trips may already be exponential [Han80]. In Section 7.1.1, we show that this problem can be solved in polynomial time. Afterward, we develop efficient algorithms for it by adapting ULTRA in Section 7.1.2 and TB in Section 7.1.3.

7.1.1 Problem Complexity

The key observation that allows this problem to be solved efficiently is that the arrival time of a Pareto-optimal journey is fully determined by its final stop event.

Lemma 7.1. *Consider a three-criteria query with source vertex v_s, target vertex v_t and departure time τ_{dep} in which the third criterion fulfills requirements (R1) and (R2). Given a fixed number of trips n and a fixed stop event ε, a Pareto set $\mathcal{J}$ for this query contains at most one journey with n trips that uses ε as the final stop event.*

Proof. Let J and J' be two Pareto-optimal journeys that both use n trips and have ε as their final stop event. Due to requirement (R2), the final transfers of both journeys are shortest paths in the transfer graph. Accordingly, their arrival time is $\tau_{\text{arr}}(\varepsilon) + \tau_{\text{tra}}(v(\varepsilon), v_t)$. Since the two journeys are equivalent in the first two criteria and both are Pareto-optimal, they must dominate each other weakly in the third criterion. Due to requirement (R1), this means that they have the same value for the third criterion and are in fact entirely equivalent. Because a Pareto set is minimal by definition, $\mathcal{J}$ includes only one of the two journeys. □

A consequence of this observation is that the number of distinct possible values for the arrival time is bounded by the number of stop events. Because the same is true for the number of trips, the size of the Pareto set is also bounded.

Lemma 7.2. *Consider a three-criteria query with source vertex v_s, target vertex v_t and departure time τ_{dep} in which the third criterion fulfills requirements (R1) and (R2). A Pareto set $\mathcal{J}$ for this query contains at most $|\mathcal{E}|^2 + 1$ journeys.*

Proof. Because the time-expanded graph is acyclic, no stop event can be visited more than once by the same journey. Accordingly, the number of trips used by a Pareto-optimal journey is at most $|\mathcal{E}|$. For each number of trips $1 \leq n \leq |\mathcal{E}|$, it follows from Lemma 7.1 that $\mathcal{J}$ contains at most $|\mathcal{E}|$ different journeys with n trips. Overall, this yields at most $|\mathcal{E}|^2$ journeys with at least one trip. Additionally, due to requirement (R2), $\mathcal{J}$ contains at most one journey with zero trips, which is a shortest v_s-v_t-path in the transfer graph. □

Depending on which third criterion is used, the upper bound of $|\mathcal{E}|$ on the number of used trips may be improved. For many criteria, it can be shown that it is never necessary to use the same trip twice or visit the same stop twice. In this case, the number of trips $|\mathcal{T}|$ or the number of stops $|\mathcal{S}|$ are tighter upper bounds. However, for some criteria these bounds do not hold: If the third criterion penalizes the time spent in a trip in some way, then there may be Pareto-optimal journeys that use the same trip T multiple times. This can occur if it is possible to exit T, transfer to a later stop along T and re-enter it there. Similarly, if the third criterion penalizes the waiting time at a stop, then there may be Pareto-optimal journeys that reduce the waiting time by travelling in a circle and visiting the same stop twice. Hence, Lemma 7.2 uses $|\mathcal{E}|$ as an upper bound because it holds regardless of the criterion.

Theorem 7.3 shows that the Pareto set can be computed in polynomial time. Note that the query algorithm used in our proof is not efficient in practice. We present a more practical algorithm in Section 7.1.3.

Theorem 7.3. *Consider a three-criteria query with source vertex v_s, target vertex v_t and departure time τ_{dep} in which the third criterion fulfills requirements (R1) and (R2). A Pareto set for this query can be found in polynomial time.*

Proof. To prove the claim, we use a modified time-expanded graph that incorporates unrestricted transfers and "unrolls" the number of trips in the same manner as previously suggested by Pyrga et al. for time-dependent graphs [PSWZ08]. The modifications can be made in polynomial time: For each pair v, w of stops such that w is reachable from v via a transfer, and for each transfer node v_i of v, identify the earliest reachable transfer node w_j of w and insert an edge (v_i, w_j). To represent initial transfers, insert a source node v_s. For each stop v, insert an edge from v_s to the earliest reachable transfer node of v. Final transfers are not represented in the graph; they are handled on the fly by the query algorithm. To encode the number of trips, create $|\mathcal{E}| + 1$ copies of the graph. For each edge from an arrival node v to a transfer node w and for each copy $0 \leq i < |\mathcal{E}|$, connect the i-th copy of v to the $(i+1)$-th copy of w. All other types of edges stay within the same copy. Thus, the copy in which a node is located corresponds to the number of trips used to reach it. Overall, this graph has $\mathcal{O}(|\mathcal{E}|^2)$ nodes and $\mathcal{O}(|\mathcal{E}|^2|\mathcal{S}|)$ edges.

A Pareto set can be found in polynomial time as follows. First, run an all-to-one Dijkstra search on the transfer graph to identify the shortest path from each stop (as well as v_s) to v_t. Afterward, run a Dijkstra search on the modified time-expanded graph to find the shortest path from v_s to each transfer node according to the edge cost function that represents the third criterion. The result set $\mathcal{J}$ is constructed as follows. First, add a journey representing the direct transfer from v_s to v_t. Then, for each reached transfer node belonging to a stop v, append the shortest v-v_t-path to the found journey and add the result to $\mathcal{J}$. By Lemma 7.2, $\mathcal{J}$ contains a Pareto set. To extract it, perform pairwise comparisons of the elements in $\mathcal{J}$ to eliminate dominated journeys. □

Our proof relies on the fact that the number of distinct values is bounded for two of the three criteria. Both bounds were previously known and are in fact already exploited by the reached index data structure of TB: Since the arrival time of all journeys to a stop event ε is equal to $\tau_{\text{arr}}(\varepsilon)$, it is sufficient to track whether ε has been reached before. Because the number of trips is bounded, this can be done individually for each possible number of trips. Thereby, TB transforms the two-criteria Pareto optimization problem into a series of reachability queries. The new observation is that we can handle a third criterion by replacing the reachability queries with shortest-path queries that optimize the third criterion.

In the following, we propose algorithms that exploit this observation. Our algorithms employ route-based pruning rules, which assume that it is never useful to enter a later trip than the earliest reachable one of a route. Our proof of Theorem 7.3 does not require this assumption, and as discussed in Chapter 4.2.3, some criteria (e.g., vehicle occupancy) are not compatible with it. However, since our primary goal is to optimize transfer time as a third criterion, we include route-based pruning in our algorithms.

7.1.2 McULTRA Shortcut Computation

To enable unlimited transfers when optimizing a third criterion, we propose McULTRA, an adaptation of ULTRA. As with the original ULTRA, McULTRA can be configured to

output either stop-to-stop or event-to-event shortcuts. The former can be combined with any algorithm that requires one-hop transfers, using the same query framework as ULTRA. The latter is intended for combination with McTB, which we present in Section 7.1.3. The shortcut computation is nearly identical for both variants and only differs in how the shortcuts for undominated candidates are retrieved.

Due to requirement (R2) for the third criterion, initial and final transfers can still be explored with single-criterion Bucket-CH searches. However, the shortcut computation must be adjusted. Its basic outline remains unchanged from ULTRA: For every stop $v_s \in \mathcal{S}$, the algorithm collects all departure times of trips at v_s and then performs a run of MCR for each departure time in descending order. The MCR runs are restricted to two rounds (each consisting of route scans followed by transfer relaxations), and vertex bags are not emptied between the runs. During the final transfer relaxation phase of each run, labels representing undominated candidates are extracted and shortcuts are inserted for them. Adding a third criterion requires the use of MCR instead of MR, and therefore vertex bags instead of mere arrival times. In the following, we adapt the running time optimizations included in the original ULTRA shortcut computation and describe additional ones for the three-criteria setting.

Route Scans. When scanning a route, MR maintains a single active trip along the route, which is the earliest trip that can be entered so far. In MCR, the active trip is replaced with a route bag containing all Pareto-optimal labels, each of which has its own active trip. Note that Lemma 7.1 implies that no two labels can have the same active trip: When exiting at any stop along the route, both labels will share the same final stop event and therefore the same arrival time. Hence, the label that is better (or equal) in the third criterion will dominate the other one.

Bags are always required in the second route scanning phase of each MCR run. If the third criterion only affects the transfer graph (which is the case for transfer time), then the simpler MR variant with a single active trip is sufficient for the first route scan. In this setting, all candidate labels have value 0 in the third criterion during the first route scan because candidates do not have an initial transfer. Therefore, the candidate with the earliest arrival time dominates all others. Although the same is not true for witnesses, it is not necessary to find all witnesses. Failing to find one may lead to superfluous shortcuts, but the query algorithm will remain correct.

Dijkstra Searches. The single-criterion Dijkstra searches of ULTRA are replaced with a standard multicriteria Dijkstra search. Given an *arrival weight* $\omega_{\text{arr}} \geq 0$ and a *transfer weight* $\omega_{\text{tra}} \geq 0$ (assuming the third criterion is transfer time), the key of a label representing a journey J in the priority queue is the sequence

$$\kappa(J) := \left\langle \frac{\omega_{\text{arr}} \cdot \tau_{\text{arr}}(J) + \omega_{\text{tra}} \cdot \tau_{\text{tra}}(J)}{\omega_{\text{arr}} + \omega_{\text{tra}}}, \tau_{\text{arr}}(J), \tau_{\text{tra}}(J) \right\rangle .$$

Sequences are compared lexicographically. The second and third elements are included as tiebreakers in case one of the two weights is 0. This ensures that the order in which labels are settled does not conflict with Pareto dominance, i.e., if the label for journey J is settled before the label for journey J', then J' may strongly not dominate J. The stopping criteria for the Dijkstra searches used by the original ULTRA shortcut computation can be carried over to McULTRA: The final Dijkstra search of each run is stopped once all candidates have been extracted from the queue. The stopping criterion for the intermediate Dijkstra search is based on the *intermediate witness limit* λ_1^{w}. If κ_{c} is the key of the last candidate extracted from the queue, the search is stopped once the key of the queue head exceeds $\kappa_{\mathrm{c}} + \lambda_1^{\mathrm{w}}$.

Canonical Journeys. To make a consistent choice between multiple equivalent Pareto-optimal journeys, we introduced the notion of the canonical journey in Chapter 5.1. The correctness of ULTRA relies on the fact that every subjourney of a canonical journey is itself canonical (cf. Lemma 5.1). This ensures that a query algorithm can always find a Pareto-optimal journey that can be decomposed into canonical candidates, for which ULTRA is guaranteed to generate a shortcut.

The addition of a third criterion in McULTRA necessitates a small modification to this definition. As before, we define total orderings on the sets of routes and vertices with a route index function $\mathrm{id}_{\mathcal{R}} : \mathcal{R} \to \mathbb{N}$ and a vertex index function $\mathrm{id}_V : V \to \mathbb{N}$. For a journey J with vertex sequence $V(J) = \langle v_{\mathrm{s}} = v_1, \ldots, v_k = v_{\mathrm{t}} \rangle$ and $k > 1$, the *route tiebreaking sequence* is given by

$$X_{\mathrm{r}}(J) := \begin{cases} \langle \mathrm{id}_{\mathcal{R}}(R(T)), i \ \rangle & \text{if } J \text{ ends with a trip segment } T[i, j], \\ \langle \infty \qquad\qquad , \infty \rangle & \text{if } J \text{ ends with an edge } (v_{k-1}, v_k), \end{cases}$$

which is unchanged from Chapter 5.1. However, the *edge tiebreaking sequence* is redefined as

$$X_{\mathrm{e}}(J) := \begin{cases} \langle \infty, \infty, \infty, \infty \rangle & \text{if } J \text{ ends with a trip segment } T[i, j], \\ \kappa(J[v_{\mathrm{s}}, v_{k-1}]) \circ \langle \mathrm{id}_V(v_{k-1}) \rangle & \text{if } J \text{ ends with an edge } (v_{k-1}, v_k). \end{cases}$$

As before, these are combined into the *local tiebreaking sequence*

$$X_\ell(J) := \langle \tau_{\mathrm{arr}}(J) \rangle \circ X_{\mathrm{r}}(J) \circ X_{\mathrm{e}}(J).$$

The overall tiebreaking sequence is the concatenation of the local sequences in reverse order:

$$X(J) := X_\ell(J[v_{\mathrm{s}}, v_k]) \circ \cdots \circ X_\ell(J[v_{\mathrm{s}}, v_2]).$$

The only difference to the original definitions is that the arrival time is replaced with the queue key in the edge tiebreaking sequence. This ensures that, all other factors being equal, journeys with a lower transfer time are preferred. Because the tiebreaking sequence is unique among all v_{s}-v_{t}-journeys, replacing arrival time with the tiebreaking sequence as a criterion

for Pareto optimality ensures that there is only one Pareto set for each query. The proof of Lemma 5.1 is oblivious to the particular definition of the edge tiebreaking sequence, so it carries over to McULTRA without modifications.

To compute canonical journeys, ULTRA uses a modification of MR called canonical MR. The differences to MR are that the routes are sorted according to $\mathrm{id}_{\mathcal{R}}$ before they are scanned and that the edge tiebreaking sequence is used as the queue key during the Dijkstra searches. We apply the same modifications to MCR and use the resulting *canonical MCR* algorithm for McULTRA. The proof of correctness for McULTRA (cf. Lemma 5.2 and Theorem 5.3) then carries over from ULTRA in a straightforward manner.

Parent Pointers. Two-criteria ULTRA uses per-stop parent pointers to extract shortcuts and to distinguish between candidates and witnesses. This is no longer sufficient for three criteria because a stop may now have multiple candidate labels that represent different shortcuts. However, by Lemma 7.1, each stop event may have at most one candidate. Thus, McULTRA maintains two pointers $\mathsf{p}_1[\varepsilon]$ and $\mathsf{p}_2[\varepsilon]$ per stop event ε, where $\mathsf{p}_k[\varepsilon]$ is the parent stop event for reaching ε with k trips. Each candidate label includes a pointer to the last stop event at which a trip was entered or exited. To distinguish them from candidates, witness labels set this pointer to $\perp$. When a candidate journey with final stop event ε_t is found, the corresponding shortcut can be extracted as $(\mathsf{p}_1[\mathsf{p}_2[\varepsilon_\mathrm{t}]], \mathsf{p}_2[\varepsilon_\mathrm{t}])$.

Final Transfer Pruning. In the original ULTRA, a candidate is discarded if it is strongly dominated by a witness or weakly dominated by a journey found in a previous run. This rule is carried over to McULTRA. Preliminary experiments showed that the final Dijkstra search of each run takes up a much larger share of the overall running time than in the original ULTRA. The reason for this is that the stopping criterion is applied later due to undominated candidate labels with a very high key. These labels are only weakly dominated by equivalent labels that were found in previous runs. To remedy this, we distinguish between *proper candidates*, which are not dominated by any journey, and *improper candidates*, which are weakly dominated by a journey found in a previous run. We make the stopping criterion of the final Dijkstra search stricter by introducing a *final witness limit* λ_2^w. Let κ_c be the key of the last proper candidate extracted from the queue. The search is stopped once the key of the queue head exceeds $\kappa_\mathrm{c} + \lambda_2^\mathrm{w}$. Afterward, all remaining improper candidates are removed from the queue and shortcuts are inserted for them.

7.1.3 McTB Query Algorithm

As a faster alternative to McRAPTOR, we propose McTB, a new three-criteria query algorithm based on TB. The original preprocessing phase of TB computes event-to-event transfers based on a one-hop transfer graph. Because we focus on a multimodal scenario, we use the set E^s of event-to-event McULTRA shortcuts instead and obtain a multimodal algorithm, ULTRA-McTB. Pseudocode for the trip enqueuing and scanning procedures of McTB is given

Algorithm 7.1: Trip enqueuing and scanning procedures of McTB.

```
1  Procedure Enqueue(T[j], Q, τ)
2      if r_tra(T, j) ≤ τ then return
3      k ← max{i ∈ {j, ..., |T| − 1} | r_tra(T, i) > τ}
4      Q ← Q ∪ {(T[j, k], τ)}
5      for each T' ≻ T do
6          for i from j to |T| − 1 do
7              r_tra(T', i) ← min(τ, r_tra(T', i))
8  Procedure ScanTrips(Q_n, Q_{n+1})
9      for each (T[j, k], τ) ∈ Q_n do
10         for i from j + 1 to k do
11             if r_tra(T, i) < τ then k ← i − 1
12             else if r_tra(T, i) = τ and pred(T) ≠ ⊥ and r_tra(pred(T), i) ≤ τ then
                   k ← i − 1
13     for each (T[j, k], τ) ∈ Q_n do
14         for i from j to k do
15             τ_f ← τ_tra(v(T[i]), v_t)
16             L ← L ∪ {(τ_arr(T[i]) + τ_f, n, τ + τ_f)}, removing dominated entries
17     for each (T[j, k], τ) ∈ Q_n do
18         if L dominates (τ_arr(T[j]), n + 1, τ) then continue
19         for i from j to k do
20             for each e = (T[i], T'[i']) ∈ E^s do
21                 Enqueue(T'[i' + 1], Q_{n+1}, τ + τ_tra(e))
```

in Algorithm 7.1 (cf. Algorithms 4.4, 4.5 and 4.6 in Chapter 4.2.4 for the original TB algorithm). For ease of exposition, we assume that the third criterion is transfer time. However, McTB supports any third criterion that fulfills requirements (R1) and (R2) and is compatible with route-based pruning rules.

McTB replaces the reached indices $r(\cdot)$ used by TB with a *reached transfer time* $r_{\text{tra}}(T, i)$ for each stop event $T[i]$. This represents the minimum transfer time that is needed to reach $T[i]$ (or ∞ if $T[i]$ is not reachable). By Lemma 7.1, the journey represented by $r_{\text{tra}}(T, i)$ is not dominated by any other journeys ending at $T[i]$ found so far. Two invariants are upheld for $r_{\text{tra}}(T, i)$: For each later trip $T' \succ T$, $r_{\text{tra}}(T', i) \leq r_{\text{tra}}(T, i)$ holds because a passenger can reach $T'[i]$ by traveling to $T[i]$ and then waiting for T'. Similarly, remaining seated in a trip does not increase the transfer time, so $r_{\text{tra}}(T, j) \leq r_{\text{tra}}(T, i)$ holds for each $i < j < |T|$.

Initial Transfer Evaluation. Like every ULTRA-based algorithm, ULTRA-McTB explores initial and final transfers with a Bucket-CH search. This yields the minimal source transfer time $\tau_{\text{tra}}(v_s, v)$ for each stop v reachable via an initial transfer, and the minimal

target transfer time $\tau_{\text{tra}}(w, v_t)$ for each stop w from which v_t is reachable via a final transfer. Afterward, the proceduce `CollectInitialTrips` identifies trip segments that can be entered via an initial transfer. This is done as in the original TB algorithm: For each stop v reachable via an initial transfer and each route visiting v, the earliest reachable trip at v is found via a binary search. Then, the `Enqueue` procedure is called to collect the corresponding trip segment. Note that two-criteria ULTRA-TB replaces this step with RAPTOR-like route scans, exploiting the fact that the earliest reachable trip cannot increase along the stop sequence of a route (cf. Algorithm 5.3 in Chapter 5.2.2). However, once a third criterion is introduced, this is no longer the case: entering the route at a later stop may decrease the value of the third criterion, which is beneficial even if it means using a later trip. Therefore, McTB reverts back to the original TB implementation of `CollectInitialTrips`.

Trip Enqueuing. Like the original TB algorithm, McTB operates in rounds, in which round n scans trip segments that were collected in queue Q_n during round $n-1$. A queue element is a tuple $(T[j,k], \tau)$ consisting of a trip segment $T[j,k]$ and the transfer time τ with which $T[j]$ was reached in the previous round. Trip segments are inserted into the queue by the `Enqueue` procedure (lines 1–7). When a trip T is entered with transfer time τ at index i, the first index at which it may be exited is $j := i+1$. If $r_{\text{tra}}(T, j) \leq \tau$ holds (line 2), then T does not improve the transfer time at $T[j]$ or any later stop event, so the trip segment is discarded. Otherwise, the end of the trip segment is set to the last index k for which $r_{\text{tra}}(T, k) > \tau$ holds (line 3). Finally, the trip segment is added to the queue (line 4) and the transfer time labels of later stop events along the route are updated to τ, satisfying the two invariants (lines 5–7).

Trip Scanning. To prevent redundant scans, the trip scanning phase starts with a new pruning step (lines 9–12). Between enqueuing a tuple $a = (T_a[j,k], \tau_a)$ and extracting it from Q_n, another tuple $b = (T_b[p,q], \tau_b)$ with $T_b \preceq T_a$, $j \leq p \leq k$ and $\tau_b \leq \tau_a$ may have been enqueued. In this case, scanning the segment $T_a[p,k]$ is redundant because it overlaps with $T_b[p,q]$. The pruning step identifies redundant portions of $T_a[j,k]$ and adjusts the end index k accordingly. If such a tuple b exists, either $T_b \prec T_a$ or $\tau_b < \tau_a$ must hold; otherwise, the check in line 2 would have discarded b. If $\tau_b < \tau_a$, then enqueuing b ensures $r_{\text{tra}}(T_a, p) < \tau_a$, which is checked in line 11. If $T_b \prec T_a$, the trip $\text{pred}(T_a)$ that immediately precedes T_a in the route must have $r_{\text{tra}}(\text{pred}(T_a), p) \leq r_{\text{tra}}(T_b, p) \leq \tau_b \leq \tau_a$. This is checked in line 12.

Following the pruning step, final transfers are evaluated for all stops along the trip segment $T[j,k]$ with transfer time τ. Line 16 adds new labels to the set $\mathcal{L}$ of Pareto-optimal labels at v_t, which is the only dynamic data structure maintained by the algorithm. Outgoing transfers to other trips are then relaxed in lines 17–21 and the reached trips are enqueued. Before this is done, target pruning is applied in line 18. Any journey with an intermediate transfer from $T[j,k]$ to another trip has an arrival time of at least $\tau_{\text{arr}}(T[j])$, uses at least $n+1$ trips and has a transfer time of at least τ. If a label with the smallest possible value for all three criteria is dominated by the current Pareto set $\mathcal{L}$, outgoing transfers do not need to be

relaxed. Note that we do not apply target pruning for each index $j \leq i \leq k$ separately by substituting $\tau_{\text{arr}}(T[i])$ for $\tau_{\text{arr}}(T[j])$. Preliminary experiments showed that this saves only a few Enqueue calls but leads to worse memory locality during the transfer relaxations. We make the dominance check more efficient by exploiting the fact that all labels added to $\mathcal{L}$ by round n use at most n trips. This makes it possible to maintain a set $\mathcal{L}^* \subseteq \mathcal{L}$ of *best labels* that are Pareto-optimal according to arrival time and transfer time (ignoring the number of trips). Then, line 18 only has to check if the label is dominated by $\mathcal{L}^*$ instead of $\mathcal{L}$.

7.2 Restricted Pareto Sets

To extract relevant solutions from the full Pareto set, we follow the approach by Delling et al. [DDP19] of computing restricted Pareto sets. In Section 7.2.1, we give a definition of the restricted Pareto set and present BM-RAPTOR, the algorithm for computing it in pure public transit networks. Afterward, we show how restricted Pareto sets can be adapted for multimodal networks by integrating them with the algorithms developed in Section 7.1. We discuss the integration with McULTRA and McRAPTOR in Section 7.2.2. Finally, we introduce a new TB-based algorithm for computing restricted Pareto sets in Section 7.2.3.

7.2.1 Definition

Consider a one-to-one, fixed departure time, Pareto optimization query with an arbitrary number of criteria in addition to arrival time and number of trips. Let $\mathcal{J}$ be a full Pareto set for this query and $\mathcal{J}^{\text{A}}$ a Pareto set for the two criteria arrival time and number of trips. The set $\mathcal{J}^{\text{A}}$ is called the *anchor set*. Each journey J in the full Pareto set $\mathcal{J}$ has a corresponding *anchor journey* $A(J)$, which is the journey in $\mathcal{J}^{\text{A}}$ with the highest number of trips not greater than $|J|$. Given a *trip slack* $\sigma_{\text{tr}} \geq 0$ and an *arrival slack* $\sigma_{\text{arr}} \geq 0$, Delling et al. define the *restricted Pareto set* as

$$\widetilde{\mathcal{J}^{\text{R}}} := \left\{J \in \mathcal{J} \mid |J| \leq |A(J)| + \sigma_{\text{tr}} \text{ and } \tau_{\text{arr}}(J) \leq \tau_{\text{arr}}(A(J)) + \sigma_{\text{arr}}\right\}.$$

The idea is that $\widetilde{\mathcal{J}^{\text{R}}}$ contains every journey from the full Pareto set that does not exceed the arrival or trip slack compared to its anchor journey. An illustration is given in Figure 7.1.

To compute restricted Pareto sets, Delling et al. present BM-RAPTOR, an extension of RAPTOR. They propose three variants of the algorithm with increasingly tight pruning rules; we focus on the fastest one (Tight-BMRAP). We observe that the restricted Pareto set as computed by BM-RAPTOR actually conforms to a slightly different definition than the one given above. Instead of a single anchor journey, each journey $J \in \mathcal{J}$ is compared to the set

$$\mathcal{J}^{\text{A}}(J) := \left\{J' \in \mathcal{J}^{\text{A}} \mid |J'| \geq |A(J)|\right\}$$

of possible anchor journeys, i.e., all journeys in the anchor set that do not use fewer trips than $A(J)$. Then the restricted Pareto set is defined as

$$\mathcal{J}^{\text{R}} := \left\{J \in \mathcal{J} \mid \exists J' \in \mathcal{J}^{\text{A}}(J) : |J| \leq |J'| + \sigma_{\text{tr}} \text{ and } \tau_{\text{arr}}(J) \leq \tau_{\text{arr}}(J') + \sigma_{\text{arr}}\right\}.$$

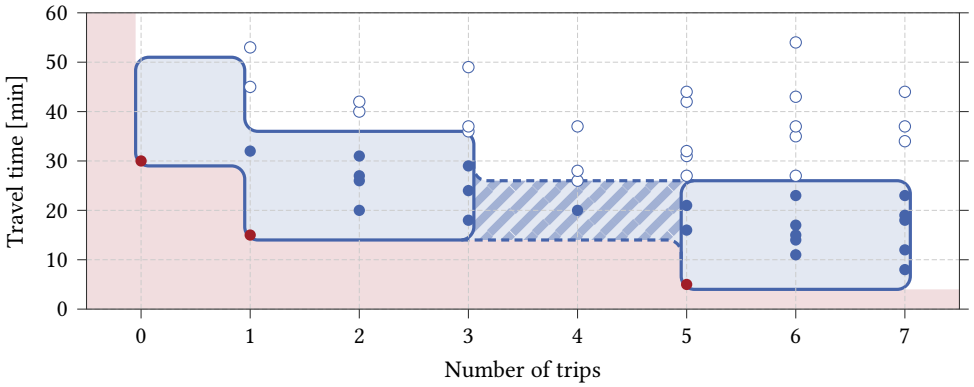


Figure 7.1: An example of a restricted Pareto set for slack values $\sigma_{\text{arr}} = 20\,\text{min}$ and $\sigma_{\text{tr}} = 2$. Circles represent journeys from the full Pareto set $\mathcal{J}$. Journeys in the anchor set $\mathcal{J}^{\text{A}}$ are drawn as red circles. Journeys excluded from the restricted Pareto set are drawn as empty circles. The area shaded in red cannot contain any journeys because they would be part of the anchor set. The area covered by the restricted Pareto set is shaded in blue. The hatched area is excluded from the original definition $\widetilde{\mathcal{J}^{\text{R}}}$ of the restricted Pareto set but included in the set $\mathcal{J}^{\text{R}}$ computed by BM-RAPTOR.

Thus, a journey J is in the restricted Pareto set if it does not exceed the slack values compared to at least one possible anchor journey. In Figure 7.1, the difference can be seen in the hatched area. In the original definition, this area is not included because it exceeds the trip slack of the corresponding anchor journey with one trip and a travel time of 15 minutes. The modified definition also considers the anchor journey with five trips and a travel time of 5 minutes, for which the area is within slack.

To obtain an equivalent definition that is easier to handle algorithmically, we introduce the notion of labels. A *label* is a tuple $\ell = (\tau, n)$ consisting of an arrival time $\tau_{\text{arr}}(\ell) := \tau$ and a number of trips $|\ell| := n$. The definitions of (two-criteria) dominance and Pareto optimality carry over from journeys to labels. Let $\mathcal{J}^{\text{A}} = \langle J_1, \ldots, J_k \rangle$ be the sequence of anchor journeys sorted from fewest trips to most. To simplify the following definitions, we additionally define $J_{k+1} := (-\infty, \infty)$. For each anchor journey J_i, we define the label

$$\ell_i := \big(\tau_{\text{arr}}(J_i) + \sigma_{\text{arr}}, \min(|J_i| + \sigma_{\text{tr}}, |J_{i+1}| - 1)\big).$$

This yields a sequence $\mathcal{L}^{\text{R}} := \langle \ell_1, \ldots, \ell_k \rangle$ of labels. In Figure 7.1, these labels represent the upper right corners of the blue boxes surrounding the anchor journeys. Theorem 7.4 shows that the restricted Pareto set consists of all journeys in $\mathcal{J}$ that weakly dominate at least one label in $\mathcal{L}^{\text{R}}$.

Theorem 7.4. *Let $\mathcal{J}$ be the full Pareto set, $\mathcal{J}^A$ the anchor set and $\sigma_{arr}, \sigma_{tr}$ slack values. The restricted Pareto set $\mathcal{J}^R$ consists of all journeys in $\mathcal{J}$ that weakly dominate at least one label in $\mathcal{L}^R$.*

Proof. For a journey $J \in \mathcal{J}$, let a be the index of its anchor journey in $\mathcal{J}^{\mathrm{A}}$, i.e., the index with $A(J) = J_a$. For $1 \leq i \leq k+1$, it follows from the definition of $A(J)$ that $|J| \geq |J_i|$ iff $i \leq a$. Furthermore, for $1 \leq i \leq k$, it follows from the definition of $\mathcal{J}^{\mathrm{A}}(J)$ that $J_i \in \mathcal{J}^{\mathrm{A}}(J)$ iff $i \geq a$. Hence, $J_i \in \mathcal{J}^{\mathrm{A}}(J)$ iff $|J| \leq |J_{i+1}| - 1$. Then it follows that

$$\begin{aligned}
\mathcal{J}^{\mathrm{R}} &= \left\{J \in \mathcal{J} \mid \exists J' \in \mathcal{J}^{\mathrm{A}}(J) : |J| \leq |J'| + \sigma_{\mathrm{tr}} \text{ and } \tau_{\mathrm{arr}}(J) \leq \tau_{\mathrm{arr}}(J') + \sigma_{\mathrm{arr}}\right\} \\
&= \left\{J \in \mathcal{J} \mid \exists J_i \in \mathcal{J}^{\mathrm{A}} : |J| \leq \min(|J_i| + \sigma_{\mathrm{tr}}, |J_{i+1}| - 1) \text{ and } \tau_{\mathrm{arr}}(J) \leq \tau_{\mathrm{arr}}(J_i) + \sigma_{\mathrm{arr}}\right\} \\
&= \left\{J \in \mathcal{J} \mid \exists \ell_i \in \mathcal{L}^{\mathrm{R}} : J \text{ weakly dominates } \ell_i\right\}.
\end{aligned}$$

□

BM-RAPTOR. To compute the restricted Pareto set, BM-RAPTOR performs three phases: The *forward pruning search* is a two-criteria RAPTOR search that computes the *earliest arrival time* $\overrightarrow{\tau_{\mathrm{arr}}}(v, n)$ per stop v and round n, and thereby the anchor set $\mathcal{J}^{\mathrm{A}}$. Then, a *backward pruning search* is run for each label in $\mathcal{L}^{\mathrm{R}}$. Collectively, these compute a *latest departure time* $\overleftarrow{\tau_{\mathrm{dep}}}(v, n)$ for each stop v and round n. This is the latest time at which a journey with n trips must reach v such that it can be extended to a journey to v_{t} that weakly dominates at least one label in $\mathcal{L}^{\mathrm{R}}$. Finally, a *main McRAPTOR search* is run, using the latest departure times to prune the search space.

The forward pruning search is identical to a regular RAPTOR search except for a change in the target pruning rule. Normally, an arrival with arrival time τ in round n is discarded if $\tau > \overrightarrow{\tau_{\mathrm{arr}}}(v_{\mathrm{t}}, n)$ because such an arrival cannot be extended to an optimal solution at v_{t}. In BM-RAPTOR, the arrival can only be discarded if it cannot be extended to a solution that weakly dominates a label in $\mathcal{L}^{\mathrm{R}}$. Hence, the condition is changed to $\tau > \overrightarrow{\tau_{\mathrm{arr}}}(v_{\mathrm{t}}, i) + \sigma_{\mathrm{arr}}$.

After the forward pruning search, $\overleftarrow{\tau_{\mathrm{dep}}}(v, n)$ is initialized with $-\infty$ for each stop v and round $0 \leq n \leq |\ell_k|$. Then, a backward pruning search is run for each label $\ell \in \mathcal{L}^{\mathrm{R}}$, in order of most used trips to fewest. This is a reverse RAPTOR search that starts at v_{t} with the arrival time $\tau_{\mathrm{arr}}(\ell)$ and is run for $|\ell|$ rounds. During round n, departure times are read from $\overleftarrow{\tau_{\mathrm{dep}}}(\cdot, |\ell| - (n-1))$ and written to $\overleftarrow{\tau_{\mathrm{dep}}}(\cdot, |\ell| - n)$. When the search finds a departure at a stop v in round n with departure time τ, it is discarded if $\tau < \overrightarrow{\tau_{\mathrm{arr}}}(v, |\ell| - n)$. Otherwise, for each round $n \leq n' \leq |\ell|$, the latest departure time $\overleftarrow{\tau_{\mathrm{dep}}}(v, |\ell| - n')$ is set to the maximum of itself and τ. The latest departure times $\overleftarrow{\tau_{\mathrm{dep}}}(\cdot, \cdot)$ are not reinitialized between backward searches. During the main search, the latest departure times are used for pruning: if the search arrives at a stop v in round n with arrival time τ, the arrival is discarded if $\tau > \overleftarrow{\tau_{\mathrm{dep}}}(v, n)$.

Discussion. As previously noted by Delling et al. [DDP19], the main advantage of restricted Pareto sets is that BM-RAPTOR offers a way to compute them both quickly and exactly. This is in contrast to previous approaches, such as the one originally proposed alongside MCR [Del+13], which is based on tightening the dominance rules. This approach

only yields exact results if the dominance rules are applied to the full Pareto set once it has already been computed. If they are applied every time a vertex is reached during the query, relevant solutions will be lost.

The manner in which restricted Pareto sets limit the solution space is particularly well suited to our multimodal scenario, in which transfer time serves as a measure of discomfort. Our aim is to find journeys beyond the anchor set that save a significant amount of transfer time without investing too much in the other two criteria. Restricted Pareto sets ensure the latter via the arrival and trip slacks. Although they may still contain undesirable journeys that only save a minuscule amount of transfer time, these can easily be filtered out in a post-processing step.

We observed that the set $\mathcal{J}^{\mathrm{R}}$ computed by BM-RAPTOR does not precisely match the original definition $\widetilde{\mathcal{J}^{\mathrm{R}}}$ of the restricted Pareto set. Because $\mathcal{J}^{\mathrm{R}}$ considers more potential anchor journeys, it holds that $\mathcal{J}^{\mathrm{R}} \supseteq \widetilde{\mathcal{J}^{\mathrm{R}}}$. The definition of the set $\mathcal{J}^{\mathrm{A}}(J)$ of possible anchor journeys for a journey J may seem arbitrary at first glance. It is unclear why journeys with more trips than J should be considered as anchor journeys, but those with fewer trips than $A(J)$ (which itself may have fewer trips than J) should be ignored. However, this eliminates a potentially undesirable feature of $\widetilde{\mathcal{J}^{\mathrm{R}}}$. In Figure 7.1, consider the journey with the label $(20\,\mathrm{min}, 4)$, which we denote as J_1, and the journey with the label $(21\,\mathrm{min}, 5)$, which we denote as J_2. According to the original definition, J_1 is not included in $\widetilde{\mathcal{J}^{\mathrm{R}}}$, but J_2 is, even though it is strongly dominated by J_1. Note that this issue can only occur if the anchor set has a gap in the number of trips that is wider than the trip slack. This gap is excluded from $\widetilde{\mathcal{J}^{\mathrm{R}}}$, but $\mathcal{J}^{\mathrm{R}}$ fills it in.

Delling et al. define restricted Pareto sets using additive arrival and trip slacks. We argue that it is more natural to use relative slacks, which depend on the length of the anchor journey. Intuitively, passengers are more willing to take long detours if the journey is already long, whereas a 60-minute detour on a 15-minute journey is not attractive. Accordingly, we redefine the restricted Pareto set based on multiplicative slacks $\sigma_{\mathrm{arr}}, \sigma_{\mathrm{tr}} \geq 1$:

$$\mathcal{J}^{\mathrm{R}} := \left\{ J \in \mathcal{J} \mid \exists J' \in \mathcal{J}^{\mathrm{A}}(J) : |J| \leq |J'| \cdot \sigma_{\mathrm{tr}} \text{ and } \tau_{\mathrm{arr}}(J) - \tau_{\mathrm{dep}} \leq (\tau_{\mathrm{arr}}(J') - \tau_{\mathrm{dep}}) \cdot \sigma_{\mathrm{arr}} \right\}.$$

Note that the arrival slack is relative to the overall travel time of the journey, not the arrival time. The label corresponding to an anchor journey J_i is now given by

$$\ell_i := \big(\tau_{\mathrm{dep}} + (\tau_{\mathrm{arr}}(J_i) - \tau_{\mathrm{dep}}) \cdot \sigma_{\mathrm{arr}}, \min(|J_i| \cdot \sigma_{\mathrm{tr}}, |J_{i+1}| - 1)\big).$$

The proof of Theorem 7.4 carries over in a straightforward manner. With the set $\mathcal{L}^{\mathrm{R}}$ of labels redefined accordingly, BM-RAPTOR can now be applied with one additional change to the target pruning rule of the forward pruning search: an arrival with arrival time τ in round n is now discarded if $\tau - \tau_{\mathrm{dep}} > (\overrightarrow{\tau_{\mathrm{arr}}}(v_{\mathrm{t}}, i) - \tau_{\mathrm{dep}}) \cdot \sigma_{\mathrm{arr}}$. Note that our choice to use multiplicative slacks is merely made to improve the solution quality. All algorithms that are presented in the following section can be easily modified to use additive slacks instead.

7.2.2 UBM-RAPTOR

We now discuss how BM-RAPTOR can be adapted to multimodal networks by integrating it with McULTRA shortcuts. We note that merely using McULTRA shortcuts in BM-RAPTOR is not sufficient. BM-RAPTOR requires that the forward and backward pruning searches find optimal arrival or departure times at each stop. However, because ULTRA-RAPTOR explores final transfers with a backward Bucket-CH search from v_t, optimal journeys to stops other than v_t may not be found if they end with a non-empty transfer.

To solve this issue, we use a slightly modified version of ULTRA-RAPTOR, which we call pRAPTOR, for the pruning searches. Aside from using the three-criteria McULTRA shortcuts for the intermediate transfers, the only difference is in the transfer relaxation phase. Normally, ULTRA-RAPTOR relaxes the outgoing shortcut edges of all stops whose arrival time was improved during the preceding route scanning phase. pRAPTOR additionally relaxes the edges of stops whose arrival time was improved during the previous transfer relaxation phase. This allows pRAPTOR to find journeys that use multiple shortcut edges in a row. However, each additional edge is counted as an additional trip.

Theorem 7.5 shows that pRAPTOR can be used to perform the forward pruning search. The proof for the backward search is analogous. By combining pRAPTOR pruning searches with ULTRA-McRAPTOR for the main search, we obtain UBM-RAPTOR (Bounded ULTRA-McRAPTOR), an algorithm for computing restricted Pareto sets in multimodal networks.

Theorem 7.5. *Consider a three-criteria query with source vertex v_s, target vertex v_t and departure time τ_{dep}. Let $\tau_{arr}(v, n)$ denote the arrival time at v found by pRAPTOR in round n for this query. Let J be a Pareto-optimal journey found by ULTRA-McRAPTOR for this query. Given a stop v visited by J, let $\tau_{arr}(J, v)$ denote the arrival time of J at v. For $1 \leq n \leq |J|$, let T_n be the n-th trip of J, v_n the stop at which T_n is entered and w_n the stop at which T_n is exited. Then $\tau_{arr}(v_n, n-1) \leq \tau_{arr}(J, v_n)$ and $\tau_{arr}(w_n, n) \leq \tau_{arr}(J, w_n)$.*

Proof. First we show that $\tau_{\text{arr}}(v_n, n-1) \leq \tau_{\text{arr}}(J, v_n)$ implies $\tau_{\text{arr}}(w_n, n) \leq \tau_{\text{arr}}(J, w_n)$. If pRAPTOR arrives at v_n no later than J, it will scan T_n or an earlier trip of the same route during the route scanning phase of round n and thereby reach w_n with an arrival time of $\tau_{\text{arr}}(J, w_n)$ or earlier. For $n < |J|$, we show that this in turn implies $\tau_{\text{arr}}(v_{n+1}, n) \leq \tau_{\text{arr}}(J, v_{n+1})$: If the arrival via the route of T_n in round n improves the previous value of $\tau_{\text{arr}}(w_n, n)$, then the following transfer phase in round n will relax the shortcut (w_n, v_{n+1}) and find a suitable arrival at v_{n+1}. Otherwise, $\tau_{\text{arr}}(w_n, n') \leq \tau_{\text{arr}}(J, w_n)$ must hold for some prior round $n' < n$. Then the transfer phase of round $n' + 1 \leq n$ will relax (w_n, v_{n+1}) and arrive at v_{n+1} in time. Because the base case $n = 1$ follows from the correctness of Bucket-CH and RAPTOR, the claim is proven by induction. □

7.2.3 UBM-TB

To achieve even faster query times, we introduce UBM-TB, a TB-based algorithm for computing restricted Pareto sets. UBM-TB follows the same query framework as BM-RAPTOR

but uses a variant of ULTRA-TB for the pruning searches and ULTRA-McTB for the main search. Switching to TB-based algorithms necessitates using different data structures for pruning. The main search of BM-RAPTOR relies on earliest arrival times $\overrightarrow{\tau_{\text{arr}}}(\cdot, n)$ and latest departure times $\overleftarrow{\tau_{\text{dep}}}(\cdot, n)$ for each round n, which are computed by the pruning searches. A natural adaptation of these data structures for TB is to replace them with a *forward reached index* $\overrightarrow{r}(T, n)$ and a *backward reached index* $\overleftarrow{r}(T, n)$ for each trip T and round n. The forward reached index $\overrightarrow{r}(T, n)$ is the first stop index along T that is reachable from v_{s} with n trips. The backward reached index $\overleftarrow{r}(T, n)$ is the last stop index i along T with the following property: if $T[i]$ is reached with n trips, then v_{t} is reachable from there while still weakly dominating at least one label in $\mathcal{L}^{\text{R}}$. These reached indices can be used for pruning in the `Enqueue` procedure: When entering a trip T at the stop event $T[i]$ in round n, a backward pruning search for a label ℓ does not enqueue the corresponding trip segment if $i < \overrightarrow{r}(T, |\ell| - n)$. Likewise, the main search does not enqueue the trip segment if $i > \overleftarrow{r}(T, n)$.

Computing Per-Round Reached Indices. The presented pruning scheme requires the pruning searches to output one reached index per stop and round, whereas the original TB query only maintains one reached index per stop across all rounds. For the forward pruning search, this can be changed by simply initializing $\overrightarrow{r}(v, n)$ with $\overrightarrow{r}(v, n-1)$ for each stop v at the start of round n. The backward pruning searches require a different approach because they do not access the rounds of $\overleftarrow{r}(\cdot, \cdot)$ in order. Before the first backward search is started, the backward reached indices for all rounds are initialized with $-\infty$. Whenever a backward reached index $\overleftarrow{r}(T, n)$ is set to a value i, this value is propagated to the preceding rounds by setting $\overleftarrow{r}(T, n')$ to $\max(\overleftarrow{r}(T, n'), i)$ for all $0 \leq n' < n$.

Shortcut Augmentation. To establish a correct pruning scheme, the TB pruning searches must fulfill a condition analogous to that of Theorem 7.5: Let $T[i]$ be a stop event at which a Pareto-optimal journey found by ULTRA-McTB exits its n-th trip. To ensure that UBM-TB can also find this journey, $\overrightarrow{r}(T, n) \leq i$ must hold after the forward pruning search. Unfortunately, simply using the set E^{s} of event-to-event McULTRA shortcuts for the pruning searches is not enough to guarantee this. Consider a shortcut $e = (T_c[i], T_b[j]) \in E^{\text{s}}$ that is relaxed by an ULTRA-McTB search in round n. If there is an earlier trip $T_a \prec T_c$ such that $T_a[i]$ is reachable by round n (albeit with a higher transfer time than $T_c[i]$), then the two-criteria forward pruning search will not scan $T_c[i]$ and therefore not relax e. Because there is no guarantee that the shortcut $(T_a[i], T_b[j])$ exists, the search may not reach $T_b[j]$.

To solve this problem, we introduce the set $E^{\text{s}}_{\text{aug}}$ of *augmented shortcuts*. An illustration is given in Figure 7.2. A simple solution is to insert the shortcut $(T_a[i], T_b[j])$ if it is missing, which yields the shortcut set

$$E^{\text{s}}_{\text{full}} := \left\{(T_a[i], T_b[j]) \mid \exists\, T_c \succeq T_a \colon (T_c[i], T_b[j]) \in E^{\text{s}}\right\}.$$

However, this is wasteful: If there is another shortcut $(T_a[k], T_d[\ell]) \in E^{\text{s}}_{\text{full}}$ with $T_d \preceq T_b$, $k \geq i$ and $\ell \leq j$, then the search will relax this shortcut and $\overrightarrow{r}(T_b, n) \leq \overrightarrow{r}(T_d, n) \leq \ell \leq j$ will

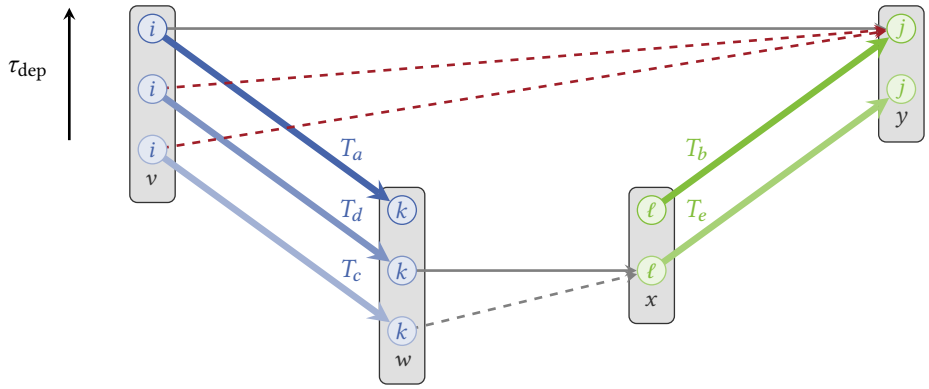


Figure 7.2: Shortcut augmentation in an example network. Trips T_a, T_c and T_d belong to the blue route, which visits stop v at index i and then w at index k. Trips T_b and T_e belong to the green route, which visits x at index ℓ and then y at index j. Earlier trips are drawn below later trips of the same route. Solid shortcuts are included in E^{s}, whereas dashed shortcuts are added to $E^{\text{s}}_{\text{full}}$. The gray shortcut $(T_c[k], T_e[\ell])$ is retained for $E^{\text{s}}_{\text{aug}}$, but the two red ones are removed because they are dominated. In particular, $(T_d[i], T_b[j])$ is dominated by $(T_d[k], T_e[\ell])$ and $(T_c[i], T_b[j])$ is dominated by $(T_c[k], T_e[\ell])$.

hold afterward. In this case, the shortcut $(T_a[i], T_b[j])$ is superfluous. We therefore define the set of *dominating shortcuts* for $(T_a[i], T_b[j])$ as

$$E^{\text{s}}_{\text{dom}}((T_a[i], T_b[j])) := \left\{(T_a[k], T_d[\ell]) \in E^{\text{s}}_{\text{full}} \mid T_d \preceq T_b, k \geq i, \ell \leq j\right\}.$$

We omit all shortcuts that are dominated by another shortcut, which yields the set

$$E^{\text{s}}_{\text{aug}} := \left\{e \in E^{\text{s}}_{\text{full}} \mid E^{\text{s}}_{\text{dom}}(e) = \{e\}\right\}.$$

The following lemmas and Theorem 7.9 prove that using an ULTRA-TB query with per-round reached indices and augmented McULTRA shortcuts for the pruning searches yields a correct pruning scheme.

Lemma 7.6. *For any shortcut $(T_c[i], T_b[j]) \in E^s$ and all trips $T_a \preceq T_c$, there exist a trip $T_d \preceq T_b$ and indices $k \geq i$, $\ell \leq j$ such that $(T_a[k], T_d[\ell]) \in E^s_{aug}$.*

Proof. For every trip $T_a \preceq T_c$, the shortcut $e = (T_a[i], T_b[j])$ is included in $E^{\text{s}}_{\text{full}}$. If $e \in E^{\text{s}}_{\text{aug}}$, the claim follows. Otherwise, there must be another shortcut $e' = (T_a[k], T_d[\ell])$ in $E^{\text{s}}_{\text{dom}}(e)$ besides e itself. Note that $E^{\text{s}}_{\text{dom}}(e') \subset E^{\text{s}}_{\text{dom}}(e)$ holds because $e \notin E^{\text{s}}_{\text{dom}}(e')$. Hence, we can choose e' such that $E^{\text{s}}_{\text{dom}}(e') = \{e'\}$ without loss of generality. Then $e' \in E^{\text{s}}_{\text{aug}}$. □

For a shortcut $e = (T_c[i], T_b[j]) \in E^{\text{s}}$ and each trip $T_a \preceq T_c$, let $k(T_a, e)$ denote the last stop index k such that a shortcut $(T_a[k], T_d[\ell]) \in E^{\text{s}}_{\text{aug}}$ exists with $k \geq i$, $\ell \leq j$ and $T_d \preceq T_b$. Note that such an index must exist by Lemma 7.6.

Lemma 7.7. *Let $e = (T_c[i], T_b[j]) \in E^s$ be a shortcut. Then for each pair of trips $T_a \preceq T_e \preceq T_c$, it holds that $k(T_a, e) \geq k(T_e, e)$.*

Proof. Let $k_1 := k(T_e, e)$. By definition of k_1, a shortcut $(T_e[k_1], T_d[\ell_1]) \in E^s_{\text{aug}}$ with $\ell_1 \leq j$ and $T_d \preceq T_b$ exists. By definition of E^s_{aug}, there must be a shortcut $(T_f[k_1], T_d[\ell_1]) \in E^s$ with $T_f \succeq T_e \succeq T_a$. Lemma 7.6 implies the existence of a shortcut $(T_a[k_2], T_g[\ell_2]) \in E^s_{\text{aug}}$ with $k_2 \geq k_1 \geq i$, $\ell_2 \leq \ell_1 \leq j$ and $T_g \preceq T_d \preceq T_b$. It follows that $k(T_a, e) \geq k_2 \geq k_1 = k(T_e, e)$. □

Lemma 7.8. *Consider a shortcut $e = (T_a[i], T_b[j]) \in E^s$. Assume that $r(T_a, n) \leq i$ holds after the n-th round of an ULTRA-TB search using E^s_{aug} as the set of shortcuts. Then there is a shortcut $(T_c[k], T_d[\ell]) \in E^s_{aug}$ with $T_c \preceq T_a$, $T_d \preceq T_b$, $k \geq i$ and $\ell \leq j$ that is relaxed by the search in some round $n' \leq n$.*

Proof. For each stop index $x \geq r(T_a, n)$, let $T(x)$ be the earliest trip of $R(T_a)$ such that $T[x]$ is scanned in some round $n' \leq n$. Note that for each index $y \geq x$, it holds that $T(y) \preceq T(x)$. Furthermore, if $x \geq r(T, n)$ for some trip T, then it follows that $T \succeq T(x)$.

To simplify notation, we write $k(T)$ instead of $k(T, e)$ for a trip T. Let $k_{\min} := k(T_a)$. It holds that $k_{\min} \geq i \geq r(T_a, n)$. Hence, we know that $T_a \succeq T(k_{\min})$. By Lemma 7.7, this yields that $k_{\min} \leq k(T(k_{\min}))$. Conversely, because $k_{\max} = |T_a| - 1$ is the highest possible index, it must hold that $k_{\max} \geq k(T(k_{\max}))$. There must be some index k with $k_{\min} \leq k \leq k_{\max}$ for which $k = k(T(k))$ because $k(T(k))$ cannot decrease as k increases by Lemma 7.7. Let $T_c := T(k)$. It follows from $k \geq k_{\min}$ that $T_c \preceq T_a$. By the definition of $k(\cdot)$, we know that there is a shortcut $(T_c[k], T_d[\ell]) \in E^s_{\text{aug}}$ with $T_d \preceq T_b$ and $\ell \leq j$. By the definition of $T(\cdot)$, we know that $T_c[k]$ is scanned in some round $n' \leq n$ and the shortcut is subsequently relaxed. □

Theorem 7.9. *Consider a three-criteria query with source vertex v_s, target vertex v_t and departure time τ_{dep}. Let J be a Pareto-optimal journey found by ULTRA-McTB for this query and $1 \leq n \leq |J|$ a number of trips. Let T_n denote the n-th trip of J and $T_n[j_n]$ the stop event at which J exits T_n. Then $r(T_n, n) \leq j_n$ holds for the reached index of an ULTRA-TB search using E^s_{aug} as the set of shortcuts.*

Proof. By induction over n. The base case $n = 1$ follows from the correctness of Bucket-CH and TB. Assume the claim is true for $n - 1$, i.e., $r(T_{n-1}, n-1) \leq j_{n-1}$. We know that the shortcut $(T_{n-1}[j_{n-1}], T_n[i_n]) \in E^s$ exists for the stop event $T_n[i_n]$ at which J enters T_n. By Lemma 7.8, there is a shortcut $(T'_{n-1}[k], T'_n[\ell]) \in E^s_{\text{aug}}$ with $T'_{n-1} \preceq T_{n-1}$, $T'_n \preceq T_n$, $k \geq j_{n-1}$ and $\ell \leq i_n$ that is relaxed in some round $j \leq n$. Thus,

$$r(T_n, n) \leq r(T'_n, n) \leq \ell + 1 \leq i_n + 1 \leq j_n.$$ □

Optimizations. As described thus far, the pruning scheme causes unnecessary work during the backward searches: The initial transfer phase of a backward search collects all routes from which v_t is reachable and scans them to find trip segments to enqueue. This is efficient for a normal ULTRA-TB query because v_t is typically reachable from most stops. In a

bounded query, however, most stops are not reachable from v_s in time to produce an optimal journey when transferring from there to v_t. For such stops, the `Enqueue` procedure will be called, but no trip segments will be enqueued because they will be pruned by the forward reached indices.

To avoid unnecessary `Enqueue` calls at these stops, we replace the forward reached indices $\overrightarrow{r}(\cdot,\cdot)$ with earliest arrival times at stops. As in the RAPTOR-based algorithm, the forward pruning search now computes an arrival time $\overrightarrow{\tau_{\text{arr}}}(v,n)$ per stop v and round n. This is the earliest arrival time among all found journeys to v with n trips that end with a trip. A corollary of Theorem 7.9 is that after round n, the arrival time $\overrightarrow{\tau_{\text{arr}}}(v,n)$ is not later than the arrival time of a Pareto-optimal journey to v with n trips that ends with a trip, allowing the backward search to use it for pruning. When entering a trip T at stop event $T[i]$ in round n, a backward search for a label ℓ does not enqueue the trip segment if $\overrightarrow{\tau_{\text{arr}}}(v(T[i]),|\ell|-n) > \tau_{\text{dep}}(T[i])$. To explore initial transfers, a backward search for a label ℓ first collects all stops v from which v_t is reachable and for which $\overrightarrow{\tau_{\text{arr}}}(|\ell|) + \tau_{\text{tra}}(v,v_t) \leq \tau_{\text{arr}}(\ell)$ holds. For each such stop v and each route visiting v, the latest reachable trip is found via binary search and `Enqueue` is called.

To compute $\overrightarrow{\tau_{\text{arr}}}(\cdot,\cdot)$, the forward pruning search adds an extra operation during the trip scanning phase. Before relaxing the outgoing transfers of a stop event $T[i]$ in round n, the arrival time $\overrightarrow{\tau_{\text{arr}}}(v(T[i]),n)$ is now set to the minimum of itself and $\tau_{\text{arr}}(T[i])$. When starting a new round n, the arrival time $\overrightarrow{\tau_{\text{arr}}}(v,n)$ for each stop v is initialized with $\overrightarrow{\tau_{\text{arr}}}(v,n-1)$.

A final optimization concerns the main search. Normally, the reached transfer times $r_{\text{tra}}(\cdot,\cdot)$ maintained by McTB are reset at the start of each query. In the context of UBM-TB, which prunes most of the search space, this is often more expensive than the main search itself. Hence, we mark each reached transfer time $r_{\text{tra}}(T,i)$ with a timestamp. When $r_{\text{tra}}(T,i)$ is accessed and its timestamp does not match that of the current query, the value is reset to ∞.

7.3 Experiments

We begin our experimental evaluation by analyzing the impact that optimizing transfer time has on the solution quality in Section 7.3.1. Section 7.3.2 evaluates the McULTRA shortcut computation and the algorithms for computing full Pareto sets on the four benchmark networks. As in Chapter 5.3, we test the shortcut hypothesis for different transfer speeds. Finally, we evaluate the algorithms for computing restricted Pareto sets in Section 7.3.3. Shortcut computations were run on the Epyc machine, all other experiments on the Xeon machine. Unless otherwise noted, walking with a constant speed of 4.5 km/h is used as the transfer mode.

7.3.1 Impact of Optimizing Transfer Time

To study the impact of optimizing transfer time on the solution quality, we compare minimal transfer times in the two-criteria Pareto set to those in restricted three-criteria Pareto sets for varying slack values. We choose restricted Pareto sets instead of a full one because they

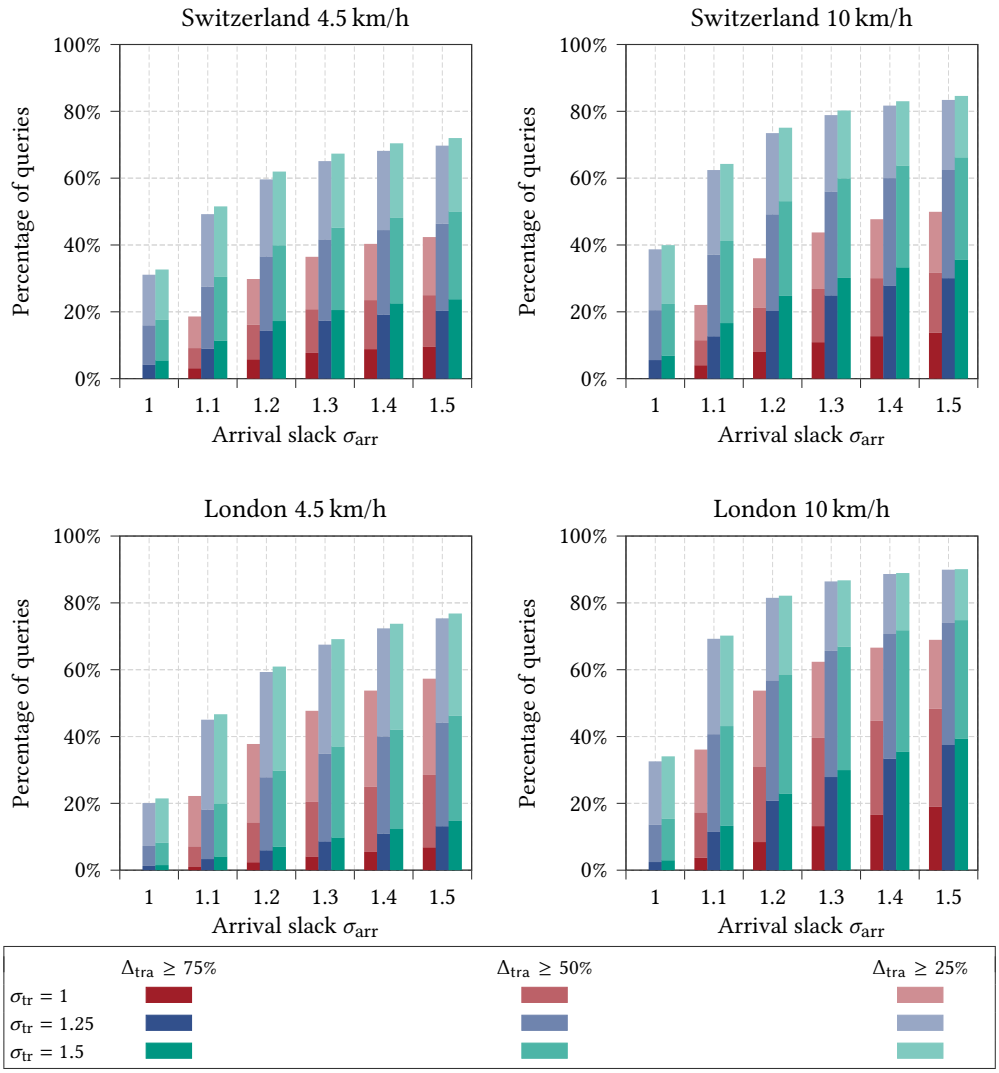


Figure 7.3: Comparison of optimal transfer times in the two-criteria Pareto set $\mathcal{J}^{\mathrm{A}}$ versus a restricted three-criteria Pareto set $\mathcal{J}^{\mathrm{R}}$ on the Switzerland and London networks for different transfer speeds. 10 000 random queries were run for each choice of slack values. We then calculated the transfer time savings as $\Delta_{\mathrm{tra}} = (\tau_A - \tau_R)/\tau_A$, where τ_R is the lowest transfer time in $\mathcal{J}^{\mathrm{R}}$ and τ_A the lowest transfer time in $\mathcal{J}^{\mathrm{A}}$. Shown is the percentage of queries for which Δ_{tra} exceeds the specified threshold.

exclude undesirable journeys with a low transfer time but excessive costs in the other criteria. Figure 7.3 shows the results on Switzerland and London for different transfer speeds. With walking as the transfer mode, more than 25% of the transfer time can be saved for most queries, even with small slack values. In up to 50% of all queries, more than half of the transfer time can be saved. This confirms that adding transfer time as a third criterion often significantly improves the quality of the found journeys. The savings become even more pronounced as the transfer speed increases. For 10 km/h (e.g., slow cycling), the vast majority of queries allow at least a moderate improvement in transfer time. Often, improvements are possible even without allowing a trip slack. This underscores that optimizing the transfer time becomes especially crucial as the transfer speed increases. Fast transfer modes are frequently competitive with public transit in terms of travel time, but not necessarily in terms of comfort. If the transfer time is not considered, optimal journeys will often avoid trips altogether in favor of long transfers.

For walking as the transfer mode, both networks behave similarly. London has a slightly larger share of queries with savings above 25%, whereas very large savings are more frequent on the Switzerland network. The impact of the transfer speed is more drastic for the London network, in which large savings are more common for a transfer speed of 10 km/h than on the Switzerland network. This can be explained by the different network topologies: The London network consists of a dense metropolitan area. Here, public transit is common, but the average travel speed is fairly low across long distances because the trips halt frequently. Thus, even moderately fast transfer modes quickly become competitive in terms of travel time. By contrast, long-distance journeys in Switzerland often involve high-speed trains, against which only cars are competitive.

7.3.2 Full Pareto Sets

For the purpose of computing full three-criteria Pareto sets, we evaluate the McULTRA shortcut computation and the McULTRA-based query algorithms.

Shortcut Computation. For the Core-CH precomputation required by the McULTRA shortcut computation algorithm, we used the same settings as in Chapter 5.3: the transfer graphs were contracted up to an average vertex degree of 20 for Germany and 14 for the other networks. As mentioned in Section 7.1.2, the first route scanning phase of each MCR run may use simplified MR-like route scans, but this may lead to superfluous shortcuts. On the Switzerland network with both witness limits set to ∞, using MCR-like route scans yields 4% fewer shortcuts for the stop-to-stop variant and 1% for the event-to-event variant. However, this increases the shortcut computation time by 23% and 11%, respectively. Hence, MR-like route scans are used for all subsequent experiments. Similarly, the intermediate witness limit is always set to $\lambda_1^{\mathrm{w}} = 0$ because this reduces the event-to-event computation time by 64% while only producing 0.4% more shortcuts.

Figure 7.4 shows the impact of the Dijkstra label key and the final witness limit λ_2^{w} on the computation time and the number of shortcuts in the event-to-event variant. Especially

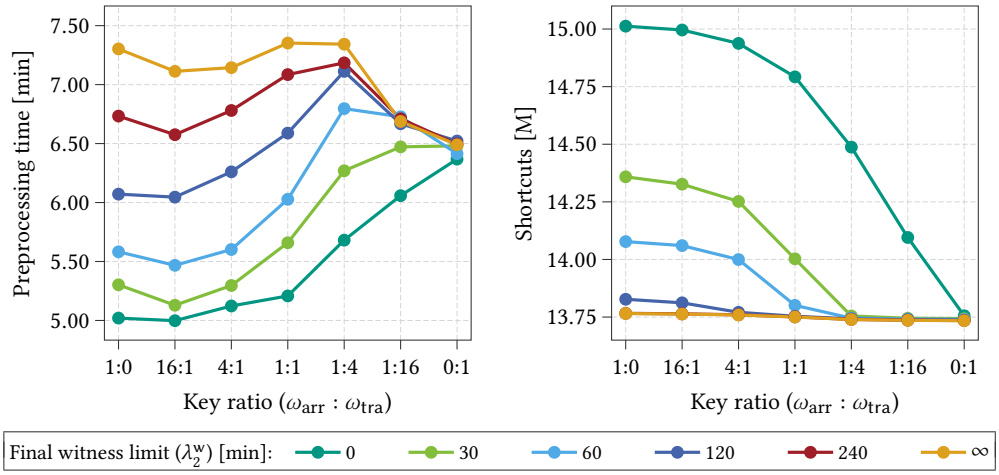


Figure 7.4: Impact of key choice and final witness limit λ_2^w on the shortcut computation time and the number of shortcuts of event-to-event McULTRA, measured on the Switzerland network. For a label with arrival time τ_{arr} and transfer time τ_{tra}, a key ratio of $\omega_{arr} : \omega_{tra}$ indicates a key value of $(\omega_{arr} \cdot \tau_{arr} + \omega_{tra} \cdot \tau_{tra})/(\omega_{arr} + \omega_{tra})$. A weight of 0 indicates that the criterion is only used as a tiebreaker.

Table 7.1: Shortcut computation results for ULTRA and McULTRA. Times are formatted as hh:mm:ss. For event-to-event McULTRA, $|\overrightarrow{E^s_{aug}}|$ and $|\overleftarrow{E^s_{aug}}|$ are the number of augmented shortcuts for the forward and backward pruning search, respectively.

Network	Variant	ULTRA		McULTRA			
		Time	# Shortcuts	Time	# Shortcuts	$\lvert\overrightarrow{E^s_{aug}}\rvert$	$\lvert\overleftarrow{E^s_{aug}}\rvert$
Stuttgart	Stop	00:00:54	83 086	00:02:43	121 251	–	–
	Event	00:01:00	1 973 321	00:03:41	5 865 012	12 212 838	12 492 437
London	Stop	00:03:44	190 388	00:16:32	135 395	–	–
	Event	00:04:17	8 576 120	00:25:56	18 955 051	38 202 817	43 439 278
Switzerland	Stop	00:01:57	170 713	00:04:23	218 899	–	–
	Event	00:02:11	6 938 012	00:05:35	14 077 529	40 231 588	40 416 364
Germany	Stop	02:53:57	2 907 691	04:07:03	2 919 962	–	–
	Event	02:55:12	77 515 291	05:55:29	197 690 344	580 541 220	590 717 058

when the arrival time is weighted heavily, imposing a final witness limit saves considerable computation time, at the cost of producing noticeably more shortcuts. The latter can be mitigated by weighting the transfer time more heavily, but this increases the computation time for strict final witness limits. This is explained by the fact that improper candidates, which are weakly dominated by journeys found in previous MCR runs, tend to have higher arrival times than proper candidates. Increasing the arrival time weight moves proper candidates toward the front of the queue, allowing the stopping criterion to be applied earlier. To strike a balance between computation time and the number of shortcuts, we choose weights $\omega_{\text{arr}} = 1$ and $\omega_{\text{tra}} = 0$ as well as a final witness limit $\lambda_2^{\text{w}} = 60$ min for all subsequent experiments.

Table 7.1 shows overall results for the shortcut computation on all networks. Compared with two-criteria ULTRA, the number of shortcuts increases by less than a factor of two for the stop-to-stop variant, and between two and three for the event-to-event variant. Computing the shortcuts takes two to four times longer on the Stuttgart, Switzerland and Germany networks, and four to six times longer on London. By contrast, three-criteria MCR has a slowdown of about 20 compared with its two-criteria variant, MR. This shows that the shortcut computation scales much better for the additional criterion than a regular one-to-one query.

A remarkable observation is that the growth in the number of stop-to-stop shortcuts is very low compared with the event-to-event shortcuts. In fact, McULTRA even manages to reduce the number of stop-to-stop shortcuts on the London network. This may be surprising at first, as adding a third criterion should lead to more Pareto-optimal candidates and thereby more shortcuts. However, it also affects the way ties are broken between journeys that have the same arrival time and number of trips. Among equivalent candidates with the same second trip, two-criteria ULTRA prefers the one that enters the trip at the earliest stop, whereas McULTRA prefers the one with the shortest intermediate transfer.

To examine the influence of the tiebreaking rule on the number of shortcuts, we implemented a variant of McULTRA that does not consider transfer time as a proper third criterion but uses it as a tiebreaker between equivalent journeys. Figure 7.5 shows the transfer time distribution for all three ULTRA variants on the London network. We observe that using transfer time as a tiebreaker increases the number of event-to-event shortcuts overall but skews the distribution toward lower transfer times. Pareto-optimizing it as a third criterion adds more shortcuts across the board, but the increase is stronger for low transfer times. As already observed for ULTRA (cf. Section 5.3.1), event-to-event shortcuts with a low transfer distance are more likely to share the same stop pair as those with a higher distance, simply because there are fewer pairs to choose from. Thus, if we switch from the event-based representation to the stop-based one, the growth in the number of shortcuts for low transfer times is dampened, whereas the loss for high transfer times is amplified. On the London network, which is very dense, this effect is strong enough to overcome the overall increase in the number of shortcuts.

Query Algorithms. We evaluate two query algorithms for computing full Pareto sets: ULTRA-McRAPTOR (combining McRAPTOR with stop-to-stop McULTRA shortcuts) and

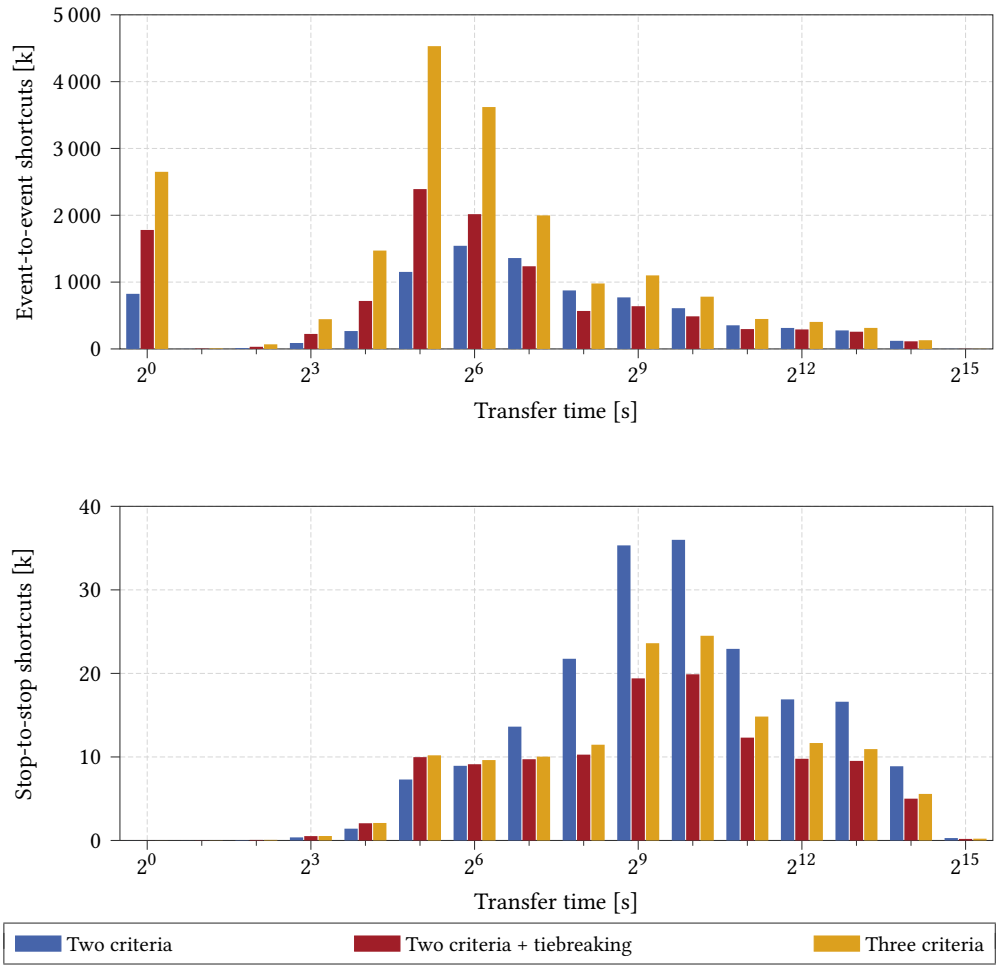


Figure 7.5: Transfer time distribution of the (Mc)ULTRA shortcuts for the London network. The blue bars correspond to the original ULTRA shortcuts for the two criteria arrival time and number of trips. The red bars add transfer time as a tiebreaker in the case of equality, whereas the yellow bars represent McULTRA, which Pareto-optimizes transfer time as a third criterion. The bars for 2^0 represent all shortcuts with a transfer time below one second. For $i > 0$, the bars for 2^i correspond to the number of shortcuts with a transfer time in the interval $[2^{i-1}, 2^i)$.

Table 7.2: Performance of full Pareto set algorithms, averaged over 10 000 random queries. RAPTOR query times are divided into phases: initialization, including clearing vertex bags and exploring initial/final transfers (*Init.*), collecting (*Collect*) and scanning (*Scan*) routes, and relaxing intermediate transfers (*Relax*). Also reported are the number of rounds (*Rnd.*) and the number of found journeys (*Jrn.*). McRAPTOR* only supports stop-to-stop queries with transitive transfers.

Network	Algorithm	Full graph	Rnd.	Jrn.	Time [ms]				
					Init.	Collect	Scan	Relax	Total
Stuttgart	McRAPTOR*	○	22.7	34.7	16.7	9.2	133.0	136.4	295.2
	MCR	●	28.9	34.2	55.2	18.4	235.2	662.2	955.2
	ULTRA-McRAPTOR	●	28.9	34.2	30.9	14.6	232.3	82.9	360.7
	ULTRA-McTB	●	28.5	34.2	–	–	–	–	100.5
London	McRAPTOR*	○	12.6	21.8	13.0	2.0	101.2	152.6	268.7
	MCR	●	14.6	30.5	30.2	3.4	128.0	296.9	458.1
	ULTRA-McRAPTOR	●	14.6	30.5	21.0	3.0	125.4	50.0	199.4
	ULTRA-McTB	●	14.8	30.5	–	–	–	–	78.6
Switzerland	McRAPTOR*	○	24.3	20.8	23.9	8.4	144.2	130.4	306.9
	MCR	●	32.8	30.5	62.2	15.3	233.1	420.7	731.3
	ULTRA-McRAPTOR	●	32.8	30.5	40.6	14.4	231.2	59.2	345.3
	ULTRA-McTB	●	33.6	30.5	–	–	–	–	147.5
Germany	McRAPTOR*	○	30.9	38.3	494.1	300.4	5 043.0	3 422.5	9 260.0
	MCR	●	36.6	58.0	1 213.4	474.6	7 628.0	20 694.5	30 010.6
	ULTRA-McRAPTOR	●	36.6	58.0	1 078.3	591.9	8 761.6	2 168.1	12 600.0
	ULTRA-McTB	●	36.3	58.0	–	–	–	–	4 780.7

ULTRA-McTB (combining McTB with event-to-event McULTRA shortcuts). Query times are reported in Table 7.2. Compared with MCR, the fastest previously known algorithm, ULTRA-McRAPTOR is about two to three times as fast and ULTRA-McTB achieves a speedup of five to ten. As in the two-criteria scenario, ULTRA-McRAPTOR achieves its speedup mostly by significantly reducing the transfer relaxation time. This results in similar query times to McRAPTOR on a transitively closed transfer graph. The slight slowdown compared to transitive McRAPTOR on most networks is explained by the significantly higher number of found journeys.

Impact of Transfer Speed. Figure 7.6 shows how McULTRA is affected by the speed of the transfer mode. For two-criteria ULTRA, we observed in Chapter 5.3.1 that the number of shortcuts declines once the transfer speed becomes competitive with public transit. This

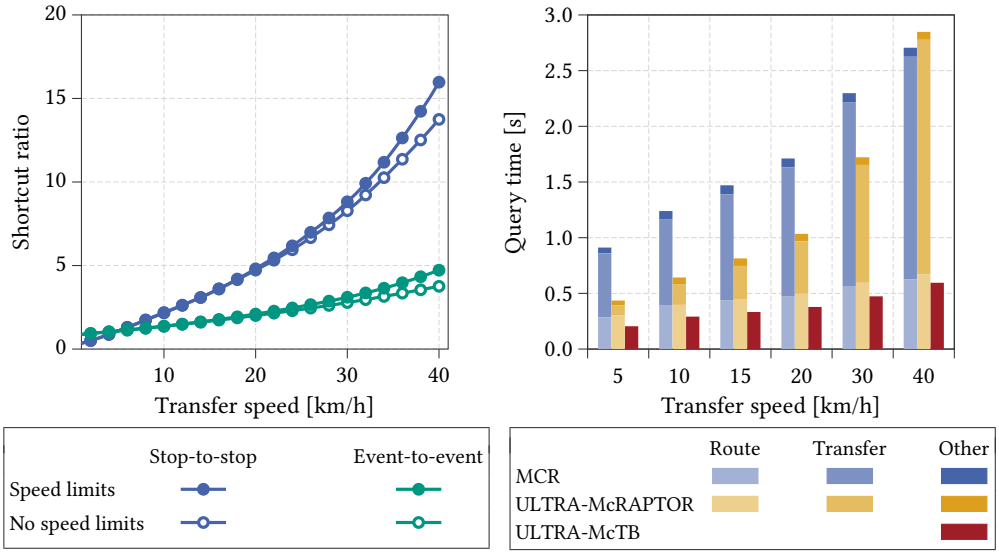


Figure 7.6: Impact of transfer speed for the Switzerland network. *Left:* Ratio of McULTRA shortcuts compared with a transfer speed of 4.5 km/h. Speed limits in the network were obeyed for the lines with filled circles and ignored for the lines with empty circles. *Right:* Query performance of MCR and ULTRA-based algorithms, averaged over 1 000 random queries. Speed limits were obeyed. For the RAPTOR-based algorithms, query times are divided into route collecting/scanning, transfer relaxation, and remaining time.

is because with arrival time and number of trips as criteria, there is no reason to use public transit unless it saves travel time. For high transfer speeds, a direct transfer from source to target is often the only Pareto-optimal journey, so fewer shortcuts for intermediate transfers are required. This is no longer the case when adding transfer time as a criterion because making a public transit detour can save transfer time. Consequently, the number of shortcuts increases for high transfer speeds. This effect is much stronger for the stop-to-stop variant than for the event-to-event variant, which indicates that most of the additional shortcuts are only required at a few specific times during the day. All query algorithms become slower for higher transfer speeds as the search space increases. ULTRA-McRAPTOR is practical for transfer speeds up to 20 km/h (which is faster than the average travel speed via bicycle or e-scooter) but loses its advantage over MCR above 30 km/h. By contrast, ULTRA-McTB maintains a speedup of up to five even for transfer speeds up to 40 km/h, due to the slower increase in the number of shortcuts.

Overall, we observe that the shortcut hypothesis only holds under the assumption that we outlined in Chapter 1: public transit is the main transportation mode and the transfer mode is mostly used for bridging gaps in poorly serviced areas. If the transfer mode is competitive

Table 7.3: Performance of the bounded query algorithms with $\sigma_{arr} = \sigma_{tr} = 1.25$, averaged over 10 000 random queries. Query times are divided into forward and backward pruning searches and main search. Also reported are the number of rounds performed by the main search (*Rnd.*) and the number of found journeys (*Jrn.*). The performance of the two-criteria algorithms is shown for comparison. (BM-)RAPTOR* only supports stop-to-stop queries with transitive transfers.

Network	Algorithm	Full graph	Rnd.	Jrn.	Time [ms]			
					Forward	Backward	Main	Total
Stuttgart	RAPTOR*	○	7.1	1.8	–	–	–	6.8
	ULTRA-RAPTOR	●	7.5	4.0	–	–	–	10.9
	ULTRA-TB	●	7.7	4.0	–	–	–	4.0
	BM-RAPTOR*	○	5.2	5.4	13.8	3.7	10.9	28.3
	UBM-RAPTOR	●	4.2	9.8	16.5	4.2	9.9	30.6
	UBM-TB	●	4.2	9.8	10.6	2.0	3.9	16.5
London	RAPTOR*	○	7.6	2.3	–	–	–	10.5
	ULTRA-RAPTOR	●	7.2	4.3	–	–	–	6.5
	ULTRA-TB	●	7.1	4.3	–	–	–	3.6
	BM-RAPTOR*	○	5.3	8.0	21.8	4.7	17.6	44.1
	UBM-RAPTOR	●	4.6	12.4	10.8	2.9	7.9	21.6
	UBM-TB	●	4.6	12.4	9.8	1.7	4.1	15.6
Switzerland	RAPTOR*	○	7.9	2.0	–	–	–	14.5
	ULTRA-RAPTOR	●	7.3	4.6	–	–	–	13.7
	ULTRA-TB	●	7.6	4.6	–	–	–	4.7
	BM-RAPTOR*	○	5.7	4.8	23.2	4.0	11.8	39.0
	UBM-RAPTOR	●	4.8	9.2	21.8	3.4	9.2	34.4
	UBM-TB	●	4.8	9.2	13.4	1.5	2.7	17.6
Germany	RAPTOR*	○	10.3	2.3	–	–	–	326.0
	ULTRA-RAPTOR	●	9.4	5.3	–	–	–	389.3
	ULTRA-TB	●	9.9	5.3	–	–	–	86.7
	BM-RAPTOR*	○	7.0	8.0	462.3	47.7	159.8	669.8
	UBM-RAPTOR	●	5.9	13.9	567.7	41.3	130.2	739.2
	UBM-TB	●	5.9	13.9	260.9	24.0	39.4	324.2

with public transit in terms of speed and also widely available, then it may become the main mode instead. If it is also more cumbersome to use than public transit and therefore transfer time should be penalized, then our assumption no longer holds and ULTRA is not a suitable approach.

Table 7.4: Performance of the bounded query algorithms for different slack values on the Switzerland network, averaged over 10 000 random queries. Query times are divided into forward and backward pruning searches and main search. Also reported are the number of rounds performed by the main search (*Rnd.*) and the number of found journeys (*Jrn.*). BM-RAPTOR* only supports stop-to-stop queries with transitive transfers.

Algorithm	σ_{tr}	σ_{arr}	Full graph	Rnd.	Jrn.	Time [ms]			
						Forward	Backward	Main	Total
BM-RAPTOR*	1.25	1.2	○	5.7	4.5	22.5	3.2	9.0	34.7
	1.25	1.25	○	5.7	4.8	23.2	4.0	11.8	39.0
	1.25	1.3	○	5.7	5.1	24.0	4.9	14.6	43.5
	1.25	1.5	○	5.7	5.8	25.4	7.8	25.1	58.4
	1.5	1.2	○	6.5	5.1	23.0	4.9	15.9	43.8
	1.5	1.25	○	6.5	5.5	23.5	6.2	20.6	50.4
	1.5	1.3	○	6.6	5.9	24.0	7.3	25.2	56.6
	1.5	1.5	○	6.6	6.9	25.6	11.6	43.5	80.7
UBM-RAPTOR	1.25	1.2	●	4.8	8.7	20.9	2.9	7.7	31.5
	1.25	1.25	●	4.8	9.2	21.8	3.4	9.2	34.4
	1.25	1.3	●	4.8	9.6	22.7	4.0	10.7	37.4
	1.25	1.5	●	4.9	10.6	25.2	6.4	16.9	48.5
	1.5	1.2	●	5.6	9.4	21.1	3.4	9.2	33.7
	1.5	1.25	●	5.6	10.0	21.9	4.1	11.0	37.1
	1.5	1.3	●	5.7	10.5	22.7	4.8	13.0	40.5
	1.5	1.5	●	5.7	11.9	25.1	7.7	21.6	54.4
UBM-TB	1.25	1.2	●	4.8	8.7	12.4	1.1	1.9	15.4
	1.25	1.25	●	4.8	9.2	13.4	1.5	2.7	17.6
	1.25	1.3	●	4.8	9.6	14.1	1.8	3.7	19.6
	1.25	1.5	●	4.8	10.6	17.1	3.6	8.0	28.8
	1.5	1.2	●	5.5	9.4	12.5	1.4	2.6	16.5
	1.5	1.25	●	5.5	10.0	13.2	1.9	3.8	18.9
	1.5	1.3	●	5.6	10.5	14.1	2.3	5.2	21.6
	1.5	1.5	●	5.7	11.9	17.1	4.5	11.4	32.9

7.3.3 Restricted Pareto Sets

We conclude by evaluating our bounded query algorithms. As shown in Table 7.1, shortcut augmentation increases the number of shortcuts by a factor of two to three. The augmentation takes a few seconds for the three smaller networks and eight minutes for Germany, which is negligible compared with the shortcut computation time. Query times for the bounded

algorithms are shown in Table 7.3. Based on the results from Figure 7.3, we choose slack values $\sigma_{\text{arr}} = \sigma_{\text{tr}} = 1.25$ as a good tradeoff between solution quality and query speed. The bounded algorithms are a factor of two to four slower than their two-criteria counterparts. This is consistent with the slowdown observed for BM-RAPTOR compared with RAPTOR on transitively closed transfer graphs. Most of the running time is taken up by the forward pruning search, which indicates that the pruning scheme is highly effective. Due to weaker target pruning and the algorithmic changes discussed in Section 7.2, the forward pruning search is slower than a two-criteria query, particularly for UBM-TB. Nevertheless, UBM-TB is about twice as fast as UBM-RAPTOR. Compared with the ULTRA-based algorithms for full Pareto sets, the bounded variants achieve a speedup of around an order of magnitude. Compared with MCR, which was previously the fastest algorithm for this problem setting, UBM-TB achieves a speedup of 30–90.

Table 7.4 shows the performance of the bounded query algorithms for different slack values on the Switzerland network. The forward pruning search is not significantly impacted by the trip slack, and only moderately by the arrival slack, due to the weaker target pruning. The backward pruning searches and main search take significantly longer for higher slack values and start to dominate the running time for $\sigma_{\text{tr}} = 1.5$ and $\sigma_{\text{arr}} = 1.5$. However, even for high slack values, the bounded algorithms remain much faster than their unbounded counterparts.

7.4 Conclusion

We showed that in order to obtain high-quality solutions in the presence of unlimited transfers, it is necessary to optimize transfer time as a third criterion. In contrast to multicriteria optimization in general, we showed that this problem can be solved in polynomial time. To exploit this, we developed McTB, a fast three-criteria algorithm that avoids costly dynamic data structures. To enable unlimited transfers, we proposed McULTRA, a three-criteria extension of ULTRA. By adapting the optimizations of ULTRA and introducing additional ones, we obtained a shortcut computation algorithm that runs in reasonable time and produces less than twice as many shortcuts as two-criteria ULTRA. The combination of McULTRA and McTB achieves a speedup of five to ten over the state of the art and remains practical even for fast transfer modes. Finally, we developed RAPTOR- and TB-based algorithms for computing restricted Pareto sets in a multimodal network. The latter is up to 90 times faster than MCR, the fastest previously known algorithm for three-criteria multimodal queries.

Although we focused on optimizing transfer time, our algorithms can be used to optimize any third criterion that fulfills requirements (R1) and (R2) and is compatible with route-based pruning rules. An example would be a criterion that represents passenger preferences between different types of public transit vehicle (e.g., preferring regional trains over more expensive high-speed trains). To ensure compatibility with route-based pruning, all trips within a route must be equivalent according to this criterion, which can be enforced by grouping the trips into routes accordingly. In particular, our algorithms may be useful for three-criteria problem settings in pure public transit networks, in which optimizing transfer time is not necessarily

important. In this context, the event-to-event variant McULTRA can serve as a three-criteria replacement for the transfer generation step of TB. It may also be interesting to generalize McTB to criteria that are not compatible with route-based pruning, such as vehicle occupancy.

Outlook. The results of our experimental evaluation provide further insights into the limits of the shortcut hypothesis. If public transit is the faster and more comfortable mode and the transfer mode is used for bridging gaps, then the shortcut hypothesis holds and can be successfully exploited with McULTRA. If the transfer mode is comfortable enough that its usage does not need to be penalized, then regardless of its speed it can be handled with two-criteria ULTRA. However, if the transfer mode is competitive in terms of speed and its usage is penalized, then the shortcut hypothesis no longer holds. The reason for this is that the modes switch places: the transfer mode becomes the "main" mode and public transit is only useful as a backup for reducing discomfort.

Most common transfer modes, including, walking, cycling and e-scooters, are slow enough to fit the shortcut hypothesis. The main exception is cars. If car usage is not penalized and cars are available for all parts of the journey, then a direct car journey dominates all solutions that include public transit, so there is no need for multimodal journey planning. However, there are still valid multimodal scenarios that include cars. We outline three examples:

Scenario 1: Taxi. Taxis are widely available and transfer time acts as a proxy for fare. Since taxis are the fastest mode and their usage is penalized, many solutions will interrupt taxi rides with public transit in order to reduce the fare. This leads to a combinatorial explosion in the number of optimal journeys, as previously observed by Delling et al. [Del+13] when they included taxis in a multimodal network. The polynomial upper bound on the number of Pareto-optimal solutions that we established in Section 7.1.1 still holds in this scenario. However, because this bound is based on the number of stop events, which is typically in the millions, the Pareto set may still be enormous. Nevertheless, it is intuitively clear that Pareto-optimal journeys that combine long car rides with short public transit detours are not interesting to passengers. The TNT approach by Bast et al. [BBS13] is an attempt to filter the Pareto set by removing such undesirable journeys, but so far no efficient algorithm has been proposed for computing the filtered set exactly. Thus, quickly identifying a small, representative subset in a scenario with unlimited car availability remains an open problem.

Scenario 2: Park and ride. A private car is available for the initial and/or final transfer but not for intermediate transfers. In this case, precomputed shortcuts are not required, whereas the initial and final transfers can be handled with Bucket-CH, as in ULTRA.

Scenario 3: Ridesharing. If public transit is combined with ridesharing services, then limited availability of the ridesharing mode may potentially offset the fact that its speed is competitive with public transit. Whether or not the shortcut hypothesis holds in this case is an open question. If so, then a more complex shortcut computation algorithm may be required to incorporate the special constraints of the ridesharing mode.

8 Multiple Transfer Modes

So far, we have only considered scenarios with a single transfer mode. In reality, there are often multiple transfer modes available, with different advantages and drawbacks. In this chapter, we therefore extend our results for bimodal networks to a more general setting with multiple competing transfer modes. As we discussed in the previous chapter, ULTRA is not a suitable approach for car-based modes, so we exclude these here. This leaves modes such as walking, private or rented bicycles, and rented e-scooters.

Chapter Outline. Allowing multiple transfer modes requires us to revisit some of our modeling assumptions. In Section 8.1, we establish and discuss a realistic model for fully multimodal journey planning with an arbitrary number of transfer modes, which we call the *multimodal discomfort scenario*. In addition to arrival time and number of trips, we Pareto-optimize the time spent in each transfer mode as an individual criterion. To ensure reasonable results, we exclude certain types of undesirable solutions, such as journeys that switch between transfer modes in the middle of a transfer.

The multimodal discomfort scenario requires algorithms that support an arbitrary number of criteria. In Section 8.2, we show that transfer shortcuts can be generated by running the three-criteria McULTRA preprocessing for each transfer mode individually, which only requires linear preprocessing effort in the number of modes. We adapt existing RAPTOR-based query algorithms to our scenario in Section 8.3. This enables the use of ULTRA-McRAPTOR to compute full Pareto sets and UBM-RAPTOR for restricted Pareto sets. We do not adapt UBM-TB because there is no apparent way to extend it to more than three criteria. Instead, Section 8.4 introduces HydRA (Hybrid Routing Algorithm), which combines the advantages of McRAPTOR and TB in scenarios with an arbitrary number of criteria. We combine our algorithm with ULTRA and restricted Pareto sets, which yields UB-HydRA (Bounded ULTRA-

HydRA). In Section 8.5, we evaluate the performance of our algorithms on the networks of Stuttgart, London and Switzerland, using walking and e-scooters as the transfer modes. On these networks, UB-HydRA achieves query times of 20–30 ms, which is faster than the state of the art by more than two orders of magnitude and enables interactive applications.

8.1 Multimodal Discomfort Scenario

Based on the findings from Chapter 7, we establish a model for multimodal journey planning with multiple competing transfer modes, which we call the *multimodal discomfort scenario*. It is based on the assumption outlined in Chapter 1: Public transit is generally the fastest and most comfortable available mode. Its main disadvantage is limited availability in rural areas and outside of peak hours. The transfer modes can bridge gaps in poorly serviced areas, but using them incurs discomfort, either because they are cumbersome (e.g., walking) or costly (e.g., e-scooter, bike-sharing). Accordingly, passengers prefer to use public transit unless using a transfer mode improves the arrival time or reduces the number of trips.

Optimization Criteria. As usual, the first two optimization criteria are arrival time and number of trips. In Chapter 7, which considered scenarios with a single transfer mode, we added transfer time as a third criterion to measure the discomfort associated with using the mode. The simplest way to extend this to multiple transfer modes is to optimize the combined transfer time across all transfer modes as a single criterion. However, this would imply that it is always preferable to use the fastest available mode. In reality, the different transfer modes have competing advantages and drawbacks, and preferences vary between users. For example, some passengers may prefer to rent a bicycle in order to be faster, whereas others may want to save money and walk instead. For others still, the decision may depend on situational factors, such as the distance traveled or the waiting time before the next trip departs. It is often difficult for users to express these considerations in precise mathematical terms that can be used algorithmically. Therefore, a common approach is to provide the users with a small, diverse selection of reasonable journeys to choose from [Del+13, BFP21]. To achieve this, we consider the transfer time for each mode as a separate criterion, independently of the other modes. This ensures that the query algorithm makes no assumptions about the user's preferences between different modes. However, this does not mean that the objective is to compute the full Pareto set, which may be extremely large and contain many uninteresting journeys that are small variations of other solutions. As in Chapter 7, we use the restricted Pareto set because it can be computed quickly and excludes some, but not necessarily all undesirable journeys. If the restricted Pareto set is still too large or contains unattractive journeys, it can be filtered in a post-processing step.

Mode Independence. To exclude a very common type of undesirable journey, we impose the following restriction: the transfer mode may not be switched in the middle of a transfer. To see why this is necessary, consider a journey that includes a bicycle ride as a transfer. If

we swap out the first edge of the transfer with walking, we obtain a second journey such that neither dominates the other because one has a lower bicycle time and the other has a lower walking time. If we repeat this for each subsequent edge along the path, none of the resulting journeys will dominate any of the others. Thus, allowing mode changes within a transfer would lead to a combinatorial explosion of nearly identical Pareto-optimal journeys. In practice, these journeys are undesirable because they disregard the fact that switching between modes takes time and incurs discomfort.

We argue that mode changes within a transfer are only useful in one circumstance: When renting a bicycle or e-scooter, passengers first have to walk to the pickup location, and after dropping it off, they have to walk to the next public transit stop. We do not treat these access/egress patterns as proper mode changes but rather as a part of the overall e-scooter/bicycle transfer. If needed, footpaths between public transit stops and pickup/dropoff points can be modeled directly in the transfer graph. The time spent using them is counted towards the e-scooter/bicycle time rather than the walking time. For our experiments, we assume that rented vehicles can be picked up and dropped off at any location. While this is a simplification, it is the most challenging scenario for the purpose of testing the shortcut hypothesis because it maximizes the availability of the rental modes. To represent the time required for access and egress, we add a fixed *mode overhead* to every transfer in the respective mode. This prevents solutions with unrealistically short scooter or bicycle transfers. For modes without rented vehicles, such as walking, we set the mode overhead to zero.

By prohibiting mode changes within a transfer, the multimodal discomfort scenario gains a crucial property that we call *mode independence*: a journey that uses a particular transfer mode for any length of time can never dominate a journey that does not. As we will show in Section 8.2, this allows us to apply ULTRA independently for each transfer mode. However, it has a side effect: Often, an illegal mode change can be circumvented by inserting a public transit trip in between as a detour. If Pareto-optimal journeys with mode changes are prohibited, these undesirable detour journeys may become Pareto-optimal in their place. However, because they incur an additional trip, they are more likely to be dominated by other, more reasonable journeys, so this choice reduces the number of undesirable solutions overall. If any such journeys are included in the restricted Pareto set, they can be removed during the post-processing step.

Free Transfers. Another side effect is caused by the way mode overheads interact with the modeling of the public transit network. Due to mode independence, if we disable a certain subset of transfer modes, the resulting Pareto set is still contained in the Pareto set that allows all modes. For the most part, this is beneficial because individual modes can be toggled on and off without having to rerun the entire query. However, it also means that the Pareto set includes journeys that avoid all modes without an overhead, including walking. There are two ways to model platforms belonging to the same station, both of which are used in real-world datasets: The first is to model each platform as its own stop and connect the stops via footpaths. The other option is to model them as a combined stop and represent the time

needed to transfer between them via the departure buffer time. Consider a scenario with two transfer modes: walking and rented e-scooters. If walking is disabled, then the only way to transfer between platforms modeled as different stops is to rent a scooter and incur the mode overhead. Especially in tightly scheduled networks (e.g., metropolitan areas), this means that a lot of crucial transfers become impossible. However, if the station is modeled as a combined stop, then the transfer is possible because no walking is required.

If this issue is not addressed, then the Pareto set will often contain unrealistic journeys that take detours to avoid multi-stop stations. This is especially problematic if we use ULTRA as a preprocessing technique, since these detour journeys produce many superfluous shortcuts that are merely artifacts of the way the network is modeled. To prevent this, we include an extra transfer mode consisting of *free transfers*. These include intra-station footpaths and other "unavoidable" transfers, e.g., connecting a train station to an adjacent bus stop. Free transfers cannot be disabled and their (negligible) transfer time is not penalized, in the same way that the departure buffer time is not counted towards the transfer time. This allows them to be used to dominate undesirable detour journeys even if all other modes are disabled.

Definitions. We denote the number of transfer modes as m. The different modes can be modeled either with m individual transfer graphs or with m edge cost functions applied to the same graph G. To simplify notation, we choose the latter option. Traveling along an edge $e = (v, w)$ in mode i requires the transfer time $\tau_{\text{tra}}(e, i)$. If an edge e cannot be traversed with mode i, we set $\tau_{\text{tra}}(e, i) = \infty$. The free transfers are represented by the edge set $F \subseteq \mathcal{S} \times \mathcal{S}$, separately from the transfer graph. These must be one-hop transfers, i.e., we require that the graph $(\mathcal{S}, F)$ is transitively closed and fulfills the triangle inequality. We also refer to the set of free transfers as mode 0 but do not count it as one of the m transfer modes. The transfer time for a free transfer $e \in F$ is denoted by $\tau_{\text{tra}}(e, 0)$.

We redefine the notion of transfers within a journey to account for free transfers: a transfer is either a path in the transfer graph that is traversed in some mode i or a free transfer that is traversed in mode 0. The time required for a transfer in mode $i \neq 0$ is the transfer time of the corresponding path in the transfer graph plus the mode overhead $\mu(i)$. For a free transfer, it is simply the transfer time of the corresponding edge in F. A journey J is evaluated with respect to $m + 2$ criteria: the arrival time $\tau_{\text{arr}}(J)$ at the target vertex, the number of used trips $|J|$, and for each transfer mode i, the sum over the transfer times of all transfers using mode i in J.

8.2 Adapting McULTRA

MCR can be easily adapted for the multimodal discomfort scenario, as we will show in Section 8.3. In order to obtain faster algorithms, we omit the costly Dijkstra searches by adapting McULTRA. So far, McULTRA only supports three criteria. A naive solution would be to extend this to an arbitrary number of criteria. However, this is not necessary due to mode independence. Consider a candidate J^c processed by McULTRA. By definition, J^c includes at most one transfer and therefore uses at most one transfer mode i. Witnesses with non-zero

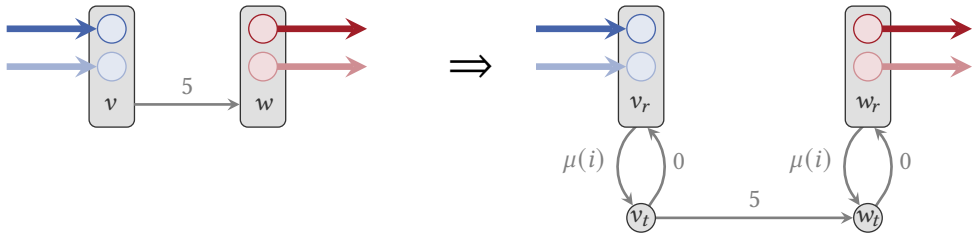


Figure 8.1: Construction of the virtual transfer graph for mode i.

transfer time in any mode besides 0 or i cannot dominate J^c. This means we can decompose the preprocessing into one three-criteria McULTRA shortcut computation per transfer mode, making the overall preprocessing effort linear in m. The McULTRA shortcut computation for mode i only considers transfers in modes 0 and i. It is guaranteed to find all relevant candidates, as well as all witnesses except those that use transfers of length zero in a mode other than 0 or i. It is reasonable to assume that if a transfer of length zero exists in any transfer mode, then a corresponding free transfer also exists in F. However, even if this is not the case, McULTRA will fail to find some witnesses and potentially generate superfluous shortcuts, but queries will remain correct.

To explore mode 0, McULTRA relaxes the outgoing free transfers of all updated stops before each Dijkstra search. Because F is already transitively closed, no stop-to-stop shortcuts need to be computed for it. Accordingly, journeys with free intermediate transfers are considered witnesses. Event-to-event shortcuts for F can be computed with two-criteria ULTRA using $(\mathcal{S}, F)$ as the transfer graph. Mode overheads are incorporated by constructing a virtual transfer graph, as shown in Figure 8.1. Each stop v is split into a route vertex v_r and a transfer vertex v_t. The two vertices are connected by directed edges $\overrightarrow{e_v} = (v_r, v_t)$ with $\tau_{\text{tra}}(\overrightarrow{e_v}, i) = \mu(i)$ and $\overleftarrow{e_v} = (v_t, v_r)$ with $\tau_{\text{tra}}(\overleftarrow{e_v}, i) = 0$. Each edge $(v, w) \in E$ in the original transfer graph is replaced with an edge (v_t, w_t) between the respective transfer vertices.

As discussed in Chapter 5.1.1, the correctness of (Mc)ULTRA relies on subjourney decomposition: every query can be answered with a Pareto-optimal journey J such that every candidate subjourney of J is also Pareto-optimal. As shown in Figure 8.2, this is no longer the case if mode changes within a transfer are prohibited. In this example, the candidate J^c is dominated by a witness J^w that begins with an initial transfer in some mode i. However, adding a transfer in another mode $j \neq i$ as a prefix induces a mode change in J^w, making it infeasible and leaving J^c as the only alternative. McULTRA will not generate a shortcut for the intermediate transfer of J^c and therefore fail to find journeys that include this transfer. However, note that this only affects journeys that would not be Pareto-optimal if mode changes were allowed. As discussed in Section 8.1, these are undesirable journeys that insert a trip detour to circumvent the forbidden mode change. Hence, although McULTRA in the multimodal discomfort scenario cannot guarantee to find all Pareto-optimal journeys, the missed journeys are known to be undesirable.

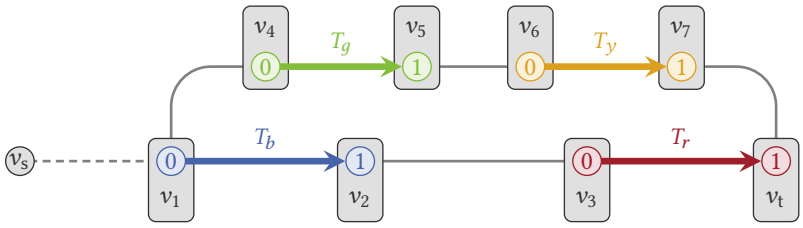


Figure 8.2: An example in which McULTRA misses an optimal shortcut in the multimodal discomfort scenario. Transfers of two different modes are drawn as dashed and solid lines, respectively. Assume that the candidate $J^c = \langle\langle v_1\rangle, T_b[0,1], \langle v_2, v_3\rangle, T_r[0,1], \langle v_t\rangle\rangle$ for the shortcut (v_2, v_3) is strongly dominated by the witness $J^w = \langle\langle v_1, v_4\rangle, T_g[0,1], \langle v_5, v_6\rangle, T_y[0,1], \langle v_7, v_t\rangle\rangle$. If we prepend the edge (v_s, v_1) to both journeys, then J^c becomes the only valid journey because J^w now includes a prohibited mode change at v_1.

8.3 RAPTOR-Based Query Algorithms

RAPTOR-based algorithms (MCR, ULTRA-McRAPTOR and UBM-RAPTOR) can be applied in the multimodal discomfort scenario with some minor changes. First, because mode changes within a transfer are prohibited, the pruning rules of McRAPTOR must be adjusted. Journeys that end with a trip may dominate journeys that end with a transfer, but not vice versa. This is because a transfer in mode i cannot be followed by a transfer in any mode other than i, whereas a trip can always be followed by a transfer. McRAPTOR can take this into account by maintaining two bags per stop v and round n: a *trip bag* $B^n_{\text{trip}}(v)$ for labels ending with a trip, and a *transfer bag* $B^n_{\text{trans}}(v)$ for labels ending in a transfer. Accordingly, the algorithm also maintains two best bags $B^*_{\text{trip}}(v)$ and $B^*_{\text{trans}}(v)$ per stop v. When a route scan in round n generates a new label ℓ at a stop v, it is compared with $B^*_{\text{trip}}(v)$ but not $B^*_{\text{trans}}(v)$. If ℓ is not dominated by $B^*_{\text{trip}}(v)$, it is merged into $B^n_{\text{trip}}(v)$. At the start of the transfer phase, $B^n_{\text{trip}}(v)$ is merged into $B^n_{\text{trans}}(v)$ for each updated stop v. This represents a direct transfer between trips at v without using a transfer mode. Finally, when a label at a stop v is generated in the transfer phase, it is compared with both $B^*_{\text{trip}}(v)$ and $B^*_{\text{trans}}(v)$. If it is not dominated by either bag, it is merged into $B^n_{\text{trans}}(v)$.

MCR additionally maintains a *Dijkstra bag* $B^i_{\text{D}}(v)$ for each transfer mode i and vertex v in order to conduct the Dijkstra searches. When a label is inserted into the trip bag $B^n_{\text{trip}}(v)$ of a stop v in round n, a copy of the label is also merged into the Dijkstra bag $B^i_{\text{D}}(v)$ for each mode i. Likewise, when a label is added to a Dijkstra bag $B^i_{\text{D}}(v)$ of a stop v during round n, a copy of it is merged into the transfer bag $B^n_{\text{trans}}(v)$.

Incorporating free transfers and mode overheads is straightforward. Because the set F of free transfers is transitively closed, no Dijkstra or Bucket-CH searches are required. It can be explored simply by relaxing the outgoing transfers of all updated stops in each round, as done

by McRAPTOR. In ULTRA-based queries, the shortcuts already include the overheads. For the initial and final transfers, they are added when evaluating the results of the Bucket-CH searches. In MCR, the overhead is added when labels are copied from the trip bags into the Dijkstra bags.

UBM-RAPTOR. Computing restricted Pareto sets requires adapting the pruning searches of UBM-RAPTOR to multiple transfer modes. Because they only optimize the arrival time and the number of trips, a transfer is always traversed using the fastest available mode. Due to overheads and limited availability of some modes, this is not necessarily the same mode for all transfers. The pruning searches therefore identify the fastest mode for each transfer individually. For initial and final transfers, this is done by merging the results of the Bucket-CH searches for each mode, choosing the minimum distance for each stop. For the intermediate transfers, the shortcut sets of all modes are merged, keeping the shortest shortcut in case of duplicates.

As explained in Chapter 7.2.2, the pruning searches of UBM-RAPTOR are allowed to relax multiple shortcuts in a row, at the cost of counting an additional trip for each shortcut. In a scenario with multiple competing transfer modes, this can cause the anchor set to include journeys with prohibited mode changes. In this case, journeys that exceed the slack compared to them are discarded, even if they are within slack for the actual anchor set without mode changes. This issue could be solved by maintaining individual earliest arrival times for each mode and ensuring that only shortcuts of the same mode can be used consecutively. However, we argue once again that the journeys that are discarded due to this phenomenon are undesirable, since they would not be part of the solution if mode changes were allowed. Therefore, we choose not to adjust the algorithm.

8.4 HydRA

In a scenario with one transfer mode and three criteria, event-to-event shortcuts enable the use of McTB, which is faster than RAPTOR. McTB avoids maintaining Pareto sets by representing the arrival time and the number of trips implicitly, which leaves transfer time as the only remaining criterion. In the multimodal discomfort scenario with an arbitrary number of criteria, this approach is no longer applicable. We therefore propose HydRA, a new algorithm that is based on McRAPTOR but incorporates some aspects of McTB. In particular, it uses event-to-event shortcuts to reduce the search space and performs simpler, more cache-efficient route scans. Because HydRA is intended for scenarios with four or more criteria, in which full Pareto sets are impractically large, we only present a variant that computes restricted Pareto sets, called UB-HydRA. High-level pseudocode is shown in Algorithm 8.1.

Initialization and Pruning Searches. Like every ULTRA-based algorithm, UB-HydRA starts by exploring the initial and final transfers in lines 1 and 2. For the free transfer mode,

Algorithm 8.1: UB-HydRA query algorithm.

Input: Public transit network $(\mathcal{S}, \Pi, \mathcal{E}, \mathcal{T}, \mathcal{R}, G)$,
source vertex v_s, departure time τ_{dep}, target vertex v_t
Output: Bags $B_{trip}(v_t, n)$ and $B_{trans}(v_t, n)$ at v_t for each number of trips n

1 **for each** mode i from 0 to m **do**
2 $\quad \tau_{tra}(v_s, \cdot, i), \tau_{tra}(\cdot, v_t, i) \leftarrow$ Explore initial and final transfers
3 $\overrightarrow{r}(\cdot,\cdot), \overrightarrow{\tau_{arr}}(\cdot,\cdot), k \leftarrow$ UBM-TB forward pruning search
4 $\overleftarrow{r}(\cdot,\cdot), \overleftarrow{\tau_{dep}}(\cdot,\cdot) \leftarrow$ UBM-TB backward pruning searches
5 **for each** $v \in \mathcal{S} \cup \{v_t\}$ **do**
6 $\quad$ **for each** n from 0 to k **do**
7 $\quad\quad B^n_{trip}(v) \leftarrow \emptyset$
8 $\quad\quad B^n_{trans}(v) \leftarrow \emptyset$
9 $\quad B^*_{trip}(v) \leftarrow \emptyset$
10 $\quad B^*_{trans}(v) \leftarrow \emptyset$
11 $\mathcal{S}_{trans} \leftarrow$ `RelaxInitialTransfers`()
12 $\mathcal{R}' \leftarrow$ `CollectRoutes`$(\mathcal{S}_{trans})$
13 $\mathcal{S}_{trip} \leftarrow$ `ScanInitialRoutes`$(\mathcal{R}')$
14 $\mathcal{S}_{trans} \leftarrow$ `RelaxTransfers`$(\mathcal{S}_{trip}, 1)$
15 **for** $n \leftarrow 2, 3, \ldots, k$ **do**
16 $\quad$ **if** $\mathcal{S}_{trans} = \emptyset$ **then break**
17 $\quad \mathcal{S}_{trip} \leftarrow$ `ScanRoutes`$(\mathcal{S}_{trans}, n)$
18 $\quad \mathcal{S}_{trans} \leftarrow$ `RelaxTransfers`$(\mathcal{S}_{trip}, n)$

this is done by relaxing the outgoing edges of the source vertex v_s and the incoming edges of the target vertex v_t, provided they coincide with stops. All other modes are explored with Bucket-CH searches. Afterward, the forward and backward pruning searches are performed in lines 3 and 4. Like UBM-TB (cf. Chapter 7.2.3), UB-HydRA uses two-criteria TB for these. For each trip T and round n, they compute a forward reached index $\overrightarrow{r}(T, n)$ and a backward reached index $\overleftarrow{r}(T, n)$. The forward reached index $\overrightarrow{r}(T, n)$ indicates the index of the first stop along T that is reachable from v_s with n trips. The backward reached index $\overleftarrow{r}(T, n)$ is the last stop index i along T with the following property: if $T[i]$ is reached with n trips, then v_t is reachable from there without exceeding the trip or arrival slack of an anchor journey. Additionally, for each stop v and round n, the forward search computes the earliest arrival time $\overrightarrow{\tau_{arr}}(v, n)$ among all journeys that arrive at v via a trip and use n trips. Unlike the UBM-TB backward search, the UB-HydRA backward search also computes an analogous latest departure time $\overleftarrow{\tau_{dep}}(v, n)$. The forward pruning search also computes the maximum number of rounds k that need to be performed by the main search.

Main Search. The overall scheme of the main search remains identical to McRAPTOR: it operates in rounds, each of which consists of a route scanning step followed by a transfer relaxation step. Each step yields a set of marked stops that are scanned in the next step; we denote these as $\mathcal{S}_{\text{trip}}$ for the route scanning phase and $\mathcal{S}_{\text{trans}}$ for the transfer relaxation phase. As in McRAPTOR, labels are maintained in a trip bag $B^n_{\text{trip}}(v)$ and a transfer bag $B^n_{\text{trans}}(v)$ for each vertex v and round n, as well as best bags $B^*_{\text{trip}}(v)$ and $B^*_{\text{trans}}(v)$ encompassing the already performed rounds. These are initialized in lines 5–10. A label $\ell = (\tau_{\text{arr}}, \tau_{\text{tra}}[\cdot], \varepsilon)$ consists of the arrival time τ_{arr}, an array $\tau_{\text{tra}}[\cdot]$ of length m that represents the transfer times for all modes, and a stop event ε. For labels in the trip bags, ε is the *exit event*, i.e., the stop event at which the last trip was exited. This information is used in the transfer relaxation phase: whereas McRAPTOR relaxes the outgoing stop-to-stop shortcuts of the corresponding stop $v(\varepsilon)$, HydRA only relaxes the outgoing event-to-event shortcuts of ε. This already yields the stop event at which the next trip will be entered, which is stored as the *entry event* in the corresponding transfer bag label. This allows the route scanning phase to skip the step of finding the earliest reachable trip for the label. Starting with the transfer relaxation phase of round 1, HydRA uses redesigned procedures that make use of the entry and exit events. For the initial transfers and the first route scanning phase, however, these events are not yet known, so the algorithm falls back on McRAPTOR-like procedures.

Pseudocode for the transfer relaxation procedures is shown in Algorithm 8.2. Two auxiliary procedures are used: `LinkTransfer` (lines 23–26) adds a transfer in mode i with transfer time $\tau^{\text{new}}_{\text{tra}}$ to an existing label ℓ, creating a new label ℓ'. This is done by adding $\tau^{\text{new}}_{\text{tra}}$ to the arrival time of ℓ and to the transfer time in mode i. Additionally, if the transfer represents an event-to-event shortcut $(\varepsilon_{\text{out}}, \varepsilon_{\text{in}})$, then the stop event of ℓ is replaced with the entry event ε_{in}. The procedure `MergeTransferLabel` (lines 27–33) merges a label ℓ into the transfer bag $B^n_{\text{trans}}(v)$ of a stop v for round n. As in UBM-RAPTOR, ℓ is discarded if its arrival time exceeds the latest departure time $\overleftarrow{\tau_{\text{dep}}}(v, n)$ computed by the backward pruning search. Otherwise, target pruning is performed by testing whether ℓ is dominated by the best trip and transfer bags of v_{t}. If not, it is merged into the best bag $B^*_{\text{trans}}(v)$ and finally $B^n_{\text{trans}}(v)$. The dominance rules for the merge operation are slightly adjusted: if two labels are equivalent in all criteria but have different exit/entry events, then both are kept. Finally, the stop is marked for the next round by adding it to the set $\mathcal{S}_{\text{trans}}$.

In round 0, transfers are relaxed with the `RelaxInitialTransfers` procedure (lines 1–11). For each transfer mode i and each stop v that is reachable with an initial transfer from v_{s} in mode i, a label ℓ' is created via `LinkTransfer` and merged into the transfer bag $B^0_{\text{trans}}(v)$ via `MergeTransferLabel`. Lines 9 and 10 do the same for a potential direct transfer to v_{t}. For later rounds $n > 0$, transfers are handled by the procedure `RelaxTransfers` (lines 12–22). For each transfer mode i, each updated stop $v \in \mathcal{S}_{\text{trip}}$ and each label in the previous trip bag $B^{n-1}_{\text{trip}}(v)$ with exit event ε_{out}, all outgoing shortcuts of ε_{out} in the set E^{s}_i of shortcuts for mode i are relaxed. For each shortcut $(\varepsilon_{\text{out}}, \varepsilon_{\text{in}})$, a new label ℓ is created that stores ε_{in} as its entry event and `MergeTransferLabel` is called to merge it into the transfer bag $B^n_{\text{trans}}(v(\varepsilon_{\text{in}}))$. A potential final transfer from v to v_{t} is handled in the same manner as in `RelaxInitialTransfers`.

Algorithm 8.2: HydRA transfer relaxation procedures.

Procedure `RelaxInitialTransfers()`
  $\mathcal{S}_{\text{trans}} \leftarrow \emptyset$
  $\ell = (\tau_{\text{dep}}, 0, \perp)$
  for each mode i from 0 to m **do**
    for each $v \in \mathcal{S}$ **do**
      if $\tau_{\text{tra}}(v_{\text{s}}, v, i) = \infty$ **then continue**
      $\ell' \leftarrow$ `LinkTransfer`$(\ell, \tau_{\text{tra}}(v_{\text{s}}, v, i), 0, \perp)$
      `MergeTransferLabel`$(v, \ell', 0, \mathcal{S}_{\text{trans}})$
    $\ell_{\text{t}} \leftarrow$ `LinkTransfer`$(\ell, \tau_{\text{tra}}(v_{\text{s}}, v_{\text{t}}, i), i, \perp)$
    `MergeTransferLabel`$(v, \ell', 0, \perp)$
  return $\mathcal{S}_{\text{trans}}$

Procedure `RelaxTransfers`$(\mathcal{S}_{trip}, n)$
  $\mathcal{S}_{\text{trans}} \leftarrow \emptyset$
  for each mode i from 0 to m **do**
    for each $v \in \mathcal{S}_{\text{trip}}$ **do**
      for each $\ell = (\tau_{\text{arr}}, \tau_{\text{tra}}[\cdot], \varepsilon_{\text{out}}) \in B^n_{\text{trip}}(v)$ **do**
        for each $e = (\varepsilon_{\text{out}}, \varepsilon_{\text{in}}) \in E^{\text{s}}_i$ **do**
          $\ell' \leftarrow$ `LinkTransfer`$(\ell, \tau_{\text{tra}}(e), i, \varepsilon_{\text{in}})$
          `MergeTransferLabel`$(v(\varepsilon_{\text{in}}), \ell', n, \mathcal{S}_{\text{trans}})$
        $\ell_{\text{t}} \leftarrow$ `LinkTransfer`$(\ell, \tau_{\text{tra}}(v, v_{\text{t}}, i), i, \perp)$
        `MergeTransferLabel`$(v_{\text{t}}, \ell_{\text{t}}, n, \perp)$
  return $\mathcal{S}_{\text{trans}}$

Procedure `LinkTransfer`$(\ell = (\tau_{arr}, \tau_{tra}[\cdot], \varepsilon_{out}), \tau^{new}_{tra}, i, \varepsilon_{in})$
  $\ell' \leftarrow (\tau_{\text{arr}} + \tau^{\text{new}}_{\text{tra}}, \tau_{\text{tra}}[\cdot], \varepsilon_{\text{in}})$
  $\tau_{\text{tra}}(\ell')[i] \leftarrow \tau_{\text{tra}}(\ell')[i] + \tau^{\text{new}}_{\text{tra}}$
  return ℓ'

Procedure `MergeTransferLabel`$(v, \ell = (\tau_{arr}, \tau_{tra}[\cdot], \varepsilon_{in}), n, \mathcal{S}_{trans})$
  if $\tau_{\text{arr}} > \overleftarrow{\tau_{\text{dep}}}(v, n)$ **then return**
  if $B^*_{\text{trip}}(v_{\text{t}})$`.dominates(`$\ell$`)` **then return**
  if $B^*_{\text{trans}}(v_{\text{t}})$`.dominates(`$\ell$`)` **then return**
  if not $B^*_{\text{trans}}(v)$`.merge(`ℓ`)` **then return**
  if not $B^n_{\text{trans}}(v)$`.merge(`ℓ`)` **then return**
  $\mathcal{S}_{\text{trans}} \leftarrow \mathcal{S}_{\text{trans}} \cup \{v\}$

Algorithm 8.3: HydRA route scanning procedures.

```
1  Procedure ScanInitialRoutes(R')
2      S_trip ← ∅
3      for each (R, j) ∈ R' do
4          B_route ← ∅
5          for i from j to |R| − 1 do
6              v ← i-th stop of R
7              for each ℓ = (τ_arr, τ_tra[·], T) ∈ B_route do
8                  if i − 1 > r←(T, 0) then continue
9                  ℓ' ← (τ_arr, τ_tra[·], T[i])
10                 MergeTripLabel(v, ℓ', 1, S_trip)
11             for each ℓ = (τ_arr, τ_tra[·], ⊥) ∈ B^0_trans(v) do
12                 T ← FindEarliestTrip(R, i, τ_arr)
13                 if T = ⊥ then continue
14                 if i > r←(T, 0) then continue
15                 ℓ' ← (τ_arr, τ_tra[·], T)
16                 B_route.merge(ℓ')
17     return S_trip
18 Procedure ScanRoutes(S_trans, n)
19     S_trip ← ∅
20     for each v ∈ S_trans do
21         for each ℓ = (τ_arr, τ_tra[·], T[i]) ∈ B^{n−1}_trans(v) do
22             for k from i + 1 to r←(T, n) + 1 do
23                 ℓ' ← (τ_arr(T[k]), τ_tra[·], T[k])
24                 MergeTripLabel(v(T[k]), ℓ', n, S_trip)
25     return S_trip
26 Procedure MergeTripLabel(v, ℓ = (τ_arr, τ_tra[·], T[i]), n, S_trip)
27     if B*_trip(v_t).dominates(ℓ) then return
28     if B*_trans(v_t).dominates(ℓ) then return
29     if not B*_trip(v).merge(ℓ) then return
30     if not B^n_trip(v).merge(ℓ) then return
31     S_trip ← S_trip ∪ {v}
```

The route scanning procedures are depicted in Algorithm 8.3. The `MergeTripLabel` procedure (lines 26–31) is analogous to `MergeTransferLabel` but omits the check for the backward pruning search. For the first round, the procedure `ScanInitialRoutes` (lines 1–17) closely resembles McRAPTOR. Given as input are the routes $\mathcal{R}'$ that are visited by at least one updated

stop, which were previously collected in `CollectRoutes`. Each route R is scanned by iterating over its stop sequence from the first marked index j onwards, maintaining a route bag B_{route}. At each index i, two steps are performed for the visited stop v: merging B_{route} into the trip bag $B^1_{\text{trip}}(v)$ (lines 7–10) and merging the transfer bag $B^0_{\text{trans}}(v)$ from the previous round into B_{route} (lines 11–16). For each label $\ell \in B_{\text{route}}$ with active trip T, the former step creates a corresponding label ℓ' with exit event $T[i]$. If $i-1$ exceeds the backward reached index $\overleftarrow{r}(T, 0)$ holds, then T cannot be entered at any of the preceding stops without exceeding the slack values, so ℓ' is discarded. Otherwise, `MergeTripLabel` is called to merge it into $B^1_{\text{trip}}(v)$. For each label ℓ in the transfer bag $B^0_{\text{trans}}(v)$, the second step adds a corresponding label ℓ' to the route bag. The active trip of ℓ' is set to the earliest reachable trip T of R, which is found with a binary search via `FindEarliestTrip`. If $i > \overleftarrow{r}(T, 0)$ holds, the label is discarded by the same argument as above. Otherwise, it is merged into the route bag B_{route}.

For the following rounds $n > 1$, the procedure `ScanRoutes` (lines 18–25) makes use of the entry events. For each updated stop v and each label in the previous transfer bag $B^{n-1}_{\text{trans}}(v)$ with entry event $T[i]$, the relevant segment of T is identified and scanned. The last stop index at which T can be entered without exceeding the slack is $\overleftarrow{r}(T, n)$. Accordingly, T can be exited at all indices k with $i + 1 \leq k \leq \overleftarrow{r} + 1$. For each such index k, a new label with exit event $T[k]$ is created and `MergeTripLabel` is called to merge it into the trip bag of $v(T[k])$.

8.5 Experiments

We evaluate our algorithms in a scenario with two competing transfer modes: walking and e-scooter. We use the same OSM transfer graph for both modes, assuming a constant speed of 4.5 km/h for walking and 15 km/h for scooter. The mode overheads are set to 0 s for walking and 300 s for scooter. We limit our experiments to the networks of London, Switzerland and Stuttgart because MCR is too slow on the Germany network (approximately 100 s per query) to evaluate 10 000 queries within a reasonable timeframe. To create the free transfer graphs, we connect all pairs of stops within a geographical distance of up to 100 meters and then build the transitive closure. This yields 35 308 edges for London, 19 128 for Switzerland, and 11 354 for Stuttgart. All shortcut computations were run on the Epyc machine, all other experiments on the Xeon machine. We evaluate the performance of the shortcut computation in Section 8.5.1. Afterward, we compare the solutions computed by the different algorithms in Section 8.5.2. Finally, we evaluate the query performance in Section 8.5.3.

8.5.1 Preprocessing

Table 8.1 reports the performance of the McULTRA shortcut computation, using the same settings as in Chapter 7.3.2. Because many short transfers are now covered by the free transfer mode, the number of shortcuts per mode is slightly lower than in the bimodal setting (cf. Table 7.1). For comparison, we also ran the shortcut computation for the scooter mode without free transfers; depending on the network, this increased the number of shortcuts

Table 8.1: Multimodal McULTRA shortcut computation results. Times are formatted as h:mm:ss. Listed are the numbers of original shortcuts $|E^s|$, augmented forward shortcuts $|\overrightarrow{E^s_{aug}}|$ and augmented backward shortcuts $|\overleftarrow{E^s_{aug}}|$.

Network	Variant	Mode	Time	Shortcuts		
				$\lvert E^s\rvert$	$\lvert\overrightarrow{E^s_{aug}}\rvert$	$\lvert\overleftarrow{E^s_{aug}}\rvert$
London	Stop	Walking	0:31:23	112 710	–	–
		Scooter	2:56:11	8 439 533	–	–
	Event	Free	0:04:39	12 920 623	21 725 937	24 219 405
		Walking	0:38:26	11 201 513	29 725 151	33 192 796
		Scooter	3:15:54	132 281 208	456 663 147	491 434 923
Switzerland	Stop	Walking	0:11:54	203 691	–	–
		Scooter	0:25:08	973 810	–	–
	Event	Free	0:00:50	7 896 506	17 917 277	18 776 547
		Walking	0:12:35	10 565 545	31 610 293	31 060 835
		Scooter	0:27:48	22 108 243	92 291 000	91 230 189
Stuttgart	Stop	Walking	0:08:18	112 406	–	–
		Scooter	0:13:59	728 042	–	–
	Event	Free	0:00:23	3 547 427	6 182 903	6 592 774
		Walking	0:07:29	4 086 469	9 520 628	9 596 946
		Scooter	0:15:28	9 460 813	35 238 616	35 247 514

Table 8.2: Impact of transfer time discretization on the McULTRA shortcut computation for e-scooters on the London network. Times are formatted as h:mm:ss.

Variant	Disc.	Time	Shortcuts		
			$\lvert E^s\rvert$	$\lvert\overrightarrow{E^s_{aug}}\rvert$	$\lvert\overleftarrow{E^s_{aug}}\rvert$
Stop	○	2:56:11	8 439 533	–	–
Stop	●	0:56:04	4 055 105	–	–
Event	○	3:15:54	132 281 208	456 663 147	491 434 923
Event	●	1:02:20	50 517 017	255 781 401	253 857 743

by a factor between 1.5 and 1.8. The preprocessing times are higher than in the bimodal case because exploring the free transfers takes additional time and incorporating the mode overheads increases the size of the network. As reported in Chapter 7.3.2, the shortcut augmentation time is negligible compared with the shortcut computation time.

For e-scooters on the London network, the preprocessing time and the number of shortcuts are extremely high due to the immense number of Pareto-optimal labels in each vertex bag. To reduce the bag sizes, we implemented a heuristic version of McULTRA that discretizes the transfer time criterion into *buckets* for the purpose of testing dominance. Let τ_{tra} be the transfer time of a label. Instead of using τ_{tra} as the criterion for testing dominance, we use $\lfloor \frac{\tau_{\text{tra}}}{x} \rfloor$, where x is the bucket size. For our experiments, we set $x = 300\,\text{s}$. As shown in Table 8.2, discretization reduces the preprocessing time by a factor of three and the number of shortcuts by a factor of two to three.

8.5.2 Result Coverage

As discussed in Section 8.2, McULTRA-based algorithms fail to find some undesirable journeys that are Pareto-optimal but dominated by journeys with prohibited mode changes. To quantify this effect, we evaluate the degree to which the exact Pareto set is covered by our algorithms. We consider two coverage metrics: exact coverage is the percentage of journeys in the (full or restricted) Pareto set that are found by the algorithm, whereas fuzzy coverage also accounts for similarity between journeys. If the algorithm does not find a journey J but another journey J' that is almost as good or better in all criteria, we consider J well covered. Like Delling et al. [Del+13], we measure similarity using fuzzy logic. Given two parameters $\chi \in (0, 1)$ and $\epsilon > 0$, the fuzzy coverage of a journey J by another journey J' for a criterion c is

$$\operatorname{cov}(J, J') := \begin{cases} \exp\left(\frac{\ln(\chi)}{\epsilon^2} \left(c(J) - c(J')\right)^2\right) & \text{if } c(J) < c(J'), \\ 1 & \text{else.} \end{cases}$$

The overall fuzzy coverage $\operatorname{cov}(J_1, J_2)$ is the minimum coverage across all criteria. The fuzzy coverage of a journey J by a set of journeys $\mathcal{J}$ is the maximum across $\mathcal{J}$, i.e., $\operatorname{cov}(J, \mathcal{J}) := \max_{J' \in \mathcal{J}} \operatorname{cov}(J, J')$. Finally, the fuzzy coverage $\operatorname{cov}(\mathcal{J}, \mathcal{J}')$ of a set of journeys $\mathcal{J}$ by another set of journeys $\mathcal{J}'$ is the mean coverage by $\mathcal{J}'$ across all journeys in $\mathcal{J}$. Following [Del+13], the fuzzy parameters (χ, ϵ) are set to $(0.8, 60\,\text{s})$ for the arrival time, $(0.1, 1)$ for the number of trips, and $(0.8, 300\,\text{s})$ for the transfer time.

Exact and fuzzy coverages are reported in Table 8.3. We observe that the vast majority of journeys in the full Pareto set is found by ULTRA-McRAPTOR. The mean percentage of missed journeys ranges from 0.54% on London to 6.08% on Stuttgart. Furthermore, a mean fuzzy coverage of 99% and above shows that the missed journeys are well covered by others. The Stuttgart network, which has a particularly large transfer graph and therefore more opportunities for mode changes, exhibits the lowest coverage overall. The coverage values for UBM-RAPTOR and UB-HydRA are significantly higher, which indicates that the missed journeys tend to fall outside the restricted Pareto set. In particular, the fuzzy coverage is near-perfect on all networks, even in the 5th percentile.

In most cases, UB-HydRA exhibits a slightly lower coverage than UBM-RAPTOR. This is because the event-to-event shortcuts are more fine-grained and therefore more prone to missing journeys. In stop-to-stop ULTRA, even if a candidate is missed, its shortcut is often

Table 8.3: Coverage of the exact Pareto sets by our algorithms for 10 000 random queries. ULTRA-McRAPTOR is compared with the full Pareto set, all others with the restricted Pareto set for slack values $\sigma_{\text{arr}} = \sigma_{\text{tr}} = 1.25$. For each metric, we report the mean coverage across all queries and the coverage of the 5th percentile. *Disc.* indicates whether shortcut discretization is enabled (•) or not (○).

Network	Algorithm	Disc.	Exact coverage [%]		Fuzzy coverage [%]	
			5th perc.	Mean	5th perc.	Mean
London	ULTRA-McRAPTOR	○	96.95	99.46	99.99	99.97
	ULTRA-McRAPTOR	•	91.55	98.16	99.94	99.97
	BM-RAPTOR	○	61.53	92.70	91.80	98.57
	UBM-RAPTOR	○	100.00	99.84	100.00	99.99
	UBM-RAPTOR	•	91.91	98.92	99.97	99.98
	UB-HydRA	○	96.87	99.43	99.99	99.96
	UB-HydRA	•	83.67	97.27	99.76	99.91
Switzerland	ULTRA-McRAPTOR	○	87.73	97.90	98.99	99.67
	BM-RAPTOR	○	55.55	92.51	80.73	97.10
	UBM-RAPTOR	○	94.59	99.12	99.98	99.83
	UB-HydRA	○	93.33	99.07	99.96	99.92
Stuttgart	ULTRA-McRAPTOR	○	81.36	95.92	93.79	98.95
	BM-RAPTOR	○	52.38	91.42	79.26	96.94
	UBM-RAPTOR	○	98.03	99.33	99.99	99.92
	UB-HydRA	○	94.11	96.10	99.97	99.92

represented by another candidate that is found. This is less likely to occur in the event-to-event variant. A notable exception is the mean fuzzy coverage on Switzerland, which is higher for UB-HydRA. This is due to the phenomenon discussed in Section 8.3. Because the pruning search of UBM-RAPTOR is allowed to relax multiple shortcuts in a row, it may find anchor journeys that include mode changes, which is not the case for the TB-based pruning search used by UB-HydRA. Altogether, these results justify our choice of prohibiting mode changes in order to reduce the number of irrelevant solutions. Although doing so introduces some new, undesirable journeys, they are often discarded by McULTRA and our experiments show that they are rare and similar to other, more relevant solutions.

Another possibility to reduce the Pareto set would be to use the transitively closed transfer graphs presented in Chapter 4.3 to explore the intermediate transfers. This would remove the need for ULTRA as a preprocessing step. To demonstrate that this negatively impacts the solution quality, we evaluate the coverage of BM-RAPTOR using transitive intermediate transfers but unlimited initial and final transfers. The exact coverage is still above 90% because most optimal journeys do not include long intermediate transfers. However, the low coverage

Table 8.4: Query performance for full Pareto sets, averaged over 10 000 random queries. *Rnd.* is the number of performed rounds, whereas *Jrn.* refers to the number of computed journeys.

Network	Algorithm	Rnd.	Jrn.	Time [ms]				
				Routes	Transfers			Total
					Free	Walking	Scooter	
London	MCR	10.6	129.4	272.1	82.9	1 455.1	2 226.9	4 064.1
	ULTRA-McRAPTOR	10.6	128.8	217.6	94.3	118.6	2 666.2	3 115.8
Switzerland	MCR	12.3	86.8	403.0	41.7	1 613.9	1 598.4	3 700.0
	ULTRA-McRAPTOR	12.3	84.2	308.1	39.6	113.2	361.2	847.9
Stuttgart	MCR	11.6	158.9	828.9	94.8	5 059.2	5 497.3	11 520.8
	ULTRA-McRAPTOR	11.6	151.4	641.1	93.4	332.4	1 366.2	2 451.0

in the 5th percentile shows that a significant portion of optimal journeys are missed and that many of these are not well covered by other solutions with limited transfers.

On the London network, we observed that discretizing transfer time when testing dominance significantly reduces the preprocessing time and the number of shortcuts. As expected, this noticeably reduces the exact coverage, although it remains much higher than with transitive intermediate transfers. On the other hand, the fuzzy coverage is hardly affected because missing shortcuts caused by discretization are guaranteed to have similar alternatives. This shows that discretization is a useful tool for limiting the preprocessing effort while maintaining a high solution quality.

8.5.3 Query Performance

Table 8.4 reports the running times of the algorithms that compute full Pareto sets. On Switzerland and Stuttgart, ULTRA-McRAPTOR achieves a speedup of four to five over MCR. Because the performance gain of ULTRA comes from speeding up the transfer phases, this is higher than the speedup of two observed in the bimodal scenario, in which the transfer phase takes up a smaller share of the running time. In the scooter mode, the performance gains are limited due to the high number of stop-to-stop shortcuts. For London, relaxing scooter shortcuts is in fact slower than a Dijkstra search, causing the overall speedup to be marginal. Regardless of the algorithm, we observe that the running times are far from practical due to the extremely high number of Pareto-optimal journeys.

We therefore investigate the performance of the algorithms for restricted Pareto sets, which is shown in Table 8.5. The number of computed journeys is reduced to less than 25, which is manageable for the algorithm but still more than what can reasonably be shown to users. The speedup of UBM-RAPTOR over MCR ranges from 35 on London to 225 on Stuttgart. HydRA speeds up the main search by a factor between 2.5 and 4 due to its more efficient route scans.

Table 8.5: Query performance for restricted Pareto sets with slack values $\sigma_{\text{arr}} = \sigma_{\text{tr}} = 1.25$, averaged over 10 000 random queries. *Rnd.* is the number of performed rounds, whereas *Jrn.* refers to the number of computed journeys. *Disc.* refers to shortcut discretization.

Network	Algorithm	Disc.	Rnd.	Jrn.	Time [ms]			
					Forward	Backward	Main	Total
London	UBM-RAPTOR	○	2.1	24.2	61.3	11.5	42.0	114.8
	UBM-RAPTOR	●	2.1	24.2	35.1	7.0	26.2	68.3
	UB-HydRA	○	2.1	24.1	18.4	8.4	9.9	36.7
	UB-HydRA	●	2.1	23.8	13.6	5.3	8.0	26.9
Switzerland	UBM-RAPTOR	○	3.5	21.9	32.2	4.4	19.1	55.8
	UB-HydRA	○	3.5	21.7	15.1	1.9	6.9	23.8
Stuttgart	UBM-RAPTOR	○	2.7	15.1	24.2	5.9	21.0	51.2
	UB-HydRA	○	2.7	15.0	10.2	2.4	8.2	20.8

Additionally, by using the more fine-grained event-to-event shortcuts, the search space is significantly reduced. Switching from RAPTOR to TB also speeds up the pruning search by a factor of two to three. On London, the performance of both algorithms is still hampered by the high number of shortcuts. Shortcut discretization mitigates this at a slight loss in solution quality, improving the query times by 41% for UBM-RAPTOR and 27% for UB-HydRA. Overall, the speedup of UB-HydRA over MCR ranges from around 150 for London and Switzerland to over 500 for Stuttgart. With query times of 20–30 ms, the performance is good enough for interactive applications and only slightly higher than for the bimodal, three-criteria setting studied in Chapter 7 (cf. Table 7.3).

8.6 Conclusion

We extended our previous results to fully multimodal networks with an arbitrary number of competing transfer modes. To ensure reasonable solutions, we established the multimodal discomfort scenario, which optimizes one discomfort criterion per transfer mode and prohibits mode changes within a transfer. We showed that McULTRA can be adapted to this scenario in a scalable fashion by preprocessing each transfer mode independently. Besides adapting existing query algorithms, we proposed HydRA, which carries over some of the advantages of TB into a setting with an arbitrary number of criteria. Our experimental evaluation shows that our algorithms achieve query times that are fast enough for interactive applications. Future work could involve incorporating more complex transfer modes such as bike-sharing, which require additional modeling [SWZ20b]. Furthermore, HydRA is a promising approach for journey planning problems that consider other criteria, such as fare or vehicle occupancy.

9 Delay-Robustness

So far, ULTRA assumes that all trips run according to schedule. However, the set of required shortcuts may change in the presence of delays: A candidate may become infeasible if its first trip is delayed so much that the second trip can no longer be entered. On the other hand, if the second trip is delayed too much, the candidate may no longer be optimal. Furthermore, delays can allow passengers to catch trips that usually depart too early. This creates new witnesses that may dominate previously optimal candidates. For these reasons, the shortcuts computed by ULTRA may no longer be sufficient if trips are delayed.

Delays are an everyday occurrence in real-world public transit networks. In order to be usable in practice, journey planning algorithms must be able to account for them. In this chapter, we therefore extend ULTRA to handle delays. As we will see, this is a highly challenging problem, so we return to the simple problem setting of Chapter 5, with one transfer mode and two criteria.

Related Work. We distinguish between two types of delay-robust algorithms: *Delay-anticipating* algorithms aim to find journeys that are robust regarding potential future delays that are not known at query time. By contrast, *delay-responsive* algorithms aim to find journeys that are optimal in the currently known delay scenario. These algorithms receive a continuous stream of *delay updates*, which they need to periodically incorporate into their query data structures in an *update phase*. This phase needs to be fast (ideally no more than a few seconds) to ensure that the query algorithm runs on reasonably up-to-date information.

In this chapter, we focus on delay-responsive algorithms. Nevertheless, we give a brief overview of research on delay-anticipating algorithms. A natural approach is to assume a stochastic delay distribution and search for journeys that are likely to be feasible under this distribution. Disser et al. [DMS08] and Delling et al. [DPW15a] compute a reliability value

for each journey based on this distribution and use it as a criterion for Pareto optimization. MEAT (Minimum Expected Arrival Time) [DPSW18] extends CSA to compute backup journeys in case the optimal journey becomes infeasible. An alternative to using a stochastic delay distribution is to rely on past observations. Böhmová et al. [Böh+13] find journeys that are reliable in the sense that they were feasible in many delay scenarios recorded in the past.

Goerigk et al. [Goe+14] study formal definitions of robustness. They only consider delay scenarios that do not exceed a specified maximum delay. A journey is *strictly robust* if every intermediate transfer is feasible in every such delay scenario. Finding strictly robust journeys is an NP-complete problem, and the found journeys often have unacceptably high travel times. To solve this problem, the authors bound the travel time compared to the fastest journey in the undelayed scenario. Among journeys within this bound, the one that minimizes the number of intermediate transfers that are infeasible in at least one possible delay scenario is called *lightly robust*. An alternative is to consider *recoverable robustness* [Goe+13]. In a recoverably robust journey, an intermediate transfer does not need to be feasible in every possible delay scenario. Instead, for all possible delays that can make the transfer infeasible, there must be a *recovery path* that allows passengers to still reach their target in time.

By comparison, delay-responsive algorithms have been studied more extensively. Both types of graph-based models for public transit networks can be made delay-responsive. In time-dependent graphs, incorporating a delay merely requires updating the travel time functions [DGWZ08], whereas time-expanded graphs require changes to the graph topology [DGWZ08, MS09]. The dynamic time-expanded model [Cio+17] is specifically engineered to allow for faster delay updates. Queries can be answered with a variant of Dijkstra's algorithm that incorporates node blocking and A* on the lower-bound graph, similar to the approach of Delling et al. [DPW09b]. Among the timetable-based algorithms, the data structures of CSA and RAPTOR are lightweight enough that they can be rebuilt from scratch during the update phase.

Speedup techniques are more challenging to make delay-responsive because their preprocessing steps are usually too expensive to be rerun in the update phase. An exception to this is TB: When a trip T receives a delay update, it is sufficient to rerun the transfer generation step for T and all trips that are connected to T via a footpath. This only takes a few milliseconds [Wit21]. Bast et al. [BSS13] show that Transfer Patterns answer almost all queries correctly even if the precomputed DAGs are not adjusted to incorporate delays. However, the queries selected in their experiments are biased towards large stations and rush hours; it is unclear whether the error rate is also low for queries selected uniformly at random. For Public Transit Labeling, D'Emidio and Khan [DK19] propose a dynamic version with update times of a few seconds on metropolitan networks. However, their approach only optimizes the arrival time.

Common to all these techniques is that they require a transitively closed transfer graph. In particular, they assume that the set of potential transfers that are incident to a trip is small and local, which makes it easy to enumerate when handling a delay update. This is no longer the case with an unrestricted transfer graph. For multimodal networks, the fastest delay-responsive algorithm so far is MR. As with RAPTOR, updates can be incorporated by

simply rebuilding the query data structures. The dynamic time-expanded model has been extended to multimodal networks [GPZ19], but query times are not competitive with MR.

Chapter Outline. In this chapter, we present Delay-ULTRA, a variant of ULTRA that anticipates possible delays during the shortcut computation phase. Doing this for arbitrarily high delays poses a conceptual challenge: Nearly every journey is optimal in at least one theoretically possible delay scenario (e.g., one that applies extreme delays to all competitors). Thus, almost every possible shortcut is required in theory. In realistic scenarios, however, most vehicles are delayed by no more than a few minutes and only a few are affected by higher delays. Delay-ULTRA exploits this with a three-phase approach: The first preprocessing phase, which does not receive any delay information, computes a set of shortcuts that is provably sufficient for all delays below a specified threshold (e.g., five minutes). During the update phase, when delays are known, irrelevant shortcuts are discarded and missing ones for higher delays are computed in a second step. Finding all required shortcuts is not feasible within the few seconds allowed by the update phase, since this would require enumerating all potential incident shortcuts of the delayed trips. Instead, we propose a heuristic *replacement search* for shortcuts that have become infeasible due to delays. Finally, the query phase uses the filtered set of precomputed shortcuts as well as the replacement shortcuts.

Unlike in previous chapters, we only propose an event-to-event variant of Delay-ULTRA, for combination with TB. We refrain from evaluating a stop-to-stop variant (e.g., for combining Delay-ULTRA with RAPTOR). Recall that the combination of ULTRA with TB yields the fastest query algorithm, whereas ULTRA-RAPTOR is only two to three times faster than MR on most networks. When incorporating delays, we expect the number of required stop-to-stop shortcuts to grow significantly. Thus, we do not expect a RAPTOR-based algorithm to be much faster than MR, whereas a TB-based algorithm could still offer a significant speedup.

The remainder of this chapter is organized as follows. Section 9.1 introduces the necessary notation and definitions. In Section 9.2, we establish a characterization of the shortcuts that are required for delays up to a certain limit. While this characterization is compact, it does not immediately imply an efficient algorithm for enumerating all such shortcuts. Section 9.3 therefore develops formulas for computing them efficiently. Based on these results, Section 9.4 outlines the Delay-ULTRA shortcut computation algorithm. Section 9.5 describes the update phase, including the replacement search. In Section 9.6, we evaluate our algorithms on the four benchmark networks, using a synthetic delay model based on real-world punctuality data. Our experiments show that the original ULTRA-TB is already fairly delay-robust, failing to find at most 1% of optimal journeys. Delay-ULTRA reduces the error rate by a factor of 4–15, yielding less than 0.02% suboptimal journeys for metropolitan and mid-sized country networks, and 0.16% for the much larger Germany network. If we take into account that the delay information is never perfectly accurate or up to date in realistic applications, these error rates are negligible. Our query algorithm, Delay-ULTRA-TB, retains a speedup of up to eight compared to MR, and is at most two times slower than ULTRA-TB. Finally, Section 9.7 summarizes our results and gives an outlook on future work.

9.1 Definitions

The addition of delays requires us to introduce some new notation and adapt some of the definitions from Chapter 2. We assume throughout this chapter that all transfers are shortest paths in the transfer graph. This allows us to disregard irrelevant corner cases, since transfers that are not shortest paths are never required to construct optimal journeys, even in the presence of delays. For a stop event $T[i]$, we denote the sets of stop events preceding and succeeding $T[i]$ in its trip T by $\overleftarrow{\mathcal{E}}(T[i])$ and $\overrightarrow{\mathcal{E}}(T[i])$, respectively.

Delays. A *delay scenario* Δ assigns to every stop event $\varepsilon \in \mathcal{E}$ an *arrival delay* $\Delta_{\text{arr}}(\varepsilon) \in \mathbb{N}_0$ and a *departure delay* $\Delta_{\text{dep}}(\varepsilon) \in \mathbb{N}_0$. This yields delayed arrival and departure times $\tau_{\text{arr}}(\Delta, \varepsilon) = \tau_{\text{arr}}(\varepsilon) + \Delta_{\text{arr}}(\varepsilon)$ and $\tau_{\text{dep}}(\Delta, \varepsilon) = \tau_{\text{dep}}(\varepsilon) + \Delta_{\text{dep}}(\varepsilon)$. Note that this definition considers the delays of all stop events independently of each other. This allows impossible delay scenarios in which vehicles travel faster than is possible, or even backward in time. We permit these to avoid introducing dependencies between stop events (cf. Section 9.3.4).

Given a journey $J = \langle P_0, T_0[i, j], \ldots, T_{k-1}[m, n], P_k \rangle$ and a delay scenario Δ, the latest possible departure time of J at the source vertex is $\tau_{\text{dep}}(\Delta, J) := \tau_{\text{dep}}(\Delta, T_0[i]) - \tau_{\text{tra}}(P_0)$ and the arrival time at the target vertex is $\tau_{\text{arr}}(\Delta, J) := \tau_{\text{arr}}(\Delta, T_{k-1}[n]) + \tau_{\text{tra}}(P_k)$. We redefine the notion of feasibility to take delays into account. We call J feasible for a departure time τ_{dep} if $\tau_{\text{dep}}(\Delta, J) \geq \tau_{\text{dep}}$ and all of its intermediate transfers are feasible. An intermediate transfer between stop events ε_{o} and ε_{d} is feasible if $\tau_{\text{dep}}(\Delta, \varepsilon_{\text{d}}) \geq \tau_{\text{arr}}(\Delta, \varepsilon_{\text{o}}) + \tau_{\text{tra}}(v(\varepsilon_{\text{o}}), v(\varepsilon_{\text{d}}))$.

Problem Statement. A *delay update* μ represents changes in the delays of a single trip T, starting at some index i. For each stop event $T[j]$ with $j \geq i$, the update μ specifies a new departure and arrival delay. A delay-responsive algorithm receives a stream $\langle \mu_1, \mu_2, \ldots \rangle$ of delay updates and a stream $\langle q_0, q_1, \ldots \rangle$ of *queries*. In the initial delay scenario Δ_0, all stop events are punctual. After receiving update μ_i, its new delays are applied to the previous scenario Δ_{i-1} to obtain the current scenario Δ_i. A query q consists of source and target vertices $v_{\text{s}}, v_{\text{t}} \in V$, an earliest departure time τ_{dep} and an *execution time* $\tau_{\text{ex}} \leq \tau_{\text{dep}}$. It must be answered with a Pareto set of v_{s}-v_{t}-journeys departing no later than τ_{dep} in the delay scenario Δ that is current at the execution time τ_{ex}.

Partial Journeys. To keep the number of shortcuts as low as possible, ULTRA only enumerates canonical journeys. The definition of canonical journeys (cf. Chapter 5.1.1) breaks all ties between equivalent journeys. This ensures that there is only one canonical Pareto set for each query. However, this definition is fairly unwieldy and complicated to use. For Delay-ULTRA, we switch to a less strict definition that still allows for some ties while breaking others. We are only interested in enumerating journeys for which all prefixes are Pareto-optimal. Additionally, we want to preserve the pruning rule of MR that ensures that trips are entered at the earliest possible stop event.

To formalize this rule, we introduce partial journeys, which start or end midway through a trip segment, and redefine the notion of prefixes to include them. A *trip segment prefix* $T[i, \cdot]$ represents a passenger entering the trip T at $T[i]$ but not yet exiting it, whereas a *trip segment suffix* $T[\cdot, j]$ represents a passenger entering T at an unspecified stop event and exiting at $T[j]$. A *journey prefix* is either a journey or a journey followed by a trip segment prefix. Likewise, a *journey suffix* is either a journey or a journey preceded by a trip segment suffix. In both cases, the incomplete trip segment counts as a used trip. Journey prefixes and suffixes are collectively called *partial journeys*. Two (partial) journeys J_1 and J_2 can be concatenated into a (partial) journey $J_1 \circ J_2$ if one of two conditions is fulfilled:

1. J_1 ends with a final transfer P_1 to a vertex v and J_2 starts with an initial transfer P_2 from the same vertex v. Then $J_1 \circ J_2$ is obtained by replacing P_1 and P_2 with the intermediate transfer $P_1 \circ P_2$.

2. J_1 ends with a trip segment prefix $T[i, \cdot]$, J_2 starts with a trip segment suffix $T[\cdot, j]$, and $i < j$. Then $J_1 \circ J_2$ contains the proper trip segment $T[i, j]$ in their place.

A journey $J = \langle P_0, \ldots, T_k[i, j], P_{k+1}, \ldots \rangle$ has two *standard prefixes* with k trips: the journey prefix $\langle P_0, \ldots, T_k[i, \cdot] \rangle$ and the proper journey $\langle P_0, \ldots, T_k[i, j], P_{k+1} \rangle$. Additionally, each journey $\langle P_0, \ldots, T_k[i, j], P'_{k+1} \rangle$ where P'_{k+1} is a prefix of P_{k+1} is a *non-standard prefix* of J. This is equivalent to the definition of journey prefixes in Chapter 2, except that it also includes partial journeys.

We extend the notion of dominance to journey prefixes J_p and J'_p: If J_p and J'_p are both proper journeys, i.e., end with final transfers, the original definition applies. If J_p ends with a trip segment prefix $T[i, \cdot]$ and J'_p with a trip segment prefix $T[j, \cdot]$ of the same trip T, then J'_p weakly dominates J_p if $|J'_\mathrm{p}| \leq |J_\mathrm{p}|$ and $j \leq i$. If $|J'_\mathrm{p}| < |J_\mathrm{p}|$ or $j < i$ also holds, then J'_p strongly dominates J_p. In all other cases, J'_p does not strongly or weakly dominate J_p. The definition of Pareto optimality carries over from proper journeys. We call a journey *prefix-optimal* if all of its prefixes are Pareto-optimal. Note that if a standard prefix that ends with a final transfer P is Pareto-optimal, then so are all non-standard prefixes ending with a prefix P' of P. Thus, a journey is prefix-optimal iff all of its standard prefixes are Pareto-optimal.

Candidate Notation. A candidate is a journey of the form

$$J^\mathrm{c} = \left\langle \langle v_\mathrm{s} \rangle, T_1[\mathrm{s}, \mathrm{o}], \langle v_\mathrm{o}, \ldots, v_\mathrm{d} \rangle, T_2[\mathrm{d}, \mathrm{t}], \langle v_\mathrm{t} \rangle \right\rangle.$$

An example is shown in Figure 9.1. We define the *source event* $\varepsilon_\mathrm{s} := T_1[\mathrm{s}]$, *origin event* $\varepsilon_\mathrm{o} := T_1[\mathrm{o}]$, *destination event* $\varepsilon_\mathrm{d} := T_2[\mathrm{d}]$ and *target event* $\varepsilon_\mathrm{t} := T_2[\mathrm{t}]$. Corresponding to these are the *source vertex* $v_\mathrm{s} := v(\varepsilon_\mathrm{s})$, *origin vertex* $v_\mathrm{o} := v(\varepsilon_\mathrm{o})$, *destination vertex* $v_\mathrm{d} := v(\varepsilon_\mathrm{d})$ and *target vertex* $v_\mathrm{t} := v(\varepsilon_\mathrm{t})$. In this chapter, we use a more compact notation to represent candidates and witnesses: We write $[\varepsilon_a, \varepsilon_b]$ for a trip segment from ε_a to ε_b, whereas a trip segment prefix starting at ε_a is simply denoted as ε_a. Since we assume that all transfers are shortest paths, they are uniquely determined by their endpoints. We therefore omit all transfers from the

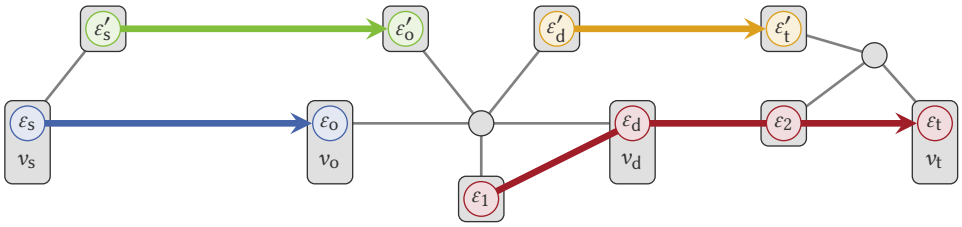


Figure 9.1: A network with a candidate $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$ and various witness types.

notation except for non-empty initial or final transfers, which are represented by the source or target vertex, respectively. Thus, a candidate is notated as $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$. Its standard prefixes are the *source prefix* $J^c_s = \langle \varepsilon_s \rangle$, the *origin prefix* $J^c_o = \langle [\varepsilon_s, \varepsilon_o], v_d \rangle$, the *destination prefix* $J^c_d = \langle [\varepsilon_s, \varepsilon_o], \varepsilon_d \rangle$, and J^c itself. The non-standard prefix that ends at a vertex v is denoted as $J^c_o(v) = \langle [\varepsilon_s, \varepsilon_o], v \rangle$. In particular, we call $J^c_o(v_o)$ the *origin stop prefix*. For a journey (prefix) J, we denote by $\mathcal{E}(J)$ the sequence of stop events in J at which a trip is entered or exited. For the candidate J^c, this sequence is $\mathcal{E}(J^c) = \langle \varepsilon_s, \varepsilon_o, \varepsilon_d, \varepsilon_t \rangle$. Standard prefixes of J^c are uniquely identified by their stop event sequence. The standard prefix that ends with a stop event $\varepsilon \in \mathcal{E}(J^c)$ is called the *ε-prefix*.

9.2 Characterizing Required Shortcuts

Since a set of shortcuts that is sufficient for all possible delay scenarios would be too large, we limit the precomputation to scenarios in which no delay exceeds a given *delay limit* $\delta^{\max}$. Our goal is to enumerate all candidates that are prefix-optimal in at least one such delay scenario. For the purpose of the shortcut computation, which we discuss throughout the following sections, we assume that all delay scenarios conform to this delay limit. We discuss how to handle scenarios beyond the delay limit in Section 9.5.

Switching from canonical candidates to prefix-optimal candidates means that we potentially enumerate more journeys than necessary for a particular delay scenario. However, this is not an issue because the additional candidates are likely to be canonical in another delay scenario that also conforms to the delay limit. To illustrate why this is the case, consider two candidates J^c_1 and J^c_2 with different target events ε_1 and ε_2 that have the same arrival time in some delay scenario Δ. If we increase the delay of ε_1 by one second compared to Δ, then J^c_1 is no longer optimal, but J^c_2 still is. Decreasing the delay by one second has the opposite effect: J^c_2 is no longer optimal, but J^c_1 still is. Unless the delay of ε_1 in Δ is 0 or $\delta^{\max}$, these changes result in valid delay scenarios that also conform to the delay limit. Hence, it is very likely that both candidates need to be enumerated anyway.

Even with the delay limit, the number of possible delay scenarios is still astronomical, so it is not feasible to consider each one individually. Instead, we develop a more succinct characterization of candidates that are prefix-optimal in at least one delay scenario. Section 9.2.1

establishes a simple characterization under the assumption that candidates and witnesses do not have any stop events in common. In Section 9.2.2, we generalize this result to allow for shared stop events. Based on our findings, Section 9.2.3 formally defines the shortcut computation problem in the presence of delays.

9.2.1 Best-Case and Virtual Delay Scenarios

To simplify proofs, we define a partial order $\preceq_{\text{eval}}$ on delay scenarios. For two delay scenarios Δ^1, Δ^2, we write $\Delta^1 \preceq_{\text{eval}} \Delta^2$ if $\Delta^1_{\text{dep}}(\varepsilon) \geq \Delta^2_{\text{dep}}(\varepsilon)$ and $\Delta^1_{\text{arr}}(\varepsilon) \leq \Delta^2_{\text{arr}}(\varepsilon)$ holds for every stop event $\varepsilon \in \mathcal{E}$. Then Δ^1 is "better" than Δ^2 in the following sense: a journey J that is feasible in Δ^1 for a departure time τ_{dep} is also feasible for τ_{dep} in Δ^2, and its arrival time in Δ^1 is not higher than in Δ^2. Consider the global *best-case* delay scenario Δ^{best}, in which all arrivals have delay 0 and all departures have delay $\delta^{\max}$. The other extreme is the *worst-case* scenario Δ^{worst}, which assumes that all departures are punctual and all arrivals have maximal delay. Then $\Delta^{\text{best}} \preceq_{\text{eval}} \Delta \preceq_{\text{eval}} \Delta^{\text{worst}}$ holds for every delay scenario Δ.

For a journey prefix J_{p}, the best-case scenario $\Delta^{\text{best}}(J_{\text{p}})$ assumes the best case for all stop events in $\mathcal{E}(J_{\text{p}})$ and the worst case otherwise. Consider a candidate J^{c} and a witness J^{w} that does not use any stop events in $\mathcal{E}(J^{\text{c}})$. If J^{w} strongly dominates a prefix of J^{c} in $\Delta^{\text{best}}(J^{\text{c}})$, then it does so in every delay scenario. Thus, if we (wrongly) assume that witnesses do not share stop events with J^{c}, we obtain the very simple condition that J^{c} is prefix-optimal in at least one delay scenario iff it is prefix-optimal in $\Delta^{\text{best}}(J^{\text{c}})$. We formalize this assumption by introducing virtual delay scenarios. A *virtual delay scenario* $\Delta^{\text{virt}} = (\Delta^{\text{can}}, \Delta^{\text{wit}})$ consists of a *candidate scenario* Δ^{can} and a *witness scenario* Δ^{wit}. A candidate prefix J^{c}_{p} is strongly dominated by a witness J^{w} in Δ^{virt} if J^{w} as evaluated in Δ^{wit} strongly dominates J^{c}_{p} as evaluated in Δ^{can}. Thus, stop events that are shared between both journeys can act as if they were not shared by assuming different delays in Δ^{can} and Δ^{wit}.

We define another partial order $\preceq_{\text{dom}}$ specifically for virtual delay scenarios. Given two virtual delay scenarios $\Delta^{\text{virt}}_1 = (\Delta^{\text{can}}_1, \Delta^{\text{wit}}_1)$ and $\Delta^{\text{virt}}_2 = (\Delta^{\text{can}}_2, \Delta^{\text{wit}}_2)$, we write $\Delta^{\text{virt}}_1 \preceq_{\text{dom}} \Delta^{\text{virt}}_2$ if $\Delta^{\text{can}}_1 \preceq_{\text{eval}} \Delta^{\text{can}}_2$ and $\Delta^{\text{wit}}_1 \succeq_{\text{eval}} \Delta^{\text{wit}}_2$. If a witness J^{w} strongly dominates a candidate prefix J^{c}_{p} in Δ^{virt}_1, it also does so in Δ^{virt}_2. Thus, if J^{c}_{p} is prefix-optimal in Δ^{virt}_2, it is also prefix-optimal in Δ^{virt}_1. Since each proper delay scenario Δ has an equivalent virtual delay scenario (Δ, Δ), virtual delay scenarios are a superset of proper delay scenarios. We therefore extend the definition of $\preceq_{\text{dom}}$ to proper delay scenarios as well.

For a candidate prefix J^{c}_{p}, the *virtual best-case scenario* $\Delta^{\text{virt}}(J^{\text{c}}_{\text{p}}) := (\Delta^{\text{best}}, \Delta^{\text{best}}(J^{\text{c}}_{\text{p}}))$ assumes the best case for J^{c}_{p} and for all witness events that are shared with it, and the worst case for everything else. For a full candidate J^{c}, $\Delta^{\text{virt}}(J^{\text{c}})$ is equivalent to $\Delta^{\text{best}}(J^{\text{c}})$. For the empty prefix $J^{\text{c}}_{\text{p}} = \langle\rangle$, the scenario $\Delta^{\text{virt}}(\langle\rangle) = (\Delta^{\text{best}}, \Delta^{\text{worst}})$ assumes the best case for J^{c} and the worst case for all witnesses. Theorem 9.1 shows that we can use $\Delta^{\text{virt}}(\langle\rangle)$ to obtain a simple characterization of prefix-optimal candidates.

Theorem 9.1. *A candidate J^c is prefix-optimal in at least one virtual delay scenario iff it is prefix-optimal in $\Delta^{virt}(\langle\rangle) = (\Delta^{best}, \Delta^{worst})$.*

Proof. Let $\Delta^{\text{virt}} = (\Delta^{\text{can}}, \Delta^{\text{wit}})$ be a virtual delay scenario. With

$$\Delta^{\text{wit}} \preceq_{\text{eval}} \Delta^{\text{worst}} = \Delta^{\text{best}}(\langle\rangle) \quad \text{and}$$
$$\Delta^{\text{can}} \succeq_{\text{eval}} \Delta^{\text{best}},$$

it follows that $\Delta^{\text{virt}} \succeq_{\text{dom}} \Delta^{\text{virt}}(\langle\rangle)$. Hence, if J^c is prefix-optimal in Δ^{virt}, it is also prefix-optimal in $\Delta^{\text{virt}}(\langle\rangle)$. □

9.2.2 Shared Stop Events

Theorem 9.1 implies that a straightforward adaptation of ULTRA that explores candidates in Δ^{best} and witnesses in Δ^{worst} will generate a sufficient set of shortcuts. However, this set will be impractically large because many of the shortcuts are only required in a virtual delay scenario, but not in any proper delay scenarios. To avoid this, we investigate the effects of shared stop events.

Hook Witnesses. In Figure 9.1, consider the witness $J^w = \langle[\varepsilon_s, \varepsilon_o], [\varepsilon'_d, \varepsilon'_t]\rangle$. If the origin event ε_o is delayed, this may cause J^w to miss its intermediate transfer, even as the candidate $J^c = \langle[\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t]\rangle$ remains feasible. This shows that it is not sufficient to consider the best case for shared stop events. In the following, we will show that ε_o is the only stop event of J^c where this is an issue. For this purpose, we introduce the concept of hook witnesses. We call a witness J^w for a journey (prefix) J a *hook witness* if $\mathcal{E}(J^w)$ can be divided into a prefix that is shared with $\mathcal{E}(J)$ (the *handle*) and a suffix that is not (the *hook*). In Figure 9.1, $\langle[\varepsilon_s, \varepsilon_o], \varepsilon_1\rangle$ is a hook witness for J^c_d while $\langle[\varepsilon_s, \varepsilon_o], [\varepsilon_1, \varepsilon_t]\rangle$ is a non-hook witness for J^c. Note that if $\mathcal{E}(J^w)$ includes a stop event $\varepsilon \in \mathcal{E}(J)$, it includes the entire ε-prefix of J. Lemma 9.2 shows that it is sufficient to consider hook witnesses, because every non-hook witness has an equivalent hook witness that replaces everything up to the last shared stop event ε with the ε-prefix of J.

Lemma 9.2. *A candidate J^c is prefix-optimal iff no prefix of J^c is strongly dominated by a hook witness.*

Proof. Let J^w be a non-hook witness for a prefix J^c_p of J^c. We construct a hook witness J' that strongly dominates J^c_p as follows: Since J^w is not a hook witness, it must share at least one stop event with J^c_p. Let ε be the last shared stop event, and let J^c_1 and J^w_1 be the ε-prefixes of J^c_p and J^w, respectively. Then there is a suffix J^w_2 such that $J^w = J^w_1 \circ J^w_2$. Because J^w_1 and J^c_1 end with the same stop event, $J' := J^c_1 \circ J^w_2$ is a valid and feasible journey (prefix). By construction, J' is a hook witness and strongly dominates J^c_p. □

A simple observation about hook witnesses is that they never use the last stop event of the journey (prefix) they strongly dominate:

Lemma 9.3. *Let J_p be a journey prefix with stop event sequence $\mathcal{E}(J_p) = \langle \varepsilon_1, \ldots, \varepsilon_k \rangle$, Δ a delay scenario, and J^w a hook witness that strongly dominates J_p in Δ. Then $\varepsilon_k \notin \mathcal{E}(J^w)$.*

Proof. Assume that $\varepsilon_k \in \mathcal{E}(J^w)$. Because J^w is a hook witness, this implies that the non-shared suffix of J^w is empty, and thus $J^w = J_p$. This contradicts the fact that J^w strongly dominates J_p. □

Parameterized Scenarios. Let $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$ be a candidate and J^w a hook witness. In order to evaluate whether J^w strongly dominates a prefix of J^c in any delay scenario, we can assume the best case for ε_s, ε_d and ε_t: If J^w includes ε_t, then J^w is identical to J^c, which cannot strongly dominate itself. If it includes ε_d, it must also include ε_o and thus have the same intermediate transfer as J^c. If it includes ε_s, then it has the same initial transfer. Thus, changing the delay of ε_s or ε_d to make J^w infeasible will also make J^c infeasible. We formalize this by defining the *parameterized best-case scenario* $\Delta^{\text{best}}(J^c_p, \delta)$ for a candidate prefix J^c_p with origin event ε_o and a delay δ. It is identical to the best-case scenario $\Delta^{\text{best}}(J^c_p)$, except that the arrival delay of ε_o is δ. Note that $\Delta^{\text{best}}(J^c_p) = \Delta^{\text{best}}(J^c_p, 0)$.

It is only necessary to consider parameterized scenarios in which the candidate is feasible. For this purpose, we define the *slack* $\text{sl}(\varepsilon_o, \varepsilon_d)$ of an intermediate transfer between two stop events ε_o and ε_d as the waiting time at $v(\varepsilon_d)$ before ε_d departs, assuming both events are punctual. Note that the slack may be negative; in this case, the transfer is only feasible if ε_d is sufficiently delayed. The *feasibility limit*

$$\lambda_f(J^c) := \min\left(0, \text{sl}(\varepsilon_o, \varepsilon_d)\right) + \delta^{\max}$$

of a candidate J^c is the maximal delay $\delta \leq \delta^{\max}$ such that J^c is feasible in $\Delta^{\text{best}}(J^c, \delta)$. Lemma 9.4 shows that it is sufficient to consider parameterized scenarios in which the delay of ε_o does not exceed $\lambda_f(J^c)$.

Lemma 9.4. *A candidate $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$ is prefix-optimal in at least one delay scenario iff there is a delay $\delta \in [0, \lambda_f(J^c)]$ such that J^c is prefix-optimal in $\Delta^{best}(J^c, \delta)$.*

Proof. Assume that for every delay $\delta \in [0, \lambda_f(J^c)]$, a prefix of J^c is strongly dominated by a witness in $\Delta^{\text{best}}(J^c, \delta)$. Let Δ be a delay scenario. By Lemma 9.2, there is a prefix J^c_p of J^c that is strongly dominated by a hook witness J^w in $\Delta^o := \Delta^{\text{best}}(J^c, \Delta_{\text{arr}}(\varepsilon_o))$. We show that J^w is feasible and strongly dominates J^c_p in Δ. Consider the delay scenario $\Delta^{\max}$ with

$$\Delta^{\max}_{\text{dep}}(\varepsilon) = \min(\Delta^o_{\text{dep}}(\varepsilon), \Delta_{\text{dep}}(\varepsilon)),$$
$$\Delta^{\max}_{\text{arr}}(\varepsilon) = \max(\Delta^o_{\text{arr}}(\varepsilon), \Delta_{\text{arr}}(\varepsilon))$$

for each stop event $\varepsilon \in \mathcal{E}$. This scenario differs from Δ^o only in the arrival delay of ε_t and in the departure delays of ε_s and ε_d. By Lemma 9.3, J^w cannot use ε_t. The departure delays of ε_s and ε_d, as well as the arrival delay of ε_o, are identical in Δ and $\Delta^{\max}$. Accordingly, the departure and intermediate transfer of J^c are feasible in $\Delta^{\max}$. If J^w contains an intermediate

transfer between some stop event $\varepsilon'_{\mathrm{o}}$ and ε_{d}, then $\varepsilon'_{\mathrm{o}} = \varepsilon_{\mathrm{o}}$ because J^{w} is a hook witness. Thus, the intermediate transfer is feasible in $\Delta^{\max}$. Likewise, if J^{w} uses ε_{s}, then its departure is feasible in $\Delta^{\max}$. Therefore, J^{w} is feasible in $\Delta^{\max}$ and its arrival time remains unchanged from Δ^{o}. Accordingly, J^{w} strongly dominates $J^{\mathrm{c}}_{\mathrm{p}}$ in the virtual delay scenario $(\Delta^{\mathrm{o}}, \Delta^{\max})$. Since $\Delta^{\max} \succeq_{\mathrm{eval}} \Delta^{\mathrm{o}}$ and $\Delta \preceq_{\mathrm{eval}} \Delta^{\max}$, this is still the case in $(\Delta^{\max}, \Delta) \succeq_{\mathrm{dom}} (\Delta^{\mathrm{o}}, \Delta^{\max})$. Since Δ and $\Delta^{\max}$ are equivalent for J^{c}, this implies that J^{w} strongly dominates $J^{\mathrm{c}}_{\mathrm{p}}$ in (Δ, Δ), which is equivalent to Δ. □

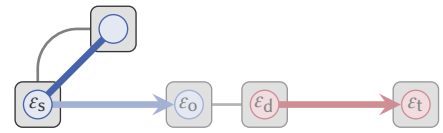
Hook Witness Classification. Lemmas 9.2 and 9.4 significantly narrow the scope of the shortcut computation problem. To determine whether a candidate J^{c} is prefix-optimal in at least one delay scenario, it is enough to consider hook witnesses and to keep the delays of all stop events except for the origin event ε_{o} fixed. To make further progress, we examine the effect that the delay of ε_{o} has on different types of hook witnesses. For this purpose, we introduce the following classification: Given a candidate $J^{\mathrm{c}} = \langle [\varepsilon_{\mathrm{s}}, \varepsilon_{\mathrm{o}}], [\varepsilon_{\mathrm{d}}, \varepsilon_{\mathrm{t}}] \rangle$ and stop events $\varepsilon_a, \varepsilon_b \in \mathcal{E}(J^{\mathrm{c}})$, we call a (partial) journey J^{w} an $(\varepsilon_a, \varepsilon_b)$-witness if it is a hook witness for the ε_b-prefix of J^{c} whose handle is the ε_a-prefix of J^{c}. If J^{w} does not share any stop events with J^{c}, we call it a $(\bot, \varepsilon_b)$-witness. Overall, this yields ten different hook witness types, which are illustrated in Figure 9.2.

Consider a parameterized delay scenario $\Delta^{\mathrm{best}}(J^{\mathrm{c}}, \delta)$ with origin delay δ. The arrival time of the origin prefix $J^{\mathrm{c}}_{\mathrm{o}} = \langle [\varepsilon_{\mathrm{s}}, \varepsilon_{\mathrm{o}}], v_{\mathrm{d}} \rangle$ increases with δ, whereas the arrival time of hook witnesses for $J^{\mathrm{c}}_{\mathrm{o}}$ is unaffected. Accordingly, they only strongly dominate J^{c} if δ is high enough. We call these witnesses, which have the type $(\bot, \varepsilon_{\mathrm{o}})$ or $(\varepsilon_{\mathrm{s}}, \varepsilon_{\mathrm{o}})$, *join* witnesses. *Split* witnesses are hook witnesses of the types $(\varepsilon_{\mathrm{o}}, \varepsilon_{\mathrm{d}})$ or $(\varepsilon_{\mathrm{o}}, \varepsilon_{\mathrm{t}})$, whose handle ends with ε_{o}. If δ is too high, their intermediate transfer becomes infeasible. All other hook witnesses are not affected by δ and are therefore called *full witnesses*. Lemma 9.5 characterizes the effect of each hook witness type on the values of δ for which J^{c} is prefix-optimal: If there is a full witness that strongly dominates J^{c} for any value of δ, then it does so for all values and J^{c} is not needed. Otherwise, join witnesses establish an upper bound on δ, whereas split witnesses establish a lower bound.

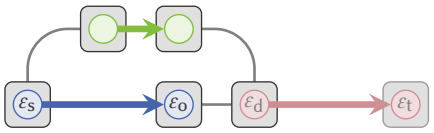
Lemma 9.5. *Let $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$ be a candidate and $\delta \in [0, \lambda_f(J^c)]$ a delay such that a prefix J^c_p of J^c is strongly dominated by a witness J^w in $\Delta^{best}(J^c, \delta)$. Then J^w strongly dominates J^c_p in every delay scenario $\Delta^{best}(J^c, \delta')$ with*

$$\delta' \in \begin{cases} [0, \lambda_f(J^c)] & \text{if } J^w \text{ is a full witness,} \\ [\delta, \lambda_f(J^c)] & \text{if } J^w \text{ is a join witness,} \\ [0, \delta] & \text{if } J^w \text{ is a split witness.} \end{cases}$$

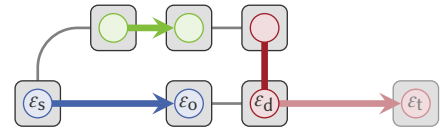
Proof. W.l.o.g., we assume that $J^{\mathrm{c}}_{\mathrm{p}}$ is a standard prefix of J^{c}. We show that J^{w} is feasible and strongly dominates $J^{\mathrm{c}}_{\mathrm{p}}$ in $\Delta^{\mathrm{best}}(J^{\mathrm{c}}, \delta')$. Only the delay of ε_{o} differs between the two scenarios. This can have two effects:



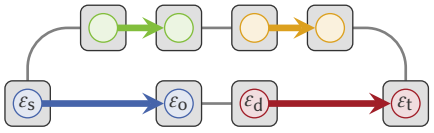
(a) A $(\bot, \varepsilon_s)$-witness (full).



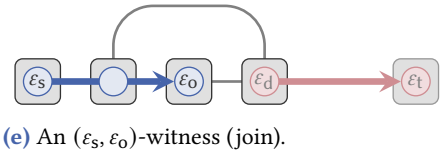
(b) A $(\bot, \varepsilon_o)$-witness (join).



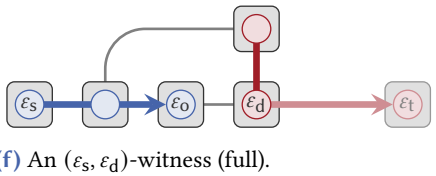
(c) A $(\bot, \varepsilon_d)$-witness (full).



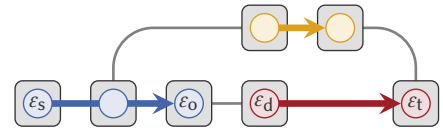
(d) A $(\bot, \varepsilon_t)$-witness (full).



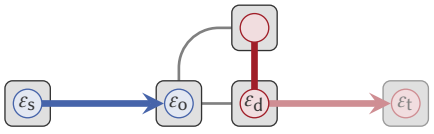
(e) An $(\varepsilon_s, \varepsilon_o)$-witness (join).



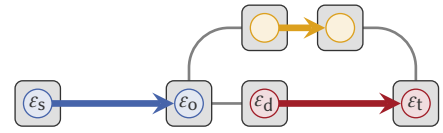
(f) An $(\varepsilon_s, \varepsilon_d)$-witness (full).



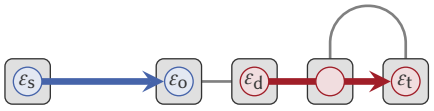
(g) An $(\varepsilon_s, \varepsilon_t)$-witness (full).



(h) An $(\varepsilon_o, \varepsilon_d)$-witness (split).



(i) An $(\varepsilon_o, \varepsilon_t)$-witness (split).



(j) An $(\varepsilon_d, \varepsilon_t)$-witness (full).

Figure 9.2: Examples of the different hook witness types for a candidate $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$. An $(\varepsilon_a, \varepsilon_b)$-witness is a hook witness for the ε_b-prefix of J^c whose shared prefix is the ε_a-prefix of J^c. If the shared prefix is empty, we write $\varepsilon_a = \bot$. The suffix that follows after the witness rejoins the candidate is grayed out. Also listed are the types according to the join/split/full classification. Witnesses of the form $(\cdot, \varepsilon_o)$ are join witnesses, whereas witnesses of the form $(\varepsilon_o, \cdot)$ are split witnesses. All other hook witnesses are full witnesses.

1. If $\delta' > \delta$ and J^w includes an intermediate transfer from ε_o to some other stop event ε'_d, this intermediate transfer can become infeasible. If $\varepsilon'_d = \varepsilon_d$, the intermediate transfer is the same as that of J^c, which is feasible in $\Delta^{\text{best}}(J^c, \delta')$. Otherwise, J^w is a split witness, which contradicts $\delta' > \delta$.

2. If $\delta' < \delta$ and J^c_p is the origin prefix $\langle [\varepsilon_s, \varepsilon_o], v_d \rangle$, then the arrival time of J^c_p decreases in $\Delta^{\text{best}}(J^c, \delta')$. If J^w does not use ε_o as well, it may no longer strongly dominate J^c_p. However, in this case J^w is a join witness, which contradicts $\delta' < \delta$. □

Final Characterization. Given a set X of witnesses, a delay scenario Δ is called *X-avoiding* for J^c if no prefix of J^c is strongly dominated by an X-witness in Δ. A delay δ is called X-avoiding if the parameterized scenario $\Delta^{\text{best}}(J^c, \delta)$ is X-avoiding. We call the lowest split-avoiding delay in $[0, \delta^{\max} + 1]$ the *split limit* $\lambda_s(J^c)$, and the highest join-avoiding delay in $(-\infty, \delta^{\max}]$ the *join limit* $\lambda_j(J^c)$. The *minimum* and *maximum origin delay* $\delta^o_{\min}(J^c)$ and $\delta^o_{\max}(J^c)$ additionally take full witnesses into account:

$$\delta^o_{\min}(J^c) := \begin{cases} \delta^{\max} + 1 & \text{if } \Delta^{\text{best}}(J^c) \text{ is not full-avoiding,} \\ \lambda_s(J^c) & \text{otherwise.} \end{cases}$$

$$\delta^o_{\max}(J^c) := \begin{cases} -\infty & \text{if } \Delta^{\text{best}}(J^c) \text{ is not full-avoiding,} \\ \min(\lambda_f(J^c), \lambda_j(J^c)) & \text{otherwise.} \end{cases}$$

Together, they form the *origin delay interval* $I^o_\delta(J^c) := [\delta^o_{\min}(J^c), \delta^o_{\max}(J^c)]$. Figure 9.3 gives an example of how it is calculated. Based on this, Theorem 9.6 establishes our final characterization of candidates that are prefix-optimal in at least one delay scenario.

Theorem 9.6. *A candidate J^c is prefix-optimal in at least one delay scenario iff $I^o_\delta(J^c) \neq \emptyset$.*

Proof. By Lemma 9.4, J^c is prefix-optimal in at least one delay scenario iff there is a delay $\delta \in [0, \lambda_f(J^c)]$ such that J^c in $\Delta^{\text{best}}(J^c, \delta)$. By Lemma 9.2, this is the case iff $\Delta^{\text{best}}(J^c, \delta)$ is hook-avoiding for J^c. By Lemma 9.5, this is the case iff $\delta \in I^o_\delta(J^c)$. Hence, J^c is prefix-optimal in at least one delay scenario iff $I^o_\delta(J^c) \neq \emptyset$. □

9.2.3 Problem Definition

Based on the condition established by Theorem 9.6, we formally define the shortcut computation problem. For a potential shortcut $e = (\varepsilon_o, \varepsilon_d)$, let $\mathcal{J}^c(e)$ be the set of candidates with an intermediate transfer from ε_o to ε_d. The shortcut is necessary if the union of the origin delay intervals for all candidates in $\mathcal{J}^c(e)$ is not empty. For the sake of simplicity, we do not consider the union (which may not form an interval itself) but rather the smallest interval containing all origin delay intervals. Let $\delta^o_{\min}(e)$ and $\delta^o_{\max}(e)$ be the lowest minimum and highest maximum origin delays among $\mathcal{J}^c(e)$, respectively. Then the interval

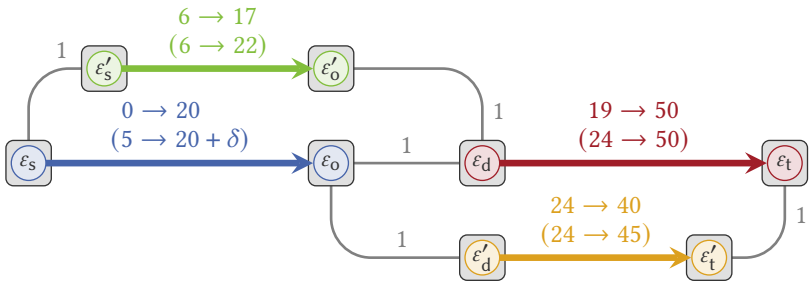


Figure 9.3: An example of how the origin delay interval is calculated for the candidate $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$, assuming a delay limit of $\delta^{\max} = 5$. Trips are labeled with the scheduled departure and arrival times. The departure and arrival times in the parameterized delay scenario $\Delta^{\text{best}}(J^c, \delta)$ are given below in parentheses. The intermediate transfer from ε_o to ε_d has a slack of $\text{sl}(\varepsilon_o, \varepsilon_d) = -2$, which yields a feasibility limit is $\lambda_f(J^c) = 3$. There are two witnesses to consider: the join witness $J^j = \langle [\varepsilon'_s, \varepsilon'_o], v_d \rangle$ and the split witness $J^s = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon'_d, \varepsilon'_t] \rangle$. The join witness J^j has an arrival time of 23 at v_d, whereas the origin prefix $J^c_o = \langle [\varepsilon_s, \varepsilon_o], v_d \rangle$ has an arrival time of $21 + \delta$. This yields a join limit of $\lambda_j(J^c) = 2$. The split witness J^s strongly dominates J^c. It is feasible if the arrival time at $v(\varepsilon'_d)$ is at most 24, which corresponds to a split limit of $\lambda_s(J^c) = 3$. Because there are no full witnesses, the minimum origin delay is $\delta^o_{\min}(J^c) = \lambda_s(J^c) = 3$ and the maximum origin delay is $\delta^o_{\max}(J^c) = \min(\lambda_f(J^c), \lambda_j(J^c)) = 2$. Thus, the origin delay interval $I^o_\delta(J^c) = [3, 2]$ is empty, which means that there is no delay scenario in which J^c is prefix-optimal.

is given by $I^o_\delta(e) := [\delta^o_{\min}(e), \delta^o_{\max}(e)]$. We will see in Section 9.3 that computing $\delta^o_{\min}(e)$ exactly is expensive, so we only ask for a lower bound $\underline{\delta^o_{\min}}(e)$. If the corresponding interval $\underline{I^o_\delta}(e) := [\underline{\delta^o_{\min}}(e), \delta^o_{\max}(e)]$ is empty, then e is not required in any delay scenario. While the converse is not necessarily true, superfluous shortcuts only affect the performance of the query algorithm, not its correctness.

Given a network $(\mathcal{S}, \Pi, \mathcal{E}, \mathcal{T}, \mathcal{R}, G)$ and a delay limit $\delta^{\max}$, the DelayShortcut problem asks for the set

$$E^s = \left\{ e \in \mathcal{E} \times \mathcal{E} \mid \underline{I^o_\delta}(e) \neq \emptyset \right\}$$

of relevant shortcuts, as well as the origin delay interval $\underline{I^o_\delta}(e)$ for each shortcut $e \in E^s$. The latter can be used to discard irrelevant shortcuts when the delay scenario is revealed in the update phase: a shortcut $e = (\varepsilon_o, \varepsilon_d)$ can be discarded in a delay scenario Δ if $\Delta_{\text{arr}}(\varepsilon_o) \notin \underline{I^o_\delta}(e)$.

9.3 Efficient Candidate Testing

Theorem 9.6 implies an algorithmic framework for solving the DelayShortcut problem: generate all possible candidates and test for each candidate J^c whether the origin delay interval $I^o_\delta(J^c)$ is empty. To perform this test, the algorithm needs to search for witnesses that strongly dominate a prefix of J^c in one of the relevant delay scenarios. These scenarios are different for each candidate, so a naive approach would perform a new search each time. To obtain a more efficient algorithm, we exploit the fact that candidates with a common prefix share the same set of witnesses for this prefix.

Given an origin stop prefix $J^c_o(v_o) = \langle[\varepsilon_s, \varepsilon_o], v_o\rangle$, we define the subproblem DelayShortcut-$J^c_o(v_o)$, which only allows candidates that begin with $J^c_o(v_o)$. The overall DelayShortcut problem can be broken down into solving DelayShortcut-$J^c_o(v_o)$ for every possible origin stop prefix $J^c_o(v_o)$ and merging the results. This has the advantage that the origin event ε_o is now fixed by the input, which means that a shortcut $e = (\varepsilon_o, \varepsilon_d)$ is uniquely identified by ε_d. Thus, the set $\mathcal{J}^c(e)$ of candidates containing e becomes the set $\mathcal{J}^c(\varepsilon_d)$ of candidates with destination prefix J^c_d. Likewise, the minimum and maximum origin delays $\delta^o_{\min}(e) = \delta^o_{\min}(\varepsilon_d)$ and $\delta^o_{\max}(e) = \delta^o_{\max}(\varepsilon_d)$, the origin delay interval $I^o_\delta(e) = I^o_\delta(\varepsilon_d)$ and their lower bounds also depend only on ε_d.

In this section, we show how an individual DelayShortcut-$J^c_o(v_o)$ problem can be solved with only two witness searches: one in $\Delta^{\text{best}}(J^c_s)$ and one in $\Delta^{\text{best}}(J^c_o)$. For this purpose, we define various arrival times:

- For a vertex v, the *candidate arrival time* of $J^c_o(v) = \langle[\varepsilon_s, \varepsilon_o], v\rangle$ in Δ^{best} is given by $\tau^c_{\text{arr}}(v) := \tau_{\text{arr}}(\varepsilon_o) + \tau_{\text{tra}}(v_o, v)$.

- For a candidate prefix J^c_p, a vertex v and a number of trips $n \leq 2$, the *witness arrival time* $\tau^w_{\text{arr}}(J^c_p, v, n)$ is the earliest arrival time among v_s-v-journeys with at most n trips that depart no earlier than $\tau_{\text{dep}}(\varepsilon_s) + \delta^{\max}$ in $\Delta^{\text{best}}(J^c_p)$.

In the following, we establish formulas for the various components of the origin delay interval that depend only on $\tau^c_{\text{arr}}(\cdot)$, $\tau^w_{\text{arr}}(J^c_s, \cdot, \cdot)$, $\tau^w_{\text{arr}}(J^c_o, \cdot, \cdot)$ and derived values. This is done in three steps: calculating the join and feasibility limits (Section 9.3.1), examining full witnesses (Section 9.3.2), and calculating the split limits (Section 9.3.3). Additionally, Section 9.3.4 discusses how to exclude impossible delay scenarios in which trips travel backward in time.

Throughout this section, we assume that the source prefix $J^c_s = \langle T_1[s]\rangle$ is Pareto-optimal in every delay scenario Δ. To see why this assumption is realistic, consider the example in Figure 9.2a. A witness that strongly dominates J^c_s can only be a $(\bot, \varepsilon_s)$-witness. This means it must have the form $J^w = \langle T_1[i]\rangle$ with $i < s$, i.e., it takes a transfer to an earlier stop along T and enters the trip there. If J^w is feasible in Δ, then $\tau_{\text{dep}}(T_1[i], \Delta) \geq \tau_{\text{dep}}(T_1[s], \Delta)$ must hold. This requires that the trip moves from index i to s instantaneously (or even backward in time), which is physically impossible.

9.3.1 Join and Feasibility Limit

The join limit $\lambda_j(J^c)$ of a candidate $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$ is the highest delay $\delta \in [-\infty, \delta^{\max}]$ such that $\Delta^{\text{best}}(J^c, \delta)$ is join-avoiding. Since join witnesses are witnesses for the origin prefix $J^c_o = \langle [\varepsilon_s, \varepsilon_o], v_d \rangle$, they do not use ε_d or ε_t. Accordingly, we can consider the parameterized scenario $\Delta^{\text{best}}(J^c_s, \delta)$ for the source prefix J^c_s instead of $\Delta^{\text{best}}(J^c, \delta)$. We define the join limit for a vertex v as

$$\lambda_j(v) := \tau^w_{\text{arr}}(J^c_s, v, 1) - \tau^c_{\text{arr}}(v). \tag{9.1}$$

Lemma 9.7 shows the relation to the join limit of a candidate.

Lemma 9.7. *For a candidate $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$, $\lambda_j(J^c) = \lambda_j(v_d)$.*

Proof. Let $\Delta := \Delta^{\text{best}}(J^c_s) = \Delta^{\text{best}}(J^c_s, \delta^{\max})$ and let J^w be a v_s-v_d-journey with at most one trip such that $\tau_{\text{arr}}(J^w, \Delta) = \tau^w_{\text{arr}}(J^c_s, v_d, 1)$. Since $\tau^c_{\text{arr}}(v_d) = \tau_{\text{arr}}(J^c_o, \Delta^{\text{best}})$ and J^w has minimal arrival time among journeys with at most one trip, it follows that

$$\lambda_j(v_d) = \tau_{\text{arr}}(J^w, \Delta) - \tau_{\text{arr}}(J^c_o, \Delta^{\text{best}}) \leq \tau_{\text{arr}}(J^c_o, \Delta) - \tau_{\text{arr}}(J^c_o, \Delta^{\text{best}}) = \delta^{\max}.$$

There are two possible cases:

1. If J^w uses ε_o, then $\tau_{\text{arr}}(J^w, \Delta) = \tau_{\text{arr}}(J^c_o, \Delta)$, so it follows that $\lambda_j(v_d) = \delta^{\max}$. Since no join witness for J^c_o has an earlier arrival time than J^w in Δ, it follows that $\delta^{\max}$ is join-avoiding.

2. If J^w does not use ε_o, it is a join witness. Consider the delay scenario $\Delta^{\text{best}}(J^c_s, \delta)$ for some delay δ. The arrival time of J^w is equal to $\tau_{\text{arr}}(J^w, \Delta) = \tau^w_{\text{arr}}(J^c_s, v_d, 1)$ and the arrival time of J^c_o is $\tau^c_{\text{arr}}(v_d) + \delta$. Thus, the scenario is join-avoiding iff $\delta \leq \tau^w_{\text{arr}}(J^c_s, v_d, 1) - \tau^c_{\text{arr}}(v_d) = \lambda_j(v_d)$.

In both cases, the highest join-avoiding delay in $[-\infty, \delta^{\max}]$ is $\lambda_j(v_d)$. □

The following two lemmas establish pruning rules based on the join limit of a vertex.

Lemma 9.8. *Let $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$ be a candidate with $\lambda_j(J^c) \geq 0$. For each vertex $v \in V$ visited by the intermediate transfer of J^c, it holds that $\lambda_j(v) \geq 0$.*

Proof. Assume that $\lambda_j(v) < 0$, i.e., $\tau^w_{\text{arr}}(J^c_s, v, 1) < \tau^c_{\text{arr}}(v)$. The transfer time between v and v_d is given by $\tau_{\text{tra}}(v, v_d)$. By the triangle inequality, $\tau^w_{\text{arr}}(J^c_s, v_d, 1) \leq \tau^w_{\text{arr}}(J^c_s, v, 1) + \tau_{\text{tra}}(v, v_d)$. Since v is visited by the intermediate transfer of J^c, it lies on a shortest v_o-v_d-path. Hence, $\tau^c_{\text{arr}}(v_d) = \tau^c_{\text{arr}}(v) + \tau_{\text{tra}}(v, v_d)$. It follows that $\tau^w_{\text{arr}}(J^c_s, v_d, 1) < \tau^c_{\text{arr}}(v_d)$ and therefore $\lambda_j(J^c) = \lambda_j(v_d) < 0$, a contradiction. □

Lemma 9.9. *Let $v \in V$ be a vertex. If $\lambda_j(v) \geq 0$, then $J^c_o(v) = \langle [\varepsilon_s, \varepsilon_o], v \rangle$ is prefix-optimal in $\Delta^{virt}(J^c_s)$.*

Proof. By our assumption at the beginning of Section 9.3, the source prefix J_s^c is Pareto-optimal in every delay scenario. It remains to be shown that there is no witness J^w that strongly dominates $J_o^c(v)$ in $\Delta^{\text{virt}}(J_s^c) = (\Delta^{\text{best}}, \Delta^{\text{best}}(J_s^c))$. The arrival time of $J_o^c(v)$ in Δ^{best} is given by $\tau_{\text{arr}}^c(v)$. The arrival time of J^w in $\Delta^{\text{best}}(J_s^c)$ is at most $\tau_{\text{arr}}^w(J_s^c, v, 1)$. Since $\lambda_j(v) = \tau_{\text{arr}}^w(J_s^c, v, 1) - \tau_{\text{arr}}^c(v) \geq 0$, it follows that J^w does not strongly dominate $J_o^c(v)$. □

For the feasibility limit $\lambda_f(J^c)$, applying the definition of $\tau_{\text{arr}}^c(\cdot)$ yields

$$\lambda_f(\varepsilon_d) = \min\left(0, \tau_{\text{dep}}(\varepsilon_d) - \tau_{\text{arr}}^c(v(\varepsilon_d))\right) + \delta^{\max}. \tag{9.2}$$

From this point onwards, we can restrict the set of relevant destination events to those with non-negative feasibility and join limits:

$$\mathcal{E}^{j/f} := \left\{\varepsilon_d \in \mathcal{E} \mid \min\left(\lambda_f(\varepsilon_d), \lambda_j(v(\varepsilon_d))\right) \geq 0\right\}.$$

All other destination events can be discarded because they cannot occur in prefix-optimal candidates.

9.3.2 Examining Full Witnesses

For each destination event $\varepsilon_d \in \mathcal{E}^{j/f}$, we need to determine the minimum and maximum origin delay $\delta_{\min}^o(\varepsilon_d)$ and $\delta_{\max}^o(\varepsilon_d)$. These depend on the minimum and maximum origin delays of the candidates that use ε_d, which are contained in $\mathcal{J}^c(\varepsilon_d)$. Candidates J^c for which $\Delta^{\text{best}}(J^c)$ is not full-avoiding do not contribute to $\delta_{\min}^o(\varepsilon_d)$ or $\delta_{\max}^o(\varepsilon_d)$, so the next step is to examine full witnesses in order to discard these candidates. Unfortunately, the witness arrival times $\tau_{\text{arr}}^w(\cdot,\cdot,\cdot)$ do not distinguish between split and full witnesses. We can circumvent this issue by considering the set $\mathcal{J}_{\text{opt}}^c(\varepsilon_d)$ of candidates $J^c \in \mathcal{J}^c(\varepsilon_d)$ for which $\delta_{\min}^o(J^c) \leq \delta^{\max}$ holds, i.e., $\Delta^{\text{best}}(J^c)$ is full-avoiding and $\Delta^{\text{best}}(J^c, \delta^{\max})$ is split-avoiding. Consider a candidate $J^c \in \mathcal{J}^c(\varepsilon_d)$. If J^c is not contained in $\mathcal{J}_{\text{opt}}^c(\varepsilon_d)$, we know that $I_\delta^o(J^c) = \emptyset$, so we can discard J^c. Otherwise, we can calculate $\delta_{\min}^o(J^c)$ and $\delta_{\max}^o(J^c)$ from the split, feasibility and join limits.

Using Virtual Delay Scenarios. Our objective is to compute the set

$$\mathcal{E}_d^{\text{opt}} := \left\{\varepsilon_d \in \mathcal{E}^{j/f} \mid \mathcal{J}_{\text{opt}}^c(\varepsilon_d) \neq \emptyset\right\}.$$

For a destination event $\varepsilon_d \in \mathcal{E}^{j/f}$, it is difficult to compute $\mathcal{J}_{\text{opt}}^c(\varepsilon_d)$ exactly. Instead, we compute the set $\mathcal{J}_{\text{virt}}^c(\varepsilon_d)$ of candidates in $\mathcal{J}^c(\varepsilon_d)$ that are prefix-optimal in $\Delta^{\text{virt}}(J_s^c)$. This allows us to use the witness arrival times for $\Delta^{\text{best}}(J_s^c)$. Lemma 9.10 shows that $\mathcal{J}_{\text{virt}}^c(\varepsilon_d)$ is a superset of $\mathcal{J}_{\text{opt}}^c(\varepsilon_d)$.

Lemma 9.10. *For every destination event $\varepsilon_d \in \mathcal{E}^{j/f}$, it holds that $\mathcal{J}_{opt}^c(\varepsilon_d) \subseteq \mathcal{J}_{virt}^c(\varepsilon_d)$.*

Proof. Consider a candidate $J^c \notin \mathcal{J}^c_{\text{virt}}(\varepsilon_d)$. By definition, there must be a prefix J^c_p of J^c is strongly dominated by a witness J^w in $\Delta^{\text{virt}}(J^c_s)$. Since $\Delta^{\text{virt}}(J^c_s) \preceq_{\text{dom}} \Delta^{\text{best}}(J^c, \delta)$ holds for every delay $\delta \in [0, \delta^{\max}]$, J^c_p is also strongly dominated by J^w in $\Delta^{\text{best}}(J^c)$ and $\Delta^{\text{best}}(J^c, \delta^{\max})$. Since $\lambda_j(v(\varepsilon_d)) \geq 0$, J^w is not a join witness. If J^w is a split witness, then $\Delta^{\text{best}}(J^c, \delta^{\max})$ is not split-avoiding, so $\lambda_s(J^c) > \delta^{\max}$. If J^w is a full witness, then $\Delta^{\text{best}}(J^c)$ is not full-avoiding. In both cases, it follows that $J^c \notin \mathcal{J}^c_{\text{opt}}(\varepsilon_d)$. □

The reason why the two sets are not equal is that $\Delta^{\text{best}}(J^c_s)$ assumes the worst case for ε_d. This underestimates $(\varepsilon_d, \varepsilon_t)$-witnesses, which are the only witnesses that use ε_d (see Figure 9.2j). Lemma 9.11 shows that this is the only difference between $\mathcal{J}^c_{\text{opt}}(\varepsilon_d)$ and $\mathcal{J}^c_{\text{virt}}(\varepsilon_d)$.

Lemma 9.11. *Let $\varepsilon_d \in \mathcal{E}^{j/f}$ be a destination event and J^c a candidate in $\mathcal{J}^c_{virt}(\varepsilon_d) \setminus \mathcal{J}^c_{opt}(\varepsilon_d)$. Then J^c is strongly dominated by an $(\varepsilon_d, \varepsilon_t)$-witness in $\Delta^{best}(J^c)$.*

Proof. Since J^c is not contained in $\mathcal{J}^c_{\text{opt}}(\varepsilon_d)$, we know that $\delta^o_{\min}(J^c) > \delta^{\max}$. There are two possible reasons for this:

1. There is a split witness J^w that strongly dominates a prefix J^c_p of J^c in $\Delta^{\text{best}}(J^c, \delta^{\max})$. Since J^w does not use ε_d or ε_t, it also strongly dominates J^c_p in $\Delta^{\text{virt}}(J^c_s)$, which contradicts $J^c \in \mathcal{J}^c_{\text{virt}}(\varepsilon_d)$.

2. There is a full witness J^w that strongly dominates a prefix J^c_p of J^c in $\Delta^{\text{best}}(J^c)$. If J^w is a $(\bot, \cdot)$- or $(\varepsilon_s, \cdot)$-witness, it does not use ε_o, ε_d or ε_t. Then it still strongly dominates J^c_p in $\Delta^{\text{virt}}(J^c_s)$, which contradicts $J^c \in \mathcal{J}^c_{\text{virt}}(\varepsilon_d)$. Hence, J^w is a $(\varepsilon_d, \varepsilon_t)$-witness. □

In order to evaluate $(\varepsilon_d, \varepsilon_t)$-witnesses exactly, we would need to perform an additional witness search in $\Delta^{\text{best}}(J^c_d)$. Because this would have to be done individually for each possible destination event in $\mathcal{E}^{j/f}$, this would be expensive. However, Lemma 9.12 shows that this is unnecessary for the purpose of computing $\mathcal{E}^{\text{opt}}_d$. Intuitively, $(\varepsilon_d, \varepsilon_t)$-witnesses are themselves candidates with a different target event than J^c. If there is a $(\varepsilon_d, \varepsilon_t)$-witness that is not strongly dominated in $\Delta^{\text{virt}}(J^c_s)$, then it is also a candidate in $\mathcal{J}^c_{\text{opt}}(\varepsilon_d)$. Thus, if $\mathcal{J}^c_{\text{virt}}(\varepsilon_d)$ is not empty, then neither is $\mathcal{J}^c_{\text{opt}}(\varepsilon_d)$, which means we can compute $\mathcal{E}^{\text{opt}}_d$ as

$$\mathcal{E}^{\text{opt}}_d = \left\{\varepsilon_d \in \mathcal{E}^{j/f} \mid \mathcal{J}^c_{\text{virt}}(\varepsilon_d) \neq \emptyset\right\}.$$

Lemma 9.12. *For every destination event $\varepsilon_d \in \mathcal{E}^{j/f}$, $\mathcal{J}^c_{virt}(\varepsilon_d) = \emptyset$ iff $\mathcal{J}^c_{opt}(\varepsilon_d) = \emptyset$.*

Proof. By Lemma 9.10, $\mathcal{J}^c_{\text{opt}}(\varepsilon_d) \subseteq \mathcal{J}^c_{\text{virt}}(\varepsilon_d)$. It therefore remains to be shown that if $\mathcal{J}^c_{\text{opt}}(\varepsilon_d)$ is empty, then so is $\mathcal{J}^c_{\text{virt}}(\varepsilon_d)$. Assume that $\mathcal{J}^c_{\text{opt}}(\varepsilon_d)$ is empty but there is at least one candidate $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle \in \mathcal{J}^c_{\text{virt}}(\varepsilon_d)$. By Lemma 9.11, J^c is strongly dominated by an $(\varepsilon_d, \varepsilon_t)$-witness $J^w = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon'_t] \rangle$ in $\Delta^{\text{best}}(J^c)$. W.l.o.g., choose J^w such that it is not strongly dominated by another $(\varepsilon_d, \varepsilon_t)$-witness in $\Delta^{\text{best}}(J^c)$. Since such a witness would not use ε'_t, it follows that J^w is also not strongly dominated by an $(\varepsilon_d, \varepsilon'_t)$-witness in $\Delta^{\text{best}}(J^w)$.

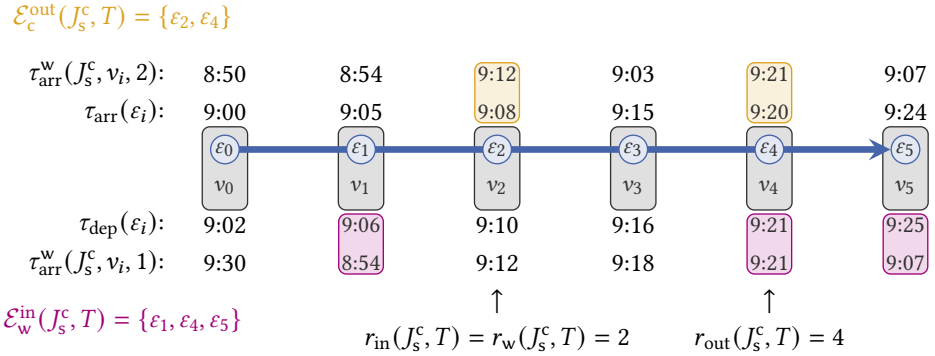


Figure 9.4: A trip T with entry index 2, exit index 4 and witness index 2.

Note that $J^w \in \mathcal{J}^c(\varepsilon_d)$ is itself a candidate. However, J^w is not included in $\mathcal{J}^c_{opt}(\varepsilon_d)$, which is empty by our assumption, and therefore not in $\mathcal{J}^c_{virt}(\varepsilon_d)$ by Lemma 9.10. Thus, J^w is strongly dominated by another witness $\overline{J^w}$ in $\Delta^{virt}(J_s^c)$. Hence, $\overline{J^w}$ as evaluated in $\Delta^{best}(J_s^c)$ strongly dominates J^w as evaluated in $\Delta^{best}(J^w)$, which in turn dominates J^w as evaluated in $\Delta^{best}(J^c)$. Since J^w strongly dominates J^c in $\Delta^{best}(J^c)$, it follows that $\overline{J^w}$ strongly dominates J^c in $\Delta^{virt}(J_s^c)$. This contradicts $J^c \in \mathcal{J}^c_{virt}(\varepsilon_d)$. □

Witness Indices. To compute $\mathcal{J}^c_{virt}(\varepsilon_d)$ efficiently, we adapt the concept of reached indices from TB. Consider a stop event ε. If $\tau_{dep}(\varepsilon) \geq \tau^w_{arr}(J_s^c, v(\varepsilon), 1)$, we call ε a *w-in event* since there is a witness in $\Delta^{best}(J_s^c)$ that is able to enter its second trip at ε. If $\tau_{arr}(\varepsilon) \leq \tau^w_{arr}(J_s^c, v(\varepsilon), 2)$, we call ε a *c-out event* since candidates ending with ε are not strongly dominated by a witness in $\Delta^{virt}(J_s^c)$. For a trip T, let $\mathcal{E}^{in}_w(J_s^c, T)$ denote the set of w-in events and $\mathcal{E}^{out}_c(J_s^c, T)$ the set of c-out events in T. We define the *entry index* $r_{in}(J_s^c, T)$ and *exit index* $r_{out}(J_s^c, T)$ as

$$r_{in}(J_s^c, T) := \begin{cases} |T| & \text{if } \mathcal{E}^{in}_w(J_s^c, T) = \emptyset, \\ \min\limits_{T[i] \in \mathcal{E}^{in}_w(J_s^c, T)} i + 1 & \text{otherwise,} \end{cases}$$

$$r_{out}(J_s^c, T) := \begin{cases} |T| & \text{if } \mathcal{E}^{out}_c(J_s^c, T) = \emptyset, \\ \max\limits_{T[i] \in \mathcal{E}^{out}_c(J_s^c, T)} i & \text{otherwise.} \end{cases}$$

The entry and exit indices are combined into the *witness index*

$$r_w(J_s^c, T) := \min\left(r_{in}(J_s^c, T), r_{out}(J_s^c, T)\right).$$

An example of how the indices are calculated is shown in Figure 9.4. Based on these definitions, we can make two observations for a destination event $T[i] \in \mathcal{E}^{j/f}$: The destination

prefix $J_\mathrm{d}^\mathrm{c} = \langle [\varepsilon_\mathrm{s}, \varepsilon_\mathrm{o}], T[i] \rangle$ is Pareto-optimal in $\Delta^\mathrm{virt}(J_\mathrm{s}^\mathrm{c})$ iff $T[i]$ is not preceded by any w-in events in T, i.e., $i < r_\mathrm{in}(J_\mathrm{s}^\mathrm{c}, T)$. Furthermore, at least one candidate in $\mathcal{J}^\mathrm{c}(T[i])$ is Pareto-optimal in $\Delta^\mathrm{virt}(J_\mathrm{s}^\mathrm{c})$ iff $T[i]$ is succeeded by at least one c-out event, i.e., $i < r_\mathrm{out}(J_\mathrm{s}^\mathrm{c}, T)$. It follows that $T[i]$ is contained in $\mathcal{E}_\mathrm{d}^\mathrm{opt}$ iff $i < r_\mathrm{w}(J_\mathrm{s}^\mathrm{c}, T)$, as shown by Lemma 9.13.

Lemma 9.13. *The set $\mathcal{E}_d^{opt}$ is equal to $\left\{T[i] \in \mathcal{E}^{j/f} \mid i < r_w(J_s^c, T)\right\}$.*

Proof. Consider a destination event $T[i] \in \mathcal{E}^{\mathrm{j/f}}$. By Lemma 9.9, it follows from $\lambda_\mathrm{j}(\nu(T[i])) \geq 0$ that the origin prefix J_o^c is prefix-optimal in $\Delta^\mathrm{virt}(J_\mathrm{s}^\mathrm{c})$. Hence, a candidate $J^\mathrm{c} \in \mathcal{J}^\mathrm{c}(T[i])$ is prefix-optimal in $\Delta^\mathrm{virt}(J_\mathrm{s}^\mathrm{c})$ iff J^c and the destination prefix $\langle [\varepsilon_\mathrm{s}, \varepsilon_\mathrm{o}], T[i] \rangle$ are Pareto-optimal in $\Delta^\mathrm{virt}(J_\mathrm{s}^\mathrm{c})$. It follows that $\mathcal{J}_\mathrm{virt}^\mathrm{c}(T[i]) \neq \emptyset$ iff $i < r_\mathrm{w}(J_\mathrm{s}^\mathrm{c}, T)$. By Lemma 9.12, this is equivalent to $\mathcal{J}_\mathrm{opt}^\mathrm{c}(T[i]) \neq \emptyset$. □

For a destination event $T[i]$, let $\mathcal{J}_\mathrm{virt}^\mathrm{c}(T[i])$ denote the set of target events that occur in the candidates from $\mathcal{E}_\mathrm{t}^\mathrm{virt}(T[i])$. Lemma 9.14 shows that these are exactly the c-out events succeeding $T[i]$ in T.

Lemma 9.14. *For a destination event $T[i] \in \mathcal{E}_d^{opt}$, $\mathcal{E}_t^{virt}(T[i]) = \mathcal{E}_c^{out}(J_s^c, T) \cap \overrightarrow{\mathcal{E}}(T[i])$.*

Proof. Consider a target event $T[j] \in \overrightarrow{\mathcal{E}}(T[i])$ and the corresponding candidate $J^\mathrm{c} = \langle [\varepsilon_\mathrm{s}, \varepsilon_\mathrm{o}], T[i, j] \rangle$. Since $T[i] \in \mathcal{E}_\mathrm{d}^\mathrm{opt}$, the destination prefix $J_\mathrm{d}^\mathrm{c} = \langle [\varepsilon_\mathrm{s}, \varepsilon_\mathrm{o}], \varepsilon_\mathrm{d} \rangle$ is prefix-optimal in $\Delta^\mathrm{virt}(J_\mathrm{s}^\mathrm{c})$. Thus, $T[j] \in \mathcal{E}_\mathrm{t}^\mathrm{virt}(T[i])$ iff J^c is Pareto-optimal in $\Delta^\mathrm{virt}(J_\mathrm{s}^\mathrm{c})$. By definition, this is the case iff $T[j] \in \mathcal{E}_\mathrm{c}^\mathrm{out}(J_\mathrm{s}^\mathrm{c}, T)$. □

9.3.3 Split Limit

The split limit $\lambda_\mathrm{s}(J^\mathrm{c})$ of a candidate $J^\mathrm{c} = \langle [\varepsilon_\mathrm{s}, \varepsilon_\mathrm{o}], [\varepsilon_\mathrm{d}, \varepsilon_\mathrm{t}] \rangle$ is the lowest delay $\delta \in [0, \delta^\mathrm{max} + 1]$ such that $\Delta^\mathrm{best}(J^\mathrm{c}, \delta)$ is split-avoiding. Since split witnesses do not use the destination event ε_d or target event ε_t, we can replace $\Delta^\mathrm{best}(J^\mathrm{c}, \delta)$ with the parameterized scenario $\Delta^\mathrm{best}(J_\mathrm{s}^\mathrm{c}, \delta)$ for the source prefix J_s^c. As shown in Figures 9.2h and 9.2i, split witnesses are either $(\varepsilon_\mathrm{o}, \varepsilon_\mathrm{d})$-witnesses for the destination prefix J_d^c or $(\varepsilon_\mathrm{o}, \varepsilon_\mathrm{t})$-witnesses for the candidate J^c itself. We divide the latter further into *direct* $(\varepsilon_o, \varepsilon_t)$*-witnesses*, which use only one trip, and *indirect* $(\varepsilon_o, \varepsilon_t)$*-witnesses*, which use two. The three split witness types are shown in Figure 9.5.

Because each split witness type has a different effect on the split limit, we divide it into three components. The *d-split limit* $\lambda_\mathrm{d.s}(\varepsilon_\mathrm{d})$ considers $(\varepsilon_\mathrm{o}, \varepsilon_\mathrm{d})$-witnesses, the *1t-split limit* $\lambda_\mathrm{1t.s}(\varepsilon_\mathrm{t})$ considers direct $(\varepsilon_\mathrm{o}, \varepsilon_\mathrm{t})$-witnesses, and the *2t-split limit* $\lambda_\mathrm{2t.s}(\varepsilon_\mathrm{t})$ considers indirect $(\varepsilon_\mathrm{o}, \varepsilon_\mathrm{t})$-witnesses. The overall split limit $\lambda_\mathrm{s}(J^\mathrm{c})$ is the maximum of all three. In the following, we establish individual formulas for the three split limits.

Direct $(\varepsilon_\mathrm{o}, \varepsilon_\mathrm{t})$-witnesses differ from the other two types in the effect that the arrival delay of the origin event ε_o has on them: For a direct $(\varepsilon_\mathrm{o}, \varepsilon_\mathrm{t})$-witness J^w, the origin delay does not affect whether J^w is feasible because it does not have an intermediate transfer, but it directly influences the arrival time and therefore whether J^w strongly dominates J^c. Thus, the 1t-split

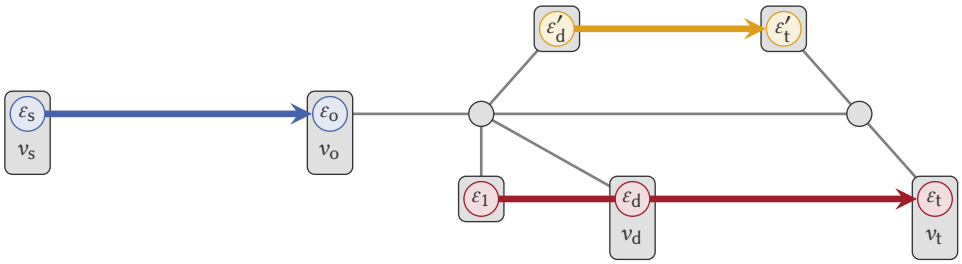


Figure 9.5: An example network showcasing the three split witness types for a candidate $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$. The partial journey $\langle [\varepsilon_s, \varepsilon_o], \varepsilon_1 \rangle$ is a $(\varepsilon_o, \varepsilon_d)$-witness. The journey $\langle [\varepsilon_s, \varepsilon_o], v_t \rangle$ that takes a direct transfer from v_o to v_t is a direct $(\varepsilon_o, \varepsilon_t)$-witness, whereas the journey $\langle [\varepsilon_s, \varepsilon_o], [\varepsilon'_d, \varepsilon'_t] \rangle$ that takes a second trip is an indirect $(\varepsilon_o, \varepsilon_t)$-witness.

limit is the lowest delay $\delta \in [0, \delta^{\max}+1]$ such that no direct $(\varepsilon_o, \varepsilon_t)$-witness has a lower arrival time than J^c in $\Delta^{\text{best}}(J^c, \delta)$.

For witnesses J^w of the other two types, the origin delay does not affect whether J^w strongly dominates J^c or J^c_d, but it determines whether the intermediate transfer is feasible. To quantify this effect, we define the *maximum witness delay* $\delta^w_{\max}(J^w)$. This is the highest delay $\delta \in [0, \delta^{\max}]$ such that J^w is feasible in $\Delta^{\text{best}}(J^c_s, \delta)$, or -1 if J^w is not feasible in any of these scenarios. If J^w contains an intermediate transfer from ε_o to a destination event ε'_d, then $\delta^w_{\max}(J^w)$ is given by

$$\delta^w_{\max}(\varepsilon'_d) := \max\Big(-1, \min\left(\tau_{\text{dep}}(\varepsilon'_d) - \tau^c_{\text{arr}}(v(\varepsilon'_d)), \delta^{\max}\right)\Big). \tag{9.3}$$

Otherwise, $\delta^w_{\max}(J^w) = \delta^{\max}$. Let X be a type of split witness (either $(\varepsilon_o, \varepsilon_d)$- or indirect $(\varepsilon_o, \varepsilon_t)$-witnesses) and let $\mathcal{J}_X$ be the set of X-witnesses that strongly dominate the respective candidate prefix (either J^c_d or J^c) in $\Delta^{\text{best}}(J^c_s)$, assuming they are feasible. If $\mathcal{J}_X$ is empty, then the X-split limit is 0. Otherwise, it is the highest maximum witness delay among $\mathcal{J}_X$ plus 1.

d-Split Limit. The set of strongly dominating $(\varepsilon_o, \varepsilon_d)$-witnesses is given by

$$\mathcal{J}_d = \left\{ \langle [\varepsilon_s, \varepsilon_o], \varepsilon'_d \rangle \mid \varepsilon'_d \in \overleftarrow{\mathcal{E}}(\varepsilon_d) \right\}.$$

Accordingly, the d-split limit can be calculated as

$$\lambda_{d.s}(\varepsilon_d) = \begin{cases} 0 & \text{if } \overleftarrow{\mathcal{E}}(\varepsilon_d) = \emptyset, \\ \max\limits_{\varepsilon'_d \in \overleftarrow{\mathcal{E}}(\varepsilon_d)} \delta^w_{\max}(\varepsilon'_d) + 1 & \text{otherwise.} \end{cases}$$

This requires iterating over the preceding stop events of ε_d and evaluating the maximum witness delay. Additionally, Lemma 9.15 shows that stop events that are not in $\mathcal{E}^{\text{opt}}_d$ can be skipped.

Lemma 9.15. *Let $\varepsilon_d = T[i] \in \mathcal{E}_d^{opt}$ be a destination event. Then*

$$\lambda_{d.s}(\varepsilon_d) = \begin{cases} 0 & \text{if } \overleftarrow{\mathcal{E}}(\varepsilon_d) \cap \mathcal{E}_d^{opt} = \emptyset, \\ \max\limits_{T[j] \in \overleftarrow{\mathcal{E}}(\varepsilon_d) \cap \mathcal{E}_d^{opt}} \delta^{w}_{max}(T[j]) + 1 & \text{otherwise.} \end{cases}$$

Proof. If $\overleftarrow{\mathcal{E}}(\varepsilon_\mathrm{d}) = \emptyset$ or $\lambda_{\mathrm{d.s}}(\varepsilon_\mathrm{d}) = 0$, the claim is trivially true. We therefore assume that $i > 0$ and $\lambda_{\mathrm{d.s}}(\varepsilon_\mathrm{d}) \geq 1$. Let $j < i$ be the smallest index such that $\lambda_{\mathrm{d.s}}(\varepsilon_\mathrm{d}) = \delta^{\mathrm{w}}_{\max}(T[j]) + 1$. We define $\varepsilon'_\mathrm{d} := T[j]$ and $v'_\mathrm{d} := v(\varepsilon'_\mathrm{d})$. We show that $\varepsilon'_\mathrm{d} \in \mathcal{E}^{\mathrm{opt}}_\mathrm{d}$ in two steps: $\varepsilon'_\mathrm{d} \in \mathcal{E}^{\mathrm{j/f}}$ and $\mathcal{J}^{\mathrm{c}}_{\mathrm{virt}}(\varepsilon'_\mathrm{d}) \neq \emptyset$ (which is equivalent to $\mathcal{J}^{\mathrm{c}}_{\mathrm{opt}}(\varepsilon'_\mathrm{d}) \neq \emptyset$ by Lemma 9.12).

Step 1: A comparison of Equations 9.2 and 9.3 shows that $\lambda_\mathrm{f}(\varepsilon'_\mathrm{d}) \geq \delta^{\mathrm{w}}_{\max}(\varepsilon'_\mathrm{d})$. Since we know that $\delta^{\mathrm{w}}_{\max}(\varepsilon'_\mathrm{d}) = \lambda_{\mathrm{d.s}}(\varepsilon_\mathrm{d}) - 1 \geq 0$, it follows that $\lambda_\mathrm{f}(\varepsilon'_\mathrm{d}) \geq 0$. By Lemma 9.13, $\varepsilon_\mathrm{d} \in \mathcal{E}^{\mathrm{opt}}_\mathrm{d}$ implies that $r_{\mathrm{in}}(J^{\mathrm{c}}_\mathrm{s}, T) > i$. Because $j < i$, this means that $\varepsilon'_\mathrm{d} = T[j]$ is not a w-in event, i.e., $\tau_{\mathrm{dep}}(\varepsilon'_\mathrm{d}) < \tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_\mathrm{s}, v'_\mathrm{d}, 1)$. Furthermore, $\delta^{\mathrm{w}}_{\max}(\varepsilon'_\mathrm{d}) \geq 0$ implies that $\tau_{\mathrm{dep}}(\varepsilon'_\mathrm{d}) \geq \tau^{\mathrm{c}}_{\mathrm{arr}}(v'_\mathrm{d})$. With Lemma 9.7, this yields

$$\lambda_\mathrm{j}(v'_\mathrm{d}) = \tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_\mathrm{s}, v'_\mathrm{d}, 1) - \tau^{\mathrm{c}}_{\mathrm{arr}}(v'_\mathrm{d}) > \tau_{\mathrm{dep}}(\varepsilon'_\mathrm{d}) - \tau^{\mathrm{c}}_{\mathrm{arr}}(v'_\mathrm{d}) \geq 0.$$

It follows from $\lambda_\mathrm{f}(\varepsilon'_\mathrm{d}) \geq 0$ and $\lambda_\mathrm{j}(v'_\mathrm{d}) \geq 0$ that $\varepsilon'_\mathrm{d} \in \mathcal{E}^{\mathrm{j/f}}$.

Step 2: Because ε_d is contained in $\mathcal{E}^{\mathrm{opt}}_\mathrm{d}$, the set $\mathcal{J}^{\mathrm{c}}_{\mathrm{virt}}(\varepsilon_\mathrm{d})$ must contain at least one candidate $J^{\mathrm{c}} = \langle [\varepsilon_\mathrm{s}, \varepsilon_\mathrm{o}], [\varepsilon_\mathrm{d}, \varepsilon_\mathrm{t}] \rangle$, which is prefix-optimal in $\Delta^{\mathrm{virt}}(J^{\mathrm{c}}_\mathrm{s})$. We show that $\overline{J^{\mathrm{c}}} = \langle [\varepsilon_\mathrm{s}, \varepsilon_\mathrm{o}], [\varepsilon'_\mathrm{d}, \varepsilon_\mathrm{t}] \rangle \in \mathcal{J}^{\mathrm{c}}(\varepsilon'_\mathrm{d})$ is also prefix-optimal in $\Delta^{\mathrm{virt}}(J^{\mathrm{c}}_\mathrm{s})$ by considering the prefixes individually:

Origin prefix: It follows from $\lambda_\mathrm{j}(v'_\mathrm{d}) \geq 0$ and Lemma 9.9 that the origin prefix $\langle [\varepsilon_\mathrm{s}, \varepsilon_\mathrm{o}], v'_\mathrm{d} \rangle$ is prefix-optimal in $\Delta^{\mathrm{virt}}(J^{\mathrm{c}}_\mathrm{s})$.

Destination prefix: Assume that the destination prefix $\overline{J^{\mathrm{c}}_\mathrm{d}} = \langle [\varepsilon_\mathrm{s}, \varepsilon_\mathrm{o}], T[j] \rangle$ is not Pareto-optimal in $\Delta^{\mathrm{virt}}(J^{\mathrm{c}}_\mathrm{s})$. Then there is a witness $J^{\mathrm{w}} = \langle [\varepsilon_\mathrm{s}, \varepsilon_\mathrm{o}], T[k] \rangle$ with $k < j$ that strongly dominates it. However, if J^{w} is feasible in $\Delta^{\mathrm{best}}(J^{\mathrm{c}}_\mathrm{s})$, it follows that $\delta^{\mathrm{w}}_{\max}(T[k]) = \delta^{\max} \geq \delta^{\mathrm{w}}_{\max}(T[j])$, which contradicts our choice of j.

Candidate: The candidate $\overline{J^{\mathrm{c}}}$ itself is Pareto-optimal in $\Delta^{\mathrm{virt}}(J^{\mathrm{c}}_\mathrm{s})$ since it shares the same target event as J^{c}.

Because $\mathcal{J}^{\mathrm{c}}_{\mathrm{virt}}(\varepsilon'_\mathrm{d})$ contains at least $\overline{J^{\mathrm{c}}}$, it is not empty. □

1t-Split Limit. Lemma 9.16 establishes a simple formula for the 1t-split limit.

Lemma 9.16. *Let $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle \in \mathcal{J}^c_{virt}(\varepsilon_d)$ be a candidate. Then*

$$\lambda_{1t.s}(\varepsilon_t) = \min(\delta^{max} + 1, \tau_{arr}(\varepsilon_t) - \tau^c_{arr}(v_t)).$$

Proof. The only possible direct $(\varepsilon_o, \varepsilon_t)$-witness for J^c is the journey $J^w = \langle [\varepsilon_s, \varepsilon_o], v_t \rangle$ that takes a final transfer directly from v_o to v_t. Since J^w does not use an intermediate transfer, it is feasible in $\Delta^{\text{best}}(J_s^c, \delta)$ for any $\delta \in [0, \delta^{\max} + 1]$. Its arrival time in $\Delta^{\text{best}}(J_s^c, \delta)$ is given by $\tau^c_{\text{arr}}(v_t) + \delta$, while the arrival time of J^c is given by $\tau_{\text{arr}}(\varepsilon_t)$. Hence, J^c is strongly dominated by a direct $(\varepsilon_o, \varepsilon_t)$-witness iff $\delta < \tau_{\text{arr}}(\varepsilon_t) - \tau^c_{\text{arr}}(v_t)$. Since $J^c \in \mathcal{J}^c_{\text{virt}}(\varepsilon_d)$, it follows that J^c is not strongly dominated by J^w in $\Delta^{\text{virt}}(J_s^c)$. Since $\Delta^{\text{virt}}(J_o^c) \preceq_{\text{dom}} \Delta^{\text{virt}}(J_s^c)$, it follows that $\tau_{\text{arr}}(\varepsilon_t) - \tau^c_{\text{arr}}(v_t) \geq 0$. □

2t-Split Limit. Determining the set of all indirect $(\varepsilon_o, \varepsilon_t)$-witnesses that strongly dominate J^c in $\Delta^{\text{best}}(J_s^c)$ is challenging because not all of them are Pareto-optimal in $\Delta^{\text{best}}(J_s^c)$. Finding all of them would potentially require examining many different delay scenarios, which would be expensive. To avoid this, we do not compute the 2t-split limit $\lambda_{2t.s}(\varepsilon_t)$ exactly. Instead, we compute a lower bound $\underline{\lambda_{2t.s}}(\varepsilon_t)$ that only considers the indirect $(\varepsilon_o, \varepsilon_t)$-witness with the lowest arrival time $\Delta^{\text{best}}(J_s^c)$. Other indirect $(\varepsilon_o, \varepsilon_t)$-witnesses, which may have a higher arrival time but a lower maximum witness delay, are ignored. For a vertex v, let $J^w(v)$ be the journey that minimizes the arrival time $\tau^w_{\text{arr}}(J_o^c, v, 2)$ in $\Delta^{\text{best}}(J_o^c)$ and let $\delta^w_{\max}(v)$ be its maximum witness delay. Given destination and target events $\varepsilon_d \in \mathcal{E}_d^{\text{opt}}$ and $\varepsilon_t \in \mathcal{E}_t^{\text{virt}}(\varepsilon_d)$, Lemma 9.17 shows that a lower bound for the 2t-split limit is given by

$$\underline{\lambda_{2t.s}}(\varepsilon_t) = \begin{cases} 0 & \text{if } \tau_{\text{arr}}(\varepsilon_t) \leq \tau^w_{\text{arr}}(J_o^c, v(\varepsilon_t), 2), \\ \delta^w_{\max}(v(\varepsilon_t)) + 1 & \text{otherwise.} \end{cases} \tag{9.4}$$

Lemma 9.17. *Let $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$ be a candidate for which $\Delta^{best}(J^c)$ is full-avoiding. Then*

$$\underline{\lambda_{2t.s}}(\varepsilon_t) \leq \lambda_{2t.s}(\varepsilon_t).$$

Proof. If $\tau_{\text{arr}}(\varepsilon_t) \leq \tau^w_{\text{arr}}(J_o^c, v(\varepsilon_t), 2)$, then $\underline{\lambda_{2t.s}}(\varepsilon_t) = 0$ and the claim is trivially true. We therefore assume that $\tau^w_{\text{arr}}(J_o^c, v(\varepsilon_t), 2) < \tau_{\text{arr}}(\varepsilon_t)$. The former is the arrival time of J^w in $\Delta^{\text{best}}(J_o^c)$ and the latter the arrival time of J^c in Δ^{best}, so it follows that J^w strongly dominates J^c in $\Delta^{\text{virt}}(J_o^c) = (\Delta^{\text{best}}, \Delta^{\text{best}}(J_o^c))$. Since $\Delta^{\text{best}} \preceq_{\text{eval}} \Delta^{\text{best}}(J^c) \preceq_{\text{eval}} \Delta^{\text{best}}(J_o^c)$, it follows that $\Delta^{\text{virt}}(J_o^c) \preceq_{\text{dom}} \Delta^{\text{best}}(J^c)$. Hence, J^w also strongly dominates J^c in $\Delta^{\text{best}}(J^c)$. Since $\Delta^{\text{best}}(J^c)$ is full-avoiding, J^w must be an indirect $(\varepsilon_o, \varepsilon_t)$-witness. By definition of $\delta^w_{\max}(v(\varepsilon_t)) = \delta^w_{\max}(J^w)$, J^w is still feasible in $\Delta^{\text{best}}(J_s^c, \delta^w_{\max}(v(\varepsilon_t)))$ and strongly dominates J^c. Hence, $\lambda_{2t.s}(\varepsilon_t) > \delta^w_{\max}(v(\varepsilon_t))$. □

Aggregating the Split Limits. The formulas established for the split limits allow us to calculate a lower bound for the minimum origin delay $\delta^o_{\min}(J^c)$ of a candidate J^c. To obtain a lower bound for the minimum origin delay $\delta^o_{\min}(\varepsilon_d)$ of a destination event $\varepsilon_d \in \mathcal{E}_d^{\text{opt}}$, the values for all candidates in $\mathcal{J}^c_{\text{opt}}(\varepsilon_d)$ must be aggregated. For this purpose, we define the *aggregated t-split limit*

$$\underline{\lambda^{\text{agg}}_{t.s}}(\varepsilon_d) := \min_{\varepsilon_t \in \mathcal{E}_t^{\text{virt}}(\varepsilon_d)} \max\left(\lambda_{1t.s}(\varepsilon_t), \underline{\lambda_{2t.s}}(\varepsilon_t)\right). \tag{9.5}$$

Lemma 9.18 shows that $\underline{\delta^{\mathrm{o}}_{\mathrm{min}}}(\varepsilon_{\mathrm{d}}) := \max\left(\lambda_{\mathrm{d.s}}(\varepsilon_{\mathrm{d}}), \underline{\lambda^{\mathrm{agg}}_{\mathrm{t.s}}}(\varepsilon_{\mathrm{d}})\right)$ is a lower bound for $\delta^{\mathrm{o}}_{\mathrm{min}}(\varepsilon_{\mathrm{d}})$.

Lemma 9.18. *For a destination event $\varepsilon_d \in \mathcal{E}_d^{opt}$, $\underline{\delta^o_{min}}(\varepsilon_d) \leq \delta^o_{min}(\varepsilon_d)$.*

Proof. Let $\mathcal{E}^{\mathrm{opt}}_{\mathrm{t}}(\varepsilon_{\mathrm{d}})$ be the set of target events that occur in $\mathcal{J}^{\mathrm{c}}_{\mathrm{opt}}(\varepsilon_{\mathrm{d}})$. We observe that

$$\delta^{\mathrm{o}}_{\mathrm{min}}(\varepsilon_{\mathrm{d}}) = \min_{J^{\mathrm{c}} \in \mathcal{J}^{\mathrm{c}}_{\mathrm{opt}}(\varepsilon_{\mathrm{d}})} \delta^{\mathrm{o}}_{\mathrm{min}}(J^{\mathrm{c}}) = \max\left(\lambda_{\mathrm{d.s}}(\varepsilon_{\mathrm{d}}), \min_{\varepsilon_{\mathrm{t}} \in \mathcal{E}^{\mathrm{opt}}_{\mathrm{t}}(\varepsilon_{\mathrm{d}})} \max\left(\lambda_{\mathrm{1t.s}}(\varepsilon_{\mathrm{t}}), \underline{\lambda_{\mathrm{2t.s}}}(\varepsilon_{\mathrm{t}})\right)\right).$$

It remains to be shown that

$$\underline{\lambda^{\mathrm{agg}}_{\mathrm{t.s}}}(\varepsilon_{\mathrm{d}}) = \min_{\varepsilon_{\mathrm{t}} \in \mathcal{E}^{\mathrm{virt}}_{\mathrm{t}}(\varepsilon_{\mathrm{d}})} \max\left(\lambda_{\mathrm{1t.s}}(\varepsilon_{\mathrm{t}}), \underline{\lambda_{\mathrm{2t.s}}}(\varepsilon_{\mathrm{t}})\right) \leq \min_{\varepsilon_{\mathrm{t}} \in \mathcal{E}^{\mathrm{opt}}_{\mathrm{t}}(\varepsilon_{\mathrm{d}})} \max\left(\lambda_{\mathrm{1t.s}}(\varepsilon_{\mathrm{t}}), \lambda_{\mathrm{2t.s}}(\varepsilon_{\mathrm{t}})\right).$$

This is the case because $\mathcal{E}^{\mathrm{virt}}_{\mathrm{t}}(\varepsilon_{\mathrm{d}}) \supseteq \mathcal{E}^{\mathrm{opt}}_{\mathrm{t}}(\varepsilon_{\mathrm{d}})$ by Lemma 9.10 and $\underline{\lambda_{\mathrm{2t.s}}}(\varepsilon_{\mathrm{t}}) \leq \lambda_{\mathrm{2t.s}}(\varepsilon_{\mathrm{t}})$ by Lemma 9.17. □

9.3.4 Preventing Time Travel

So far, we have focused on finding shortcuts that are required in at least one delay scenario. However, many delay scenarios cannot occur in reality because they require vehicles to travel faster than their maximum speed, or even backward in time. Unfortunately, prohibiting time travel introduces dependencies between the delays of stop events. This makes it much more complicated to characterize the conditions under which a candidate is prefix-optimal. We therefore eschew a full characterization and limit ourselves to two optimizations that prevent some types of time travel within the two candidate trips.

First Trip. Let $J^{\mathrm{c}}_{\mathrm{o}} = \langle [\varepsilon_{\mathrm{s}}, \varepsilon_{\mathrm{o}}], v_{\mathrm{d}} \rangle$ be an origin prefix with $\varepsilon_{\mathrm{o}} = T[i]$. By Lemma 9.9, the join limit $\lambda_{\mathrm{j}}(v_{\mathrm{d}})$ is non-negative iff $J^{\mathrm{c}}_{\mathrm{o}}$ is not strongly dominated by a join witness in the virtual delay scenario $\Delta^{\mathrm{virt}}(J^{\mathrm{c}}_{\mathrm{s}})$. Consider a join witness of the form $J^{\mathrm{w}} = \langle [\varepsilon_{\mathrm{s}}, T[j]], v_{\mathrm{d}} \rangle$ with $j < i$. Then $\Delta^{\mathrm{virt}}(J^{\mathrm{c}}_{\mathrm{s}})$ assumes maximum arrival delay for $T[j]$, but no arrival delay for $T[i]$. If the difference in the arrival times of both stop events is less than $\delta^{\max}$, this is physically impossible because T would have to travel backward in time. To avoid this type of time travel, we define the *time travel delay scenario* $\Delta^{\mathrm{tt}}(T[i])$. For stop events $T[i]$ and $T[j]$ of the same trip T, we define the arrival time difference

$$\tau^{\mathrm{diff}}_{\mathrm{arr}}(T, i, j) := \min\left(\delta^{\max}, \tau_{\mathrm{arr}}(T[i]) - \tau_{\mathrm{arr}}(T[j])\right).$$

Then the departure and arrival delay of a stop event ε in $\Delta^{\mathrm{tt}}(T[i])$ are given by

$$\begin{aligned} \Delta^{\mathrm{tt}}_{\mathrm{dep}}(T[i])(\varepsilon) &= \Delta^{\mathrm{best}}_{\mathrm{dep}}(J^{\mathrm{c}}_{\mathrm{s}})(\varepsilon) \\ \Delta^{\mathrm{tt}}_{\mathrm{arr}}(T[i])(\varepsilon) &= \begin{cases} \tau^{\mathrm{diff}}_{\mathrm{arr}}(T, i, j) & \text{if } \varepsilon = T[j] \text{ with } j < i, \\ \delta^{\max} & \text{otherwise.} \end{cases} \end{aligned} \tag{9.6}$$

For each vertex v, let $\tau^{\mathrm{tt}}_{\mathrm{arr}}(T[i], v)$ be the earliest arrival time at v among journeys with at most one trip that depart no earlier than $\tau_{\mathrm{dep}}(J^{\mathrm{c}}_{\mathrm{s}}, \Delta^{\mathrm{best}})$ in $\Delta^{\mathrm{tt}}(T[i])$.

Lemma 9.19. *Let $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$ be a candidate and $v \in \mathcal{S}$ a vertex visited by the intermediate transfer of J^c. If $\tau^c_{arr}(v) > \tau^{tt}_{arr}(T[i], v)$, then J^c is not prefix-optimal in any delay scenario without time travel.*

Proof. If $\tau^{\mathrm{c}}_{\mathrm{arr}}(v) > \tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{s}}, v, 1)$, i.e., $\lambda_{\mathrm{j}}(v) < 0$, then $\lambda_{\mathrm{j}}(J^{\mathrm{c}}) < 0$ by Lemma 9.8 and therefore J^{c} is not prefix-optimal in any delay scenario. Since $\Delta^{\mathrm{tt}}(T[i]) \preceq_{\mathrm{eval}} \Delta^{\mathrm{best}}(J^{\mathrm{c}}_{\mathrm{s}})$, we know that $\tau^{\mathrm{tt}}_{\mathrm{arr}}(T[i], v) \leq \tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{s}}, v, 1)$. Assume therefore that $\tau^{\mathrm{tt}}_{\mathrm{arr}}(T[i], v) < \tau^{\mathrm{c}}_{\mathrm{arr}}(v) \leq \tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{s}}, v, 1)$. Let $T[i] := \varepsilon_{\mathrm{o}}$ and let J^{w} be a v_{s}-v-journey with at most one trip and arrival time $\tau^{\mathrm{tt}}_{\mathrm{arr}}(T[i], v)$ at v that departs no later than J^{c} in $\Delta^{\mathrm{tt}}(T[i])$. Since the arrival time of J^{w} is earlier in $\Delta^{\mathrm{tt}}(T[i])$ than in $\Delta^{\mathrm{best}}(J^{\mathrm{c}}_{\mathrm{s}})$, the final stop event of J^{w} must be some $T[j]$ with $j < i$ and $\tau_{\mathrm{arr}}(T[i]) - \tau_{\mathrm{arr}}(T[j]) < \delta^{\max}$ must hold. In any delay scenario Δ without time travel, $\tau_{\mathrm{arr}}(\Delta, T[j]) \leq \tau_{\mathrm{arr}}(\Delta, T[i])$ must hold. This is equivalent to $\Delta_{\mathrm{arr}}(T[j]) - \Delta_{\mathrm{arr}}(T[i]) \leq \tau_{\mathrm{arr}}(T[i]) - \tau_{\mathrm{arr}}(T[j]) = \Delta^{\mathrm{tt}}_{\mathrm{arr}}(T[i])(T[j])$. It follows that

$$\begin{aligned}
\tau_{\mathrm{arr}}(\Delta, J^{\mathrm{w}}) &= \tau_{\mathrm{arr}}(\Delta^{\mathrm{tt}}(T[i]), J^{\mathrm{w}}) + \Delta_{\mathrm{arr}}(T[j]) - \Delta^{\mathrm{tt}}_{\mathrm{arr}}(T[i])(T[j]) \\
&= \tau^{\mathrm{tt}}_{\mathrm{arr}}(T[i], v) + \Delta_{\mathrm{arr}}(T[j]) - \Delta^{\mathrm{tt}}_{\mathrm{arr}}(T[i])(T[j]) \\
&< \tau^{\mathrm{c}}_{\mathrm{arr}}(v) + \Delta_{\mathrm{arr}}(T[j]) - \Delta^{\mathrm{tt}}_{\mathrm{arr}}(T[i])(T[j]) \\
&\leq \tau^{\mathrm{c}}_{\mathrm{arr}}(v) + \Delta_{\mathrm{arr}}(T[i]) \\
&= \tau_{\mathrm{arr}}(\Delta, J^{\mathrm{c}}_{\mathrm{o}}(v)).
\end{aligned}$$

Thus, J^{c} is not prefix-optimal in Δ because the prefix $J^{\mathrm{c}}_{\mathrm{o}}(v)$ is strongly dominated by J^{w}. □

Lemma 9.19 shows that if $\tau^{\mathrm{c}}_{\mathrm{arr}}(v) > \tau^{\mathrm{tt}}_{\mathrm{arr}}(T[i], v)$, then no candidate starting with $J^{\mathrm{c}}_{\mathrm{o}}(v)$ can be prefix-optimal in any delay scenario without time travel. Because $\tau^{\mathrm{tt}}_{\mathrm{arr}}(T[i], v) \leq \tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{s}}, v, 1)$ holds, this also implies $\lambda_{\mathrm{j}}(v) < 0$, so Lemma 9.19 is a stronger version of Lemma 9.8.

Second Trip. Consider a candidate $J^{\mathrm{c}} = \langle [\varepsilon_{\mathrm{s}}, \varepsilon_{\mathrm{o}}], [\varepsilon_{\mathrm{d}}, \varepsilon_{\mathrm{t}}] \rangle$. If J^{c} is Pareto-optimal in a delay scenario Δ, the arrival time of the earliest witness at v_{t} in Δ is an upper bound for the arrival time of ε_{t}. Because the second trip of J^{c} cannot travel backward in time, this is also an upper bound for the arrival time of the origin prefix $J^{\mathrm{c}}_{\mathrm{o}}$ at v_{d}. We define the *time travel limit*

$$\lambda_{\mathrm{tt}}(\varepsilon_{\mathrm{d}}, \varepsilon_{\mathrm{t}}) := \tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{s}}, v(\varepsilon_{\mathrm{t}}), 2) - \tau^{\mathrm{c}}_{\mathrm{arr}}(v(\varepsilon_{\mathrm{d}})).$$

Lemma 9.20. *Let $J^c = \langle [\varepsilon_s, \varepsilon_o], [\varepsilon_d, \varepsilon_t] \rangle$ be a candidate and Δ a delay scenario without time travel. If J^c is prefix-optimal in Δ, then $\tau_{arr}(\Delta, \varepsilon_o) \leq \lambda_{tt}(\varepsilon_d, \varepsilon_t)$.*

Proof. Assume that $J^{\mathrm{c}} = \langle [\varepsilon_{\mathrm{s}}, \varepsilon_{\mathrm{o}}], [\varepsilon_{\mathrm{d}}, \varepsilon_{\mathrm{t}}] \rangle$ is prefix-optimal in Δ. Then $J^{\mathrm{c}} \in \mathcal{J}^{\mathrm{c}}_{\mathrm{opt}}(\varepsilon_{\mathrm{d}}) \subseteq \mathcal{J}^{\mathrm{c}}_{\mathrm{virt}}(\varepsilon_{\mathrm{d}})$ and therefore $\tau_{\mathrm{arr}}(\Delta, \varepsilon_{\mathrm{t}}) \leq \tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{s}}, v_{\mathrm{t}}, 2)$ must hold. Since no time travel occurs in Δ, it follows that $\tau_{\mathrm{dep}}(\Delta, \varepsilon_{\mathrm{d}}) \leq \tau_{\mathrm{arr}}(\Delta, \varepsilon_{\mathrm{t}})$. Finally, since J^{c} is feasible, $\tau_{\mathrm{arr}}(\Delta, J^{\mathrm{c}}_{\mathrm{o}}) = \tau^{\mathrm{c}}_{\mathrm{arr}}(v_{\mathrm{d}}) + \Delta_{\mathrm{arr}}(\varepsilon_{\mathrm{o}}) \leq \tau_{\mathrm{dep}}(\Delta, \varepsilon_{\mathrm{d}})$ must hold. Altogether, this yields $\Delta_{\mathrm{arr}}(\varepsilon_{\mathrm{o}}) \leq \tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{s}}, v_{\mathrm{t}}, 2) - \tau^{\mathrm{c}}_{\mathrm{arr}}(v_{\mathrm{d}}) = \lambda_{\mathrm{tt}}(\varepsilon_{\mathrm{d}}, \varepsilon_{\mathrm{t}})$. □

For a destination event $\varepsilon_d \in \mathcal{E}_d^{\text{opt}}$, we define the *maximum witness time*

$$\tau^{\text{w}}_{\text{max}}(\varepsilon_d) := \max_{\varepsilon_t \in \mathcal{E}_t^{\text{virt}}(\varepsilon_d)} \tau^{\text{w}}_{\text{arr}}(J_s^c, v(\varepsilon_t), 2) \tag{9.7}$$

and the *aggregated time travel limit*

$$\lambda^{\text{agg}}_{\text{tt}}(\varepsilon_d) := \max_{\varepsilon_t \in \mathcal{E}_t^{\text{virt}}(\varepsilon_d)} \lambda_{\text{tt}}(\varepsilon_d, \varepsilon_t) = \tau^{\text{w}}_{\text{max}}(\varepsilon_d) - \tau^{\text{c}}_{\text{arr}}(v(\varepsilon_d)). \tag{9.8}$$

If the delay of ε_o in a delay scenario Δ without time travel exceeds $\lambda^{\text{agg}}_{\text{tt}}(\varepsilon_d)$, then by Lemma 9.20 the shortcut $(\varepsilon_o, \varepsilon_d)$ is not required in Δ. We therefore redefine the maximum origin delay as

$$\delta^{\text{o}}_{\text{max}}(\varepsilon_d) := \min\left\{\lambda_f(\varepsilon_d), \lambda_j(v(\varepsilon_d)), \lambda^{\text{agg}}_{\text{tt}}(\varepsilon_d)\right\}.$$

9.4 Delay-ULTRA Shortcut Computation

The Delay-ULTRA shortcut computation retains the basic outline of ULTRA: For each source stop v_s (in parallel), it enumerates all prefix-optimal candidates originating at v_s. For each possible candidate departure time τ_j at v_s (in descending order), candidates departing at τ_j and witnesses departing within $[\tau_j, \tau_{j+1})$ are explored. ULTRA does this with an MR run that is limited to the first two rounds. Delay-ULTRA must additionally take different delay scenarios into account. Section 9.3 showed that three delay scenarios are sufficient to examine a candidate J^c: The candidate itself is explored in Δ^{best}. Witnesses are explored in $\Delta^{\text{best}}(J_s^c)$ and $\Delta^{\text{best}}(J_o^c)$, yielding arrival times $\tau^{\text{w}}_{\text{arr}}(J_s^c, \cdot, \cdot)$ and $\tau^{\text{w}}_{\text{arr}}(J_o^c, \cdot, \cdot)$, respectively. While Δ^{best} is the same for all candidates, the witness delay scenarios depend on the candidate prefixes. Thus, instead of a single witness search per run, Delay-ULTRA performs one for each source and origin prefix.

Self-Pruning. To avoid redundant work, Delay-ULTRA performs the witness searches in a staggered manner and extends the self-pruning approach used by rRAPTOR and ULTRA. A run with departure time τ_j begins with a two-round MR witness search in Δ^{worst}. For each vertex v and round $n \leq 2$, this finds the earliest arrival time $\tau^{\text{w}}_{\text{arr}}(\tau_j, v, n)$ among journeys to v with at most n trips that depart no earlier than τ_j. The original self-pruning rule is applied here: Because $\tau^{\text{w}}_{\text{arr}}(\tau_j, v, n) \leq \tau^{\text{w}}_{\text{arr}}(\tau_{j+1}, v, n)$ holds for each vertex v and round n, the search for τ_j initializes the former with the latter and only explores journeys departing before τ_{j+1}. Since $\tau^{\text{w}}_{\text{arr}}(J_s^c, v, n) \leq \tau^{\text{w}}_{\text{arr}}(\tau_j, v, n)$ holds as well, the witness search in $\Delta^{\text{best}}(J_s^c)$ reuses the results for Δ^{worst} in the same manner. Moreover, since the two scenarios only differ in the departure delay of ε_s, the witness search in $\Delta^{\text{best}}(J_s^c)$ only explores journeys starting with J_s^c. By the same argument, the witness search for $\Delta^{\text{best}}(J_o^c)$ reuses the results for $\Delta^{\text{best}}(J_s^c)$ and only explores journeys starting with J_o^c. Thus, each witness search is pruned with results from the previous ones.

Algorithm 9.1: Delay-ULTRA shortcut computation.

Input: Public transit network $(\mathcal{S}, \Pi, \mathcal{E}, \mathcal{T}, \mathcal{R}, G)$, source stop $v_s \in \mathcal{S}$
Output: Shortcuts E^s

1 $\mathcal{D} \leftarrow$ Collect departures of trips at v_s
2 **for each** $(\tau_{dep}, \mathcal{E}_s) \in \mathcal{D}$ in descending order of τ_{dep} **do**
3 $\quad \tau^w_{arr}(\tau_{dep}, \cdot, \cdot), r_{in}(\tau_{dep}, \cdot) \leftarrow$ Find witnesses in Δ^{worst}
4 $\quad$ **for each** $\varepsilon_s \in \mathcal{E}_s$ **do**
$\quad\quad$ `// Solve DelayShortcut-`$J^c_o(v(\varepsilon_o))$
5 $\quad\quad \tau^w_{arr}(J^c_s, \cdot, \cdot), r_{in}(J^c_s, \cdot) \leftarrow$ Find witnesses in $\Delta^{best}(J^c_s)$
6 $\quad\quad$ **for each** $\varepsilon_o \in \overrightarrow{\mathcal{E}}(\varepsilon_s)$ **do**
7 $\quad\quad\quad \tau^c_{arr}(\cdot), \lambda_j(\cdot) \leftarrow$ Explore intermediate transfers of candidates
8 $\quad\quad\quad r_{out}(J^c_s, \cdot) \leftarrow$ Compute exit indices
9 $\quad\quad\quad \mathcal{E}^{opt}_d, \lambda_f(\cdot) \leftarrow$ Find destination events
10 $\quad\quad\quad \lambda_{d.s}(\cdot) \leftarrow$ Compute d-split limits
11 $\quad\quad\quad \tau^w_{arr}(J^c_o, \cdot, \cdot), \delta^w_{max}(\cdot) \leftarrow$ Find witnesses in $\Delta^{best}(J^c_o)$
12 $\quad\quad\quad E^s_{new}, \underline{\delta^o_{min}}(\cdot), \delta^o_{max}(\cdot) \leftarrow$ Scan second trips of candidates
13 $\quad\quad\quad$ Merge E^s_{new} into E^s

A similar approach is used for the entry index $r_{in}(J^c_s, T)$ of each trip T. Whereas $r_{in}(J^c_s, T)$ is based on witness arrival times in $\Delta^{best}(J^c_s)$, we define $r_{in}(\tau_j, T)$ as the entry index based on witnesses in Δ^{worst} that depart no earlier than τ_j. To compute it, we augment the MR witness search as follows: whenever a stop event $T[i]$ is entered during a route scan, $r_{in}(\tau_j, T')$ is updated to $\min(r_{in}(\tau_j, T'), i)$ for all trips $T \preceq T'$. Based on the inequalities $r_{in}(J^c_s, T) \geq r_{in}(\tau_j, T) \geq r_{in}(\tau_{j+1}, T)$, self-pruning is applied to the entry indices as well.

9.4.1 Overview

Algorithm 9.1 shows a high-level overview of the shortcut computation for a source stop v_s. Further details are provided in Section 9.4.2. Line 1 collects all stop events departing at v_s in a set $\mathcal{D}$, grouped by their departure time. For a tuple $(\tau_{dep}, \mathcal{E}_s) \in \mathcal{D}$, $\mathcal{E}_s$ contains all stop events departing at v_s with scheduled departure time $\tau_{dep} - \delta^{max}$. These tuples are processed in descending order of departure time. Line 3 computes $\tau^w_{arr}(\tau_{dep}, \cdot, \cdot)$ and $r_{in}(\tau_{dep}, \cdot)$ with a witness search in Δ^{worst}. For each source event $\varepsilon_s \in \mathcal{E}_s$, the algorithm then explores candidates beginning with the source prefix $J^c_s = \langle \varepsilon_s \rangle$. Line 5 computes $\tau^w_{arr}(J^c_s, \cdot, \cdot)$ and $r_{in}(J^c_s, \cdot)$ with a self-pruned witness search in $\Delta^{best}(J^c_s)$. For each event ε_o succeeding ε_s in $T(\varepsilon_s)$, the algorithm now solves the subproblem DelayShortcut-$J^c_o(v_o)$ for the origin stop prefix $J^c_o(v_o) = \langle [\varepsilon_s, \varepsilon_o], v_o \rangle$ with origin stop $v_o = v(\varepsilon_o)$.

Line 7 performs a Dijkstra search from v_o in Δ^{best}. For each vertex v, this computes the candidate arrival time $\tau^c_{arr}(v)$ and the join limit $\lambda_j(v)$ by Equation 9.1. For each trip T that can

be entered by a candidate, line 8 computes the exit index $r_{\text{out}}(J_s^c, T)$. For each event $\varepsilon_d := T[i]$ with $i < r_w(J_s^c, T)$, line 9 computes the feasibility limit $\lambda_f(\varepsilon_d)$ by Equation 9.2. If it is non-negative, then ε_d is added to $\mathcal{E}_d^{\text{opt}}$. Line 10 computes the d-split limits for the events in $\mathcal{E}_d^{\text{opt}}$ by Lemma 9.15. For the t-split limits, line 11 computes the witness arrival times $\tau_{\text{arr}}^w(J_o^c, \cdot, \cdot)$ and the maximum witness delays $\delta_{\text{max}}^w(\cdot)$ with a modified MR search in $\Delta^{\text{best}}(J_o^c)$. For each destination event $\varepsilon_d \in \mathcal{E}_d^{\text{opt}}$, line 12 calculates the aggregated t-split limit $\underline{\lambda_{\text{t.s}}^{\text{agg}}}(\varepsilon_d)$ by Equation 9.5 and the aggregated time travel limit $\lambda_{\text{tt}}^{\text{agg}}(\varepsilon_d)$ by Equation 9.8. From these, the maximum origin delay $\delta_{\text{max}}^o(\varepsilon_d)$ and a lower bound $\underline{\delta_{\text{min}}^o}(\varepsilon_d)$ for the minimum origin delay are computed. If $\underline{\delta_{\text{min}}^o}(\varepsilon_d) \leq \delta_{\text{max}}^o(\varepsilon_d)$, the shortcut $(\varepsilon_o, \varepsilon_d)$ is generated and merged into the result set E^s in line 13.

9.4.2 Details

We now discuss further details that are not shown in Algorithm 9.1.

Exploring Intermediate Transfers. Line 7 of Algorithm 9.1 explores the intermediate transfers of candidates beginning with the origin stop prefix $J_o^c(v_o)$. This is done with a Dijkstra search starting from v_o in the best-case delay scenario Δ^{best}. For each vertex v visited by the search, this yields the candidate arrival time $\tau_{\text{arr}}^c(v)$ and the join limit $\lambda_j(v)$ by Equation 9.1. By Lemma 9.8, if $\lambda_j(v) < 0$, then all candidates beginning with $J_o^c(v)$ are irrelevant because they have a negative join limit as well. In this case, the search is pruned at v, i.e., the outgoing edges are not explored. For an even stronger pruning rule, a Dijkstra search from v_o in the time travel delay scenario $\Delta^{\text{tt}}(\varepsilon_o)$ can be performed before line 7 to compute $\tau_{\text{arr}}^{\text{tt}}(\varepsilon_o, v)$ for each vertex v. Then by Lemma 9.19, the candidate search can be pruned at a vertex v if $\tau_{\text{arr}}^{\text{tt}}(\varepsilon_o, v) < 0$.

Exit Indices. Line 8 of Algorithm 9.1 collects all routes that can be entered by a candidate and calculates the exit indices of their trips. A route R can be entered if its last trip T_{max} can be entered, i.e., there is at least one stop event $T_{\text{max}}[i]$ such that $\tau_{\text{arr}}^c(v(T_{\text{max}}[i])) \leq \tau_{\text{dep}}(T_{\text{max}}[i]) + \delta^{\text{max}}$. For each trip T of R, the exit index $r_{\text{out}}(J_s^c, T)$ is then calculated. Since this value is based on the delay scenario $\Delta^{\text{best}}(J_s^c)$, which does not depend on the choice of ε_o, the value of $r_{\text{out}}(J_s^c, T)$ cannot change between iterations of the loop in line 6. To avoid unnecessary recalculations, the algorithm maintains a timestamp for each route R that indicates whether the exit indices for the trips of R were already calculated for the current choice of ε_s.

Finding Destination Events. Line 9 of Algorithm 9.1 computes the set $\mathcal{E}_d^{\text{opt}}$ of relevant destination events and the feasibility limit $\lambda_f(\varepsilon_d)$ for each destination event $\varepsilon_d \in \mathcal{E}_d^{\text{opt}}$. By Lemma 9.13, a stop event $T[i] \in \mathcal{E}^{j/f}$ is contained in $\mathcal{E}_d^{\text{opt}}$ iff $i < r_w(J_s^c, T)$. To find the stop events fulfilling this condition efficiently, each route collected in line 8 is processed

Algorithm 9.2: Find destination events.

Input: Route R, first reachable stop index j
Output: Destination events $\mathcal{E}_{\mathrm{d}}^{\mathrm{opt}}$, feasibility limits $\lambda_{\mathrm{f}}(\cdot)$

```
1  T_b ← first trip of R
2  T_e ← last trip of R
3  for j ≤ i < r_w(J_s^c, T_b) in ascending order do
4      while r_w(J_s^c, T_e) ≤ i do
5          T_e ← predecessor of T_e in R
6      for T_e ⪰ T ⪰ T_b in descending order do
7          ε_d ← T[i]
8          ν_d ← ν(ε_d)
9          λ_f(ε_d) ← min(0, τ_dep(ε_d) − τ_arr^c(ν_d)) + δ^max
10         if λ_f(ε_d) < 0 then break
11         E_d^opt ← E_d^opt ∪ {ε_d}
```

individually. Detailed pseudocode for the examination of a route R is given in Algorithm 9.2. Lines 1 and 2 initialize the variables T_b and T_e, which initially refer to the first and last trip of R, respectively. The algorithm then iterates over the stops of R, starting at the first stop that was reached by the Dijkstra search in line 7 of Algorithm 9.1 and ending at $r_{\mathrm{w}}(J_{\mathrm{s}}^{\mathrm{c}}, T_b) - 1$. For each stop index i, line 4 decreases T_e until $r_{\mathrm{w}}(J_{\mathrm{s}}^{\mathrm{c}}, T_e) > i$ holds. This ensures that the trips after T_e, which contain no further stop events in $\mathcal{E}_{\mathrm{d}}^{\mathrm{opt}}$, are ignored. Then, in descending order from T_e to T_b, the algorithm examines for each trip T the destination event $\varepsilon_{\mathrm{d}} := T[i]$ departing at the current stop. Line 9 computes the feasibility limit $\lambda_{\mathrm{f}}(\varepsilon_{\mathrm{d}})$ according to Equation 9.2. If this is below 0, then $\varepsilon_{\mathrm{d}} \notin \mathcal{E}^{\mathrm{j/f}}$, so it can be discarded. Since the feasibility limit will also be below 0 for all of the trips preceding T, the loop is exited altogether. Otherwise, ε_{d} is added to $\mathcal{E}_{\mathrm{d}}^{\mathrm{opt}}$ in line 11.

d-Split Limit. Line 10 of Algorithm 9.1 computes the d-split limits for the events in $\mathcal{E}_{\mathrm{d}}^{\mathrm{opt}}$. By Equation 9.15, the d-split limit of a destination event $T[i]$ depends on the maximum witness delays of the events in $\mathcal{E}_{\mathrm{d}}^{\mathrm{opt}}$ that precede it in T. For each trip T with at least one stop event in $\mathcal{E}_{\mathrm{d}}^{\mathrm{opt}}$, the algorithm sweeps over its stop events from first to last and maintains a value $\lambda_{\mathrm{d.s}}(T)$, which is initialized with 0. If a stop event $T[i]$ is not in $\mathcal{E}_{\mathrm{d}}^{\mathrm{opt}}$, it is skipped. Otherwise, its d-split limit $\lambda_{\mathrm{d.s}}(T[i])$ is set to $\lambda_{\mathrm{d.s}}(T)$. Afterward, the maximum witness delay $\delta_{\mathrm{max}}^{\mathrm{w}}(T[i])$ is calculated according to Equation 9.3 and $\lambda_{\mathrm{d.s}}(T)$ is set to the maximum of itself and $\delta_{\mathrm{max}}^{\mathrm{w}}(T[i]) + 1$.

Modified MR Search for $(\varepsilon_{\mathbf{o}}, \varepsilon_{\mathbf{t}})$-Witnesses. Line 11 of Algorithm 9.1 performs a witness search in $\Delta^{\mathrm{best}}(J_{\mathrm{o}}^{\mathrm{c}})$ to calculate the witness arrival times $\tau_{\mathrm{arr}}^{\mathrm{w}}(J_{\mathrm{o}}^{\mathrm{c}}, \cdot, \cdot)$ and the maximum witness delays $\delta_{\mathrm{max}}^{\mathrm{w}}(\cdot)$. This is done with an MR search that is modified as follows.

Since $\tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{o}}, v, 1) = \min(\tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{s}}, v, 1), \tau^{\mathrm{c}}_{\mathrm{arr}}(v))$ holds for each vertex v, the first MR round is skipped and $\tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{o}}, v, 1)$ is initialized accordingly. Afterward, a second MR round is performed to find second trips and final transfers. For each vertex v, this search calculates the earliest arrival time $\tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{o}}, v, 2)$ at v among journeys with at most two trips in $\Delta^{\mathrm{best}}(J^{\mathrm{c}}_{\mathrm{o}})$, as well as the maximum witness delay $\delta^{\mathrm{w}}_{\max}(v)$ of the corresponding witness. This search finds exactly those witnesses that include an intermediate transfer from ε_{o} to a destination event $\varepsilon'_{\mathrm{d}}$. Since all other witnesses have maximum witness delay $\delta^{\max}$, $\delta^{\mathrm{w}}_{\max}(v)$ is initialized with $\delta^{\max}$ for each vertex v at the start of the search.

To calculate the maximum witness delays, the scanning procedure for a route R is modified as follows. In addition to the active trip T_{min}, it maintains an *active maximum witness delay* $\delta^{\mathrm{w}}_{\max}(T_{\mathsf{min}})$, which is initialized with $\delta^{\max}$. If T_{min} is improved by entering R at a destination event $\varepsilon'_{\mathrm{d}}$, then $\delta^{\mathrm{w}}_{\max}(T_{\mathsf{min}})$ is set to $\delta^{\mathrm{w}}_{\max}(\varepsilon'_{\mathrm{d}})$, which is calculated according to Equation 9.3. If exiting R at a stop v improves the arrival time $\tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{o}}, v, 2)$, then $\delta^{\mathrm{w}}_{\max}(v)$ is set to $\delta^{\mathrm{w}}_{\max}(T_{\mathsf{min}})$. During the Dijkstra search, when an edge (v, w) is relaxed and $\tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{o}}, w, 2)$ is improved, then $\delta^{\mathrm{w}}_{\max}(w)$ is set to $\delta^{\mathrm{w}}_{\max}(v)$. When a new iteration of the loop in line 6 is started, i.e., a new origin event is processed, $\tau^{\mathrm{w}}_{\mathrm{arr}}(J^{\mathrm{c}}_{\mathrm{o}}, \cdot, \cdot)$ and $\delta^{\mathrm{w}}_{\max}(\cdot)$ are reset.

Second Candidate Trip Scan. Line 12 of Algorithm 9.1 scans all trips that contain at least one destination event $\varepsilon_{\mathrm{d}} \in \mathcal{E}^{\mathrm{opt}}_{\mathrm{d}}$. This scan calculates a lower bound $\underline{\delta^{\mathrm{o}}_{\mathrm{min}}}(\varepsilon_{\mathrm{d}})$ for the minimum origin delay and the maximum origin delay $\delta^{\mathrm{o}}_{\max}(\varepsilon_{\mathrm{d}})$ and generates the shortcut $(\varepsilon_{\mathrm{o}}, \varepsilon_{\mathrm{d}})$ if it is required. Detailed pseudocode for the scan of a trip T is given in Algorithm 9.3. For each stop event $T[i] \in \mathcal{E}^{\mathrm{opt}}_{\mathrm{d}}$, the algorithm must calculate a lower bound $\underline{\lambda^{\mathrm{agg}}_{\mathrm{t.s}}}(T[i])$ for the aggregated target split limit by Equation 9.5 and the aggregated time travel limit $\lambda^{\mathrm{agg}}_{\mathrm{tt}}(T[i])$ by Equation 9.8. The latter in turn requires the maximum witness time $\tau^{\mathrm{w}}_{\max}(T[i])$, which is calculated by Equation 9.7. Computing these values requires aggregating over the set $\mathcal{E}^{\mathrm{virt}}_{\mathrm{t}}(T[i])$ of relevant target events. According to Lemma 9.14, this is given by $\mathcal{E}^{\mathrm{out}}_{\mathrm{c}}(J^{\mathrm{c}}_{\mathrm{s}}, T) \cap \overrightarrow{\mathcal{E}}(T[i])$, i.e., the set of c-out events succeeding $T[i]$ in T.

In order to consider exactly the set $\mathcal{E}^{\mathrm{virt}}_{\mathrm{t}}(T[i])$ for each destination event $T[i]$, the procedure performs one sweep across the stop events of T in reverse, starting at the index $r_{\mathrm{out}}(J^{\mathrm{c}}_{\mathrm{s}}, T) - 1$ and ending at the first index j for which $T[j] \in \mathcal{E}^{\mathrm{opt}}_{\mathrm{d}}$. The algorithm maintains the values $\underline{\lambda^{\mathrm{agg}}_{\mathrm{t.s}}}(T)$ and $\tau^{\mathrm{w}}_{\max}(T)$, which fulfill the invariants that $\underline{\lambda^{\mathrm{agg}}_{\mathrm{t.s}}}(T) = \underline{\lambda^{\mathrm{agg}}_{\mathrm{t.s}}}(T[i])$ and $\tau^{\mathrm{w}}_{\max}(T) = \tau^{\mathrm{w}}_{\max}(T[i])$ at the point when $T[i]$ is scanned. They are initialized with ∞ in line 1 and $-\infty$ in line 2, respectively. In the step for destination stop index i, the target event $\varepsilon_{\mathrm{t}} := T[i+1]$ is examined. Line 7 tests whether ε_{t} is a c-out event. If so, then $\lambda_{1\mathrm{t.s}}(\varepsilon_{\mathrm{t}})$ and $\underline{\lambda_{2\mathrm{t.s}}}(\varepsilon_{\mathrm{t}})$ are calculated in lines 8–9. Lines 10 and 11 updates $\underline{\lambda^{\mathrm{agg}}_{\mathrm{t.s}}}(T)$ and $\tau^{\mathrm{w}}_{\max}(T)$, respectively, to uphold the invariants. Line 14 incorporates $\underline{\lambda^{\mathrm{agg}}_{\mathrm{t.s}}}(T)$ into $\underline{\delta^{\mathrm{o}}_{\mathrm{min}}}(\varepsilon_{\mathrm{d}})$. Line 15 calculates $\lambda^{\mathrm{agg}}_{\mathrm{tt}}(\varepsilon_{\mathrm{d}})$ from $\tau^{\mathrm{w}}_{\max}(T)$ and line 16 incorporates this into $\delta^{\mathrm{o}}_{\max}(\varepsilon_{\mathrm{d}})$. If $\underline{\delta^{\mathrm{o}}_{\mathrm{min}}}(\varepsilon_{\mathrm{d}}) \leq \delta^{\mathrm{o}}_{\max}(\varepsilon_{\mathrm{d}})$, the shortcut $(\varepsilon_{\mathrm{o}}, \varepsilon_{\mathrm{d}})$ is generated in line 18.

Algorithm 9.3: Scan second candidate trip.

Input: Trip T, destination events $\mathcal{E}_{\mathrm{d}}^{\mathrm{opt}}$
Output: New shortcut edges $E_{\mathrm{new}}^{\mathrm{s}}$, lower bounds $\underline{\delta_{\mathrm{min}}^{\mathrm{o}}}(\cdot)$ for the minimum origin delay, maximum origin delays $\delta_{\mathrm{max}}^{\mathrm{o}}(\cdot)$

1 $\underline{\lambda_{\mathrm{t.s}}^{\mathrm{agg}}}(T) \leftarrow \infty$
2 $\tau_{\mathrm{max}}^{\mathrm{w}}(T) \leftarrow -\infty$
3 $j \leftarrow \min\left\{i \mid T[i] \in \mathcal{E}_{\mathrm{d}}^{\mathrm{opt}}\right\}$
4 **for** $r_{\mathrm{out}}(J_{\mathrm{s}}^{\mathrm{c}}, T) > i \geq j$ in descending order **do**
5 $\quad \varepsilon_{\mathrm{t}} \leftarrow T[i+1]$
6 $\quad v_{\mathrm{t}} \leftarrow v(\varepsilon_{\mathrm{t}})$
7 $\quad$ **if** $\tau_{\mathrm{arr}}(\varepsilon_{\mathrm{t}}) \leq \tau_{\mathrm{arr}}^{\mathrm{w}}(J_{\mathrm{s}}^{\mathrm{c}}, v_{\mathrm{t}}, 2)$ **then**
8 $\quad\quad$ Calculate $\lambda_{\mathrm{1t.s}}(\varepsilon_{\mathrm{t}})$ by Lemma 9.16
9 $\quad\quad$ Calculate $\underline{\lambda_{\mathrm{2t.s}}}(\varepsilon_{\mathrm{t}})$ by Equation 9.4
10 $\quad\quad \underline{\lambda_{\mathrm{t.s}}^{\mathrm{agg}}}(T) \leftarrow \min(\underline{\lambda_{\mathrm{t.s}}^{\mathrm{agg}}}(T), \max(\lambda_{\mathrm{1t.s}}(\varepsilon_{\mathrm{t}}), \underline{\lambda_{\mathrm{2t.s}}}(\varepsilon_{\mathrm{t}})))$
11 $\quad\quad \tau_{\mathrm{max}}^{\mathrm{w}}(T) \leftarrow \max(\tau_{\mathrm{max}}^{\mathrm{w}}(T), \tau_{\mathrm{arr}}^{\mathrm{w}}(J_{\mathrm{s}}^{\mathrm{c}}, v_{\mathrm{t}}, 2))$
12 $\quad \varepsilon_{\mathrm{d}} \leftarrow T[i]$
13 $\quad v_{\mathrm{d}} \leftarrow v(\varepsilon_{\mathrm{d}})$
14 $\quad \underline{\delta_{\mathrm{min}}^{\mathrm{o}}}(\varepsilon_{\mathrm{d}}) \leftarrow \max(\lambda_{\mathrm{d.s}}(\varepsilon_{\mathrm{d}}), \underline{\lambda_{\mathrm{t.s}}^{\mathrm{agg}}}(T))$
15 $\quad \lambda_{\mathrm{tt}}^{\mathrm{agg}}(\varepsilon_{\mathrm{d}}) \leftarrow \tau_{\mathrm{max}}^{\mathrm{w}}(T) - \tau_{\mathrm{arr}}^{\mathrm{c}}(v_{\mathrm{d}})$
16 $\quad \delta_{\mathrm{max}}^{\mathrm{o}}(\varepsilon_{\mathrm{d}}) \leftarrow \min\left\{\lambda_{\mathrm{f}}(\varepsilon_{\mathrm{d}}), \lambda_{\mathrm{j}}(v_{\mathrm{d}}), \lambda_{\mathrm{tt}}^{\mathrm{agg}}(\varepsilon_{\mathrm{d}})\right\}$
17 $\quad$ **if** $\underline{\delta_{\mathrm{min}}^{\mathrm{o}}}(\varepsilon_{\mathrm{d}}) \leq \delta_{\mathrm{max}}^{\mathrm{o}}(\varepsilon_{\mathrm{d}})$ **then**
18 $\quad\quad E_{\mathrm{new}}^{\mathrm{s}} \leftarrow E_{\mathrm{new}}^{\mathrm{s}} \cup \left\{(\varepsilon_{\mathrm{o}}, \varepsilon_{\mathrm{d}})\right\}$

Merging Shortcuts. Line 13 of Algorithm 9.1 merges the newly generated shortcuts $E_{\mathrm{new}}^{\mathrm{s}}$ into the result set E^{s}. If a shortcut $e = (\varepsilon_{\mathrm{o}}, \varepsilon_{\mathrm{d}}) \in E_{\mathrm{new}}^{\mathrm{s}}$ is already contained in E^{s}, then $\underline{\delta_{\mathrm{min}}^{\mathrm{o}}}(e)$ is set to $\min(\underline{\delta_{\mathrm{min}}^{\mathrm{o}}}(e), \underline{\delta_{\mathrm{min}}^{\mathrm{o}}}(\varepsilon_{\mathrm{d}}))$ and $\delta_{\mathrm{max}}^{\mathrm{o}}(e)$ to $\max(\delta_{\mathrm{max}}^{\mathrm{o}}(e), \delta_{\mathrm{max}}^{\mathrm{o}}(\varepsilon_{\mathrm{d}}))$. Otherwise, they are initialized with $\underline{\delta_{\mathrm{min}}^{\mathrm{o}}}(\varepsilon_{\mathrm{d}})$ and $\delta_{\mathrm{max}}^{\mathrm{o}}(\varepsilon_{\mathrm{d}})$ and the shortcut is added to E^{s}.

9.5 Update Phase

The update phase incorporates the delay update stream $\langle \mu_1, \mu_2, \dots \rangle$ into the query data structures. We present two variants of the update phase: a basic one in Section 9.5.1 that only updates the data structures and removes irrelevant shortcuts, and an advanced version in Section 9.5.2 that searches for missing shortcuts. For simplicity, we assume that each update is processed individually. In practice, this may not be viable because updates arrive too frequently. Thus, updates that are received while an update phase is running are buffered. Once it has finished, the buffered updates are combined into a single input for the next phase.

9.5.1 Basic

For a delay update μ, the basic update phase first computes the new delay scenario Δ by incorporating μ into the previous scenario. It then recalculates the set of routes such that no trips of the same route overtake each other in Δ and rebuilds the TB query data structures accordingly. Finally, two types of irrelevant shortcuts are removed from the set E^{s} of Delay-ULTRA shortcuts: The first is the set

$$E^{\mathrm{s}}_{\mathrm{inf}}(\Delta) := \left\{ e = (\varepsilon_{\mathrm{o}}, \varepsilon_{\mathrm{d}}) \in E^{\mathrm{s}} \mid \tau_{\mathrm{dep}}(\Delta, \varepsilon_{\mathrm{d}}) < \tau_{\mathrm{arr}}(\Delta, \varepsilon_{\mathrm{o}}) + \tau_{\mathrm{tra}}(e) \right\}$$

of shortcuts that are infeasible in Δ. The second type is shortcuts $e = (\varepsilon_{\mathrm{o}}, \varepsilon_{\mathrm{d}})$ for which the arrival delay $\Delta_{\mathrm{arr}}(\varepsilon_{\mathrm{o}})$ is not within the computed origin delay interval $\underline{I^{\mathrm{o}}_{\delta}}(e)$.

9.5.2 Advanced

The advanced variant searches for shortcuts that are required due to delays above $\delta^{\max}$ but are not in E^{s}. An exhaustive search for all missing shortcuts would take too long for the update phase, so we propose a heuristic *replacement search* for candidates that were made infeasible by the last delay update. A candidate with intermediate transfer $(\varepsilon_{\mathrm{o}}, \varepsilon_{\mathrm{d}})$ becomes infeasible if the arrival delay of ε_{o} increases to the point at which ε_{d} is no longer reachable. The update phase therefore collects the set $\mathcal{E}_{\mathrm{del}}$ of stop events whose arrival delay was increased by the current update. For each stop event in this set, the algorithm collects the affected candidates and searches for replacements that are prefix-optimal in the current delay scenario Δ.

Replacement Search Routine. The core of the replacement search algorithm is the routine `FindReplacements`. It takes as input a set $\mathcal{E}_{\mathrm{s}}$ of source events, a set V_{t} of target stops and an upper bound $\overline{\tau}_{\mathrm{arr}}(v)$ for the arrival time at each target stop $v \in V_{\mathrm{t}}$. For each pair of source event $\varepsilon_{\mathrm{s}} \in \mathcal{E}_{\mathrm{s}}$ and target stop $v_{\mathrm{t}} \in V_{\mathrm{t}}$, it searches for a Pareto-optimal $v(\varepsilon_{\mathrm{s}})$-$v_{\mathrm{t}}$-journey with exactly two trips that departs no earlier than $\tau_{\mathrm{dep}}(\varepsilon_{\mathrm{s}})$ and arrives no later than $\overline{\tau}_{\mathrm{arr}}(v_{\mathrm{t}})$. If one exists, a shortcut representing its intermediate transfer is generated.

The routine uses one-to-all MR searches restricted to two rounds as its basic building blocks. First, a backward MR search establishes a latest departure time $\overleftarrow{\tau_{\mathrm{dep}}}(v, n)$ at each vertex v for each round n. This is the latest departure time at v such that at least one target stop $v_{\mathrm{t}} \in V_{\mathrm{t}}$ can be reached from v with at most i trips no later than $\overline{\tau}_{\mathrm{arr}}(v_{\mathrm{t}})$. For each target stop $v_{\mathrm{t}} \in V_{\mathrm{t}}$, $\overleftarrow{\tau_{\mathrm{dep}}}(v_{\mathrm{t}}, 0)$ is initialized with $\overline{\tau}_{\mathrm{arr}}(v_{\mathrm{t}})$. The backward MR search is then run with the following pruning rule: Let τ_{min} be the earliest departure time in Δ of any source event in $\mathcal{E}_{\mathrm{s}}$. Whenever the search improves $\overleftarrow{\tau_{\mathrm{dep}}}(v, n)$ for a vertex v in round n, it checks whether $\overleftarrow{\tau_{\mathrm{dep}}}(v, n) < \tau_{\mathrm{min}}$. If so, the search is pruned at v since it cannot be reached in time from any source event in $\mathcal{E}_{\mathrm{s}}$.

Afterward, for each source event $\varepsilon_{\mathrm{s}} \in \mathcal{E}_{\mathrm{s}}$, a forward MR search from $v(\varepsilon_{\mathrm{s}})$ with departure time $\tau_{\mathrm{dep}}(\varepsilon_{\mathrm{s}}, \Delta)$ is performed, which computes the earliest arrival time $\overrightarrow{\tau_{\mathrm{arr}}}(v, n)$ at each vertex v for each round n. Whenever $\overrightarrow{\tau_{\mathrm{arr}}}(v, n)$ is improved for a vertex v in round n, the algorithm checks whether $\overrightarrow{\tau_{\mathrm{arr}}}(v, n) > \overleftarrow{\tau_{\mathrm{dep}}}(v, 2 - n)$. If so, the search is pruned at v since no target

Algorithm 9.4: Replacement search.

Input: Shortcuts E^{s}, delay scenario Δ, infeasible shortcuts $E^{\mathrm{s}}_{\mathrm{inf}}(\Delta)$, delayed origin event $\varepsilon_{\mathrm{o}} \in \mathcal{E}_{\mathrm{del}}$
Output: Replacement shortcuts $E^{\mathrm{s}}_{\mathrm{r}}$

1 $\mathcal{E}_{\mathrm{s}} \leftarrow \overleftarrow{\mathcal{E}}(\varepsilon_{\mathrm{o}})$
2 $\mathcal{E}_{\mathrm{d}} \leftarrow \{\varepsilon_{\mathrm{d}} \mid (\varepsilon_{\mathrm{o}}, \varepsilon_{\mathrm{d}}) \in E^{\mathrm{s}}_{\mathrm{inf}}(\Delta)\}$
3 $V_{\mathrm{t}} \leftarrow \{v(\varepsilon_{\mathrm{t}}) \mid \varepsilon_{\mathrm{t}} \in \overrightarrow{\mathcal{E}}(\varepsilon_{\mathrm{d}}), \varepsilon_{\mathrm{d}} \in \mathcal{E}_{\mathrm{d}}\}$
4 **for each** $v_{\mathrm{t}} \in V_{\mathrm{t}}$ **do**
5 $\tau^{\mathrm{o}}_{\mathrm{arr}}(v_{\mathrm{t}}) \leftarrow \tau_{\mathrm{arr}}(\Delta, \varepsilon_{\mathrm{o}}) + \tau_{\mathrm{tra}}(v_{\mathrm{o}}, v_{\mathrm{t}})$
6 $\overline{\tau}_{\mathrm{arr}}(v_{\mathrm{t}}) \leftarrow \tau^{\mathrm{o}}_{\mathrm{arr}}(v_{\mathrm{t}})$
7 **for each** $T[i] \in \mathcal{E}_{\mathrm{d}}$ **do**
8 $T' \leftarrow$ `FindEarliestTrip`(T, i, Δ)
9 **for each** $\varepsilon_{\mathrm{t}} \in \overrightarrow{\mathcal{E}}(T'[i])$ **do**
10 $\overline{\tau}_{\mathrm{arr}}(v(\varepsilon_{\mathrm{t}})) \leftarrow \min(\overline{\tau}_{\mathrm{arr}}(v(\varepsilon_{\mathrm{t}})), \tau_{\mathrm{arr}}(\varepsilon_{\mathrm{t}}))$
11 $E^{\mathrm{s}}_{\mathrm{r}} \leftarrow$ `FindReplacements`$(\mathcal{E}_{\mathrm{s}}, V_{\mathrm{t}}, \overline{\tau}_{\mathrm{arr}}(\cdot))$

stop $v_{\mathrm{t}} \in V_{\mathrm{t}}$ can be reached from there without exceeding $\overline{\tau}_{\mathrm{arr}}(v_{\mathrm{t}})$. Finally, all journeys with exactly two trips that have been found at a target stop $v_{\mathrm{t}} \in V_{\mathrm{t}}$ are extracted and shortcuts for their intermediate transfers are added to the set $E^{\mathrm{s}}_{\mathrm{r}}$ of replacement shortcuts.

Calling the Routine. The replacement search for an origin event $\varepsilon_{\mathrm{o}} \in \mathcal{E}_{\mathrm{del}}$ is depicted in Algorithm 9.4. The set $\mathcal{E}_{\mathrm{s}}$ of potential source events consists of all stop events preceding ε_{o} in $T(\varepsilon_{\mathrm{o}})$. For each shortcut $(\varepsilon_{\mathrm{o}}, \varepsilon_{\mathrm{d}})$ that is infeasible in Δ, the destination event ε_{d} is added to $\mathcal{E}_{\mathrm{d}}$ and the stops visited by its successor events in $T(\varepsilon_{\mathrm{d}})$ are added to the set V_{t} of target stops. Lines 4–10 compute the upper bounds $\overline{\tau}_{\mathrm{arr}}(\cdot)$. Two alternatives are considered for an infeasible candidate $J^{\mathrm{c}} = \langle [\varepsilon_{\mathrm{s}}, \varepsilon_{\mathrm{o}}], [\varepsilon_{\mathrm{d}}, \varepsilon_{\mathrm{t}}] \rangle$ with $v(\varepsilon_{\mathrm{t}}) = v_{\mathrm{t}}$. One is the journey $\langle [\varepsilon_{\mathrm{s}}, \varepsilon_{\mathrm{o}}], v_{\mathrm{t}} \rangle$ that transfers directly from v_{o} to v_{t}. Line 6 initializes $\overline{\tau}_{\mathrm{arr}}(v_{\mathrm{t}})$ with the arrival time $\tau^{\mathrm{o}}_{\mathrm{arr}}(v_{\mathrm{t}}) := \tau_{\mathrm{arr}}(\Delta, \varepsilon_{\mathrm{o}}) + \tau_{\mathrm{tra}}(v_{\mathrm{o}}, v_{\mathrm{t}})$ of this journey. The other option is to wait at v_{d} for the next trip with the same stop sequence as $T(\varepsilon_{\mathrm{d}})$. For each destination event $T[i] \in \mathcal{E}_{\mathrm{d}}$, line 8 calls the subroutine `FindEarliestTrip`(T, i, Δ) to find the earliest trip T' with the same stop sequence as T such that $\tau^{\mathrm{o}}_{\mathrm{arr}}(v(T[i])) \leq \tau_{\mathrm{dep}}(\Delta, T'[i])$. Then, for each stop event ε_{t} succeeding $T'[i]$ in T', line 10 incorporates $\tau_{\mathrm{arr}}(\varepsilon_{\mathrm{t}})$ into $\overline{\tau}_{\mathrm{arr}}(v(\varepsilon_{\mathrm{t}}))$. Finally, the `FindReplacements` routine is invoked in line 11.

Algorithm 9.4 can be sped up further by batching the `FindReplacements` calls. For all delayed stop events belonging to the same trip, the computed inputs are merged and the routine is called just once on the merged inputs. This reduces the number of MR searches, at the cost of worse upper bounds $\overline{\tau}_{\mathrm{arr}}(\cdot)$. Furthermore, the individual `FindReplacements` calls are independent of each other and can be run in parallel.

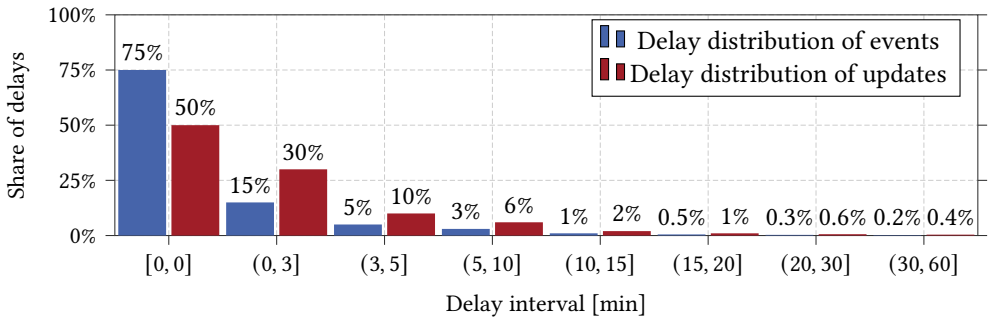


Figure 9.6: Delay distribution in our synthetically generated delay scenarios. There are eight delay intervals, ranging from 0 to 60 minutes. The percentage of stop events whose delay falls within each interval is shown in blue, and the percentage of delay updates is shown in red. Within an interval, delays are distributed uniformly.

9.6 Experiments

To generate delay scenarios for our experimental evaluation, we use a synthetic delay model. In Section 9.6.1, we describe the model in detail, discuss how we generate test queries and analyze the impact of delays on the four benchmark networks. Afterward, we evaluate the performance of Delay-ULTRA, the update phases and the resulting query algorithms in Section 9.6.2. For all experiments, we use walking with a constant speed of 4.5 km/h as the transfer mode. Shortcut computations were run on the Epyc machine, all other experiments on the Xeon machine.

9.6.1 Delay Model

Since we do not have access to proper real-world delay data, we generate synthetic delay scenarios using the model by Bast et al. [BSS13]: We create one delay update per trip T, with the index i of the first affected event chosen uniformly at random and the delay δ chosen at random according to the probability distribution outlined below. The arrival and departure delays of each stop event $T[j]$ with $j \geq i$ are set to δ; updates with $\delta = 0$ are discarded. For the reveal time, we choose the arrival time $\tau_{\text{arr}}(T[i])$ of the first affected event. The distribution for the expected delay of a stop event is shown in blue in Figure 9.6; it is roughly based on aggregated real-world punctuality data for London[1]. Note that since our model divides each trip into an undelayed first portion and a second portion with delay δ, half of all stop events remain unaffected by delays in expectation. To obtain the desired expected delay distribution, we reduce the probability of delay 0 to 50% and double all other probabilities; the resulting probability distribution is shown in red in Figure 9.6. To allow for a reasonably fast simulation

[1] `https://www.raildeliverygroup.com/punctuality.html`. Accessed May 6th, 2022.

Table 9.1: Impact of delays on queries in the synthetic delay scenarios. For each network, random queries were generated until 1 000 affected ones were found. A query is affected if MR without delay information returns at least one suboptimal or infeasible journey. Also reported are the number of returned journeys that are suboptimal and infeasible in the synthetic delay scenarios, respectively. For each infeasible journey, the highest slack among its infeasible transfers was measured. The reported slack is the median value across all infeasible journeys.

Network	Queries		Journeys			Slack [s]
	Total	Affected	Total	Affected	Infeasible	
London	5 572	1 000 (17.94%)	25 532	1 585 (6.20%)	956 (3.74%)	54
Germany	10 182	1 000 (9.82%)	54 852	1 699 (3.09%)	783 (1.42%)	68
Stuttgart	102 362	1 000 (0.97%)	424 480	1 528 (0.35%)	724 (0.17%)	136
Switzerland	30 375	1 000 (3.29%)	143 570	1 601 (1.11%)	894 (0.62%)	120

of the update phase, we limit our delay scenarios to the interval between 12 and 1 PM of the first covered day. Updates with a reveal time after 1 PM are discarded. For updates with a reveal time before 12 PM, we omit all affected stop events before 12 PM from the update.

Note that our synthetic model is heavily simplified, for example by assuming that there is at most one cause of delay along each trip and that this delay is never caught up. However, most of our simplifications make the synthetic scenarios more challenging than real ones. Since delays are chosen uniformly at random within each interval, instead of preferring lower delays, the average delay is overestimated. Delay updates are revealed at the latest possible moment, which puts more pressure on the update phase to process them in time. Finally, our model does not reproduce knock-on effects caused by trips waiting for each other or a delayed train blocking a platform. In practice, these effects make it more likely that precomputed transfers stay feasible in the presence of delays.

Impact of Delays. To investigate how many queries are affected by delays in each network, we generate queries until 1 000 affected ones are found. We choose source and target vertices uniformly at random and departure times uniformly at random between 12 and 1 PM. For the query execution times, we use the departure times. This maximizes the number of delays that are already known when a query is executed, making it more likely to be affected by delays and therefore more challenging to answer correctly. In reality, the execution time often lies well before the departure time. For each query, we perform one MR search in an undelayed scenario Δ_{base} and one in the scenario Δ_{del} that incorporates all delay updates that are known at execution time. This yields Pareto sets $\mathcal{J}_{\text{base}}$ and $\mathcal{J}_{\text{del}}$, respectively. If any journey $J \in \mathcal{J}_{\text{base}}$ is infeasible in Δ_{del} or has a later arrival time than a journey in $\mathcal{J}_{\text{del}}$ with at most $|J|$ trips, the query counts as affected. The results are shown in Table 9.1. We observe that among the affected queries, the share of affected and infeasible journeys is similar across

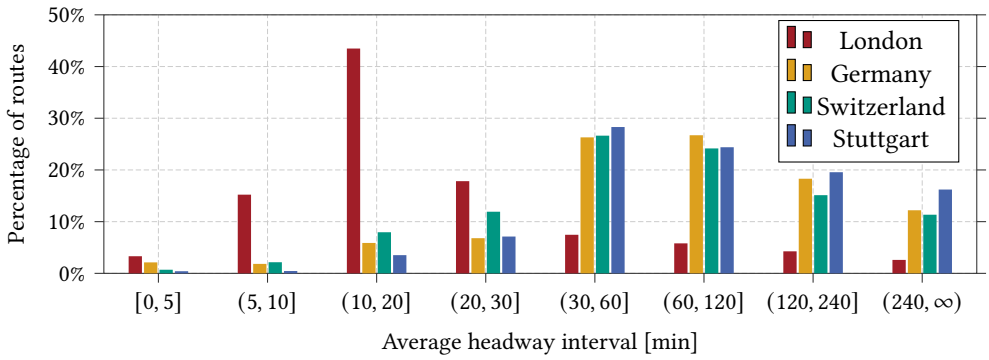


Figure 9.7: Headway distribution per network. Only trips that depart during the first service day are counted. Routes with fewer than two trips are ignored.

all networks: around a third are affected and about half of those are infeasible. However, the share of affected queries varies heavily between networks. In particular, it is much higher for London and Germany than for Switzerland and Stuttgart.

To explain these differences, we measure the *average headway* of a route, i.e., the mean difference in departure time between two consecutive trips of the route. Figure 9.7 shows the distribution of the average headways for each network. We notice a stark difference between London, which is a large metropolitan area, and the other networks, which also contain many rural regions. For London, the distribution centers around headways between 10 and 20 minutes, whereas the other networks mostly contain trips with headways above 30 minutes.

If headways are low, intermediate transfers have low slacks and are therefore more likely to become infeasible in the presence of delays. To confirm this, we measure the highest slack among the infeasible transfers of each infeasible journey. As shown in Table 9.1, the median slack across all infeasible journeys is particularly low for London and Germany. While the headway distribution of Germany is closer to that of Switzerland and Stuttgart, it has a comparatively high share of routes with an average headway below five minutes. This indicates its greater structural diversity: While its overall structure is similar to that of Switzerland and Stuttgart, it contains some metropolitan areas that are closer in structure and size to London. Accordingly, journeys that span these areas tend to use routes with lower headways and are therefore more likely to be affected by delays.

9.6.2 Algorithm Performance

We now evaluate the performance of Delay-ULTRA. First we describe our experimental setup. For Switzerland, we then analyze how the delay limit affects the algorithm's performance. For the other networks, we focus on a smaller selection of delay limits but analyze other aspects in more detail.

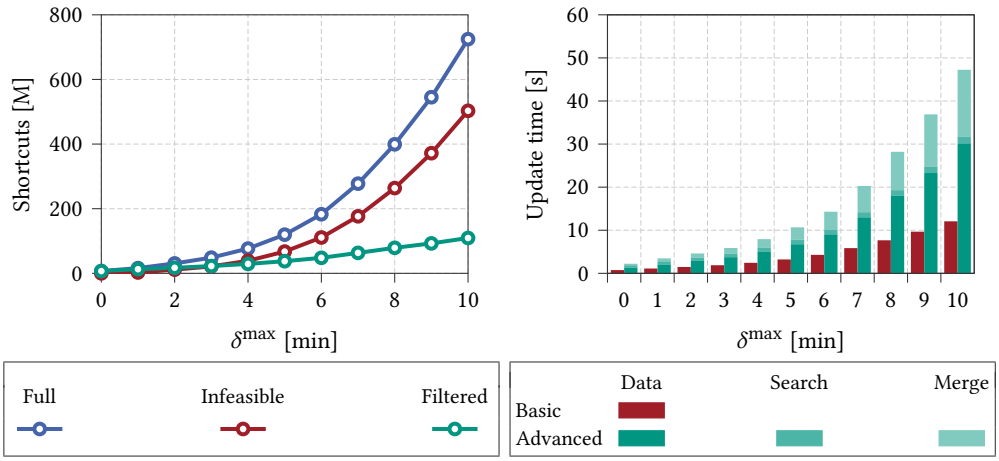


Figure 9.8: Impact of the delay limit $\delta^{\max}$ on the number of precomputed shortcuts (left) and the running time of the update phase (right), measured on the Switzerland network. We report the full set of shortcuts and, for an undelayed scenario, the number of infeasible shortcuts and those remaining after filtering. For the advanced update phase, the running time is split into updating the data structures, performing the replacement search and merging the replacement shortcuts.

Experimental Setup. To evaluate the update phase and the result quality of our Delay-ULTRA-TB query algorithm, we simulate the update algorithm within the interval from 12 PM to 1 PM and then run the test queries from Section 9.6.1. For a query with execution time τ_{ex}, we run MR on the delay scenario that is current at τ_{ex} and Delay-ULTRA-TB on the data produced by the last update phase that was fully finished before τ_{ex}. We then compute two error rates: The *query error rate* is the share of queries for which Delay-ULTRA-TB fails to find at least one Pareto-optimal journey. The *journey error rate* is the share of Pareto-optimal journeys (aggregated across all queries) that are not found. Since the update phase takes non-negligible time, delays do not become known to the query algorithm instantaneously. To quantify the effect of this on the result quality, we also compute *hypothetical error rates*: we repeat the same experiment, but this time we run a query with execution time τ_{ex} on the data produced by the last update phase started (rather than finished) by τ_{ex}.

While this setup is useful for evaluating the result quality, it is not suitable for measuring query times because it switches between answering queries and processing delay updates with a high frequency. Each update entails rebuilding the TB data structures, which effectively flushes the machine's cache and thereby distorts the query time measurements. We therefore use a different setup for measuring query times: We generate 10 000 random queries with the departure time chosen uniformly at random between 1 and 2 PM. Then we simulate the update phases within the interval between 12 and 1 PM and run the queries on the output of the last

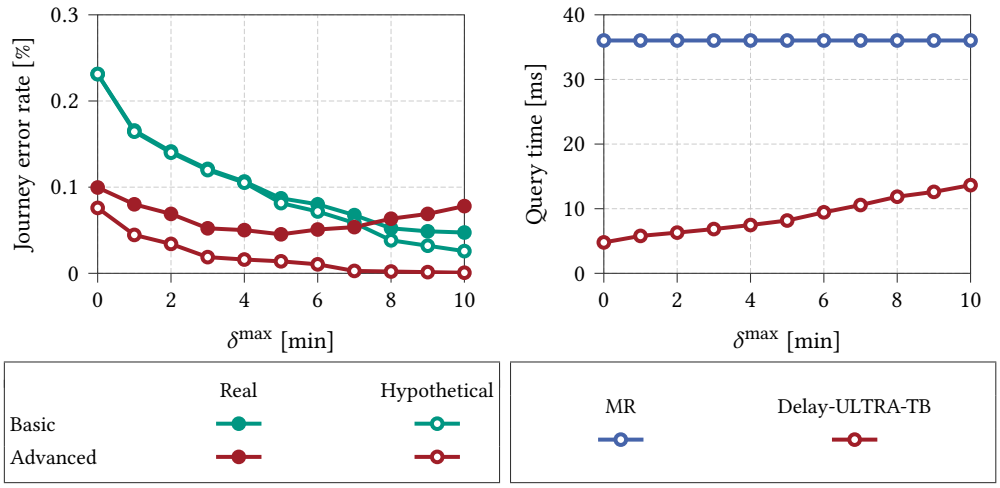


Figure 9.9: Impact of the delay limit $\delta^{\max}$ on the query algorithm, measured on the Switzerland network. *Left:* Journey error rate for various configurations of the update phase, averaged over the test queries generated in Section 9.6.1. *Right:* Mean running times of Delay-ULTRA-TB and MR for 10 000 random queries.

update. Note that this results in a set of queries whose execution time and departure time do not match: the execution time is always 1 PM while the departure time is up to one hour later. To quantify the impact that this difference has on the result quality, we also measure the error rates for this set of queries.

Impact of Delay Limit. Figure 9.8 (left) shows the impact of the delay limit on the number of shortcuts computed by Delay-ULTRA. For a scenario without any delays, we also report how many shortcuts are infeasible and how many remain after removing both infeasible and irrelevant shortcuts. Subsequent experiments (see Table 9.3) show that these numbers are almost identical in scenarios with delays. The growth in the number of total and infeasible shortcuts is roughly quadratic. The number of filtered shortcuts, which influences the speed of the query algorithm, grows more slowly. However, the size of the unfiltered shortcut set still affects the running time of the update phase, which also grows quadratically, as shown in Figure 9.8 (right). The advanced update phase takes three to four times longer than the basic one. This is not primarily due to the replacement search itself, but due to building the additional data structures required by it and merging the replacement shortcuts.

Note that our implementation of TB is geared towards query performance rather than quick updates. Hence, we believe that the running time of the update phase could be improved significantly. To improve cache locality during a query, we assign consecutive IDs to all trips and stop events belonging to the same route. The set E^{s} of shortcut edges is stored as a

Table 9.2: Results for the Delay-ULTRA shortcut computation. Reported are the full set of shortcuts and the number of infeasible and filtered shortcuts in an undelayed scenario.

Network	$\delta^{\max}$ [s]	Time [hh:mm:ss]	# Shortcuts		
			Full	Infeasible	Filtered
London	0	00:03:44	8 576 120	0 (0.00%)	8 576 120 (100.00%)
	120	00:10:26	86 573 982	36 435 150 (42.09%)	39 665 271 (45.82%)
	180	00:10:41	176 630 627	93 727 958 (53.06%)	57 941 841 (32.80%)
	300	00:14:37	609 565 189	398 706 627 (65.41%)	114 153 168 (18.73%)
Germany	0	02:53:57	77 515 291	0 (0.00%)	77 515 291 (100.00%)
	180	11:24:36	588 027 879	209 113 002 (35.56%)	327 040 211 (55.62%)
	300	11:51:22	1 401 861 101	717 161 810 (51.16%)	533 061 808 (38.03%)
Stuttgart	0	00:00:54	1 973 321	0 (0.00%)	1 973 321 (100.00%)
	180	00:02:35	13 388 174	5 632 875 (42.07%)	6 576 102 (49.12%)
	300	00:02:59	29 236 505	15 106 602 (51.67%)	10 629 297 (36.36%)
Switzerland	0	00:01:57	6 938 012	0 (0.00%)	6 938 012 (100.00%)
	180	00:06:46	48 529 326	21 420 171 (44.14%)	22 481 726 (46.33%)
	300	00:07:56	119 664 041	67 335 101 (56.27%)	37 552 597 (31.38%)

graph in adjacency array representation, where the outgoing edges (v, w) of a stop event v are sorted according to the ID of w. After regrouping the set of routes in order to avoid overtaking, the update phase must reorder the trips and stop events to ensure that their IDs are still consecutive. Afterward, E^s must be reordered as well. Our implementations of these reordering steps are not fully optimized. Furthermore, they could be omitted entirely by using an implementation of TB that does not rely on consecutive IDs.

Figure 9.9 (left) reports the journey error rate. Original ULTRA-TB with basic updates is already fairly delay-robust: slightly above 0.2% of optimal journeys are missed, compared to 1.1% without updates. The replacement search reduces this to 0.1%. Delay-ULTRA shortcuts offer drastic improvements for low delay limits, but these are eventually offset by the longer update time. With advanced updates, the real and hypothetical error rates diverge significantly: the latter reaches near-zero while the former stagnates and then increases again. This shows that a faster implementation of the update phase would significantly improve the result quality. The smallest errors are achieved with a limit of 10 min for basic updates and 5 min for advanced updates. As shown in Figure 9.9 (right), the latter configuration is preferable overall since it produces fewer shortcuts and therefore yields lower query times. Overall, query times exhibit moderate growth with an increasing delay limit: compared to the original ULTRA-TB, they roughly double for a delay limit of 5 min and triple for 10 min. Compared to MR, the speedup is reduced from 7.5 to 4.4 and 2.6, respectively.

Table 9.3: Results for the update phases. *Adv.* indicates whether the basic (○) or advanced update phase (●) was performed. *Updates* is the average number of processed delay updates per phase. For the reported shortcuts, *Query* is the number of shortcuts given as input to the query algorithm, averaged across all performed update phases, while *Added* is the total number of replacement shortcuts found across all phases. Running times are averaged across the evaluated one-hour interval.

Network	$\delta^{\max}$ [s]	Adv.	Updates	Shortcuts		Time [s]			
				Query	Added	Update	Search	Merge	Total
London	0	○	6.0	8 508 656	0	–	–	–	1.0
	0	●	6.0	8 586 763	98 149	2.0	0.5	0.6	3.1
	120	○	6.0	39 625 853	0	–	–	–	3.6
	120	●	9.8	39 668 596	54 547	7.6	0.8	2.9	11.2
	180	○	6.0	58 051 426	0	–	–	–	5.6
	180	●	15.7	58 082 966	39 714	12.1	0.9	5.1	18.1
Germany	0	○	344.2	77 286 195	0	–	–	–	10.8
	0	●	469.3	77 536 055	314 714	17.7	17.8	5.8	41.2
	180	○	344.2	326 591 619	0	–	–	–	26.0
	180	●	1 290.6	326 691 285	131 255	47.1	51.2	18.6	116.9
Stuttgart	0	○	17.9	1 968 084	0	–	–	–	0.2
	0	●	17.9	1 971 959	6 096	0.4	0.3	0.1	0.9
	300	○	17.9	10 620 259	0	–	–	–	0.9
	300	●	18.8	10 621 317	1 945	2.1	0.5	0.7	3.3
Switzerland	0	○	13.6	6 920 689	0	–	–	–	0.8
	0	●	22.1	6 934 598	19 407	1.3	0.6	0.4	2.2
	300	○	25.7	37 551 717	0	–	–	–	3.2
	300	●	42.1	37 555 783	5 856	6.7	1.0	2.9	10.7

Shortcut Computation. Results for the shortcut precomputation on all networks are reported in Table 9.2. Compared to the original ULTRA, the preprocessing time is three to four times longer. For Stuttgart and Germany, the growth in the number of shortcuts depending on the delay limit is similar to what we observed for Switzerland. By contrast, London exhibits a much faster growth but also a significantly higher share of infeasible shortcuts. Again, this is explained by the fact that transfers in the London network have much lower slacks on average and therefore become infeasible much faster.

Update Phase. Table 9.3 shows statistics for the update phase. We observe that the number of replacement shortcuts decreases as the delay limit increases, even though there are

Table 9.4: Query result quality, averaged over the test queries from Section 9.6.1. *Adv.* indicates whether the basic (○) or advanced update phase (●) was performed. We report the real and hypothetical error rates for queries (*Qr.*) and journeys (*Jrn.*), as well as the median detours (*Det.*) of suboptimal journeys compared to their optimal counterparts with the same number of trips. For the real update phase, we also report the number of returned journeys that are infeasible and the number of queries with at least one infeasible journey.

Network	$\delta^{\max}$ [s]	Adv.	Error rate (real)			Error rate (hypo.)			Infeasible (real)	
			Qr.	Jrn.	Det.	Qr.	Jrn.	Det.	Qr.	Jrn.
London	0	○	7.54%	2.30%	6.21%	7.54%	2.30%	6.21%	0.00%	0.00%
	0	●	0.88%	0.26%	6.53%	0.32%	0.09%	6.80%	0.00%	0.00%
	120	○	4.00%	1.18%	5.81%	3.98%	1.18%	5.85%	0.02%	0.00%
	120	●	0.50%	0.16%	6.45%	0.14%	0.05%	5.94%	0.14%	0.04%
	180	○	3.03%	0.85%	5.90%	3.01%	0.85%	5.93%	0.02%	0.00%
	180	●	0.65%	0.20%	7.43%	0.07%	0.02%	5.95%	0.22%	0.06%
Germany	0	○	5.10%	1.51%	8.13%	5.10%	1.51%	8.13%	0.00%	0.00%
	0	●	3.11%	0.90%	7.56%	2.28%	0.64%	7.27%	0.09%	0.03%
	180	○	2.31%	0.67%	8.90%	2.25%	0.65%	8.87%	0.04%	0.01%
	180	●	1.20%	0.37%	11.31%	0.43%	0.13%	8.50%	0.25%	0.08%
Stuttgart	0	○	0.27%	0.09%	10.40%	0.27%	0.09%	10.40%	0.00%	0.00%
	0	●	0.15%	0.05%	8.33%	0.13%	0.04%	7.62%	0.00%	0.00%
	300	○	0.09%	0.03%	13.85%	0.09%	0.03%	13.85%	0.00%	0.00%
	300	●	0.04%	0.01%	14.66%	0.01%	0.00%	10.64%	0.00%	0.00%
Switzerland	0	○	0.85%	0.22%	10.31%	0.85%	0.22%	10.31%	0.00%	0.00%
	0	●	0.35%	0.10%	10.81%	0.27%	0.08%	10.45%	0.00%	0.00%
	300	○	0.34%	0.09%	10.70%	0.33%	0.08%	10.65%	0.00%	0.00%
	300	●	0.15%	0.05%	11.24%	0.05%	0.01%	8.66%	0.02%	0.01%

more infeasible shortcuts that need to be replaced. This indicates that most replacements are already included in the precomputed shortcut set. Furthermore, the number of replacement shortcuts is negligible compared to the number of filtered Delay-ULTRA shortcuts. Hence, even when running the update phase for an entire day, the replacement shortcuts will not significantly impact the query time. Even in the most expensive configuration, the update phase takes at most a few seconds on the three smaller networks. This is different for Germany due to the sheer size of the network. Here, the time required for the replacement search starts to become a significant factor.

Table 9.5: Running times of MR and (Delay-)ULTRA-TB for 10 000 random queries. Also reported are the mean error rates of Delay-ULTRA-TB for these queries, with the basic and advanced update phases.

Network	$\delta^{\max}$ [s]	Query times [ms]		Error rate (basic)		Error rate (adv.)	
		MR	TB	Queries	Journeys	Queries	Journeys
London	0	18.4	4.0	1.94%	0.56%	0.28%	0.06%
	120	18.4	8.0	1.21%	0.35%	0.11%	0.02%
	180	18.4	9.8	0.71%	0.21%	0.05%	0.01%
Germany	0	920.4	78.8	3.57%	1.02%	2.14%	0.60%
	180	920.4	113.2	1.34%	0.39%	0.57%	0.16%
Stuttgart	0	36.8	3.8	0.08%	0.02%	0.02%	0.00%
	300	36.8	6.1	0.02%	0.00%	0.00%	0.00%
Switzerland	0	36.0	4.8	0.37%	0.11%	0.14%	0.03%
	300	36.0	8.2	0.08%	0.02%	0.01%	0.00%

Error Rate. Table 9.4 reports error rates for all networks. ULTRA-TB with basic updates already answers 50–75% of delay-affected queries correctly. Delay-ULTRA with advanced updates reduces the error rate by a factor of 4–15, while a hypothetical instant update phase would reduce it by more than a factor of 10 on all networks. The median detours of suboptimal journeys compared to their optimal counterparts are not significantly influenced by the delay limit or the type of update phase; they are between 5% and 15% depending on the network. The replacement search is particularly effective for London, where it resolves almost 90% of incorrectly answered queries on its own. This effect narrows the gap in the error rate compared to the other networks, which have many fewer delay-affected queries to begin with. Still, London and Germany exhibit higher error rates than the other networks. We observe a tradeoff between the overall error rate and the number of infeasible journeys: a higher delay limit reduces the error rate overall, but updates take longer to incorporate, which increases the risk of returning journeys that are already known to be infeasible at execution time.

Query Performance. Finally, Table 9.5 reports query times. The speedup of Delay-ULTRA-TB over MR ranges from 2.3 for London to 8.0 for Germany. Compared to ULTRA-TB without delay information, this corresponds to a slowdown of 1.4–2.0. For this different set of queries, whose execution time is up to an hour before the departure time, the error rates are significantly lower. The effect is particularly drastic for London, where trips and journeys are shorter on average and the impact of an individual delay therefore dissipates more quickly. As a result, the error rate for $\delta^{\max} = 3$ min is lower here than for 2 min, even though the opposite was the case in Table 9.4. This shows that incorporating a delay update too late mostly affects

queries that depart right away, whereas the replacement shortcuts provide a more long-term benefit. Still, $\delta^{\max} = 2$ min is the preferable configuration overall since the query time is lower. With a moderate delay limit and advanced updates, the journey error rate is minuscule on all networks except for Germany, where it is still very low at 0.16%. Overall, this shows that Delay-ULTRA offers a significant speedup over MR at near-perfect solution quality.

9.7 Conclusion

We showed that the benefits of ULTRA can be retained in the presence of delays by handling small delays exactly during the shortcut computation and larger ones heuristically in the update phase. Together, both steps reduce the error rate for random queries by up to a factor of 30 compared to an algorithm without any delay information. Hence, Delay-ULTRA offers near-perfect solution quality while achieving a speedup between two and eight over the fastest delay-robust competitor. In the future, the replacement search could be applied to trip cancellations as well. Further insights could also be gained from evaluating our algorithms on real or more sophisticated synthetic delay scenarios.

10 Conclusion

In order to close the multimodal performance gap, we identified two main tasks: developing a preprocessing technique that allows existing public transit algorithms to be applied to multimodal networks, and generalizing the fastest query algorithms to more complex problem settings. For the first task, we built on the shortcut hypothesis by Sauer [Sau18], which states that the intermediate transfers that occur in optimal journeys can be condensed into a small set of shortcuts. In Chapter 5, we presented ULTRA, a speedup technique based on the shortcut hypothesis. It can be combined with any public transit algorithm that requires one-hop transfers, thereby offering a solution for the first task. Following the cyclical approach of Algorithm Engineering, we then tested the shortcut hypothesis in various extended multimodal scenarios, adapting ULTRA along the way. These include one-to-many queries in Chapter 6, additional criteria in Chapter 7, multiple competing transfer modes in Chapter 8, and delays in Chapter 9. Especially for the multicriteria problems considered in Chapters 7 and 8, our modeling decisions were informed by the experimental results of the previous chapters. We saw that optimizing transfer time is important in order to obtain high-quality solutions. We also identified the limits of the shortcut hypothesis: it holds as long as public transit is the fastest and more comfortable mode and the transfer modes are mainly used for bridging gaps. Many, but not all multimodal scenarios fit this assumption. In particular, ULTRA is a poor fit for most car-based modes, which compete with public transit in terms of speed.

ULTRA can be configured to generate shortcuts between pairs of stop events, which allows it to serve as a multimodal replacement for the transfer generation step of TB. This enables the use of TB in multimodal networks, which made TB a natural candidate for approaching the second task. We adapted TB to a variety of extended problem settings, including one-to-many search, additional criteria and restricted Pareto sets. We showed that three-criteria Pareto

optimization is solvable in polynomial time for particular combinations of criteria because the solution size is bounded. This was exploited in the design of McTB, which does not require costly dynamic data structures and retains the performance benefits of TB in a three-criteria setting. Since there is no apparent way to extend this result to four or more criteria, we combined the advantages of McRAPTOR and TB into a hybrid algorithm called HydRA. Overall, the combination of ULTRA and TB-based query algorithms allowed us to close the performance gap in a variety of multimodal scenarios. In all cases, the achieved query times are fast enough for interactive applications. In scenarios for which comparable pure public transit algorithms are available, our algorithms match their performance. Compared to the previous state of the art for multimodal journey planning, they achieve speedups ranging from an order of magnitude in the two-criteria setting to nearly three orders of magnitude in the most complex scenario.

Further Applications. Although multimodal journey planning is a complex problem, the success of the shortcut hypothesis and ULTRA suggests that a good way to make progress is to decompose it into modular subproblems. When tackling a new problem, this often allows us to reuse some of these modules without changes. We observed several examples of this throughout this thesis: Adapting ULTRA to one-to-many queries in Chapter 6 only required us to exchange the component that handles the final transfers. When adding transfer time as a third criterion in Chapter 7, we were able to retain the Bucket-CH searches for the initial and final transfers. Finally, when considering multiple competing transfer modes in Chapter 8, we were able to decompose the shortcut computation by mode, which allowed us to reuse the existing bimodal McULTRA algorithm.

Overall, the algorithms developed in this thesis make up a versatile toolbox that can help carry over future advances in public transit journey planning to multimodal networks. One open problem for which this might prove useful is designing an integrated multimodal speedup technique that accelerates the exploration of all network parts. So far, such a technique has proven elusive due to the structural differences between public transit and road networks. ULTRA potentially offers a way forward by condensing the road-based modes into small sets of shortcuts, allowing the design to focus on the public transit component.

While this thesis focused on multimodal journey planning, many of the developed techniques also have applications in pure public transit networks. We observed that the exploration of these networks can be sped up by combining multiple pruning rules to restrict the search space, but care has to be taken to ensure that these rules are compatible with each other. For example, in Chapter 5, a conflict arose between the self-pruning rule used by profile searches and the tiebreaking choices of RAPTOR. Our solution for this problem may be useful in other settings that involve profile search. For example, Großmann et al. [GSSS23] show that the combination of Arc-Flags with TB promises to achieve extremely fast query times with only moderate space consumption. However, due to a conflict between self-pruning and route-based pruning in the preprocessing phase, some queries are answered incorrectly. A similar approach to the one used in ULTRA may be able to resolve this issue.

In Chapters 7 and 8, we showed that the event-based approach of TB can be extended to scenarios with additional criteria. The resulting query algorithms, McTB and HydRA, are therefore promising candidates for answering public transit queries with additional criteria, such as fare or vehicle occupancy. Even though it may seem counterintuitive, the ULTRA shortcut computation also has applications in scenarios with limited transfers. We saw that event-to-event ULTRA produces a smaller transfer set than the original TB transfer generation algorithm because it employs stronger pruning rules. Therefore, a variant of ULTRA that is tailored towards one-hop transfer graphs could improve the performance of TB in an unimodal context, whereas McULTRA could serve as a replacement for the TB preprocessing step in scenarios with additional criteria.

One of the biggest remaining issues with ULTRA is that its preprocessing time is quadratic in the number of stops, so it does not scale well for larger networks. This is in line with existing public transit speedup techniques, such as Transfer Patterns. It is unclear whether anything can be done about this while still computing a provably optimal set of shortcuts. However, this shortcoming is mitigated in scenarios with limited transfers. For example, we saw that for faster transfer modes, such as bicycles or e-scooters, the approach of guaranteeing a few minutes of transfer time and then computing the transitive closure is no longer feasible. On the other hand, (Mc)ULTRA is slower for these modes than for walking and produces more shortcuts. A potential solution that represents a compromise between unrestricted and transitively closed transfers is to allow McULTRA to explore the full transfer graph but limit the transfer duration. With such a limit, the size of the region that can be reached from a given stop with at most two trips becomes near-constant in the network size. Hence, we expect the preprocessing effort to be much closer to linear in this setting.

Open Questions. Although we examined a wide range of multimodal scenarios in this thesis, there are still more left to study. On the one hand, this includes combinations of multiple problem extensions. For example, we only considered one-to-many searches and delay-robustness in the simple scenario with one transfer mode and two criteria. On the other hand, there are multimodal scenarios that do not fit the assumptions of the shortcut hypothesis (e.g., taxis) or for which it is unknown whether they do (e.g., ridesharing). These may require algorithmic approaches that are altogether different. Furthermore, we used simplified models for the transfer modes, delays and query distributions throughout this thesis. Using more realistic models would provide further insights into the shortcut hypothesis and the structure of multimodal networks. Since we designed our models to represent the worst case, it is possible that ULTRA exhibits even better performance for more realistic models.

Finally, while this thesis provides ample experimental support for the shortcut hypothesis, the phenomenon is not yet understood from a theoretical perspective. A logical next step would be to identify structural properties of multimodal networks that explain these results. Ideally, this would allow us to establish provable bounds on the number of shortcuts. These could be turned into performance guarantees for public transit algorithms when they are applied to multimodal networks.

Bibliography

[Abr+16] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. **Highway Dimension and Provably Efficient Shortest Path Algorithms**. In *Journal of the ACM* volume 63:5, 41:1–41:26, Association for Computing Machinery (ACM), 2016. DOI: 10.1145/2985473.
Cited on pages 2, 3, 21.

[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. DOI: 10.1007/978-3-642-20662-7_20.
Cited on page 20.

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato 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 (LNCS), pages 24–35. Springer, 2012. DOI: 10.1007/978-3-642-33090-2_4.
Cited on page 20.

[ALD21] Rusul L. Abduljabbar, Sohani Liyanage, and Hussein Dia. **The Role of Micro-Mobility in Shaping Sustainable Cities: A Systematic Literature Review**. In *Transportation Research Part D: Transport and Environment* volume 92:39, pages 102734:1–102734:19, Elsevier, 2021. DOI: 10.1016/j.trd.2021.102734.
Cited on page 1.

[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 (LNCS), pages 55–66. Springer, 2013. DOI: 10.1007/978-3-642-38527-8_7.
Cited on page 21.

[Bar+09] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav Marathe, and Dorothea Wagner. **Engineering Label-Constrained Shortest-Path Algorithms**. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 309–319. American Mathematical Society (AMS), 2009. DOI: 10.1090/dimacs/074/12.
Cited on page 28.

[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 (LNCS), pages 290–301. Springer, 2010. DOI: 10.1007/978-3-642-15775-2_25.
Cited on pages 3, 24, 26, 48, 90.

[Bas+16] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. **Route Planning in Transportation Networks**. In *Algorithm Engineering: Selected Results and Surveys*. Volume 9220. Lecture Notes in Computer Science (LNCS). Springer, 2016, pages 19–80. DOI: 10.1007/978-3-319-49487-6_2.
Cited on pages 2, 17, 18, 24, 48.

[Bas09] Hannah Bast. **Car or Public Transport – Two Worlds**. In *Efficient Algorithms*. Volume 5760 of Lecture Notes in Computer Science (LNCS), pages 355–367. Springer, 2009. DOI: 10.1007/978-3-642-03456-5_24.
Cited on pages 2, 23, 25.

[Bau+10] 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. DOI: 10.1145/1671970.1671976.
Cited on pages 21, 25, 34.

[Bau+16] Moritz Baum, Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner. **Scalable Exact Visualization of Isocontours in Road Networks via Minimum-Link Paths**. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA'16)*. Volume 57 of Leibniz International Proceedings in Informatics (LIPIcs), pages 7:1–7:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. DOI: 10.4230/LIPIcs.ESA.2016.7.
Cited on page 21.

[Bau+19] 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. DOI: 10.4230/LIPIcs.ESA.2019.14.
Cited on page 9.

[Bau+23] Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. **ULTRA: Unlimited Transfers for Efficient Multimodal Journey Planning**. In *Transportation Science*, INFORMS, 2023. DOI: 10.1287/trsc.2022.0198.
Cited on page 9.

[Bau17] Moritz Baum. **Engineering Route Planning Algorithms for Battery Electric Vehicles**. PhD thesis. Karlsruhe Institute of Technology, 2017. DOI: 10.5445/IR/1000082225.
Cited on page 33.

[BBDW19] Moritz Baum, Valentin Buchhold, Julian Dibbelt, and Dorothea Wagner. **Fast Exact Computation of Isocontours in Road Networks**. In *Journal of Experimental Algorithmics (JEA)* volume 24, pages 1.18:1–1.18:26, Association for Computing Machinery (ACM), 2019. DOI: 10.1145/3355514.
Cited on pages 21, 103.

[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 (OASIcs), pages 123–136. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013. DOI: 10.4230/OASIcs.ATMOS.2013.123.
Cited on pages 6, 29, 106, 134.

[BD10] Reinhard Bauer and Daniel Delling. **SHARC: Fast and Robust Unidirectional Routing**. In *Journal of Experimental Algorithmics (JEA)* volume 14, pages 4:1–4:29, Association for Computing Machinery (ACM), 2010. DOI: 10.1145/1498698.1537599.
Cited on page 21.

[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 (OASIcs), pages 2:1–2:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2009. DOI: 10.4230/OASIcs.ATMOS.2009.2148.
Cited on pages 25, 48.

[BDPW16] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. **Dynamic Time-Dependent Route Planning in Road Networks with User Preferences**. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*. Volume 9685 of Lecture Notes in Computer Science (LNCS), pages 33–49. Springer, 2016. DOI: 10.1007/978-3-319-38851-9_3.
Cited on page 22.

[BDW11] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. **Experimental Study of Speed Up Techniques for Timetable Information Systems**. In *Networks* volume 57:1, pages 38–52, Wiley, 2011. DOI: 10.1002/net.20382.
Cited on page 25.

[Bel58] Richard Bellman. **On a Routing Problem**. In *Quarterly of Applied Mathematics* volume 16:1, pages 87–90, Brown University, 1958. DOI: 10.1090/qam/102435.
Cited on page 17.

[BFM09] Hannah Bast, Stefan Funke, and Domagoj Matijevic. **Ultrafast Shortest-Path Queries via Transit Nodes**. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 175–192. American Mathematical Society (AMS), 2009. DOI: 10.1090/dimacs/074/07.
Cited on page 20.

[BFP21] Florian Barth, Stefan Funke, and Claudius Proissl. **Preference-Based Trajectory Clustering – An Application of Geometric Hitting Sets**. In *Proceedings of the 32nd International Symposium on Algorithms and Computation (ISAAC'21)*. Leibniz International Proceedings in Informatics (LIPIcs), pages 15:1–15:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. DOI: 10.4230/LIPIcs.ISAAC.2021.15.
Cited on pages 23, 136.

[BFP22] Florian Barth, Stefan Funke, and Claudius Proissl. **An Upper Bound on the Number of Extreme Shortest Paths in Arbitrary Dimensions**. In *Proceedings of the 30th Annual European Symposium on Algorithms (ESA'22)*. Leibniz International Proceedings in Informatics (LIPIcs), pages 14:1–14:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. DOI: 10.4230/LIPIcs.ESA.2022.14.
Cited on page 23.

[BFS19] Florian Barth, Stefan Funke, and Sabine Storandt. **Alternative Multicriteria Routes**. In *Proceedings of the 21st Workshop on Algorithm Engineering and Experiments (ALENEX'19)*, pages 66–80. Society for Industrial and Applied Mathematics (SIAM), 2019. DOI: 10.1137/1.9781611975499.6.
Cited on page 23.

[BFS21] Johannes Blum, Stefan Funke, and Sabine Storandt. **Sublinear Search Spaces for Shortest Path Planning in Grid and Road Networks**. In *Journal of Combinatorial Optimization* volume 42, pages 231–257, Springer, 2021. DOI: 10.1007/s10878-021-00777-3.
Cited on pages 3, 21.

[BFSS07] Hannah Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. **Fast Routing in Road Networks with Transit Nodes**. In *Science* volume 316:5824, pages 566–566, American Association for the Advancement of Science (AAAS), 2007. DOI: 10.1126/science.1137521.
Cited on pages 2, 20.

[BGSV13] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. **Minimum Time-Dependent Travel Times with Contraction Hierarchies**. In *Journal of Experimental Algorithmics (JEA)* volume 18, pages 1.4:1–1.4:43, Association for Computing Machinery (ACM), 2013. DOI: 10.1145/2444016.2444020.
Cited on page 22.

[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. Society for Industrial and Applied Mathematics (SIAM), 2016. DOI: 10.1137/1.9781611974317.2.
Cited on pages 26, 28, 48.

[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 (ATMOS'03)*. Volume 92 of, pages 3–15. Elsevier, 2004. DOI: 10.1016/j.entcs.2003.12.019.
Cited on page 23.

[BJM00] Chris Barrett, Riko Jacob, and Madhav Marathe. **Formal-Language-Constrained Path Problems**. In *SIAM Journal on Computing (SICOMP)* volume 30:3, pages 809–837, Society for Industrial and Applied Mathematics (SIAM), 2000. DOI: 10.1137/S0097539798337716.
Cited on page 28.

[Böh+13] Katerina Böhmová, Matús Mihalák, Tobias Pröger, Rastislav Srámek, and Peter Widmayer. **Robust Routing in Urban Public Transportation: How to Find Reliable Journeys Based on Past Observations**. In *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13)*. Volume 33 of OpenAccess Series in Informatics (OASIcs), pages 27–41. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013. DOI: 10.4230/OASIcs.ATMOS.2013.27.
Cited on page 154.

[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 (LIPIcs), pages 20:1–20:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. DOI: 10.4230/LIPIcs.SEA.2017.20.
Cited on pages 47, 89.

[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 (ACM), 2014. DOI: 10.1145/2666310.2666405.
Cited on pages 25, 26, 28, 48.

[BS24] Dominik Bez and Jonas Sauer. **Fast and Delay-Robust Multimodal Journey Planning**. In *Proceedings of the 26th Workshop on Algorithm Engineering and Experiments (ALENEX'24)*. Society for Industrial and Applied Mathematics (SIAM), 2024. Accepted for publication.
Cited on page 10.

[BSS13] Hannah Bast, Jonas Sternisko, and Sabine Storandt. **Delay-Robustness of Transfer Patterns in Public Transportation 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 (OASIcs), pages 42–54. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013. DOI: 10.4230/OASIcs.ATMOS.2013.42.
Cited on pages 154, 185.

[CGR96] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. **Shortest Paths Algorithms: Theory and Experimental Evaluation**. In *Mathematical Programming* volume 73:2, pages 129–174, Springer, 1996. DOI: 10.1007/BF02592101.
Cited on page 32.

[CHKZ03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. **Reachability and Distance Queries via 2-Hop Labels**. In *SIAM Journal on Computing (SICOMP)* volume 32:5, pages 1338–1355, Society for Industrial and Applied Mathematics (SIAM), 2003. DOI: 10.1137/S0097539702403098.
Cited on page 20.

[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, Elsevier, 2017. DOI: 10.1016/j.jda.2017.09.001.
Cited on page 154.

[Dan63] George B. Dantzig. **Linear Programming and Extensions**. Princeton University Press, 1963. DOI: 10.1515/9781400884179.
Cited on pages 18, 32.

[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. Society for Industrial and Applied Mathematics (SIAM), 2019. DOI: 10.1137/1.9781611975499.5.
Cited on pages 6, 24, 27, 28, 29, 106, 115, 117.

[DDPW15] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. **Public Transit Labeling**. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Volume 9125 of Lecture Notes in Computer Science (LNCS), pages 273–285. Springer, 2015. DOI: 10.1007/978-3-319-20086-6_21.
Cited on pages 3, 15, 26, 48.

[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 (OASIcs), pages 8:1–8:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. DOI: 10.4230/OASIcs.ATMOS.2017.8.
Cited on pages 27, 48.

[Del+09] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. **High-Performance Multi-Level Routing**. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 73–91. American Mathematical Society (AMS), 2009. DOI: 10.1090/dimacs/074/04.
Cited on page 20.

[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 (LNCS), pages 260–271. Springer, 2013. DOI: 10.1007/978-3-642-38527-8_24.
Cited on pages 5, 6, 29, 34, 42, 48, 54, 105, 106, 107, 117, 134, 136, 148.

[Del11] Daniel Delling. **Time-Dependent SHARC-Routing**. In *Algorithmica* volume 60, pages 60–94, Springer, 2011. DOI: 10.1007/s00453-009-9341-0.
Cited on pages 22, 25.

[DGNW13] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. **PHAST: Hardware-Accelerated Shortest Path Trees**. In *Journal of Parallel and Distributed Computing* volume 73:7, pages 940–952, Elsevier, 2013. DOI: 10.1016/j.jpdc.2012.02.007.
Cited on pages 4, 9, 22, 35, 92, 94.

[DGPW14] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. **Robust Distance Queries on Massive Networks**. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*. Volume 8737 of Lecture Notes in Computer Science (LNCS), pages 321–333. Springer, 2014. DOI: 10.1007/978-3-662-44777-2_27.
Cited on page 20.

[DGPW17] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. **Customizable Route Planning in Road Networks**. In *Transportation Science* volume 51:2, pages 566–591, INFORMS, 2017. DOI: 10.1287/trsc.2014.0579.
Cited on page 20.

[DGRW11] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. **Graph Partitioning with Natural Cuts**. In *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS'11)*, pages 1135–1146. IEEE Computer Society, 2011. DOI: 10.1109/IPDPS.2011.108.
Cited on page 19.

[DGW11] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. **Faster Batched Shortest Paths in Road Networks**. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'11)*. Volume 20 of OpenAccess Series in Informatics (OASIcs), pages 53–63. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011. DOI: 10.4230/OASIcs.ATMOS.2011.52.
Cited on pages 22, 96.

[DGWZ08] Daniel Delling, Kalliopi Giannakopoulou, Dorothea Wagner, and Christos Zaroliagis. **Timetable Information Updating in Case of Delays: Modeling Issues**. In Technical report, 2008.
Cited on page 154.

[Dij59] Edsger W. Dijkstra. **A Note on Two Problems in Connexion with Graphs**. In *Numerische Mathematik* volume 1, pages 269–271, 1959. DOI: 10.1007/BF01386390.
Cited on pages 2, 17, 31.

[DK19] Mattia D'Emidio and Imran Khan. **Dynamic Public Transit Labeling**. In *Proceedings of the 19th International Conference on Computational Science and Its Applications (ICCSA'19)*. Volume 11619 of Lecture Notes in Computer Science (LNCS), pages 103–117. Springer, 2019. DOI: 10.1007/978-3-030-24289-3_9.
Cited on page 154.

[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. DOI: 10.1145/2133803.2345678.
Cited on pages 24, 25, 28.

[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 (LNCS), pages 347–361. Springer, 2008. DOI: 10.1007/978-3-540-68552-4_26.
Cited on pages 23, 24, 25, 27, 33, 153.

[DN12] Daniel Delling and Giacomo Nannicini. **Core Routing on Dynamic Time-Dependent Road Networks**. In *INFORMS Journal on Computing* volume 24:2, pages 187–201, INFORMS, 2012. DOI: 10.1287/ijoc.1110.0448.
Cited on page 22.

[DPSW18] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. **Connection Scan Algorithm**. In *Journal of Experimental Algorithmics (JEA)* volume 23, pages 1.7:1–1.7:56, Association for Computing Machinery (ACM), 2018. DOI: 10.1145/3274661.
Cited on pages 3, 26, 27, 37, 48, 154.

[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 (ES'09)*. Volume 5757 of Lecture Notes in Computer Science (LNCS), pages 587–598. Springer, 2009. DOI: 10.1007/978-3-642-04128-0_53.
Cited on pages 28, 90.

[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 (LNCS). Springer, 2009, pages 182–206. DOI: 10.1007/978-3-642-05465-5_7.
Cited on pages 24, 25, 37, 154.

[DPW15a] Daniel Delling, Thomas Pajor, and Renato F. Werneck. **Round-Based Public Transit Routing**. In *Transportation Science* volume 49:3, pages 591–604, INFORMS, 2015. DOI: 10.1287/trsc.2014.0534.
Cited on pages 3, 5, 26, 27, 39, 41, 42, 48, 60, 153.

[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. DOI: 10.1145/2699886.
Cited on pages 29, 34.

[DPWZ09] Daniel Delling, Thomas Pajor, Dorothea Wagner, and Christos Zaroliagis. **Efficient Route Planning in Flight Networks**. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'09)*. Volume 12 of OpenAccess Series in Informatics (OASIcs), pages 7:1–7:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2009. DOI: 10.4230/OASIcs.ATMOS.2009.2145.
Cited on page 28.

[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 66:1–6:4. Association for Computing Machinery (ACM), 2015. DOI: 10.1145/2820783.2820856.
Cited on page 51.

[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. DOI: 10.1145/2886843.
Cited on pages 2, 19, 20.

[DW09] Daniel Delling and Dorothea Wagner. **Pareto Paths with SHARC**. In *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*. Volume 5526 of Lecture Notes in Computer Science (LNCS), pages 125–136. Springer, 2009. DOI: 10.1007/978-3-642-02011-7_13.
Cited on page 22.

[DW15] Daniel Delling and Renato F. Werneck. **Customizable Point-of-Interest Queries in Road Networks**. In *IEEE Transactions on Knowledge and Data Engineering* volume 27:3, pages 686–698, IEEE Computer Society, 2015. DOI: 10.1109/TKDE.2014.2345386.
Cited on pages 21, 103.

[EFS11] Jochen Eisner, Stefan Funke, and Sabine Storandt. **Optimal Route Planning for Electric Vehicles in Large Networks**. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI'11)*, pages 1108–1113. AAAI Press, 2011. DOI: 10.1609/aaai.v25i1.7991.
Cited on page 17.

[EG08] David Eppstein and Michael T. Goodrich. **Studying (Non-Planar) Road Networks Through an Algorithmic Lens**. In *Proceedings of the 16th SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 16:1–16:10. Association for Computing Machinery (ACM), 2008. DOI: 10.1145/1463434.1463455.
Cited on pages 2, 19.

[EP14] Alexandros Efentakis and Dieter Pfoser. **GRASP. Extending Graph Separators for the Single-Source Shortest-Path Problem**. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*. Volume 8737 of Lecture Notes in Computer Science (LNCS), pages 358–370. Springer, 2014. DOI: 10.1007/978-3-662-44777-2_30.
Cited on pages 21, 22, 103.

[For56] L. R. Ford. **Network Flow Theory**. Rand Corporation, 1956. DOI: 10.1007/0-387-22633-8_9.
Cited on page 17.

[FS13] Stefan Funke and Sabine Storandt. **Polynomial-Time Construction of Contraction Hierarchies for Multi-Criteria Objectives**. In *Proceedings of the 15th Workshop on Algorithm Engineering and Experiments (ALENEX'13)*, pages 41–54. Society for Industrial and Applied Mathematics (SIAM), 2013. DOI: 10.1137/1.9781611972931.4.
Cited on page 23.

[FS15] Stefan Funke and Sabine Storandt. **Personalized Route Planning in Road Networks**. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 45:1–45:10. Association for Computing Machinery (ACM), 2015. DOI: 10.1145/2820783.2820830.
Cited on pages 22, 23.

[FT87] Michael L. Fredman and Robert E. Tarjan. **Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms**. In *Journal of the ACM* volume 34:3, pages 596–615, Association for Computing Machinery (ACM), 1987. DOI: 10.1145/28869.28874.
Cited on page 32.

[GBCI11] Johann Gamper, Michael Böhlen, Willi Cometti, and Markus Innerebner. **Defining Isochrones in Multimodal Spatial Networks**. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM'11)*, pages 2381–2384. Association for Computing Machinery (ACM), 2011. DOI: 10.1145/2063576.2063972.
Cited on page 89.

[GBI12] Johann Gamper, Michael Böhlen, and Markus Innerebner. **Scalable Computation of Isochrones with Network Expiration**. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management (SSDBM'12)*. Volume 7338 of Lecture Notes in Computer Science (LNCS), pages 526–543. Springer, 2012. DOI: 10.1007/978-3-642-31235-9_35.
Cited on page 89.

[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. DOI: 10.1007/978-3-642-13193-6_7.
Cited on page 25.

[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. Society for Industrial and Applied Mathematics (SIAM), 2005. DOI: 10.5555/1070432.1070455.
Cited on page 18.

[GKS10] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. **Route Planning with Flexible Objective Functions**. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, pages 124–137. Society for Industrial and Applied Mathematics (SIAM), 2010. DOI: 10.1137/1.9781611972900.12.
Cited on page 23.

[Goe+13] Marc Goerigk, Sascha Heße, Matthias Müller-Hannemann, Marie Schmidt, and Anita Schöbel. **Recoverable Robust Timetable Information**. In *Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'13)*. Volume 33 of OpenAccess Series in Informatics (OASIcs), pages 1–14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013. DOI: 10.4230/OASIcs.ATMOS.2013.1.
Cited on page 154.

[Goe+14] Marc Goerigk, Marie Schmidt, Anita Schöbel, Martin Knoth, and Matthias Müller-Hannemann. **The Price of Strict and Light Robustness in Timetable Information**. In *Transportation Science* volume 48:2, pages 225–242, INFORMS, 2014. DOI: 10.1287/trsc.2013.0470.
Cited on page 154.

[GPZ19] Kalliopi Giannakopoulou, Andreas Paraskevopoulos, and Christos D. Zaroliagis. **Multimodal Dynamic Journey-Planning**. In *Algorithms* volume 12:10, pages 213:1–213:16, Multidisciplinary Digital Publishing Institute (MDPI), 2019. DOI: 10.3390/a12100213.
Cited on pages 28, 29, 155.

[Gre18] Greater London Authority. **Mayor's Transport Strategy**. Accessed October 23rd, 2023. 2018. URL: https://www.london.gov.uk/sites/default/files/mayors-transport-strategy-2018.pdf.
Cited on page 1.

[GSSS23] Ernestine Großmann, Jonas Sauer, Christian Schulz, and Patrick Steil. **Arc-Flags Meet Trip-Based Public Transit Routing**. In *Proceedings of the 21st International Symposium on Experimental Algorithms (SEA'23)*. Volume 265 of Leibniz International Proceedings in Informatics (LIPIcs), pages 16:1–16:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. DOI: 10.4230/LIPIcs.SEA.2023.16.
Cited on page 196.

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. **Exact Routing in Large Road Networks Using Contraction Hierarchies**. In *Transportation Science* volume 46:3, pages 388–404, INFORMS, 2012. DOI: 10.1287/trsc.1110.0401.
Cited on pages 2, 19, 22, 33, 35.

[Han80] Pierre Hansen. **Bicriterion Path Problems**. In *Multiple Criteria Decision Making Theory and Application*, pages 109–127. Springer, 1980. DOI: 10.1007/978-3-642-48782-8_9.
Cited on pages 5, 22, 32, 107.

[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: Ninth DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 41–72. American Mathematical Society (AMS), 2009. DOI: 10.1090/dimacs/074/03.
Cited on page 18.

[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 4:2, pages 100–107, IEEE Computer Society, 1968. DOI: 10.1109/TSSC.1968.300136.
Cited on page 18.

[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 2.5:1–2.5:26, Association for Computing Machinery (ACM), 2009. DOI: 10.1145/1412228.1412239.
Cited on page 20.

[Joh73] Donald B. Johnson. **A Note on Dijkstra's Shortest Path Algorithm**. In *Journal of the ACM* volume 20:3, pages 385–388, Association for Computing Machinery (ACM), 1973. DOI: 10.1145/321765.321768.
Cited on page 17.

[KBB16] Roland Kager, Luca Bertolini, and Marco te Brömmelstroet. **Characterisation of and Reflections on the Synergy of Bicycles and Public Transport**. In *Transportation Research Part A: Policy and Practice* volume 85:1, pages 208–219, Elsevier, 2016. DOI: 10.1016/j.tra.2016.01.015.
Cited on page 1.

[KLC12] Dominik Kirchler, Leo Liberti, and Roberto Wolfler Calvo. **A Label Correcting Algorithm for the Shortest Path Problem on a Multi-Modal Route Network**. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*. Volume 7276 of Lecture Notes in Computer Science (LNCS), pages 236–247. Springer, 2012. DOI: 10.1007/978-3-642-30850-5_21.
Cited on page 29.

[KLPC11] Dominik Kirchler, Leo Liberti, Thomas Pajor, and Roberto Wolfler Calvo. **UniALT for Regular Language Contrained Shortest Paths on a Multi-Modal Transportation Network**. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'11)*. Volume 20 of OpenAccess Series in Informatics (OASIcs), pages 64–75. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011. DOI: 10.4230/OASIcs.ATMOS.2011.64.
Cited on page 29.

[KMPZ22] Spyros Kontogiannis, Paraskevi-Maria-Malevi Machaira, Andreas Paraskevopoulos, and Christos Zaroliagis. **REX: A Realistic Time-Dependent Model for Multimodal Public Transport**. In *Proceedings of the 22nd Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'22)*. Volume 106 of OpenAccess Series in Informatics (OASIcs), pages 12:1–12:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. DOI: 10.4230/OASIcs.ATMOS.2022.12.
Cited on page 23.

[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)*, pages 36–45. Society for Industrial and Applied Mathematics (SIAM), 2007. DOI: 10.5555/2791188.2791192.
Cited on pages 21, 22, 35.

[KSSG17] Nikolaus Krismer, Doris Silbernagl, Günther Specht, and Johann Gamper. **Computing Isochrones in Multimodal Spatial Networks Using Tile Regions**. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM'17)*, pages 33:1–33:6. Association for Computing Machinery (ACM), 2017. DOI: 10.1145/3085504.3085538.
Cited on page 89.

[KV17] Adrian Kosowski and Laurent Viennot. **Beyond Highway Dimension: Small Distance Labels Using Tree Skeletons**. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'17)*, pages 1462–1478. Society for Industrial and Applied Mathematics (SIAM), 2017. DOI: 10.5555/3039686.3039781.
Cited on pages 3, 21.

[Lau09] Ulrich Lauther. **An Experimental Evaluation of Point-to-Point Shortest Path Calculation on Road Networks with Precalculated Edge-Flags**. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 19–40. American Mathematical Society (AMS), 2009. DOI: 10.1090/dimacs/074/02.
Cited on page 18.

[LL20] Vassilissa Lehoux and Christelle Loiodice. **Faster Preprocessing for the Trip-Based Public Transit Routing Algorithm**. In *Proceedings of the 20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'20)*. Volume 85 of OpenAccess Series in Informatics (OASIcs), pages 3:1–3:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. DOI: 10.4230/OASIcs.ATMOS.2020.3.
Cited on pages 43, 73, 74.

[LT79] Richard J. Lipton and Robert E. Tarjan. **A Separator Theorem for Planar Graphs**. In *SIAM Journal on Applied Mathematics* volume 36:2, pages 177–189, Society for Industrial and Applied Mathematics (SIAM), 1979. DOI: 10.1137/0136016.
Cited on page 19.

[Mar07] Karel Martens. **Promoting Bike-and-Ride: The Dutch Experience**. In *Transportation Research Part A: Policy and Practice* volume 41:4, pages 326–338, Elsevier, 2007. DOI: 10.1016/j.tra.2006.09.010.
Cited on page 1.

[Mar84] Ernesto Queirós Vieira Martins. **On a Multicriteria Shortest Path Problem**. In *European Journal of Operational Research* volume 16:2, pages 236–245, Elsevier, 1984. DOI: 10.1016/0377-2217(84)90077-8.
Cited on pages 22, 32.

[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. DOI: 10.1016/j.procs.2013.06.114.
Cited on page 47.

[Möh+06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. **Partitioning Graphs to Speed Up Dijkstra's Algorithm**. In *Journal of Experimental Algorithmics (JEA)* volume 11, pages 2.8:1–2.8:29, Association for Computing Machinery (ACM), 2006. DOI: 10.1007/11427186_18.
Cited on page 18.

[MS07] Matthias Müller-Hannemann and Mathias Schnee. **Finding All Attractive Train Connections by Multi-Criteria Pareto Search**. In *Algorithmic Methods for Railway Optimization*. Volume 4359. Lecture Notes in Computer Science (LNCS). Springer, 2007, pages 246–263. DOI: 10.1007/978-3-540-74247-0_13.
Cited on pages 24, 25.

[MS09] Matthias Müller-Hannemann and Mathias Schnee. **Efficient Timetable Information in the Presence of Delays**. In *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*. Volume 6630. Lecture Notes in Computer Science (LNCS). Springer, 2009, pages 249–272. DOI: 10.1007/978-3-642-05465-5_10.
Cited on page 154.

[MS10] Matthias Müller-Hannemann and Stefan Schirra (editors). **Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice**. Volume 5971 of Lecture Notes in Computer Science (LNCS). Springer, 2010. DOI: https://doi.org/10.1007/978-3-642-14866-8.
Cited on pages 3, 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 (LNCS), pages 67–90. Springer, 2007. DOI: 10.1007/978-3-540-74247-0_3.
Cited on pages 23, 24, 27, 37.

[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 (LNCS), pages 185–198. Springer, 2001. DOI: 10.1007/3-540-44688-5_15.
Cited on page 24.

[Nac95] Karl Nachtigall. **Time Depending Shortest-Path Problems with Applications to Railway Networks**. In *European Journal of Operational Research* volume 83:1, pages 154–166, Elsevier, 1995. DOI: 10.1016/0377-2217(94)E0349-G.
Cited on page 24.

[NDSL12] Giacomo Nannicini, Daniel Delling, Dominik Schultes, and Leo Liberti. **Bidirectional A* Search on Time-Dependent Road Networks**. In *Networks* volume 59:2, pages 240–251, Wiley, 2012. DOI: 10.1002/net.20438.
Cited on page 22.

[Nic66] T. A. J. Nicholson. **Finding the Shortest Route Between Two Points in a Network**. In *The Computer Journal* volume 9, pages 275–280, Oxford University Press, 1966. DOI: 10.1093/comjnl/9.3.275.
Cited on pages 18, 32.

[OCC20] Giulia Oeschger, Páraic Carroll, and Brian Caulfield. **Micromobility and Public Transport Integration: The Current State of Knowledge**. In *Transportation Research Part D: Transport and Environment* volume 89:4, pages 102628:1–102628:21, Elsevier, 2020. DOI: 10.1016/j.trd.2020.102628.
Cited on pages 1, 2.

[Paj09] Thomas Pajor. **Multi-Modal Route Planning**. Diploma Thesis. Karlsruhe Institute of Technology, 2009.
Cited on page 28.

[PS22a] Moritz Potthoff and Jonas Sauer. **Efficient Algorithms for Fully Multimodal Journey Planning**. In *Proceedings of the 22nd Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'22)*. Volume 106 of OpenAccess Series in Informatics (OASIcs), pages 14:1–14:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. DOI: 10.4230/OASIcs.ATMOS.2022.14.
Cited on page 10.

[PS22b] Moritz Potthoff and Jonas Sauer. **Fast Multimodal Journey Planning for Three Criteria**. In *Proceedings of the 24th Workshop on Algorithm Engineering and Experiments (ALENEX'22)*, pages 145–157. Society for Industrial and Applied Mathematics (SIAM), 2022. DOI: 10.1137/1.9781611977042.12.
Cited on page 9.

[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. DOI: 10.1007/978-1-4615-5757-9_11.
Cited on page 24.

[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. DOI: 10.1145/1227161.1227166.
Cited on pages 3, 15, 23, 24, 35, 109.

[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 (SEA^2)*. Volume 11544 of Lecture Notes in Computer Science (LNCS), pages 237–247. Springer, 2019. DOI: 10.1007/978-3-030-34029-2_16.
Cited on pages 5, 29, 48, 53, 71, 85.

[San09] Peter Sanders. **Algorithm Engineering – An Attempt at a Definition**. In *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*. Volume 5760 of Lecture Notes in Computer Science (LNCS), pages 321–340. Springer, 2009. DOI: 10.1007/978-3-642-03456-5_22.
Cited on pages 3, 4.

[Sau18] Jonas Sauer. **Faster Public Transit Routing with Unrestricted Walking**. Master's Thesis. Karlsruhe Institute of Technology, 2018.
Cited on pages 6, 7, 53, 195.

[SGv14] Alexander Spickermann, Volker Grienitz, and Heiko A. von der Gracht. **Heading Towards a Multimodal City of the Future?: Multi-Stakeholder Scenarios for Urban Mobility**. In *Technological Forecasting and Social Change* volume 89, pages 201–221, Elsevier, 2014. DOI: 10.1016/j.techfore.2013.08.036.
Cited on page 1.

[SHP11] Johannes Schlaich, Udo Heidl, and Regine Pohlner. **Verkehrsmodellierung für die Region Stuttgart – Schlussbericht**. Unpublished, 2011.
Cited on page 47.

[SS07] Peter Sanders and Dominik Schultes. **Dynamic Highway-Node Routing**. In *Proceedings of the 6th International Workshop on Experimental and Efficient Algorithms (WEA'07)*. Volume 4525 of Lecture Notes in Computer Science (LNCS), pages 66–79. Springer, 2007. DOI: 10.1007/978-3-540-72845-0_6.
Cited on page 19.

[SS09] Peter Sanders and Dominik Schultes. **Robust, Almost Constant Time Shortest-Path Queries in Road Networks**. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 193–218. American Mathematical Society (AMS), 2009. DOI: 10.1090/dimacs/074/08.
Cited on page 20.

[SS12] Peter Sanders and Dominik Schultes. **Engineering Highway Hierarchies**. In *Journal of Experimental Algorithmics (JEA)* volume 17, pages 1.6:1–1.6:40, Association for Computing Machinery (ACM), 2012. DOI: 10.1145/2133803.2330080.
Cited on page 19.

[Ste23] Patrick Steil. **Optimal FIFO Grouping in Public Transit Networks**. In *ArXiv e-prints* 2308.06629, URL: https://arxiv.org/abs/2308.06629. Technical report, 2023.
Cited on page 50.

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. **Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport**. In *Journal of Experimental Algorithmics (JEA)* volume 5, pages 12:1–12:23, Association for Computing Machinery (ACM), 2000. DOI: 10.1145/351827.384254.
Cited on page 19.

[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 (LNCS), pages 43–59. Springer, 2002. DOI: 10.1007/3-540-45643-0_4.
Cited on pages 19, 25.

[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, URL: https://arxiv.org/abs/1909.08519. Technical report, 2019.
Cited on pages 47, 89.

[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 (ACM), 2019. DOI: 10.1145/3347146.3359354.
Cited on page 89.

[SWZ20a] Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. **An Efficient Solution for One-To-Many Multi-Modal Journey Planning**. In *Proceedings of the 20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'20)*. Volume 85 of OpenAccess Series in Informatics (OASIcs), pages 1:1–1:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. DOI: 10.4230/OASIcs.ATMOS.2020.1.
Cited on page 9.

[SWZ20b] 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. DOI: 10.4230/LIPIcs.SEA.2020.16.
Cited on pages 7, 151.

[SWZ20c] 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 (OASIcs), pages 4:1–4:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. DOI: 10.4230/OASIcs.ATMOS.2020.4.
Cited on page 9.

[SWZ21] Ben Strasser, Dorothea Wagner, and Tim Zeitz. **Space-Efficient, Fast and Exact Routing in Time-Dependent Road Networks**. In *Algorithms* volume 14:3, pages 90:1–90:31, Multidisciplinary Digital Publishing Institute (MDPI), 2021. DOI: 10.3390/a14030090.
Cited on page 22.

[Tig+11] Miles Tight, Paul Timms, David Banister, Jemma Bowmaker, Jonathan Copas, Andy Day, David Drinkwater, Moshe Givoni, Astrid Gühnemann, Mary Lawler, James Macmillen, Andrew Miles, Niamh Moore, Rita Newton, Dong Ngoduy, Marcus Ormerod, Maria O'Sullivan, and David Watling. **Visions for a Walking and Cycling Focussed Urban Transport System**. In *Journal of Transport Geography* volume 19:6, pages 1580–1589, Elsevier, 2011. DOI: 10.1016/j.jtrangeo.2011.03.011.
Cited on page 1.

[vB04] Rob van Nes and Piet H. L. Bovy. **Multimodal Traveling and Its Impact on Urban Transit Network Design**. In *Journal of Advanced Transportation* volume 38:3, pages 225–241, Wiley, 2004. DOI: 10.1002/atr.5670380302.
Cited on page 1.

[WBN17] Christoph Willing, Tobias Brandt, and Dirk Neumann. **Intermodal Mobility**. In *Business & Information Systems Engineering* volume 59:3, pages 173–179, Springer, 2017. DOI: 10.1007/s12599-017-0471-7.
Cited on pages 1, 2.

[Wit15] Sascha Witt. **Trip-Based Public Transit Routing**. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA'15)*. Volume 9294 of Lecture Notes in Computer Science (LNCS), pages 1025–1036. Springer, 2015. DOI: 10.1007/978-3-662-48350-3_85.
Cited on pages 3, 27, 42, 48, 84.

[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 (ATMOS'16)*. Volume 54 of OpenAccess Series in Informatics (OASIcs), pages 10:1–10:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. DOI: 10.4230/OASIcs.ATMOS.2016.10.
Cited on pages 27, 48, 94.

[Wit21] Sascha Witt. **Extending the Time Horizon: Efficient Public Transit Routing on Arbitrary-Length Timetables**. In *ArXiv e-prints* 2109.14143, URL: https://arxiv.org/abs/2109.14143. Technical report, 2021.
Cited on page 154.

[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 (ATMOS'17)*. Volume 59 of OpenAccess Series in Informatics (OASIcs), pages 7:1–7:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. DOI: 10.4230/OASIcs.ATMOS.2017.7.
Cited on pages 4, 6, 28, 48, 50, 51, 80.

[Zün20] Tobias Zündorf. **Multimodal Journey Planning and Assignment in Public Transportation Networks**. PhD thesis. Karlsruhe Institute of Technology, 2020. DOI: 10.5445/IR/1000145076.
Cited on page 4.

List of Acronyms

ALT	A*, landmarks, triangle inequality Used on pages 18, 19, 21, 22, 25, 29
ANR	Access Node Routing Used on page 29
BFS	Breadth-first search Used on pages 3, 27, 35, 39, 91
BM-RAPTOR	Bounded McRAPTOR Used on pages 106, 107, 113, 115–119, 128, 129, 132, 149, 150
CATCHUp	Customizable Approximated Time-Dependent CH Through Unpacking Used on page 22
CCH	Customizable Contraction Hierarchies Used on pages 20, 22, 23
CH	Contraction Hierarchies Used on pages 2, 9, 19–23, 25, 29, 31, 33–35, 42, 53–55, 66–73, 75, 80–82, 89–92, 97–100, 103, 109, 112, 118, 119, 122, 126, 141, 142, 194
CHASE	CH + Arc-Flags + HNR Used on pages 21, 25

CRP	Customizable Route Planning Used on pages 20, 22, 23
CSA	Connection Scan Algorithm Used on pages 3, 7, 9, 26–28, 30, 37–40, 47, 53–55, 69, 79–81, 85, 87, 89, 90, 93, 94, 96–101, 103, 154
DAG	Directed acyclic graph Used on pages 3, 12, 22, 26, 27, 34, 37
DFS	Depth-first search Used on page 92
FIFO	First-in, first-out Used on pages 43, 44, 94
HH	Highway Hierarchies Used on pages 19, 22
HL	Hub Labeling Used on pages 20, 21, 26, 30, 47, 53, 54, 71, 85, 86
HNR	Highway-Node Routing Used on page 19
HydRA	Hybrid Routing Algorithm Used on pages 135, 142–144, 146, 150–152, 194
LCSPP	Label-constrained shortest path problem Used on page 29
MCR	Multimodal Multicriteria RAPTOR Used on pages 4, 29, 30, 34, 42, 47, 53, 107, 109, 111, 117, 125–127, 130, 132, 133, 138, 140, 141, 145, 150, 151
McRAPTOR	Multicriteria RAPTOR Used on pages 27, 29, 40, 42, 47, 106, 112, 113, 116, 118, 125, 130, 135, 140–143, 145, 148–151, 194
MCSA	Multimodal CSA Used on pages 79–81, 85, 96–101, 103
McTB	Multicriteria Trip-Based Routing Used on pages 106, 107, 109, 112, 114, 119–122, 125, 130, 132, 133, 141, 142, 194
McULTRA	Multicriteria ULTRA Used on pages 107, 109–113, 118, 119, 121, 123, 126, 128–133, 135, 139, 140, 147–149, 151, 194, 195

MLO	Multi-level overlays Used on pages 19–21
MR	Multimodal RAPTOR Used on pages 42, 53–55, 58–62, 64–66, 80–87, 94, 96, 100–103, 109, 111, 126, 128, 154–156, 174–176, 178, 180–188, 192
OSM	OpenStreetMap Used on pages 48, 49, 145
PHAST	PHAST Hardware-Accelerated Shortest Path Trees Used on pages 9, 22, 35, 67, 90–92, 94, 96–101, 103
POI	Point-of-interest Used on pages 21, 89, 103
RAM	Random-access machine Used on page 7
RAPTOR	Round-bAsed Public Transit Optimized Router Used on pages 3–5, 7–10, 27, 28, 30, 38–43, 45, 47, 53–55, 59, 60, 69, 70, 79–86, 90, 93, 94, 100–103, 105, 106, 112, 115, 116, 118, 119, 122, 125, 127, 128, 132, 133, 135, 141, 150, 154, 155
RPHAST	Restricted PHAST Used on pages 22, 35, 90–94, 100, 101, 103
rRAPTOR	Range RAPTOR Used on pages 42, 54, 60–64, 174
SHARC	Shortcuts + Arc-Flags Used on pages 21, 22, 25
SPCS	Self-Pruning Connection-Setting Used on page 25
TB	Trip-Based Routing Used on pages 3, 4, 7–10, 27, 28, 42–46, 48, 53, 55, 66, 67, 69–71, 73–75, 78, 79, 81–84, 87, 90, 94–96, 100–103, 105–108, 112, 114, 119, 121, 122, 128, 133, 135, 142, 149–151, 154, 155, 167, 180, 185–188, 192–194
TB-CST	Trip-Based Routing with Condensed Search Trees Used on pages 27, 47
TNR	Transit-Node Routing Used on pages 20, 21, 29
TNT	Types aNd Thresholds Used on pages 30, 133

TP	Transfer Patterns Used on pages 26, 28
UB-HydRA	Bounded ULTRA-HydRA Used on pages 136, 142, 143, 149–151
UBM-RAPTOR	Bounded ULTRA-McRAPTOR Used on pages 118, 128, 129, 132, 135, 140, 141, 143, 149–151
UBM-TB	Bounded ULTRA-McTB Used on pages 107, 119, 122, 128, 129, 132, 135, 142, 143
UCCH	User-Constrained Contraction Hierarchies Used on pages 29, 34
ULTRA	UnLimited TRAnsfers Used on pages 8–10, 15, 45, 54, 55, 59–63, 65–75, 77–87, 89–92, 94–103, 105, 107, 109–112, 118–123, 125, 127–135, 137–142, 148–151, 153, 155, 156, 160, 174, 175, 180, 182, 185–189, 192–195