Karlsruhe Institute of Technology

# Faster Public Transit Routing with Unrestricted Walking

Master Thesis of

Jonas Sauer

At the Department of Informatics
Institute of Theoretical Informatics

| | |
|---|---|
| Reviewers: | Prof. Dr. Dorothea Wagner |
| | Prof. Dr. Peter Sanders |
| Advisors: | Tobias Zündorf, M.Sc. |
| | Valentin Buchhold, M.Sc. |
| | Moritz Baum, M.Sc. |

Time Period: 1st November 2017 – 30th April 2018

**Statement of Authorship**

Karlsruhe, 30th April, 2018

## Abstract

Many routing algorithms for public transit networks restrict the amount of walking that is allowed between stops. It is often argued that long footpaths rarely improve travel times and therefore do not need to be represented. However, this assumption has recently been called into question [WZ17]. On the other hand, existing algorithms that allow for unrestricted walking use expensive Dijkstra searches to explore the footpaths and are therefore significantly slower than algorithms with restricted walking. This thesis is therefore aimed at developing faster algorithms for public transit routing with unrestricted walking. To this end, we present a speedup technique for the RAPTOR algorithm that precomputes necessary footpaths, thereby removing the need for Dijkstra searches and speeding up the footpath exploration. Furthermore, we propose two algorithms that adapt existing speedup techniques originally developed for other scenarios: Route-Flags, which is based on Arc-Flags, and ML-RAPTOR, which adapts ideas from Customizable Route Planing and Connection Scan Accelerated. An experimental study on the public transit network of Switzerland shows that the precomputation of footpaths offers a substantial speedup compared to existing algorithms, while Route-Flags and ML-RAPTOR only achieve small speedups.

## Deutsche Zusammenfassung

Viele Routing-Algorithmen für öffentliche Verkehrsnetzwerke beschränken die Länge von Fußwegen, die zwischen Stationen erlaubt sind. Es wird oft argumentiert, dass lange Fußwege nur selten die Reisezeit verbessern und deswegen nicht repräsentiert werden muss. Diese Annahme wurde jedoch kürzlich in Frage gestellt [WZ17]. Auf der anderen Seite verwenden existierende Algorithmen, die unbeschränktes Laufen erlauben, teure Dijkstra-Suchen, um die Fußwege zu erkunden, und sind deshalb deutlich langsamer als Algorithmen mit beschränktem Laufen. Diese Arbeit beschäftigt sich deshalb damit, schnellere Routing-Algorithmen für öffentliche Verkehrsnetzwerke mit unbeschränktem Laufen zu entwickeln. Hierzu präsentieren wir eine Beschleunigungstechnik für den RAPTOR-Algorithmus, die benötigte Fußwege vorberechnet. Dadurch werden die Dijkstra-Suchen überflüssig und die Erkundung der Fußwege wird beschleunigt. Weiterhin stellen wir zwei Algorithmen vor, die an existierende Beschleunigungstechniken angelehnt sind, die für andere Szenarien entwickelt wurden: Route-Flags basiert auf Arc-Flags, während ML-RAPTOR Ideen von Customizable Route Planning und Connection Scan Accelerated aufgreift. Eine experimentelle Studie auf dem öffentlichen Verkehrsnetzwerk der Schweiz zeigt, dass die Vorberechnungstechnik für Fußwege eine deutliche Beschleunigung gegenüber existierenden Algorithmen aufweist, während Route-Flags and ML-RAPTOR nur eine geringe Beschleunigung erzielen.

# Contents

# 1. Introduction

Route planning in road networks has seen considerable advances over the last decade. Fundamentally, the routing problem can be solved by modeling the road network as a graph and using Dijkstra's algorithm [Dij59] to find shortest paths. However, Dijkstra's algorithm is too slow in practice and is therefore commonly sped up with auxiliary data computed in one or more preprocessing phases. A variety of such speedup techniques have been developed, with different tradeoffs between preprocessing time, memory consumption and query time. An overview of the state of the art is given in [BDG+16].

Many speedup techniques exploit the hierarchical nature of road networks, where certain roads, such as highways, are more important for long-distance journeys than others. One such technique is Contraction Hierarchies (CH) [Gei08], which ranks vertices according to their "importance" and iteratively removes less important vertices while inserting shortcut edges that bypass them. This creates a hierarchy of shortcuts that lead toward more important vertices in the road network. Another hierarchical technique is Customizable Route Planning (CRP) [DGPW11], which uses two preprocessing phases: In the first phase, the graph is recursively partitioned into cells to create a nested multi-level partition. The second phase precomputes shortest paths between the boundary vertices of each cell, allowing the query to bypass unimportant cells quickly. The advantage of using two preprocessing phases is that the first phase is independent of the metric that is used for the edge weights. This makes it possible to change the metric without having to repeat the first phase, which takes much longer than the second phase. This is useful for scenarios where the metric changes frequently, for example when incorporating up-to-date traffic information.

Another class of speedup techniques is based on using precomputed data to direct the search towards the goal. A classic example of such a technique is A* search [HNR68], which precomputes a *potential function* that gives a lower bound on the remaining distance to the target vertex. A more recent technique is Arc-Flags [HKMS09], which divides the graph into cells and labels each edge with a set of flags that indicate whether the edge is necessary for reaching a certain cell.

Besides road networks, another common application of routing algorithms are public transit networks, which represent public transportation vehicles such as buses and trains. Unlike road networks, public transit networks are inherently time-dependent, since the vehicles depart at predetermined times. Instead of finding shortest paths, it is therefore common to ask for journeys that minimize the arrival time at the target. However, this is often not enough. Passengers are typically interested in avoiding transfers between different vehicles as much as possible, since they are inconvenient and pose the risk of missing the next connection if a vehicle is delayed. It is therefore common in public transit routing to ask for a set of journeys that are Pareto-optimal according to the two criteria of arrival time and number of transfers.

Compared to road networks, the state of the art for routing on public transit networks is less advanced. Early approaches typically modeled the public transit network as a graph on which Dijkstra's algorithm could be applied, just as for road networks. The two most commonly used models were the *time-dependent* and *time-expanded* models. In the time-dependent model, the stops of the public transit network are modeled as vertices of the graph while the connections between stops are modeled as edges. In order to represent the time-dependency of the public transit connections, time-dependent functions are used as edge weights. This leads to a compact graph representation of the network, but it requires algorithms that can handle functions as edge weights [BJ04]. The time-expanded model removes the need for time-dependent edges by encoding the time dependencies directly into the structure of the network. Here, vertices represent events such as the arrival or departure of a vehicle at a stop, while edges represent elementary connections between two stops at a specific point in time. This enables a straightforward Dijkstra search to be used, but at the expense of a much larger graph [PS98].

When using graph-based models, speedup techniques for Dijkstra's algorithm that were originally developed for road networks can be applied directly to public transit networks. However, these direct adaptations are often less successful than on road networks, offering smaller speedups [BDGM09] [BDG$^+$16]. This is at least partly due to the fact that many of these speedup techniques exploit structural features of road networks that are less pronounced in public transit networks [Bas09]. An example of a speedup technique that was developed specifically for public transit networks with a graph-based model is Transfer Patterns [BCE$^+$10]. This technique precomputes for each optimal journey in the network the sequence of stops at which passengers must switch between vehicles. These sequences are stored in a directed acyclic graph, which can be used to speed up queries. This allows for very fast queries, but at the expense of a high preprocessing time and memory consumption. In order to make this approach feasible for very large networks, a speedup technique called Scalable Transfer Patterns [BHS16] has been developed, which employs clustering to reduce the preprocessing effort.

As a reaction to the problems associated with graph-based modeling approaches, several algorithms have been developed in recent years that are not based on graphs and Dijkstra's algorithm and instead use models that adhere more closely to the inherent timetable structure of public transit networks. One such technique is RAPTOR (Round-bAsed Public Transit Optimized Router) [DPW15], whose model is based on *trips*, which are sequences of stops served by a specific vehicle. RAPTOR employs a round-based dynamic programming approach where each round extends optimal solutions by one more trip. A similar approach is called trip-based public transit routing [Wit15], which precomputes the set of all necessary transfers between trips and explores the network in a manner similar to breadth-first search. The Connection Scan Algorithm (CSA) [DPSW13] is based on the even smaller unit of *connections*, which represent a vehicle traveling from one stop to another without any intermediate stops. CSA scans all connections in the network in ascending order of departure time to determine if they can be used to extend previously found optimal journeys.

While these algorithms typically outperform unpreprocessed graph-based approaches, they are still too slow for interactive queries on larger, country-sized networks. To remedy this, several speedup techniques have been developed for these algorithms: Connection Scan

Accelerated (ACSA) [SW14] adapts the ideas of CRP to the CSA algorithm, using multi-level partitions to prune the set of connections that need to be scanned. HypRAPTOR [DDPZ17], a partitioning-based speedup technique for RAPTOR, partitions the routes of the network instead of the stops. This is achieved by constructing a hypergraph where the vertices represent the routes and hyperedges represent stops that are shared by multiple routes. Queries can then be restricted to the cells containing the source and target vertex, as well as a set of precomputed routes in-between.

**Unrestricted Walking.** Public transit networks are commonly augmented with footpaths which allow passengers to transfer between stops by walking. However, most algorithms place restrictions on these footpaths. Both RAPTOR and CSA require that the graph that represents the footpaths is transitively closed. This has the advantage that optimal footpaths between stops are always represented by a single edge, but it also severely restricts the amount of footpaths that can be represented. The restriction can be lifted by employing Dijkstra's algorithm to find footpaths between stops. While this makes it possible to use arbitrary, unrestricted footpath graphs, it also severely impacts the performance of the algorithms.

Until recently, unrestricted walking had only been considered in the context of multimodal routing, where multiple modes of transportation are combined. In this context, the unrestricted footpath graph simply constitutes one of several available transportation networks. Multimodal algorithms that can solve public transit queries with unrestricted walking include UCCH (User-Constrained Contraction Hierarchies) [DPW12] and MCR (multimodal multicriteria RAPTOR) [DDP$^+$13]. The special case where unrestricted walking is only allowed at the start and the end of the journey can be handled by Access Node Routing (ANR) [DPW09]. However, none of these algorithms can solve *range queries*, where we are given a range of departure times and are interested in optimal journeys for every departure time in that range.

Wagner and Zündorf [WZ17] recently developed an algorithm based on RAPTOR that can solve point-to-point range queries with unrestricted walking. It is based on running alternating forward and backward RAPTOR queries in order to determine the boundaries of the time interval in which each found journey is optimal. However, because it employs both forward and backward searches, it cannot solve *one-to-all* queries, where we ask for optimal journeys from a given source stop to every other stop in the network. Furthermore, like most algorithms that allow for unrestricted walking, its running time is dominated by the time it takes to explore the footpath graph.

Imposing restrictions on the footpath graph causes optimal journeys to not be found. This is already evidenced by the fact that in a Pareto-optimal setting where we minimize both travel time and number of transfers, the shortest journey involving direct walking is always Pareto-optimal, since it does not include any transfers. More generally, for any Pareto-optimal journey with $k$ trips, there is also a Pareto-optimal journey with $k-1$ trips that involves replacing one trip with a footpath. Especially for long-distance journeys, this footpath can be very long and may not be included in a restricted footpath graph. Of course, in practice there are limits to how far a passenger is willing to walk during a journey. For example, no passenger would consider direct walking viable when traveling from Karlsruhe to Berlin.

The footpath restrictions are often justified with the argument that walking beyond a certain distance or time limit (e.g., 15 minutes) is not desired by users and generally does not improve the travel time in realistic journeys. Both of these claims must be called into question. Wagner and Zündorf [WZ17] conducted experiments on the public transit networks of Switzerland and Germany that compared the travel times achieved with

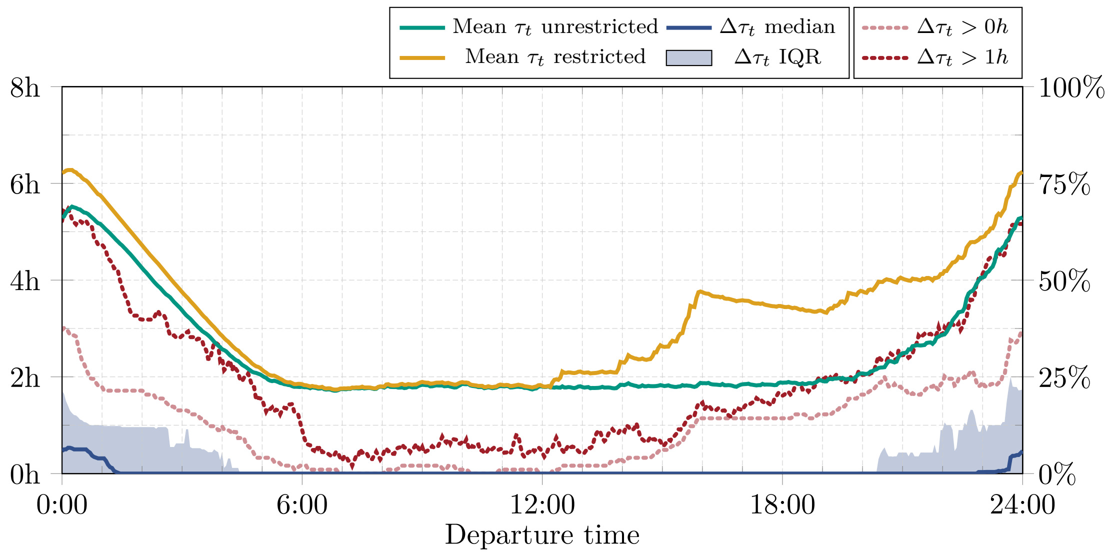


Figure 1.1: Comparison of the optimal travel times throughout the day on the public transit network of Switzerland, based on an experiment from [WZ17]. 100 random queries with distance rank $2^{16}$ were evaluated. The green curve depicts the average travel time with unrestricted walking, while the yellow curve depicts the average travel time when using a transitively closed footpath where direct footpaths are restricted to 15 minutes. The blue curve shows the median of the travel time difference, while the light blue shaded area depicts the interquartile range (IQR). The two red dotted curves use the right y-axis instead of the left one. The light red dotted curve depicts the percentage of queries where the travel times found on both network instances were not identical. The percentage of queries where the travel time difference was greater than one hour is indicated by the dark red dotted curve.

unrestricted walking to those achieved with a transitively closed footpath graph. Figure 1.1 shows the results of a recreation of this experiment on the Switzerland network, using a transitively closed footpath graph where direct footpaths are restricted to 15 minutes. It can be seen that the percentage of optimal journeys whose arrival time is improved by allowing unrestricted walking averages around 10% during the daytime. During the nighttime, the percentage is much higher, reaching more than 50% at some points, due to fewer public transit connections being available and walking therefore being more common. The amount of journeys whose travel time is reduced by more than one hour is smaller, but still non-negligible and even rising to more than 35% around midnight.

The majority of long footpaths are needed at the start and the end of a journey, before and after the first trip. This is especially common when the source or target stop is located in a rural area served by few vehicles. Especially when departing at night, there are often no departing vehicles available until the next morning, so it is often faster to walk to a nearby town than to wait until the next morning when the next bus departs. However, there are also optimal journeys that require long intermediate footpaths between trips. Figures 1.2 and 1.3 show examples of journeys that are improved by long intermediate footpaths. The journey in Figure 1.2 is representative of many such journeys, featuring hour-long walks during the night. Obviously, these journeys would not be considered viable by most users. However, Figure 1.3 shows a more reasonable case where a 23-minute walk can significantly reduce the length of a morning-time commute. While these cases are fairly rare, they do exist and are frequent enough to warrant allowing unrestricted walking.

Regarding the claim that long walks are not desired by most users, while it is certainly true that extremely long walks such as the one from Karlsruhe to Berlin are clearly undesirable

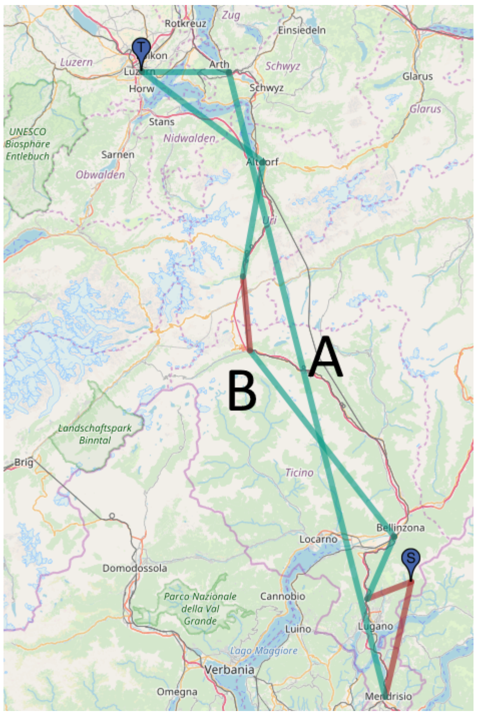


Figure 1.2: Two journeys from Maglio di Collia, Paese to the Lucerne main railway station for the departure time 20:56:19. Footpaths are drawn in red, while trip segments are shown in green. Journey A starts with a seven-hour walking leg to Madrisio and continues with two trip segments, arriving at 7:42:11 on the following day. Journey B starts with a two-and-a-half hour walking leg to Taverne-Torricella. After two trip segments, it features a four-and-a-half hour intermediate walking leg from Airolo to Göschenen, followed by two more trip segments. It arrives at 7:20:00 on the following day and thus saves 22:11 minutes of travel time.

to all users, individual passengers may have different thresholds for how much walking they will accept. Ideally, a routing application should allow users to specify this threshold themselves during queries, rather than setting a predetermined threshold in advance. The 23-minute walk in Figure 1.3 is an example of a walking distance that will be considered unacceptable by some users, but acceptable by others. Allowing a user-determined walking threshold makes it possible to show this journey only to the users who are interested in it.

**Contribution.** The straightforward solution for incorporating unrestricted walking, which involves adding a full footpath graph and traversing it with Dijkstra's algorithm between trips, has the major drawback that Dijkstra's algorithm is fairly slow. Therefore the exploration of the footpath graph will dominate the running time of the algorithm, making public transit routing with unrestricted walking much slower than restricted approaches. Conventional speedup techniques for Dijkstra's algorithm such as goal direction or hierarchical techniques have limited use since the Dijkstra searches that need to be performed have multiple source and target vertices.

The goal of this thesis is to develop speedup techniques aimed at closing the performance gap between unrestricted and restricted approaches, first by reducing the time needed to explore the footpath graph so that it no longer dominates the running time, and then by speeding up the exploration of the actual public transit network as well. We achieve the former with a preprocessing technique that precomputes shortcut edges for all footpaths that are needed for intermediate walking between trips. We then use these shortcuts to speed up the footpath exploration in Wagner and Zündorf's profile RAPTOR algorithm

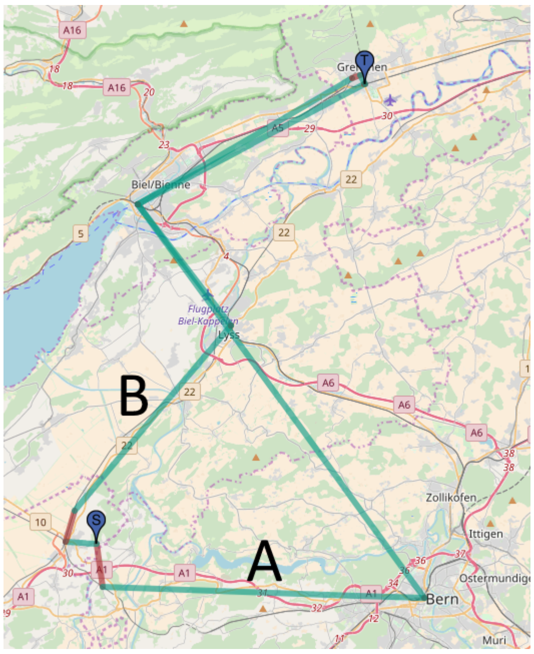


Figure 1.3: Two journeys from the Kerzers Schwimmbad bus stop to the Grenchen Süd railway station for the departure time 7:26:10. Footpaths are drawn in red, while trip segments are shown in green. Journey A starts with a 38-minute walking leg to Ferenbalm-Gurbrü, followed by three trip segments and a short 5-minute walking leg to the destination. It arrives at 10:12:33. Journey B starts with a bus ride to the Kerzers main railway station, followed by a 23-minute intermediate walking leg to another railway station. After four more trip segments and another 9-minute intermediate walking leg, it arrives at the destination at 9:11:00, thus saving slightly over one hour of travel time while requiring less walking overall.

[WZ17]. To achieve the latter goal, we discuss several existing speedup techniques and how they can be adapted for the RAPTOR algorithm. We develop adaptations of Arc-Flags and Connection Scan Accelerated and evaluate them on the Switzerland network, showing that while they achieve a small speedup, they are much less successful than the original techniques due to being less suitable for RAPTOR's route-based model of public transit networks.

**Outline.** The remainder of this thesis is organized as follows: In Chapter 2, we introduce the necessary foundations for the algorithms developed later, including the modeling of the public transit networks, definitions of routing problems and existing algorithms for solving these problems. The speedup techniques developed in the later chapters are based on precomputing subjourneys of optimal journeys. This poses several challenges that do not arise in the context of road networks; these are discussed and solutions are presented in Chapter 3. Chapter 4 presents our preprocessing technique for speeding up the footpath exploration, while techniques for speeding up the entire RAPTOR algorithm, including the route exploration, are presented in Chapter 5. We evaluate the performance of the presented techniques on realistic public transit network data in Chapter 6. Finally, we summarize our findings and give recommendations for future work on the subject in Chapter 7.

# 2. Preliminaries

This chapter recapitulates the basic concepts and notation related to public transit routing that are referenced throughout this thesis, including the modeling of the public transit network, routing problem definitions and algorithms for solving these routing problems.

## 2.1 Modeling the Public Transit Network

As discussed in the introduction, there are many ways of modeling a public transit network, and the best model to use typically depends on the algorithm that is used to solve queries. Early approaches to public transit routing tended to model the network as a graph, where vertices represent stops or stop events and edges represent public transit connections. More recent approaches tend to favor models that adhere more closely to the timetable structure inherent to public transit networks. The most prominent examples are RAPTOR and CSA: RAPTOR's model is based on trips, which are sequences of stops served by a particular vehicle. The model of the CSA algorithm separates the trips further into elementary connections, which represent a vehicle driving directly from one stop to the next.

Since the algorithms developed in this thesis are based on RAPTOR, we will employ RAPTOR's model for the most part. The notation introduced below is based on the original formulation of RAPTOR [DPW15] as well as the notation used by Wagner and Zündorf's algorithm for public transit routing with unrestricted walking [WZ17]. Other models will be needed only to explain ideas from existing algorithms, and will be introduced in more detail as they are needed.

A *public transit network* is a 4-tuple $(\mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, consisting of a set of stops $\mathcal{S}$, a set of trips $\mathcal{T}$, a set of routes $\mathcal{R}$ and a footpath graph $\mathcal{F}$. A small example of a public transit network can be seen in Figure 2.1. A *stop* $v \in \mathcal{S}$ is a location in the public transit network where one or more vehicles halt, allowing passengers to enter or leave the vehicle. Examples of vehicles include trains and buses, so stops can be thought of as train stations or bus stops. The *minimum change time* $\tau_{\text{ch}}(v)$ indicates the minimum time that is required for a passenger to leave one vehicle and enter another at $v$. Including a minimum change time in the model is necessary because stops may represent large stations with multiple platforms. In this case, passengers exiting a vehicle need some time to walk from the platform where they arrived to the one where the next vehicle departs. This is modeled by the minimum change time.

A *trip* $\text{tr} \in \mathcal{T}$ is a sequence of stops that are served consecutively by a specific vehicle. Associated with each stop $v$ in a trip tr are an arrival time $\tau_{\text{arr}}(\text{tr}, v)$ and a departure

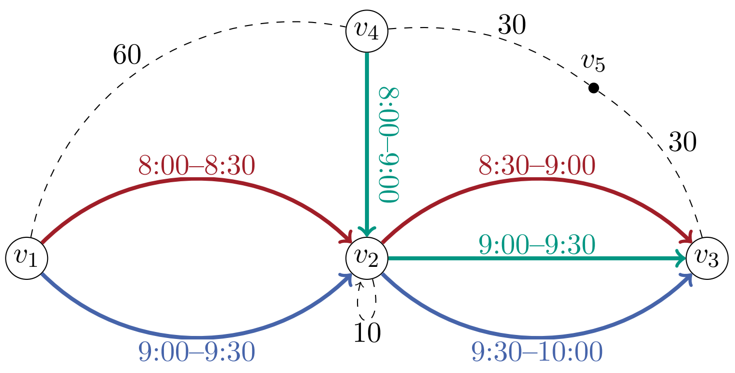


Figure 2.1: An excerpt of a public transit network. Stops are represented by large circles, while vertices of the footpath graph are drawn as small dots. Trips are drawn as a series of directed edges between stops; each edge represents the connection between two consecutive stops $u$ and $v$ in the trip and is labeled with the departure time at $u$ and the arrival time at $v$. Connections belonging to the same trip are drawn in the same color. The red and blue trip belong to the same route since they serve the same sequence and do not overtake each other. Dashed edges represent footpaths and are labeled with their walking time. Minimum change times are represented by dashed loop edges.

time $\tau_{\text{dep}}(\text{tr}, v)$. Naturally, we require that $\tau_{\text{arr}}(\text{tr}, v) \leq \tau_{\text{dep}}(\text{tr}, v)$. The arrival time of the first stop and the departure time of the last stop in a trip are undefined. The 4-tuple $(\text{tr}, v, \tau_{\text{arr}}(\text{tr}, v), \tau_{\text{dep}}(\text{tr}, v))$ is called a *stop event* of tr at $v$.

The trips of the public transit network are partitioned into *routes*: Two trips belong to the same route $r \in \mathcal{R}$ if they serve the same sequence of stops and neither trip overtakes the other. We say that a trip $\text{tr}_1 \in \mathcal{T}$ overtakes another trip $\text{tr}_2 \in \mathcal{T}$ if there are two stops $u, v \in \mathcal{S}$ such that $\text{tr}_1$ arrives or departs at $u$ before $\text{tr}_2$ and $\text{tr}_2$ arrives or departs at $v$ before $\text{tr}_1$. In this case, the two trips would be assigned to different routes despite serving the same sequence of stops. Typically, a route will consist of multiple trips that serve the same sequence of stops at different times. For example, if a route represents a bus line, its trips correspond to the individual buses that serve the line throughout the day.

Finally, the *footgraph graph* is a weighted graph $\mathcal{F} = (V, E, \tau_w)$ with vertices $V$, edges $E$ and an edge weight function $\tau_w : E \to \mathbb{R}_{\geq 0}$. We require that each stop is represented by a vertex in the footpath graph, i.e., $\mathcal{S} \subseteq V$. Each edge $e \in E$ represents a footpath which requires the *walking time* $\tau_w(e)$ to traverse. For the sake of simplicity, we will assume in the following that the shortest path between two vertices $u, v \in V$ is always unique, and denote it by $P_{uv}$. If there are multiple shortest paths between $u$ and $v$, it always suffices to arbitrarily choose one of them. The walking time function can be extended naturally from edges to paths: the walking time of a path $P = (e_1, \ldots, e_k)$ is defined as the sum of the walking times of its edges, i.e., $\tau_w(P) = \sum_{i=1}^{k} \tau_w(e_i)$.

The networks used in this thesis represent the public transit timetable of a single day, i.e., all trips in the network start within the time interval $[0h, 24h]$. Note that trips that start before $24h$ may include stop events after $24h$. In order to ensure that our routing algorithms are able to find overnight journeys that start on one day and end on the next, we make copies of all trips and shift them forward by 24 hours to the interval $[24h, 48h]$. This makes the simplified assumption that the public transit timetable is periodic and repeats daily. However, our model is also able to handle aperiodic timetables that span multiple days.

## 2.2 Journeys

The fundamental objective of public transit route planning is to compute a *journey* between two stops $s, t \in \mathcal{S}$, i.e., a sequence of trips and footpaths that allows passengers to travel from $s$ to $t$. Formally, we define a journey as a sequence of trip segments. A *trip segment* is a triple $(\mathrm{tr}, u, v)$ where $\mathrm{tr} \in \mathcal{T}$ is a trip and $u, v \in \mathcal{S}$ are stops that are served by tr in that order. The trip segment describes the journey of a passenger with tr where they enter the trip at $u$ and exit it at $v$; consequently, we call $u$ the *entering stop* and $v$ the *exiting stop* of the trip segment.

Given source and target stops $s, t \in \mathcal{S}$, an $s$-$t$-journey is a sequence of trip segments $J = ((\mathrm{tr}_1, u_1, v_1), \ldots, (\mathrm{tr}_k, u_k, v_k))$. In order to form a valid journey, it must be possible to transfer between each pair of subsequent trip segments. Transferring can take place either by switching to a different trip at the same stop or by walking to a different stop and entering another trip there. Additionally, if $s \neq u_1$ or $v_k \neq t$, $\mathcal{F}$ must contain a footpath from $s$ to $u_1$ or from $v_k$ to $t$, respectively. Formally, a journey is valid if and only if all of the following conditions hold:

- For each consecutive pair of trip segments $(\mathrm{tr}_i, u_i, v_i)$ and $(\mathrm{tr}_{i+1}, u_{i+1}, v_{i+1})$ $(1 \leq i < k)$, it must be possible to transfer between them. Transferring can take place in one of two ways:
  - If $v_i = u_{i+1}$, we require that $\mathrm{tr}_i \neq \mathrm{tr}_{i+1}$ (otherwise, the two trip segments could be merged into one trip segment $(\mathrm{tr}_i, u_i, v_{i+1})$). The minimum change time at $v_i$ must be sufficient to switch from $\mathrm{tr}_i$ to $\mathrm{tr}_{i+1}$, i.e., $\tau_{\mathrm{arr}}(\mathrm{tr}_i, v_i) + \tau_{\mathrm{ch}}(v_i) < \tau_{\mathrm{dep}}(\mathrm{tr}_{i+1}, v_i)$.
  - If $v_i \neq u_{i+1}$, then $\mathcal{F}$ must contain a $v_i$-$u_{i+1}$-footpath $P$ that allows passengers to reach $u_{i+1}$ in time to enter $\mathrm{tr}_{i+1}$, i.e., $\tau_{\mathrm{arr}}(\mathrm{tr}_i, v_i) + \tau_w(P) < \tau_{\mathrm{dep}}(\mathrm{tr}_{i+1}, u_{i+1})$.
- If $s \neq u_1$, then $s$ and $u_1$ must be connected by a footpath in $\mathcal{F}$.
- If $v_k \neq t$, then $v_k$ and $t$ must be connected by a footpath in $\mathcal{F}$.
- If $J$ is an empty sequence, then $s$ and $t$ must be connected by a footpath in $\mathcal{F}$. We call such a journey a *pure walking journey.*

The trip segments and their connecting footpaths are collectively called the *legs* of the journey. A journey with $k$ trip segments may contain between $k$ and $2k+1$ legs, depending on how many footpaths are needed to connect the trip segments.

Note that the trip sequence alone does not uniquely define a journey, since journeys with different source or target stops may share the same trip sequence. A journey is uniquely defined by the combination of source stop, target stop and trip sequence. In the following, the source and target stop will always be specified when describing a journey to avoid ambiguities.

**Journey Properties.** To determine which journeys are desirable to the user, we consider a number of properties. Given an $s$-$t$-journey $J = ((\mathrm{tr}_1, u_1, v_1), \ldots, (\mathrm{tr}_k, u_k, v_k))$, we define the following properties of $J$:

- The *departure time* $\tau_{\mathrm{dep}}(J)$ of $J$ is the departure time of the first trip at its entering stop, minus the time it takes to walk there from the source stop: $\tau_{\mathrm{dep}}(J) = \tau_{\mathrm{dep}}(\mathrm{tr}_1, u_1) - \tau_w(s, u_1)$. If $s = u_1$, no walking is required and the departure time is simply that of the first trip.
- The *arrival time* $\tau_{\mathrm{arr}}(J)$ of $J$ is the arrival time of its last trip at its exiting stop, plus the time it takes to walk to the target stop from there: $\tau_{\mathrm{arr}}(J) = \tau_{\mathrm{arr}}(\mathrm{tr}_k, v_k) + \tau_w(v_k, t)$. If $v_k = t$, no walking is required and the arrival time is simply that of the last trip.

- The *travel time* $\tau_t(J)$ of $J$ is the difference between its arrival time and its departure time: $\tau_t(J) = \tau_{\text{dep}}(J) - \tau_{\text{arr}}(J)$.
- The *number of trips* in $J$, denoted by $|J|$, is the number of trip segments $k$. An alternative is to measure the *number of transfers*, which is $k-1$.

In public transit routing, we are typically interested in routes that depart no earlier than a given minimum departure time $\tau_{\text{dep}}$. We call $J$ *feasible* for $\tau_{\text{dep}}$ if $\tau_{\text{dep}}(J) \geq \tau_{\text{dep}}$. This means it is possible to enter $\text{tr}_1$ at $u_1$ when departing from $s$ at $\tau_{\text{dep}}$.

**Subjourneys.** Several of the algorithms discussed in this thesis work by dividing journeys into smaller *subjourneys*. Defining subjourneys in our public transit network model is not as straightforward as defining subpaths in a graph. Since paths are defined as sequences of edges, a subpath of a path $P = (e_1, \dots, e_k)$ can be defined as a subsequence $P_{ij} = (e_i, \dots, e_j)$ for some choice of $1 \leq i \leq j \leq k$. Since journeys are defined as sequences of trip segments, it seems natural to define subjourneys as subsequences of trip segments. However, this does not capture subjourneys that begin or end midway through a trip segment or a walking leg in the original journey. To solve this problem, we make a distinction between *proper* and *improper* subjourneys.

**Definition 2.1.** *Let $J = ((tr_1, u_1, v_1), \dots, (tr_k, u_k, v_k))$ be an $s$-$t$-journey. For any $1 \leq i \leq j \leq k$, the $u_i$-$v_j$-journey that consists of the subsequence $J_{ij} = ((tr_i, u_i, v_i), \dots, (tr_j, u_j, v_j))$ is called a proper subjourney of $J$.*

This subjourney definition is analogous to the subpath definition in graphs. A proper subjourney always begins with entering a trip and ends with exiting a trip. As a consequence, a proper subjourney always contains at least one trip segment. In order to define improper subjourneys, we first introduce the notion of *trip subsegments*:

**Definition 2.2.** *Let $(tr, u, v)$ be a trip segment. The trip segment $(tr, u', v')$ is called a trip subsegment of $(tr, u, v)$ if $u'$ comes no earlier than $u$ in $tr$ and $v'$ comes no later than $v$.*

To avoid edge cases in the following definition, we define $v_0 := s$ and $u_{k+1} := t$ for an $s$-$t$-journey of length $k$. Improper subjourneys can then be defined as follows:

**Definition 2.3.** *Let $J = ((tr_1, u_1, v_1), \dots, (tr_k, u_k, v_k))$ be an $s$-$t$-journey. For any $1 \leq i < j \leq k$, the $s'$-$t'$-journey that consists of the sequence $J'_{ij} = ((tr_i, u'_i, v_i), (tr_{i+1}, u_{i+1}, v_{i+1}), \dots, (tr_{j-1}, u_{j-1}, v_{j-1})(tr_j, u_j, v'_j))$ is called an improper subjourney of $J$ if both of the following conditions are fulfilled:*

- *$u'_i \in \{s', u_i\}$. If $u'_i = s'$, $u'_i$ must be a stop in $tr_i$ that comes between $u_i$ and $v_i$. If $u'_i = u_i$, then $s' \in P_{v_{i-1}u_i}$ must hold.*
- *$v'_j \in \{t', v_j\}$. If $v'_j = t'$, $v'_j$ must be a stop in $tr_j$ that comes between $u_j$ and $v_j$. If $v'_j = v_j$, then $t' \in P_{v_j u_{j+1}}$ must hold.*

*Additionally, the $s'$-$t'$-journey $J = ()$ is called an improper subjourney if $s' \in P_{v_i u_{i+1}}$ and $t' \in P_{s' u_{i+1}}$ for some $0 \leq i \leq k$.*

Journeys can also be interpreted as sequences of legs, containing both trip segments and walking legs. In this view, an improper subjourney can be seen as a subsequence of legs except for possibly the first and last leg, which may be a subpath or trip subsegment of the corresponding leg in the original journey. This allows for subjourneys that begin or end midway through a trip segment or footpath.

The reason for making the distinction between proper and improper subjourneys will become apparent in Chapter 3, where we will show that there always exists an optimal journey whose proper subjourneys are all optimal themselves. The same is not true of improper subjourneys.

## 2.3 Problem Definitions

This section describes the routing problems considered in this thesis. Factors that influence a public transit routing problem include the choice of source and target stop, the considered departure time(s), and the definition of which journeys are considered optimal. Initially, we will only consider *one-to-one* queries, which ask for optimal journeys from a single source stop $s \in \mathcal{S}$ to a single target stop $t \in \mathcal{S}$. Regarding the departure time, we initially assume a fixed minimum departure time $\tau_{\text{dep}}$ and only ask for journeys that are feasible for that departure time, i.e., journeys that depart no earlier than $\tau_{\text{dep}}$. Later on in this section, we will discuss different choices for the source/target stop and the departure time.

**Optimality.** In routing scenarios with graph-based networks and fixed edge costs, such as time-independent road networks, the simplest type of routing problem asks for the shortest path from a source vertex to a target vertex. The equivalent of this for public transit networks is asking for a journey that minimizes the arrival time at the target stop:

**Definition 2.4.** Earliest Arrival Problem. *We are given a source stop $s$, a target stop $t$ and a departure time $\tau_{dep}$. Among all $s$-$t$-journeys $J$ with departure time $\tau_{dep}(J) \geq \tau_{dep}$, we ask for a journey that minimizes the arrival time $\tau_{dep}(J)$.*

Since we minimize only one criterion, there is only one optimal arrival time, although there may be several journeys with that arrival time. If that is the case, we can arbitrarily pick one optimal journey as the solution. Alternatively, we can pick a journey with a minimal number of trips among all optimal journeys. This prevents unnecessary transfers, which are bothersome to passengers.

However, simply minimizing the arrival time does not always lead to journeys that would be considered the most desirable journey by a typical passenger. Consider a situation where there are two possible journeys, one of which arrives slightly earlier but uses significantly more trips than the other journey. Many passengers would be willing to accept the slightly later arrival time of the other journey if it means having to transfer between trips less often, while other passengers would prefer the journey with the earlier arrival time. This implies that rather than computing just a single optimal journey, it makes sense to compute multiple journeys with different tradeoffs between the two relevant criteria, arrival time and number of trips. To represent these solutions, we use *Pareto sets*, which are commonly used for multi-criteria optimization problems, including multi-criteria shortest path problems [Mar84].

In a multi-criteria optimization problem, we have a search space $\mathcal{S}$ containing all possible solutions, as well as a number of criteria by which we evaluate the solutions. The criteria are represented by objective functions $f_1, \ldots, f_n : \mathcal{S} \to \mathbb{R}$, which we want to minimize. We say that a solution $s_1 \in \mathcal{S}$ *dominates* another solution $s_2 \in \mathcal{S}$ if $s_1$ is equal to or better than $s_2$ according to all objective functions, i.e., if $f_i(s_1) \leq f_i(s_2)$ for all $1 \leq i \leq n$. A solution $s \in \mathcal{S}$ is called *Pareto-optimal* if every solution that dominates $s$ is equivalent to $s$ according to all objective functions. A *Pareto set* is a minimal set $\mathcal{P} \subseteq \mathcal{S}$ of Pareto-optimal solutions such that every solution in $\mathcal{S}$ is dominated by at least one solution in $\mathcal{P}$.[1]

In our context, the search space is the set of all feasible journeys for $\tau_{\text{dep}}$, while the objective functions are arrival time and number of trips, both of which we want to minimize. Additional criteria can be included, such as fare zones [DPW15], but in this thesis we will

[1]In traditional definitions of dominance, $s_1$ only dominates $s_2$ if $s_1$ is strictly better than $s_2$ according to at least one objective function. Under this definition, a solution is Pareto-optimal if it is not dominated by any other solution. A Pareto set can then be defined simply as the set of all Pareto-optimal solutions. If there are multiple equivalent Pareto-optimal solutions, they will all be included in the Pareto set. This is avoided in our definition, where a Pareto set only includes one of these solutions.

limit ourselves to arrival time and number of trips. This leads to the following definition of *bicriteria dominance*:

**Definition 2.5.** *A journey $J_1$ bicriteria-dominates a journey $J_2$ if $\tau_{arr}(J_1) \leq \tau_{arr}(J_2)$ and $|J_1| \leq |J_2|$.*

From this, the definition of *bicriteria optimality* follows naturally. Unless otherwise noted, the definitions of dominance and optimality used throughout this thesis are bicriteria dominance and bicriteria optimality, respectively. Using this definition of optimality, we can define the following routing problem:

**Definition 2.6.** BICRITERIA PROBLEM. *We are given a source stop $s$, a target stop $t$ and a departure time $\tau_{dep}$. Among all journeys $J$ with departure time $\tau_{dep}(J) \geq \tau_{dep}$, we ask for a Pareto set of bicriteria-optimal journeys.*

**Range Problem.** The routing problems introduced so far only ask for optimal solutions for a single, fixed minimum departure time. In practice, users often do not have a fixed departure time in mind, but rather a range of possible departure times. We model this range as a time interval and define the following routing problem:

**Definition 2.7.** RANGE PROBLEM. *We are given a source stop $s$, a target stop $t$ and a departure interval $I = [\tau_{min}, \tau_{max}]$. For each departure time $\tau_{dep} \in I$, we ask for a Pareto set of optimal journeys from $s$ to $t$ among all journeys $J$ with departure time $\tau_{dep}(J) \geq \tau_{dep}$.*

The Range Problem can be combined with any definition of optimality, including the earliest arrival optimality and bicriteria optimality discussed above.

Essentially, the Range Problem consists of a series of fixed departure time problems, one for each time in $I$. In practice, the resulting Pareto sets for departure times that are close to each other will overlap significantly and in many cases will be identical. As a consequence, even though $I$ is continuous and therefore contains infinitely many departure times, the solution to the Range Problem is finite in practice. It is possible to interpret the Range Problem as a multi-criteria problem where the departure time of a journey is an additional criterion that must be maximized. This yields a single Pareto set that contains all optimal solutions for each individual departure time.

The Range Problem is also important for many preprocessing techniques that involve precomputing all optimal journeys between certain source and target vertices for all possible departure times. In our scenario, where the network models public transit connections spanning a single day, this can be achieved by solving the Range Problem for the departure interval $[0h, 24h]$.

**Source and Target Stops.** In addition to the one-to-one problems discussed so far, where a single source and target stop are supplied as part of the query, we also consider routing problems for multiple source or target stops:

- **One-to-Many**: Asks for optimal journeys from a single source stop $s \in \mathcal{S}$ to each in a set $T \subseteq \mathcal{S}$ of target stops.
- **Many-to-One**: Asks for optimal journeys from each in a set $S \subseteq \mathcal{S}$ of source stops to a single target stop $t \in \mathcal{S}$.
- **One-to-All**: Asks for optimal journeys from a single source stop $s \in \mathcal{S}$ to each target stop $t \in \mathcal{S}$.
- **All-to-One**: Asks for optimal journeys from each source stop $s \in \mathcal{S}$ to a single target stop $t \in \mathcal{S}$.

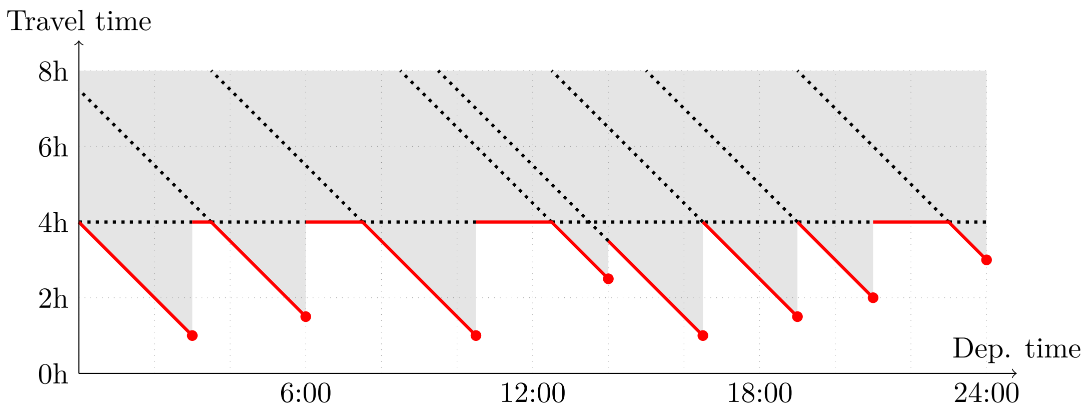


Figure 2.2: An example of a profile function for the departure time interval $[0h, 24h]$, drawn in red. The segments with slope $-1$ represent journeys containing at least one trip. The segments with slope 0 represent the optimal pure walking journey. Dotted lines represent parts of journeys that are dominated by another optimal journey. The entire area dominated by the profile function is highlighted in gray.

- **Many-to-Many**: Given sets $S, T \subseteq \mathcal{S}$ of source and target stops, asks for optimal journeys between each pair of source stop $s \in S$ and target stop $t \in T$.
- **All-to-All**: Asks for optimal journeys between each pair of source and target stops $s, T \in \mathcal{S}$.

These problems are mainly relevant for the preprocessing techniques discussed later on.

**Profiles.** As discussed above, a solution to the Range Problem can be represented as a Pareto set using departure time as an additional criterion. We call the Pareto set for a given source and target stop $s, t \in \mathcal{S}$ the *s-t-profile*. A more natural way to represent profiles is by using functions that map departure time to travel time. Consider first the simplest version of the Range Problem using the arrival time as the sole criterion for optimality. In this case, for each departure time $\tau_{\text{dep}} \in I$, the profile contains exactly one journey that is optimal for $\tau_{\text{dep}}$. We therefore associate the profile with a function $f : I \to \mathbb{R}$ that maps departure time to travel time. For each departure time $\tau_{\text{dep}} \in I$, the optimal travel time is the difference between the arrival time of the optimal journey $J$ and $\tau_{\text{dep}}$ itself, i.e., $f(\tau_{\text{dep}}) = \tau_{\text{arr}}(J) - \tau_{\text{dep}}$.

Assume momentarily that all optimal journeys contain at least one trip. Then a journey $J$ is feasible for all points in the interval $[\tau_{\text{min}}, \tau_{\text{dep}}(J)]$. We can represent the travel time of $J$ with a function $f_J$:

$$f_J(\tau_{\text{dep}}) = \begin{cases} \tau_{\text{arr}}(J) - \tau_{\text{dep}} & \text{if } \tau_{\text{dep}} \in [\tau_{\text{min}}, \tau_{\text{dep}}(J)] \\ \infty & \text{otherwise.} \end{cases}$$

Since the arrival time is fixed and does not depend on the specific departure time, the travel time function is linear with slope $-1$ in $[\tau_{\text{min}}, \tau_{\text{dep}}(J)]$. It reaches its minimum value $\tau_t(J)$ for $\tau_{\text{dep}}(J)$. For any earlier departure time, the travel time increases by the amount of time it takes to wait for the first trip to depart.

The profile function $f$ can be calculated as the piecewise minimum over the travel time functions $f_J$ for each journey $J$ in the profile. Since the travel time functions for the individual journeys are all linear with the same slope, they cannot intersect. This means that each journey $J$ is optimal in the interval $(\tau_{\text{dep}}(J'), \tau_{\text{dep}}(J)]$, where $J'$ is the journey with the latest departure time $\tau_{\text{dep}}(J') \leq \tau_{\text{dep}}(J)$. This allows us to represent $f$ as a piecewise linear function where all the segments have slope $-1$ except for a possible segment of constant value $\infty$ at the end.

If we consider pure walking journeys as well, things get slightly more complicated. A pure walking journey contains no trips and consists entirely of a path in the footpath graph, which is time-independent. This means the travel time is fixed and not dependent on the specific departure time. Furthermore, pure walking journeys are feasible for any departure time. Therefore, pure walking journeys contribute to the profile function as segments with slope 0. An example of a profile function can be seen in Figure 2.2.

A piecewise linear function $f$ can be represented as a sequence of *breakpoints*, where each breakpoint represents one linear segment of $f$. A breakpoint is a triple $(x, y, s)$ where $x$ denotes the endpoint of the segment, $y$ is the value of $f$ at $x$, and $s$ is the slope of $f$ in the segment. Given a sequence of breakpoints $((x_1, y_1, s_1), \ldots, (x_k, y_k, s_k))$, the corresponding piecewise linear function $f$ is defined as:

$$f(x) := \begin{cases} x_1 - s_1(x_1 - x) & \text{if } x \leq x_1 \\ x_2 - s_2(x_2 - x) & \text{if } x_1 < x \leq x_2 \\ \vdots & \\ x_k - s_k(x_k - x) & \text{if } x_{k-1} < x \leq x_k \\ \infty & \text{if } x > x_k \end{cases}$$

In our case, each optimal journey $J$ with at least one trip contributes the breakpoint $(\tau_{\text{dep}}(J), \tau_t(J), -1)$. Trip-based journeys can intersect with pure walking journeys, but the profile can include at most one optimal pure walking journey $J_w$ (since all other pure walking journeys are dominated by it). Whenever $J_w$ intersects with a trip-based journey at a point $x$, another breakpoint $(x, \tau_t(J_w), 0)$ is added. Since $J_w$ can intersect at most once with each trip-based journey, the total number of breakpoints is linear in the number of optimal journeys.

Since there can be only one pure walking journey, it is more space-efficient to store it separately from the rest of the profile. We therefore represent the profile as a function $f$ that ignores pure walking journeys and the optimal walking time $\tau_w$. This allows us to drop the slope value from the representation of the breakpoints, since all intervals in the function have slope $-1$. Thus, the breakpoint representing a journey $J$ now has the form $(\tau_{\text{dep}}(J), \tau_t(J))$. The value of the profile at $x$ can be calculated as $\min(f(x), \tau_w)$.

Now we consider the more complicated case of the Range Problem with bicriteria optimality, where the number of trips is also relevant as an optimization criterion. In this case, there might be more than one optimal journey for a given departure time. However, for a given departure time $\tau_{\text{dep}}$ and a number of trips $k$, the profile contains only one optimal journey that departs no earlier than $\tau_{\text{dep}}$ and uses at most $k$ trips. Thus, we can represent the profile for a bicriteria problem with several piecewise linear functions, one for each number of trips. If $K$ is the maximum number of trips used by any journey in the profile, then we can represent the profile with exactly $K + 1$ piecewise linear functions $f_i : I \to \mathbb{R}$ $(0 \leq i \leq K)$. In practice, since we store $\tau_w$ as a separate value, we do not need to store $f_0 \equiv \tau_w$ explicitly, so $K$ functions suffice.

An alternative representation of profiles is to map departure time to arrival time instead of travel time. In this case, the trip-based journeys are represented by intervals with slope 0 and the pure walking journeys is represented by intervals with slope 1.

## 2.4 RAPTOR

In this section we describe the RAPTOR algorithm, with its original restriction that the footpath graph must be transitively closed. The formulation of RAPTOR presented here differs slightly from the original formulation in [DPW15], mostly in the way the minimum change times are handled. RAPTOR solves both the Earliest Arrival Problem and the Bicriteria Problem for a fixed departure time. In its basic form, it can solve one-to-all problems, i.e., for a given source stop $s \in \mathcal{S}$, it computes optimal journeys to all other stops. However, it can also efficiently solve one-to-one problems via *target pruning*.

A key observation is that the Bicriteria Problem can also be interpreted as a series of Earliest Arrival Problems: For each number of trips $k$, we ask for a journey with minimal arrival time among all feasible journeys with at most $k$ trips. Described in this manner, it becomes clear that the resulting Pareto set contains at most one optimal solution for each number of trips. The RAPTOR algorithm exploits this observation with a dynamic programming approach that operates in rounds. In round $k$, it computes optimal solutions with at most $k$ trips. This is done by taking the solutions from the previous round and extending them by one trip.

Each round consists of two phases: a route phase in which routes are scanned and a transfer phase in which footpaths are relaxed. In the route phase, all routes that visit a stop whose solution was improved in the previous phase are collected. Each of these routes is then scanned, starting at the first stop whose solution was improved. At each stop visited by the route, we identify the earliest trip that can be entered and follow it to the next stop to check if it improves the solution from the previous round. In the transfer phase, all stops whose solution was improved in the previous phase are collected and all their outgoing edges in the footpath graph are relaxed. Since we require that the footpath graph is transitively closed, relaxing single edges is sufficient for finding optimal paths between stops. Additionally, at each stop whose solution was improved, we add the minimum change time to the solution since it is required to enter trips in the next round.

The main data structure of the algorithm is the *round table* $R$. For each round $k$ executed so far and each stop $v$, $R[2k][v]$ contains the earliest arrival time among all journeys that use exactly $k$ trips and end with a trip, while $R[2k+1][v]$ contains the earliest arrival time among all journeys that use exactly $k$ trips and end with a transfer. Additionally, we maintain labels for the earliest arrival time regardless of the number of trips: $\tau_r(v)$ is the earliest arrival time among all journeys that end with a trip, while $\tau_t(v)$ is the earliest arrival time among all journeys that end with a transfer. The arrivals via transfers and via trips are stored separately because they cannot be compared directly: An arrival via a transfer already obeys the minimum change time and can thus be immediately extended by another trip, while an arrival via a trip cannot. Therefore, it might be necessary to store an arrival via a transfer even if there is an earlier arrival via a trip, since the arrival via transfer might allow passengers to enter trips that the arrival via trip does not. Finally, we maintain sets $U_r$ and $U_t$, which collect all stops that were updated in the last route or transfer phase, respectively.

Pseudocode for the RAPTOR algorithm is given in Algorithm 2.1. Besides the public transit network, the algorithm receives a source stop $s \in \mathcal{S}$, a target stop $t \in \mathcal{S}$ and a departure time $\tau_{\text{dep}}$ as input. The algorithm starts with an initialization phase in which the round table entries for the first round are initialized with $\infty$ for all stops except $s$, which is initialized with $\tau_{\text{dep}}$. The same is done for the earliest arrival time labels $\tau_r$ and $\tau_t$. The source stop $s$ is then marked as updated and the first transfer phase is started, the results of which are stored in $R[1]$.

From this point onward, the algorithm iterates through a series of rounds, alternating between route and transfer phases. At the start of each round $k$, the round table entries

**Algorithm 2.1:** RAPTOR

```
Input: Network $(\mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, source and target stop $s, t \in \mathcal{S}$, departure time $\tau_{\text{dep}}$
Data: Round table $R[\cdot][\cdot]$, earliest arrival times by route $\tau_r(\cdot)$ and by transfer
      $\tau_t(\cdot)$, stops updated by route $U_r$ and by transfer $U_t$, routes to scan $C$
1  function RAPTOR:
       // Initialization
2      forall $v \in \mathcal{S}$ do
3          $R[0][v] \leftarrow \infty$
4          $R[1][v] \leftarrow \infty$
5          $\tau_r(v) \leftarrow \infty$
6          $\tau_t(v) \leftarrow \infty$
7      $R[0][s] \leftarrow \tau_{\text{dep}}$
8      $\tau_r(s) \leftarrow \tau_{\text{dep}}$
9      $U_r$.INSERT$(s)$
10     RELAXTRANSFERS$(1,$ false$)$
       // Main loop
11     forall $k \leftarrow 1, 2, \dots$ do
12         forall $v \in \mathcal{S}$ do
13             $R[2k][v] \leftarrow \infty$
14             $R[2k+1][v] \leftarrow \infty$
           // Collect routes serving updated stops
15         $C$.CLEAR()
16         forall $v \in U_t$ do
17             forall routes $r$ serving $v$ do
18                 if $C$.CONTAINSKEY$(r)$ then
19                     $C(r) \leftarrow \min(v, C(r))$
20                 else
21                     $C$.INSERT$((r, v))$
22         SCANROUTES$(2k)$
23         if $U_r$.ISEMPTY() then return
24         RELAXTRANSFERS$(2k+1,$ true$)$
```

in $R[2k]$ and $R[2k+1]$ are initialized with $\infty$. Then all routes that serve stops updated in the previous transfer phase (which are collected in $U_t$) are collected in $C$, which stores key-value-pairs $(r, v)$ consisting of a route $r$ and the first stop $v$ in $r$ that was updated. Afterwards, the route phase is performed, which scans the routes in $C$. Note that it is not necessary to scan routes that are not contained in $C$ – any such route does not visit any stop whose arrival time was improved in the previous round, so taking a trip belonging to the route cannot lead to arrival times that are better than those found in the previous rounds.

Pseudocode for the route phase is given in Algorithm 2.2. For each pair $(r, u) \in C$, the route $r$ is scanned beginning at the stop $u$. We want to scan the earliest departing trip that can be entered at $u$. The arrival time at $u$ is given by the round table entry from the previous phase, $R[2k-1][u]$. We then call the function ET$(r, u)$ to compute the earliest trip $\text{tr} \in r$ for which $\tau_{\text{dep}}(\text{tr}, u) \geq R[2k-1][u]$ holds. We then iterate over the stops in $r$, beginning at $u$. Whenever we arrive at a stop $v$, we call the function ARRIVALBYROUTE

**Algorithm 2.2:** SCANROUTES

```
function SCANROUTES($k$):
    $U_r$.CLEAR()
    forall $(r, u) \in C$ do
        tr $\leftarrow \perp$
        forall stops $v \in r$ beginning with $v$ do
            if $tr \neq \perp$ then
                ARRIVALBYROUTE($k, v, \tau_{arr}(tr, v)$)
            if $R[k-1][v] \leq \tau_{dep}(tr, v)$ then
                tr $\leftarrow$ ET($r, v$)

function ARRIVALBYROUTE($k, v, \tau_{arr}$):
    if $\tau_{arr} < \min(\tau_r(v), \tau_r(t))$ then
        $R[k][v] \leftarrow \tau_{\text{arr}}$
        $\tau_r(v) \leftarrow \tau_{\text{arr}}$
        $U_r$.INSERT($v$)
```

to determine if the arrival time at $v$ can be improved. This is the case if the arrival time $\tau_{\text{arr}}(\text{tr}, v)$ via tr is better than the earliest arrival time $\tau_r(v)$ found so far. If so, we update $R[2k][v]$ and $\tau_r(v)$ and mark $v$ as reached by inserting it into $U_r$. Before we continue to the next stop, we check if we can switch to an earlier trip at $v$. This may occur if the arrival time from the previous phase, $R[2k-1][v]$, is better than the departure time of tr, $\tau_{\text{dep}}(\text{tr}, v)$. In this case, we call ET$(r, v)$ again to find the earliest trip that can be entered.

After the route phase has concluded, $U_r$ contains all stops that were updated by it. We then perform the transfer phase, which relaxes all outgoing transfers from these stops. Pseudocode for the transfer phase is given in Algorithm 2.3. For each stop $u \in U_r$, we relax all outgoing edges in the footpath graph $\mathcal{F}$. For each edge $e = (u, v)$, we call the function ARRIVALBYTRANSFER to determine if taking this edge improves the arrival time at $v$. The arrival time via the edge is given by the arrival time $R[2k][u]$ at $u$ from the previous round, plus the walking time $\tau_w(e)$. If this improves on the earliest arrival time $\tau_t(v)$ found so far, we update $R[2k+1][v]$ and $\tau_t(v)$ and mark $v$ as reached by inserting it into $U_t$. Afterwards, we call ARRIVALBYTRANSFER again to handle the minimum change time at $u$, now using $R[2k][u] + \tau_{\text{ch}}(u)$ as the arrival time. This ensures that departures via a trip scanned in the next round observe the minimum change time. An exception to this is made during the very first transfer phase in round 0, which relaxes outgoing transfer from $s$. Here we do not add the minimum change time to the arrival time, since we did not arrive at $s$ via a trip.

Since the main loop of the algorithm continues adding new rounds indefinitely, we need a termination criterion to determine when the algorithm can be stopped. We do this by checking after the route phase in each round $k$ if $U_r$ is empty. If this is the case, the route phase was unable to improve any arrival times from the previous round. This means that no journeys with $k$ trips exist that have an earlier arrival time than those with less than $k$ trips, and therefore we can stop the algorithm since no new journeys will be found.

**Target Pruning.** The algorithm as described so far is able to solve one-to-all queries, since it maintains round table entries for each stop and only terminates when no new journeys to any target stop are found. If we are only interested in solving one-to-one queries to a given target stop $t \in \mathcal{S}$, we can improve the running time with a target pruning condition: The functions ARRIVALBYROUTE and ARRIVALBYTRANSFER check if a found arrival time $\tau_{\text{arr}}$ at a stop $v$ improves on previously found arrival times by comparing it to

**Algorithm 2.3:** relaxTransfers

```
function relaxTransfers($k$, $addMinChangeTime$):
    $U_t$.clear()
    forall $u \in U_r$ do
        forall edges $e = (u, v) \in \mathcal{F}$ do
            arrivalByTransfer($k, v, R[k-1][u] + \tau_w(e)$)
        if $addMinChangeTime$ then
            arrivalByTransfer($k, u, R[k-1][u] + \tau_{ch}(u)$)
        else
            arrivalByTransfer($k, u, R[k-1][u]$)
function arrivalByTransfer($k, v, \tau_{arr}$):
    if $\tau_{arr} < \min(\tau_t(v), \tau_t(t))$ then
        $R[k][v] \leftarrow \tau_{\text{arr}}$
        $\tau_t(v) \leftarrow \tau_{\text{arr}}$
        $U_t$.insert($v$)
```

the earliest arrival time found so far, which is given by $\tau_r(v)$ or $\tau_t(v)$, respectively. We now additionally compare $\tau_{\text{arr}}$ to the earliest arrival time at the target, i.e., $\tau_r(t)$ or $\tau_t(t)$. If $\tau_{\text{arr}}$ does not improve on the best arrival time at $t$ found so far, it cannot be extended to an optimal journey to $t$, so we do not update the round table entry.

**Running Time.** We now give an upper bound for the worst-case running time of the RAPTOR algorithm. The data structures $U_r$, $U_t$ and $C$ can be implemented as indexed sets/maps, which allows insertion and element access operations to be implemented in constant time. Accordingly, the functions arrivalByRoute and arrivalByTransfer, which are used by the route and transfer phase, respectively, take constant time. The running time of the initialization phase, excluding the first transfer phase, is linear in the number of stops $|\mathcal{S}|$.

When collecting the routes that serve updated stops, each route is considered at most once for every stop served by the route. The time needed for collecting the routes in each round is therefore linear in the sum of all route sizes, i.e., $\sum_{r \in \mathcal{R}} |r|$. During the route phase of each round, each route is scanned at most once, and during each route scan, each stop of the route is scanned at most once as well. The arrival time of the currently scanned trip can only decrease over the course of a route scan, so et can be implemented by iterating over the trips in descending order of departure time. This allows us to implement the route phase in time $\mathcal{O}(|\mathcal{T}| + \sum_{r \in \mathcal{R}} |r|)$.

The transfer phase of each round scans each stop at most once. For each scanned stop, each outgoing edges is relaxed at most once. Handling the minimum change time takes constant time, so overall the transfer phase can be implemented in time $\mathcal{O}(|\mathcal{S}| + |E|)$.

Assuming that each stop is visited by at least one route, we observe that $|\mathcal{S}| \leq \sum_{r \in \mathcal{R}} |r|$. Applying this observation yields a running time of $\mathcal{O}(|\mathcal{T}| + |E| + \sum_{r \in \mathcal{R}} |r|)$ for each round. If $K$ is the total number of rounds performed by the algorithm, we obtain a total running time of $\mathcal{O}(K(|\mathcal{T}| + |E| + \sum_{r \in \mathcal{R}} |r|))$. Note that because RAPTOR does not use a priority queue, the running time lacks the logarithmic factor seen in Dijkstra's algorithm.

**Retrieving Optimal Arrivals and Journeys.** When solving the Earliest Arrival Problem for a target stop $t \in \mathcal{S}$, the earliest arrival time at $t$ is given by $\min_k(R[k][t])$ or, alternatively, $\min(\tau_r(t), \tau_t(t))$. In the case of the Bicriteria Problem, we are interested, for each number of trips $k$, in the earliest arrival time among all journeys with at most $k$ trips. For each round $k$, $\min(R[2k][t], R[2k+1][t])$ is the earliest arrival time among all journeys with exactly $k$ trips. We can therefore collect a Pareto set of arrival times by iterating over the number of trips $k$ and adding the earliest arrival time for $k$ trips to the Pareto set if it is not dominated by the arrival times already found for less than $k$ trips.

In order to retrieve full descriptions of the optimal journeys, we need to augment the round table entries with parent pointers. If a round table entry $R[k][u]$ has the parent pointer $v$, that means that the arrival at $u$ in phase $k$ was extended from the arrival at $v$ in phase $k-1$, which is given by $R[k-1][v]$. If phase $k$ was a route phase, $u$ was therefore reached via a trip leg from $v$. If phase $k$ was a transfer phase, $u$ was either reached via a walking leg from $v$ or, if $u = v$, by exiting a trip and observing the minimum change time. We can thus recursively unpack journeys by following the parent pointers across the round table and adding the corresponding legs to the journey until we reach $s$ in round 0.

**Backward Search.** RAPTOR can be easily adapted to perform backward searches from a target stop $t \in \mathcal{S}$ to a source stop $s \in \mathcal{S}$ or all stops in the graph for a given arrival time $\tau_{\text{arr}}$. The outline of the algorithm remains the same, but a few changes are necessary: Instead of maintaining earliest arrival times, we now maintain latest departure times. Routes are scanned in reverse order. During a route scan, when scanning a stop $v$, the forward search is given an arrival time $\tau_{\text{arr}}(v)$ at $v$ and searches for the earliest trip ET that departs after $\tau_{\text{arr}}(v)$. When performing a backward search, we are instead given a departure time $\tau_{\text{dep}}(v)$ and must search for the latest trip LT that arrives at $v$ before $\tau_{\text{dep}}(v)$. In the transfer phase, walking times and minimum change times are subtracted from the current departure time instead of being added to the current arrival time.

## 2.5 RAPTOR with Unrestricted Walking

In its original formulation, RAPTOR requires that the footpath graph is transitively closed. This severely limits the number of footpaths that can be represented. In a transitively closed graph, each connected component is a clique. However, unrestricted footpath graphs are usually fully (or almost fully) connected. Thus, a transitive closure would produce a complete graph with $\mathcal{O}(|V|^2)$ edges. On country-sized networks, such a graph would be much too large to store in memory. We therefore need public transit algorithms that can handle footpath graphs that are not transitively closed.

In a transitively closed graph, there is always a shortest path consisting of a single edge between any pair of connected vertices. RAPTOR exploits this fact in its transfer phase by only relaxing the outgoing edges of each updated stop instead of searching for shortest paths consisting of multiple edges. If we drop the requirement that the graph is transitively closed, we can no longer do this and instead must perform a full multi-source Dijkstra search, using $U_r$ as the set of source vertices. This approach was first proposed by Delling et al. for their MCR (multimodal multicriteria RAPTOR) algorithm [DDP$^+$13], which solves multicriteria queries in multimodal networks. In the multimodal scenario, the footpath graph is simply one of several possible unrestricted networks that can be explored between the route phases.

For a round $k$, the modified transfer phase now performs the following stops: Each stop $v \in U_r$ is added to the priority queue of Dijkstra's algorithm with the arrival time $R[2k][v]$ as its key. Afterwards, Dijkstra's algorithm is run. Whenever a stop $v$ is reached via an edge, ARRIVALBYTRANSFER is called with the arrival time obtained from the Dijkstra search. This updates $R[2k+1][v]$ if necessary. As in the original RAPTOR algorithm, arrivalByTransfer is also called once for each stop $v \in U_r$ to handle the minimum change time.

The original RAPTOR algorithm maintains an earliest arrival time $\tau_t(v)$ for each stop $v$ to prune arrivals that do not improve on the arrival times from previous rounds. We extend $\tau_t$ to all vertices to prevent unnecessary arrivals at non-stop vertices from being found. Whenever we relax an edge $(u, v)$ during the Dijkstra search, we compare the arrival time to $\tau_t(v)$ and prune the search at $v$ if it is not an improvement.

### 2.5.1 Contraction

The main drawback of incorporating a Dijkstra search into the transfer phase, and the reason why most public transit algorithms restrict walking, is that the transfer phase now takes up the vast majority of the overall running time. MCR mitigates this somewhat by contracting the footpath graph [DDP+13] in a similar manner to the Contraction Hierachies algorithm [Gei08]. Since we are only interested in computing journeys between pairs of stops, vertices in the footpath graph which are not stops can be removed as long as the correct distances between all stops are preserved. This can be achieved by inserting shortcuts between the neighbors of removed vertices.

The *contraction* operation for a vertex $u \in V \setminus \mathcal{S}$ works as follows: First $u$ and all its adjacent edges are removed. For each pair $v, w$ of neighbors of $u$, we check if the optimal path from $v$ to $w$ consists of the edges $e_1 = (v, u)$ and $e_2 = (u, w)$. If this is the case, we add a new shortcut edge $e_3 = (v, w)$ with the weight $\tau_w(e_3) = \tau_w(e_1) + \tau_w(e_2)$ to the graph. This edge ensures that the correct distance between $v$ and $w$ is preserved. The contraction operation can lead to multi-edges if a shortcut is inserted between two vertices that are already connected by an edge. These multi-edges are removed by only keeping one edge with a minimal weight.

In order to reduce the size of the footpath graph, the vertices in $V \setminus \mathcal{S}$ are contracted iteratively. The order in which the vertices are contracted is important since it impacts the number of inserted shortcuts. Determining a suitable contraction order is a complex topic and will not be explored here; details can be found in [Gei08]. Typically, heuristics are used to establish a contraction order, based on a metric that measures the "importance" of vertices. Factors impacting importance include the vertex degree, the number of edges removed by contracting the vertex and the number of shortcuts created by contracting the vertex. In general, it is advisable to contract vertices with a small degree early on since they produce few shortcuts.

Contracting all vertices in $V \setminus \mathcal{S}$ would result in a transitively closed footpath graph containing only stops. As we have already established, this is not feasible, so we need to stop the contraction before it finishes. There are several possible termination criteria, including setting a limit on the number of vertices remaining in the graph or the average vertex degree.

### 2.5.2 Issues with Minimum Change Time

Handling unrestricted walking via a multi-source Dijkstra search leads to a number of problems related to the minimum change time. These problems will be discussed in this section, alongside possible solutions. The underlying issue beneath all these problems is that the minimum change time is only added when switching between two trips at a stop, and not when switching between a trip and a footpath.

Let $u \in U_r$ be a source stop during a Dijkstra search and let $\tau_{\text{arr}}(u)$ be its arrival time via route in the previous phase. At the beginning of the Dijkstra search, the Dijkstra label of

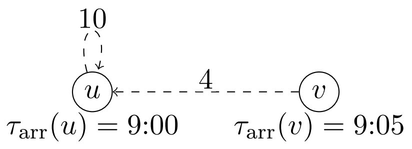

10
u
4
v
$\tau_{\text{arr}}(u) = 9{:}00$
$\tau_{\text{arr}}(v) = 9{:}05$

Figure 2.3: An example of a situation where arriving at $v$ and walking to $u$ is superior to arriving directly at $u$. Due to the minimum change time of 10 minutes, the earliest viable departure time from $u$ when arriving there at 9:00 is 9:10. When arriving at $v$ and walking to $u$, however, the earliest viable departure time is 9:09.

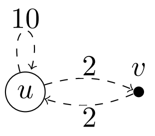


Figure 2.4: An example of a cyclical path that allows passengers to circumvent the minimum change time at a stop. In this case, the minimum change time at $u$ is 10 minutes, but walking to $v$ and back to $u$ only takes 4 minutes.

$u$ is set to $\tau_{\text{arr}}(u)$, but this arrival time is not added to the round table because $u$ was not reached via an edge. This is important because the arrival time at $u$ has to incorporate the minimum change time $\tau_{\text{ch}}(u)$. The minimum change time is handled after the Dijkstra search has finished, but if the arrival time $\tau_{\text{arr}}(u)$ were entered into the round table by the Dijkstra search, the correct arrival time $\tau_{\text{arr}}(u) + \tau_{\text{ch}}(u)$ would be dominated.

Let $v \in U_r$ be another source stop whose arrival time via route is $\tau_{\text{arr}}(v)$. If $\tau_{\text{arr}}(v) + \tau_w(P_{vu}) < \tau_{\text{arr}}(u) + \tau_{\text{ch}}(u)$, arriving via $v$ and walking to $u$ is better than arriving directly at $u$ and waiting for the required minimum change time. This scenario is depicted in Figure 2.3. In this case, our Dijkstra search should find the arrival via $P_{vu}$ and update the round table entry for $u$. However, because the Dijkstra label of $u$ was set to $\tau_{\text{arr}}(u)$ at the start of the search. If $\tau_{\text{arr}}(u) < \tau_{\text{arr}}(v) + \tau_w(P_{vu})$, the arrival via $P_{vu}$ is dominated and not added to the round table, even though $\tau_{\text{arr}}(u)$ itself was not added to the round table. To prevent this, we do not start the Dijkstra search at the stops in $U_r$. Instead, we relax the outgoing edges of the stops in $U_r$ and then start the Dijkstra search at the vertices reached this way. This way, we do not initialize the Dijkstra labels of the stops in $U_r$ with infeasible arrival times that can dominate legitimate arrivals.

However, this introduces another problem: If there is a path $P$ in the footpath graph that starts and ends at a stop $u \in U_r$ and has a walking time $\tau_w(P) < \tau_{\text{ch}}(u)$, taking this path would allows passengers to circumvent the minimum change time at $u$ by walking in a circle. This scenario is depicted in Figure 2.4. This is clearly unrealistic and counter-intuitive and it can be argued that any occurrence of this situation in real public transit network data constitutes a modeling error. Unfortunately, it turns out that these situations actually occur in commonly available public transit network data. Note that this problem does not occur when in restricted walking scenarios where the graph must be transitively closed and loop edges are prohibited.

A straightforward solution to this problem is to prevent the Dijkstra search from finding cyclical paths. This can be achieved by adding parent pointers to the labels of the Dijkstra search that points to the source stop where the respective footpath originated. We can then prohibit relaxing edges $(u, v)$ where $v$ is the parent of the label at $u$. This prevents cyclical paths from being found. However, this has undesirable consequences.

Consider the scenario depicted in Figure 2.5, where two stops $u, v \in \mathcal{S}$ with arrival times $\tau_{\text{arr}}(u)$ and $\tau_{\text{arr}}(v)$ are connected via a footpath $P$ that visits the non-stop vertex $w$. Because

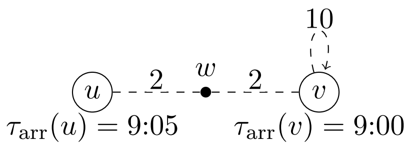

10
$u$
2
$w$
2
$v$
$\tau_{\text{arr}}(u) = 9{:}05$
$\tau_{\text{arr}}(v) = 9{:}00$

Figure 2.5: In this scenario, arriving at $u$ and walking to $v$ via the footpath $P = ((u,w),(w,v))$ is superior to arriving directly at $u$ and waiting from the required minimum change time of 10 minutes. However, the earliest possible arrival at $w$ comes via $v$ at 9:02. This dominates the arrival via $u$ at 9:07, which is needed to find the optimal arrival at $u$ via $P$.

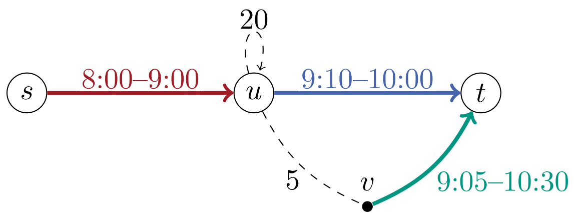


Figure 2.6: An example of an optimal journey with a suboptimal proper subjourney. The optimal $s$-$t$-journey $J$ for the departure time 8:00 involves taking the footpath from $u$ to $v$ and entering a trip to $t$ there. The trip that departs at $u$ at 9:10 arrives earlier at $t$, but it cannot be entered due to the minimum change time at $u$. The proper $v$-$t$-subjourney of $J$ is not optimal, since walking to $u$ and entering the trip to $t$ is superior.

$\tau_{\text{arr}}(u) + \tau_w(P) < \tau_{\text{arr}}(v) + \tau_{\text{ch}}(v)$, our Dijkstra search needs to find the arrival via $P$ at $v$ (or an even better one). However, in this situation $\tau_{\text{arr}}(u) + \tau_w(P_{uw}) \geq \tau_{\text{arr}}(v) + \tau_w(P_{vw})$ also holds, i.e., walking from $v$ to $w$ is better than walking to it from $u$. This prevents the Dijkstra search from finding the arrival via $P$ at $v$, since $P$ is no longer explored after being dominated at $w$. Note that this scenario requires that $\tau_w(P_{vw}) + \tau_w(P_{wv}) < \tau_{\text{ch}}(v)$, i.e., $v$ has a cyclical path visiting $w$ that allows passengers to circumvent the minimum change time. As this example demonstrates, if we disallow such paths, other paths that are dominated by them are no longer found even though they should be. A possible solution to this problem is to precompute shortcuts between all pairs of stops whose distance is smaller than the minimum change times of either stops. These shortcuts then allow the Dijkstra search to bypass intermediate vertices and thereby avoid this problem.

Another problem involves the preprocessing technique presented in Chapter 4. This technique relies on the *subjourney property*, which states that for any optimal journey, an equivalent optimal journey can be constructed out of optimal proper subjourneys, which we will prove in Chapter 3. This property is lost if cyclical footpaths are prohibited. Consider the example in Figure 2.6. At $u$, the trip departing at 9:10 cannot be entered due to the minimum change time of 20 minutes. However, it would be possible to walk to $v$ and back to $u$ in 10 minutes, allowing passengers to enter the trip departing at 9:10 and bypass the minimum change time. If cyclical paths are prohibited, the optimal $s$-$t$-journey instead involves walking to $v$ and entering a trip there that arrives at 10:30, later than the other trip which cannot be entered. However, if we consider the $v$-$t$-subjourney of that journey, we find that it is not optimal because the optimal journey would involve walking back to $u$ as described before. If we only consider the $v$-$t$-journey by itself, this is not prohibited, but once we consider it as a subjourney in the larger $s$-$t$-journey, it involves a prohibited cyclical path. Thus, the only feasible $v$-$t$-subjourney is not optimal.

Since we want to retain the subjourney property, we cannot solve the issue of cyclical paths that bypass the minimum change time by prohibiting them. If we take the standpoint that such paths constitute a modeling error, the natural solution is to adjust either the minimum change times or the walking times of the footpaths so that the problem can no longer occur. Lowering the minimum change times so that they are not longer than any circumventing cyclical path could lead to unrealistically low minimum change times. Instead, we take the opposite approach: For any edge $e$ adjacent to a stop $v$, if $\tau_w(e) \leq \frac{\tau_{\text{ch}}(v)}{2}$, we set $\tau_w(e)$ to $\frac{\tau_{\text{ch}}(v)}{2} + \epsilon$, where $\epsilon$ is a very small unit of time. As a consequence, every cyclical path that starts and end in $v$ now has a walking time that is greater than $\tau_{\text{ch}}(v)$.

All the problems presented in this section are caused by the modeling decision that the minimum change time only applies when switching between trips and not when switching between a trip and a footpath. Although this model is commonly used, it is arguably not realistic. Consider a stop that represents a multi-platform train station. The justification behind including a minimum change time in the model is that passengers exiting a train at one platform need some time to walk to the platform where the next train departs. However, no walking time is considered when exiting a train and continuing via a footpath, even though in reality passengers would still need to walk from the platform to the station entrance, where the footpath starts.

A more suitable model is to intepret the minimum change time as a *minimum entering or exiting time*, which is added every time a trip is entered or exited, regardless of whether the previous or next leg of the journey is a footpath or another trip. This also means that the minimum change time is applied twice when switching between trips, once for exiting the first trip and then again for entering the second trip. Under this view, the minimum change time represents the time it takes for passengers to walk between the station entrance and a platform. Doubling that time gives an upper bound on the time it takes to walk between any two platforms in the station, which is why it makes sense to apply the minimum change time twice when changing between trips. This approach is similar to how minimum change times are handled in graph-based models: Here the platforms are modeled as individual vertices that are connected to a central station vertex via edges whose weights represent the minimum change time. In this model, switching between trips requires a detour to the station vertex during which the minimum change time is added [PSWZ08]. If the edges between the station vertex and the platform vertices are given weights that reflect the actual walking time between them, this model becomes equivalent to explicitly applying the minimum entering and exiting times.

The RAPTOR algorithm has to be adapted slightly to handle minimum entering and exiting times. Instead of applying the minimum change time during the transfer phase, independently of the footpaths, it must be applied during the route phase: When entering a trip, the minimum entering time is added to the arrival time retrieved from the round table before the trip is scanned. When reaching a stop via a trip, the minimum exiting time is added to the arrival time before writing it to the round table. This ensures that the minimum exiting time is observed even when continuing the journey with a footpath in the next phase.

Using this model solves all the problems presented in this section: Since the minimum change time is added before the transfer phase starts, the labels at the source stops of the Dijkstra search incorporate it as well and therefore do not have to be handled in a special manner. Cyclical footpaths that circumvent the minimum change time are now impossible because it is added to the start and the end of every footpath. As a result, Dijkstra's algorithm can be used for the transfer phase without needing any special handling.

For this thesis, we stick to the original minimum change time model, since it is more established in the context of the RAPTOR algorithm and trip-based network models. For future work, however, we recommend using the minimum entering/exiting time model proposed here.

## 2.6 Solving the Range Problem with RAPTOR

In this section, we will present algorithms for solving the Range Problem, where we are given a source stop $s \in \mathcal{S}$, a target stop $t \in \mathcal{S}$ and a range of departure times $I = [\tau_{\min}, \tau_{\max}]$. We want to find a Pareto set of journeys that contains, for each departure time $\tau_{\text{dep}} \in I$, all optimal journeys that depart no earlier than $\tau_{\text{dep}}$.

### 2.6.1 rRAPTOR

The rRAPTOR algorithm was proposed alongside the original RAPTOR algorithm in [DPW15]. It adapts the ideas of the Self-Pruning Connection-Setting (SPCS) algorithm [DKP09], an efficient algorithm for solving the Range Problem in a time-dependent graph-based network model, to RAPTOR's trip-based model. Like RAPTOR itself, it is naturally able to solve one-to-all queries, which will be important for the speedup techniques discussed later on, but it can also employ target pruning to solve one-to-one queries efficiently.

The algorithm is based on the following observation: If we assume that walking from $s$ to other stops is not possible, the only possible departure times for journeys starting at $s$ are those of the trips that depart from $s$. In order to solve the Range Problem, one could therefore collect all trips tr that visit $s$ and for which $\tau_{\text{dep}}(\text{tr}, s) \in I$, and perform a regular RAPTOR search with the departure time $\tau_{\text{dep}}(\text{tr}, s)$. Additionally, a RAPTOR search must be performed for the departure time $\tau_{\max}$, since it is possible that there are no trips that depart exactly at $\tau_{\max}$, in which case a later trip must be used.

When initial walking from $s$ is possible, this approach has to be adjusted accordingly: Let $S$ be the set of stops that can be reached via initial walking from $s$. For each stop $v \in S$ and each trip tr that visits $v$, we must check if $\tau_{\text{dep}}(\text{tr}, v) - \tau_w(P_{sv}) \in I$. If so, we perform a RAPTOR search with the departure time $\tau_{\text{dep}}(\text{tr}, v) - \tau_w(P_{sv})$. This search will find any journey that begins with walking to $v$ and entering tr there.

In order to obtain a Pareto set of optimal solutions to the Range Problem, we can just collect the optimal solutions of the individual RAPTOR searches and remove duplicates. Here it becomes clear that this approach will lead to a lot of redundant work. Not all trips that depart at $s$ or can be reached by walking are the start of an optimal journey. When performing a RAPTOR search for the departure time of an unneeded trip tr, each journey found by the search begins with a different trip $\text{tr}' \neq \text{tr}$ and is therefore also found by the RAPTOR search associated with $\text{tr}'$.

To prevent redundant searches, we perform the RAPTOR searches in descending order of departure time and retain the round table between searches instead of clearing it. This is possible because a journey that is feasible for a departure time $\tau_{\text{dep}}$ is still feasible for all earlier departure times $\tau'_{\text{dep}} < \tau_{\text{dep}}$. By keeping journeys from previous searches in the round table, later searches do not have to find them again and can immediately use them to prune suboptimal journeys. This property is called the *self-pruning* property.

Unlike the round table, the earliest arrival labels $\tau_r(\cdot)$ and $\tau_t(\cdot)$ cannot be carried over between RAPTOR executions. If the earliest arrival for a stop comes from round $k$ of a previous RAPTOR execution, it does not dominate arrivals found in rounds $k' < k$ during the current RAPTOR execution. Instead of using earliest arrival labels, we therefore change the definition of the round table: Instead of storing solutions with exactly $k$ trips, $R[2k]$ and $R[2k+1]$ now store solutions with at most $k$ trips. This can be achieved by initializing $R[2k]$ and $R[2k+1]$ with the entries from the previous round, i.e., $R[2k-2]$ and $R[2k-1]$, at the start of each round $k$.

**Initial Walking.** The algorithm as described so far repeats the initial transfer phase in round 0 for each RAPTOR execution. This is redundant and wastes time, especially when using an unrestricted footpath graph. Instead, we perform the initial transfer phase only once at the start of the algorithm, using a departure time of 0. This means that for each stop $v \in \mathcal{S}$, the arrival time $\tau_{\text{arr}}(v)$ found by this initial transfer phase is equal to the optimal walking time $\tau_w(P_{sv})$ from $s$ to $v$. We save these optimal walking times to be reused by the following RAPTOR executions. During a RAPTOR execution with departure time $\tau_{\text{dep}}$, instead of performing the initial transfer phase again, we simply update $R[1]$ with the information from the initial transfer scan. For each stop $v$ reached via an initial transfer, we set $R[1] = \tau_{\text{dep}} + \tau_w(P_{sv}))$.

The regular RAPTOR algorithm would then insert all of these stops into $U_t$ to be scanned in the next route phase, but this is not necessary in this case. Let $S$ be the set that contains every stop $u$ that is visited by a trip tr with the departure time $\tau_{\text{dep}}(\text{tr}, u) = \tau_{\text{dep}} + \tau_w(P_{su})$. Only the stops in $S$ need to be added to $U_t$. For any other stop $v \notin S$, if passengers depart at $\tau_{\text{dep}}$ and then walk to $v$, there is no trip that will depart immediately, so they will have to wait for the next trip. Instead of waiting at $v$, passengers could just depart later at $s$ so that they arrive just in time for the next trip. This later departure time is included in the set of scanned departure times, so a RAPTOR execution for it was already performed by the time $\tau_{\text{dep}}$ is scanned, and $v$ was added to $U_t$ during that RAPTOR execution. An exception to this is the RAPTOR execution for $\tau_{\max}$, which is performed first. Here we have to insert all stops reached by initial walking into $U_t$.

**Constructing Profiles.** The profile for a target stop $t \in \mathcal{S}$ can be constructed from a rRAPTOR execution in the following manner: The optimal walking time $\tau_w = \tau_w(P_{st})$ can be obtained from the initial transfer phase that is performed at the start of the algorithm. After a RAPTOR execution with departure time $\tau_{\text{dep}}$, a Pareto set of optimal journeys (or a single optimal journey for the Earliest Arrival Problem) can be obtained as described in Section 2.4. For each optimal journey with arrival time $\tau_{\text{arr}}$ and $k$ trips that was not already found in an earlier RAPTOR execution, the breakpoint $(\tau_{\text{dep}}, \tau_{\text{arr}})$ is then added to the function $f_k$ representing optimal journeys with $k$ trips.

**Problems with Unrestricted Walking.** The rRAPTOR algorithm was originally developed for scenarios where the footpath is transitively closed. While it can be trivially adapted to unrestricted walking by simply using a RAPTOR algorithm that can handle unrestricted walking, doing so leads to problems. Typically, unrestricted footpath graphs are fully (or almost fully) connected, so every stop can be reached from every other stop by walking. This means that when collecting the departure times that must be scanned, every trip departure at every stop will be relevant, so the rRAPTOR algorithm must perform one RAPTOR execution for every possible departure time in the entire network. This severely impacts the performance, leading to running times of several minutes on country-scale networks.

Among the RAPTOR executions will be many that are clearly absurd and will not lead to optimal journeys, e.g., a three-day walk to a train station in another city, just to take a train there. RAPTOR executions of this kind are very likely to finish almost immediately due to self-pruning, since scanning the first trip is unlikely to improve the round table entries of any stops along the trip. To ensure that the rRAPTOR algorithm remains at least somewhat efficient, it is important that RAPTOR executions that finish very quickly take up as little time as possible. In the current formulation of the algorithm, the round table of all first stop are initialized at the start of each round. If the round itself finishes quickly, this can easily dominate the running time. This can be prevented with the use of timestamps: Each round table entry maintains a timestamp that indicates when it was last updated. Whenever a round table entry is accessed, we check if its timestamp is up to date. If not, we initialize the entry as we would normally do at the start of the round and update the timestamp. This ensures that entries are only initialized when they are needed. As a result, RAPTOR executions that finish quickly and only access very few round table entries do not spend unnecessary time initializing the rest.

### 2.6.2 One-to-One Profile RAPTOR

As discussed above, the rRAPTOR algorithm partially loses its efficiency when using with unrestricted footpath graphs. As an alternative, Wagner and Zündorf [WZ17] proposed an algorithm for the Range Problem that is more efficient in unrestricted walking scenarios, but can only solve one-to-one queries. We will initially describe a simplified version of the algorithm for the Earliest Arrival problem and then discuss how it has to be adapted for Bicriteria Problem.

**Earliest Arrival Problem.** The algorithm is motivated by the following observation: In an $s$-$t$-profile for the Range Problem with earliest arrival optimality, each journey $J$ in the profile is optimal for a certain subinterval $I_J = [\tau_{\text{dep}}, \tau'_{\text{dep}}] \subseteq I$. If we know $\tau_{\text{dep}}$, we can do a RAPTOR search with departure time $\tau_{\text{dep}}$ to find $J$. If we knew $\tau'_{\text{dep}}$ as well, we could add a breakpoint $(\tau'_{\text{dep}}, \tau_{\text{arr}}(J))$ to the profile and continue with the next optimal journey, whose interval would start at $\tau'_{\text{dep}} + \epsilon$, where $\epsilon$ is an infinitesimally small time unit. [2] We could therefore find the next journey by performing a RAPTOR search with $\tau'_{\text{dep}} + \epsilon$. In fact, we cannot determine $\tau'_{\text{dep}}$ from knowing $J$ alone, but we can determine it by performing a backward RAPTOR search from $t$ with the arrival time $\tau_{\text{arr}}(J)$. This will find an $s$-$t$-journey $J'$ with the latest possible departure time $\tau_{\text{dep}}(J') = \tau'_{\text{dep}}$ at $s$. This allows us to complete the subinterval $I_J$ and move on to the next journey.

The basic algorithm for the Earliest Arrival Problem thus works as follows:

1. The current departure time $\tau_{\text{dep}}$ is initialized with $\tau_{\text{min}}$.
2. A forward RAPTOR search with departure time $\tau_{\text{dep}}$ is run from $s$. This finds a journey $J$ with minimal arrival time $\tau_{\text{arr}}(J)$ at $t$.
3. A backward RAPTOR search with arrival time $\tau_{\text{arr}}(J)$ is run from $t$. This finds a journey with maximal departure time $\tau'_{\text{dep}}$ at $s$.
4. The journey $J$ is added to the profile as the optimal journey for the interval $[\tau_{\text{dep}}, \tau'_{\text{dep}}]$, i.e., with the breakpoint $(\tau'_{\text{dep}}, \tau_{\text{arr}}(J))$. At this point the profile is complete for the interval $[\tau_{\text{min}}, \tau'_{\text{dep}}]$, while journeys still have to be found for the interval $[\tau'_{\text{dep}}+\epsilon, \tau_{\text{max}}]$.
5. If $\tau'_{\text{dep}} \geq \tau_{\text{max}}$, the algorithm is finished. Otherwise, the current departure time $\tau_{\text{dep}}$ is set to $\tau'_{\text{dep}} + \epsilon$ and the algorithm repeats from step 2.

**Pure Walking Journeys.** The algorithm as described so far fails if an optimal journey $J$ consists entirely of walking and thus contains no trips. If $J$ was found during a forward search with the departure time $\tau_{\text{dep}}$, the backward search will also yield a departure time of $\tau_{\text{dep}}$ because pure walking journeys do not involving waiting for a trip. This would mean the algorithm adds $J$ to the profile for the interval $[\tau_{\text{dep}}, \tau_{\text{dep}}]$ and continues with the departure time $\tau_{\text{dep}} + \epsilon$, thus making only infinitesimal progress. The optimal journey found for $\tau_{\text{dep}} + \epsilon$ is likely going to be $J$ again, so the algorithm progresses in steps of $\epsilon$ while finding the same journey each time.

To prevent this from occurring, we prohibit the RAPTOR executions from finding pure walking journeys, which can be achieved by not updating the round table entry for $t$ if it is

[2]In practice, we use a discrete representation of time, usually with a resolution of one second, so in that case we set $\epsilon = 1s$.

reached during the initial transfer phase. As with rRAPTOR, it is sufficient to perform the initial transfer phase only once and reuse the results for each RAPTOR execution. We can therefore compute the optimal pure walking journey during this initial transfer phase and immediately add it to the *s*-*t*-profile. Since the optimal pure walking journey is not added to the round table, journeys that are dominated by it may be found during the RAPTOR executions. We therefore compare each found journey to the pure walking journey and only add it to the profile if it is not dominated.

**Bicriteria Problem.** When using bicriteria optimality, the profile consists of one function for each number of trips. When running a forward RAPTOR search with departure time $\tau_{\text{dep}}$, we no longer find just a single journey, but rather a Pareto set $\mathcal{P}$ containing up to one journeys for each possible number of trips. To handle these journeys, we maintain a priority queue $Q$ which contains journeys for which no backward RAPTOR search has yet been performed. Whenever a forward RAPTOR search finishes, producing a Pareto set $\mathcal{P}$ of optimal journeys, all journeys in $\mathcal{P}$ are inserted into $Q$.

Afterwards, we extract a journey $J$ with minimal arrival time $\tau_{\text{arr}}(J)$ from $Q$. Let $\tau_{\text{dep}}$ be the departure time of the forward search that found $J$ and $k$ the number of trips in $J$. We perform a backward RAPTOR search for the arrival time $\tau_{\text{arr}}(J)$ from $t$ in order to obtain the maximal departure time $\tau'_{\text{dep}}$ for $k$ trips at $s$. We can then add $J$ as a solution to the function $f_k$ for $k$ trips in the interval $[\tau_{\text{dep}}, \tau'_{\text{dep}}]$. If there are multiple solutions with arrival time $\tau_{\text{arr}}(J)$ in $Q$, we can extract optimal departure time from the same backward search and add them to the profile as well.

For each journey that was added to the profile with the maximal departure time $\tau'_{\text{dep}}$, we then perform a forward search for the departure time $\tau'_{\text{dep}} + \epsilon$. Overall, the algorithm therefore alternates between performing a backward search for the minimal arrival time in $Q$ and performing the next forward searches to follow up on all journeys whose intervals were completed by the backward search.

# 3. Computing Optimal Subjourneys

Preprocessing techniques for shortest path problems on graphs with scalar edge weights (e.g., road networks without time-dependence) often exploit the *subpath property*: Subpaths of shortest paths are always shortest paths themselves.

**Theorem 3.1.** *Let $G = (V, E)$ be a graph with a scalar edge weight function $w : E \to \mathbb{R}$ and let $P = (v_1, \dots, v_k)$ be a shortest $v_1$-$v_k$-path in $G$. Then for each $1 \leq i \leq j \leq k$, the subpath $P_{ij}$ is a shortest $v_i$-$v_j$-path.*

*Proof.* Assume for the sake of contradiction that $P_{ij}$ is not a shortest $v_i$-$v_j$-path. This means that another $v_i$-$v_j$ path $P'_{ij}$ exists with $w(P'_{ij}) < w(P_{ij})$. Let $P'$ be the $v_1$-$v_k$-path that is obtained by replacing $P_{ij}$ with $P'_{ij}$ in $P$. Then $w(P') = w(P) - w(P_{ij}) + w(P'_{ij}) < w(P)$ holds, and thus $P$ is not a shortest $v_i$-$v_j$-path. □

Preprocessing techniques can exploit this property in numerous ways. Contraction-based techniques (including CH and CRP) compute shortcuts that condense optimal paths into a single edge. Due to the subpath property, these shortcuts can be used as subpaths in longer optimal paths. Arc-Flags partitions the graph into a set of cells $\mathcal{Z}$. An edge is flagged as necessary for a cell $z \in \mathcal{Z}$ if it lies on the shortest path tree of any boundary vertex of $z$. The subpath property guarantees that this flags all edges outside of $z$ that lie on the shortest path tree of any vertex in $z$. Without the subpath property, these techniques would not work because shortest paths might include suboptimal subpaths. In this case, the query algorithm used in the preprocessing phase would have to be adapted to find these necessary suboptimal paths as well.

In order to adapt speedup techniques for graph-based networks to the public transit scenario, it is necessary to define an analogous property to the subpath property for journeys. This presents a number of challenges:

Figure 3.1 shows an example of a suboptimal subjourney that is nevertheless part of an optimal journey because there is a waiting buffer before the next trip departs. However, we observe that replacing the suboptimal subjourney with an optimal subjourney still yields an optimal path with the same arrival time and number of trips. We therefore resolve this issue by using slightly relaxed requirements for the subjourney property: Instead of requiring that every optimal journey consists entirely of optimal subjourneys, we only require that at least one optimal journey with that property exists for every possible number of trips.

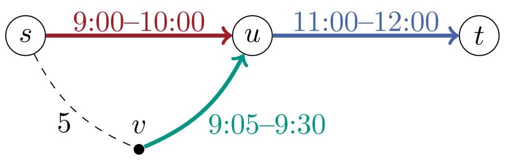


Figure 3.1: In this example, there are two optimal $s$-$t$-journeys, one involving the direct trip from $s$ to $u$ and the other involving a footpath to $v$ and trip to $u$ from there. In both cases, passengers must wait until 11:00 at $u$ to enter the trip to $t$. When considering the $s$-$u$-subjourney, only the journey via $v$ is optimal, since it arrives earlier at $u$. Thus, the direct trip from $s$ to $u$ is not an optimal journey, but it can be used as a subjourney in an optimal $s$-$t$-journey.

This is sufficient because it still allows us to construct a full Pareto set of optimal journeys from optimal subjourneys.

Another issue is that the proof for Theorem 3.1 is based on the observation that exchanging a subpath in a valid path still yields a valid path. This is not the case with public transit journeys due to their time-dependence. For example, consider an $s$-$t$-journey $J = ((\mathrm{tr}_1, u_1, v_1), \dots, (\mathrm{tr}_k, u_k, v_k))$ for some source and target stop $s, t \in \mathcal{S}$ and an $s'$-$t'$ subjourney $J_{uv}$ for vertices $u, v \in V$ that are visited by $J$. Replacing $J_{uv}$ with a different subjourney $J'_{uv}$ only yields a valid journey if it is possible to enter $J'_{uv}$ at $u$ and enter the next trip of $J$ at $v$. At the least, this requires that $J'_{uv}$ departs at $u$ no earlier than $J$ arrives, and that it arrives at $v$ no later than $J$ departs.

Additional requirements are introduced by the minimum change time: If $u$ or $v$ is a stop and inserting $J'_{uv}$ requires a trip switch at $u$ or $v$, the minimum change time must be obeyed. To explore the impact of these restrictions, we first consider the case of proper subjourneys and then the more complicated case of improper subjourneys.

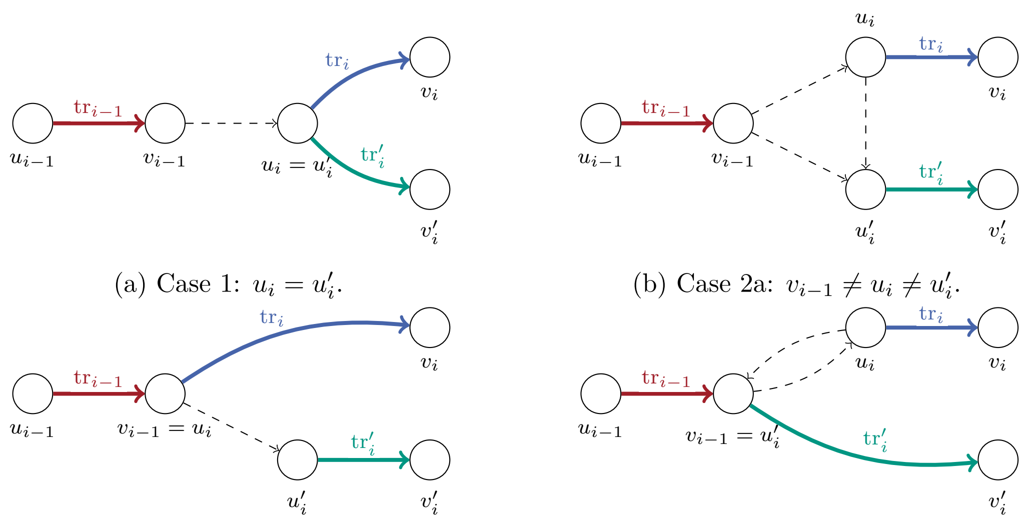
## 3.1 Proper Subjourneys

The defining characteristic of proper subjourneys is that they never begin or end midway through a trip or footpath – a proper subjourney always begins/ends at a point where the full journey enters/exits a trip, respectively. This is crucial because it means that replacing a subjourney with another one cannot introduce new trip switches at the start-/endpoint. This allows us to state a subjourney property for proper subjourneys:

**Theorem 3.2.** *Let $s, t \in \mathcal{S}$ be stops, $\tau$ a departure time and $k$ a number of trips. If an optimal $s$-$t$-journey for $\tau$ with $k$ trips exists, then there also exists an optimal $s$-$t$-journey $J = ((tr_1, u_1, v_1), \dots, (tr_k, u_k, v_k))$ for $\tau$ with $k$ trips such that every proper subjourney of $J$ is optimal as well, i.e., for each $1 \le i < j \le k$, the subjourney $J_{ij} = ((tr_i, u_i, v_i), \dots, (tr_j, u_j, v_j))$ is an optimal $u_i$-$v_j$-journey for $\tau_{dep}(J_{ij})$.*

*Proof.* We prove this theorem by induction over the number of trips $k$. For $k = 0$, the theorem is trivially true, since a journey with no trips has no proper subjourneys. Assume now that the theorem has already been proven for all journeys with less than $k$ trips. We will then show that it also holds for journeys with exactly $k$ trips.

Let $J = ((\mathrm{tr}_1, u_1, v_1), \dots, (\mathrm{tr}_k, u_k, v_k))$ be an optimal $s$-$t$-journey with $k$ trips. We show that if $J$ has at least one proper subjourney that is not optimal, we can construct another $s$-$t$-journey with $k$ trips that is also optimal but has fewer proper subjourneys that are not optimal. By repeating this construction iteratively, we can construct an optimal journey with $k$ trips whose proper subjourneys are all optimal.



(a) Case 1: $u_i = u'_i$.

(b) Case 2a: $v_{i-1} \neq u_i \neq u'_i$.

(c) Case 2b: $v_{i-1} = u_i \neq u'_i$.

(d) Case 2c: $v_{i-1} = u'_i \neq u_i$.

Figure 3.2: Illustrations of the four possible cases that can occur when replacing $\mathrm{tr}_i$ with $\mathrm{tr}'_i$ in the proof of Theorem 3.2.

Let $J_{ij} = ((\mathrm{tr}_i, u_i, v_i), \ldots, (\mathrm{tr}_j, u_j, v_j))$ be a proper subjourney of $J$ that is not an optimal $u_i$-$v_j$-journey for the departure time $\tau_{\mathrm{dep}}(J_{ij}) = \tau_{\mathrm{dep}}(\mathrm{tr}_i, u_i)$. This means that an optimal $u_i$-$v_j$-journey $J'_{ij}$ exists that dominates $J_{ij}$. Let $k_{ij} := j - i$ be the number of trips in $J_{ij}$ and $k'_{ij}$ the number of trips in $J'_{ij}$. Since $J_{ij} \neq J$ and $J'_{ij}$ dominates $J_{ij}$, we know that $k'_{ij} \leq k_{ij} < k$. We will demonstrate later that in fact $k'_{ij} = k_{ij}$ must hold. According to the induction hypothesis, there exists an optimal $u_i$-$v_j$-journey with $k_{ij}$ trips whose proper subjourneys are all optimal. Without loss of generality, let $J'_{ij} = ((\mathrm{tr}'_i, u'_i, v'_i), \ldots, (\mathrm{tr}'_j, u'_j, v'_j))$ be that journey. We now show that by replacing $J_{ij}$ with $J'_{ij}$ in $J$, we receive a journey $J'$ that dominates $J$.

First we must show that $J'$ is a valid journey. In particular, we must show that the transfers between the first and last trip of $J'_{ij}$ and the remainder of $J'$ are valid. We will show that the first trip $\mathrm{tr}'_i$ of $J'_{ij}$ can be entered. A symmetrical argument can be applied to show that it is possible to enter the next trip $\mathrm{tr}_{j+1}$ after $J_{ij}$, if it exists.

If $i = 1$, it must be possible to walk from $s$ to $u'_i$ in time to enter $\mathrm{tr}'_i$, i.e., $\tau + \tau_w(P_{su'_i}) < \tau_{\mathrm{dep}}(\mathrm{tr}'_i, u'_i)$ must hold. We already know that $J'_{ij}$ is a valid $u_i$-$v_j$-journey for the departure time $\tau_{\mathrm{dep}}(\mathrm{tr}_i, u_i)$, so $\tau_{\mathrm{dep}}(\mathrm{tr}_i, u_i) + \tau_w(P_{u_i u'_i}) < \tau_{\mathrm{dep}}(\mathrm{tr}'_i, u'_i)$ must hold. Likewise, we know that $J_{ij}$ is a valid journey for the departure time $\tau$, so $\tau + \tau_w(P_{su_i}) < \tau_{\mathrm{dep}}(\mathrm{tr}_i, u_i)$ must hold. Together with the triangle inequality, this yields $\tau + \tau_w(P_{su'_i}) \leq \tau + \tau_w(P_{su_i}) + \tau_w(P_{u_i u'_i}) < \tau_{\mathrm{dep}}(\mathrm{tr}_i, u_i) + \tau_w(P_{u_i u'_i}) < \tau_{\mathrm{dep}}(\mathrm{tr}'_i, u'_i)$. Note that this argument also holds in the special case $s = u'_i$, where no walking is necessary.

If $i > 1$, it must be possible to transfer from $\mathrm{tr}_{i-1}$ to $\mathrm{tr}_i$. We already know that $J'_{ij}$ is valid for the departure time $\tau_{\mathrm{dep}}(J_{ij})$ at $u_i$. There are several cases to consider, depending on the relation between $v_{i-1}$, $u_i$ and $u'_i$. Illustrations of the cases are given in Figure 3.2. The cases are as follows:

- **Case 1.** $u_i = u'_i$: In this case, $J'_{ij}$ begins at the same stop as $J_{ij}$. Since $\tau_{\mathrm{dep}}(J'_{ij}) \geq \tau_{\mathrm{dep}}(J_{ij})$ and $\mathrm{tr}_i \neq \mathrm{tr}_{i-1}$ can be entered, $\mathrm{tr}'_i$ can be entered as well.
- **Case 2a.** $v_{i-1} \neq u_i \neq u'_i$: In this case, a walking leg to $u_i$ is changed to a walking leg to $u'_i$. Here the same argument as for $i = 1$ applies. Since there is enough time to walk from $v_{i-1}$ to $u_i$ and from $u_i$ to $u'_i$, there is also enough time to walk directly from $v_{i-1}$ to $u'_i$.

- **Case 2b.** $v_{i-1} = u_i \neq u'_i$: In this case, a direct trip switch at $v_{i-1}$ turns into a walking leg to $u'_i$. Since there is enough time to walk to $u'_i$ when departing at $\tau_{\text{dep}}(\text{tr}_i, u_i)$, there is also enough time when departing at $\tau_{\text{arr}}(\text{tr}_{i-1}, v_{i-1}) \leq \tau_{\text{dep}}(\text{tr}_i, u_i)$.
- **Case 2c.** $v_{i-1} = u'_i \neq u_i$: This case is slightly more complicated because a walking leg is turned into a direct trip switch, so the minimum change time at $v_{i-1}$ must be obeyed. Using the same argument as for $i = 1$ and Case 2a, it can be be shown that $\tau_{\text{arr}}(\text{tr}_{i-1}, v_{i-1}) + \tau_w(P_{v_{i-1}u_i}) + \tau_w(P_{u_i v_{i-1}}) \leq \tau_{\text{dep}}(\text{tr}'_i, u'_i)$ holds, i.e., $J'_{ij}$ can be entered by taking the cyclical footpath to $u_i$ and back again. In Chapter 2.5.2, we introduced the restriction that cyclical footpaths of this sort are always shorter than the minimum change time, so it is possible to transfer to $\text{tr}'_i$.

Let $k'$ be the number of trips in $J'$. We now show that $J'$ dominates $J$, i.e., $\tau_{\text{arr}}(J') \leq \tau_{\text{arr}}(J)$ and $k' \leq k$. If $J_{ij}$ is not a suffix of $J$, then the arrival time of the journey remains unchanged by substituting $J_{ij}$ for $J'_{ij}$. If it is a suffix, we know that $\tau_{\text{arr}}(J') = \tau_{\text{arr}}(J'_{ij}) \leq \tau_{\text{arr}}(J_{ij}) = \tau_{\text{arr}}(J)$. Regarding the number of trips, we know that $k' \leq k - k_{ij} + k'_{ij} \leq k$ [1] Thus, $J'$ is not worse than $J$ according to any domination criteria and therefore dominates $J$. Because $J$ was already optimal, this means $J'$ must be optimal as well. Furthermore, it means that $J'$ has exactly $k$ trips, because if it had fewer trips than $J$, $J$ would not be optimal. From this it also follows that $k'_{ij} = k_{ij}$.

It remains to be shown that $J'$ has fewer suboptimal proper subjourneys than $J$. Because the number of trips did not change, the number of proper subjourneys remains the same. Let $J_{mn}$ be a proper subjourney of $J$ that is optimal. We will show that $J'_{mn}$ must be optimal as well. Since the number of trips is the same, it must be shown in particular that $\tau_{\text{arr}}(J'_{mn}) = \tau_{\text{arr}}(\text{tr}'_n, v'_n) \leq \tau_{\text{arr}}(\text{tr}_n, v_n) = \tau_{\text{arr}}(J_{mn})$. If $\text{tr}_n$ is not part of $J_{ij}$, then $\text{tr}'_n = \text{tr}_n$ and $v'_n = v_n$, and therefore the arrival time remains the same. If $\text{tr}_n$ is part of $J_{ij}$, consider the subjourney $J_{in}$. As a proper subjourney of $J_{ij}$, it is optimal, so $\tau_{\text{arr}}(\text{tr}'_n, v'_n) \leq \tau_{\text{arr}}(\text{tr}_n, v_n)$. This proves that $J'$ cannot add new suboptimal proper subjourneys. Since $J_{ij}$ is not optimal but $J'_{ij}$ is, $J'$ has at least one fewer suboptimal proper subjourney than $J$. □

By generalizing to an arbitrary number of trips, we obtain the following result:

**Corollary 3.3.** *Let $s, t \in \mathcal{S}$ be stops and $\tau$ a departure time. Then there exists a Pareto set $\mathcal{P}$ of optimal $s$-$t$-journeys for $\tau$ such that every proper subjourney of every journey $J \in \mathcal{P}$ is optimal as well.*

In other words, there is always an optimal solution that fulfills the subpath property. This means that speedup techniques that are based on precomputing proper subjourneys only have to compute optimal journeys and can therefore use an unmodified RAPTOR algorithm. A query algorithm using the precomputed subjourneys can always find an optimal journey.

## 3.2 Switch-Optimality

Unfortunately, Theorem 3.2 cannot be extended directly to improper subjourneys. The crucial part of the proof that does not extend to improper subjourneys relates to switching between trips. A proper subjourney always begins with entering a trip at some stop $v$. If the full journey arrives at $v$ with a trip, we know that the minimum change time is already obeyed there, which makes it possible to replace the subjourney with another optimal journey, even if it begins with a different trip. The same is true at the end of the proper

[1] It is possible that $k' < k - k_{ij} + k'_{ij}$ if $\text{tr}_{i-1} = \text{tr}'_i$ or $\text{tr}'_j = \text{tr}_{j+1}$. In this case, a trip switch is substituted with a trip continuation and the continued trips are only counted once.

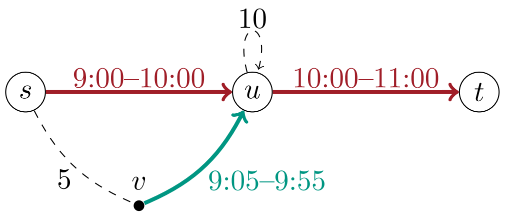


Figure 3.3: An example of a situation where no optimal $s$-$t$-journey exists whose improper subjourneys are all optimal. The trip segments connecting $s$ and $u$ as well as $u$ and $t$ belong to the same trip. Therefore, it is possible to take the two trip segments consecutively without having to switch between trips at $u$. Doing so yields an optimal $s$-$t$-journey for the departure time 9:00. The $s$-$u$-subjourney is an improper subjourney because it ends midway through a trip. It is not optimal because the journey involving walking to $v$ and taking the trip to $u$ arrives 5 minutes earlier. However, this superior $s$-$u$-journey cannot be extended to an $s$-$t$-journey because there is not enough time to switch to the trip departing at 10:00 within the required minimum change time. Thus, there is only one optimal $s$-$t$-journey and one of its improper subjourneys is not optimal.

subjourney, which always ends with exiting a trip. This is no longer true if we consider improper subjourneys. An improper subjourney may start or end in the middle of a trip segment or a footpath in the full journey, where no trip is entered or exited. If we try to substitute a different subjourney that starts or ends with another trip, we may introduce a new trip switch that was not there originally. This trip switch may not be possible due to the minimum change time. An example where the trip switch is not possible for any optimal subjourney can be seen in 3.3. As a result, no optimal $s$-$t$-journey exists whose improper subjourneys are all optimal. This demonstrates that the subjourney property we proved for proper subjourneys does not hold for improper subjourneys.

If we want to apply speedup techniques that rely on precomputed improper subjourneys, it does not suffice to compute only optimal improper subjourneys. We have to compute all journeys that can potentially be improper subjourneys of an optimal journey. We call this property *switch-optimality.*

**Definition 3.4.** *For a given source and target stop $s, t \in \mathcal{S}$ and a minimum departure time $\tau_{dep}$, an $s$-$t$-journey $J_1$ source-switch-dominates an $s$-$t$-journey $J_2$ iff both of the following conditions hold:*

1. *$J_1$ dominates $J_2$.*
2. *For each trip tr that arrives at $s$ with an arrival time $\tau_{arr}(tr, s) \leq \tau_{dep}$, at least one of the following conditions must be fulfilled:*
   a) *$J_2$ departs from $s$ via a trip $tr_2 \neq tr$ for which $\tau_{dep}(tr_2, s) < \tau_{arr}(tr, s) + \tau_{ch}(s)$.*
   b) *$J_1$ departs from $s$ via a footpath or tr.*
   c) *$J_1$ departs from $s$ via a trip $tr_1 \neq tr$ and $\tau_{dep}(tr_1, s) \geq \tau_{arr}(tr, s) + \tau_{ch}(s)$ holds.*

Condition 2 ensures that for any arrival at $s$ before $\tau_{\text{dep}}$ that allows passengers to enter $J_2$, it is also be possible to enter $J_1$. This is obviously possible for arrivals via footpaths since $J_1$ is feasible for $\tau_{\text{dep}}$. For any arrival via a trip tr, a trip switch is necessary to enter $J_1$ unless $J_1$ starts with a footpath or tr itself, which is Condition 2b. If Condition 2a is fulfilled, it is not possible to switch from tr to $J_2$ within the minimum change time, so it does not need to be possible to switch to $J_1$ either. Otherwise, Condition 2c ensures that the trip switch from tr to $J_1$ is possible within the minimum change time.

**Definition 3.5.** *For a given source and target stop $s, t \in \mathcal{S}$, an s-t-journey $J_1$ target-switch-dominates an s-t-journey $J_2$ iff both of the following conditions hold:*

1. *$J_1$ dominates $J_2$.*
2. *For each trip tr that departs at t with a departure time $\tau_{dep}(tr, t) \geq \tau_{arr}(J_2)$, at least one of the following conditions must be fulfilled:*
   a) *$J_2$ arrives at t via a trip $tr_2 \neq tr$.*
   b) *$J_1$ arrives at t via a footpath or tr.*
   c) *$J_1$ arrives at t via a trip $tr_1 \neq tr$ and $\tau_{arr}(tr_1, t) + \tau_{ch}(t) \leq \tau_{dep}(tr, t)$ holds.*

Here, the second condition ensures that any departure at $t$ that passengers can take if they arrived via $J_2$ can also be taken if they arrived via $J_1$. For departures via footpaths, this requires $\tau_{\text{arr}}(J_1) \leq \tau_{\text{arr}}(J_2)$, which is already fulfilled if $J_1$ dominates $J_2$. For any departure via a trip tr, a trip switch is necessary to enter tr from $J_1$ unless $J_1$ ends with a footpath or tr itself, which is Condition 2b. If Condition 2a is fulfilled, a trip switch is also necessary to enter tr from $J_2$. If this is possible within the minimum change time, it is also possible for $J_1$, which arrives at $t$ no later than $J_2$. Otherwise, Condition 2c ensures that the trip switch from $J_1$ to tr is possible within the minimum change time.

**Definition 3.6.** *For a given source and target stop $s, t \in \mathcal{S}$ and a minimum departure time $\tau_{dep}$, an s-t-journey $J_1$ switch-dominates an s-t-journey $J_2$ iff $J_1$ source-switch-dominates and target-switch-dominates $J_2$.*

We call a journey *(source-/target-)switch-optimal* if it is Pareto-optimal according to (source-/target-)switch-dominance.

The additional conditions introduced for switch-dominance ensure that a journey $J$ can only switch-dominate another journey $J'$ if it can always replace $J'$ as a subjourney in a longer journey. As a result, journeys that may be needed as subjourneys are not switch-dominated and are included in a Pareto set of switch-optimal journeys. Because suboptimal journeys may be switch-optimal, the Earliest Arrival Problem can have more than one switch-optimal solution and the Bicriteria Problem can have more than one switch-optimal solution per number of trips. The remainder of this chapter will discuss how the RAPTOR algorithm can be adapted to compute (source-/target-)switch-optimal journeys.

## 3.3 Computing Target-Switch-Optimal Journeys

Our approach for computing target-switch-optimal journeys is based on the following observation:

**Theorem 3.7.** *Let $s, t \in \mathcal{S}$ be stops, $\tau$ a departure time, $k$ a number of trips and $R$ the round table after a RAPTOR query for s, t and $\tau$. Let $\mathcal{P}'$ be the set of all s-t-journeys $J = ((tr_1, u_1, v_1), \ldots, (tr_k, u_k, v_k))$ for $\tau$ with $k$ trips that have the following properties:*

1. *If $J$ ends with a footpath ($v_k \neq t$), $R[2k+1][t] = \tau_{arr}(J)$.*
2. *If $J$ ends with a trip ($v_k = t$), $R[2k-1][u_k] = \tau_{arr}(J_t)$ for the s-$u_k$-journey $J_t = ((tr_1, u_1, v_1), \ldots, (tr_{k-1}, u_{k-1}, v_{k-1}))$.*

*Then every s-t-journey for $\tau$ with $k$ trips that is not in $\mathcal{P}'$ is target-switch-dominated by a journey in $\mathcal{P}'$.*

*Proof.* Let $J = ((\mathrm{tr}_1, u_1, v_1), \dots, (\mathrm{tr}_k, u_k, v_k))$ be an $s$-$t$-journey that is not in $\mathcal{P}'$. We show that it is target-switch-dominated by a journey in $\mathcal{P}'$.

If $J$ ends with a footpath, let $J' \in \mathcal{P}'$ be the journey with $R[2k+1][t] = \tau_{\mathrm{arr}}(J')$. Obviously $\tau_{\mathrm{arr}}(J') \leq \tau_{\mathrm{arr}}(J)$ must hold, since otherwise $\tau_{\mathrm{arr}}(J)$ would be in the round table instead of $\tau_{\mathrm{arr}}(J')$. This means that $J'$ dominates $J$, and since $J'$ ends with a footpath, it also target-switch-dominates $J$.

If $J$ ends with a trip, let $J_t = ((\mathrm{tr}_1, u_1, v_1), \dots, (\mathrm{tr}_{k-1}, u_{k-1}, v_{k-1}))$ be its $s$-$u_k$-subjourney and let $J' = ((\mathrm{tr}'_1, u'_1, v'_1), \dots, (\mathrm{tr}'_{k-1}, u'_{k-1}, v'_{k-1}), (\mathrm{tr}_k, u_k, v_k)) \in \mathcal{P}'$ be the journey for which $R[2k-1][u_k] = \tau_{\mathrm{arr}}(J_t)$ holds for $J'_t = ((\mathrm{tr}'_1, u'_1, v'_1), \dots, (\mathrm{tr}'_{k-1}, u'_{k-1}, v'_{k-1}))$. Obviously $\tau_{\mathrm{arr}}(J'_t) \leq \tau_{\mathrm{arr}}(J_t)$ must hold, since otherwise $\tau_{\mathrm{arr}}(J_t)$ would be in the round table instead of $\tau_{\mathrm{arr}}(J'_t)$. Since $J$ and $J'$ both arrive at $v_k$ with the same trip, $J'$ target-switch-dominates $J$. □

A direct consequence of this theorem is that $\mathcal{P}'$ contains a full Pareto set $\mathcal{P}$ of target-switch-optimal journeys for $\tau$ and $k$ trips. This means that it is possible to find $\mathcal{P}$ by extending journeys in the round table of a normal RAPTOR execution by at most one trip. This allows us to keep most of the RAPTOR algorithm unchanged, particularly the basic structure of the round table. Additionally, at every potential target stop $t$, we maintain one Pareto set $\mathcal{P}[k][t]$ per number of trips $k$ that contains labels for the target-switch-optimal journeys. To enable journey retrieval, we add a parent pointer to each journey in $\mathcal{P}$ that points to the optimal journey from the previous round that it extends.

A naive approach for maintaining $\mathcal{P}[k][t]$ is to compare each new journey $J$ that is found to every existing journey $J' \in \mathcal{P}[k][t]$. If any $J'$ target-switch-dominates $J$, $J$ can be discarded. Otherwise, $J$ is inserted and all journeys $J'$ that are target-switch-dominated by $J$ are removed. This requires a routine for determining target-switch-dominance, which is called $\mathcal{O}(|\mathcal{P}[k][t]|)$ times when adding a journey.

Given two $s$-$t$-journeys $J_1$ and $J_2$ for the departure time $\tau_{\mathrm{dep}}$ with $\leq k$ trips, the following steps need to be performed to determine if $J_1$ target-switch-dominates $J_2$:

- If $J_1$ does not dominate $J_2$, i.e., $\tau_{\mathrm{arr}}(J_1) \leq \tau_{\mathrm{arr}}(J_2)$ or $|J_1| \leq |J_2|$ does not hold, $J_1$ does not target-switch-dominate $J_2$.
- Assume now that $J_1$ dominates $J_2$. If $J_1$ ends with a footpath, passengers exiting $J_1$ can enter any trip tr that departs at $t$ with a departure time $\tau_{\mathrm{dep}}(\mathrm{tr}, t) \geq \tau_{\mathrm{arr}}(J_2) \geq \tau_{\mathrm{arr}}(J_1)$. Thus, $J_1$ target-switch-dominates $J_2$.
- If $J_1$ does not end with a footpath, let $\mathrm{tr}_1$ be the final trip of $J_1$. If $J_2$ also ends with $\mathrm{tr}_1$ or a trip that ends at $t$, passengers that exit $J_2$ can only enter trips that obey the minimum change time, except for $\mathrm{tr}_1$. Since passengers exiting $J_1$ can also enter those trips, $J_1$ target-switch-dominates $J_2$.
- If $J_2$ ends with a footpath, passengers exiting $J_2$ can enter every trip that departs after $\tau_{\mathrm{arr}}(J_2)$, while passengers exiting $J_1$ must obey the minimum change time to enter any trip except $\mathrm{tr}_1$. $J_1$ target-switch-dominates $J_2$ if $\mathrm{tr}_1$ is the only departing trip within the minimum departure time. To check this, find the earliest departing trip $\mathrm{tr}' \neq \mathrm{tr}_1$ with $\tau_{\mathrm{dep}}(\mathrm{tr}', t) \geq \tau_{\mathrm{arr}}(J_2)$. If no such trip exists, $J_1$ target-switch-dominates $J_2$. Otherwise, $J_1$ target-switch-dominates $J_2$ if $\tau_{\mathrm{arr}}(J_1) + \tau_{\mathrm{ch}}(t) \leq \tau_{\mathrm{dep}}(\mathrm{tr}', t)$.
- If $J_2$ ends with a trip $\mathrm{tr}_2 \neq \mathrm{tr}_1$ that does not end at $t$, passengers exiting either journey must obey the minimum change time except for $\mathrm{tr}_1$ in the case of $J_1$ and $\mathrm{tr}_2$ in the case of $J_2$. $J_1$ target-switch-dominates $J_2$ if passengers exiting $J_1$ can enter $\mathrm{tr}_2$, i.e., $\tau_{\mathrm{arr}}(J_1) + \tau_{\mathrm{ch}}(t) \leq \tau_{\mathrm{dep}}(\mathrm{tr}_2, t)$.

| $J_1$ | $J_2$ | | |
|---|---|---|---|
| | Footpath | Non-con. trip | Con. trip $\mathrm{tr}_2$ |
| Footpath | $\tau_\mathrm{a}(J_1) \leq \tau_\mathrm{a}(J_2)$ | $\tau_\mathrm{a}(J_1) \leq \tau_\mathrm{a}(J_2)$ | $\tau_\mathrm{a}(J_1) \leq \tau_\mathrm{a}(J_2)$ |
| Non-con. trip | $\tau_\mathrm{a}(J_1) + \tau_\mathrm{c}(t) \leq \tau_\mathrm{a}(J_2)$ | $\tau_\mathrm{a}(J_1) \leq \tau_\mathrm{a}(J_2)$ | $\tau_\mathrm{a}(J_1) + \tau_\mathrm{c}(t) \leq \tau_\mathrm{a}(J_2)$ |
| Con. trip $\mathrm{tr}_1$ | $\tau_\mathrm{a}(J_1) + \tau_\mathrm{c}(t) \leq \tau_\mathrm{a}(J_2)$ | $\tau_\mathrm{a}(J_1) \leq \tau_\mathrm{a}(J_2)$ | $\mathrm{tr}_1 = \mathrm{tr}_2 \vee$ $\tau_\mathrm{a}(J_1) + \tau_\mathrm{c}(t) \leq \tau_\mathrm{a}(J_2)$ |

Table 3.1: Conditions for simplified target-switch-dominance. Each cell lists the conditions under which $J_1$ target-switch-dominates $J_2$, depending on the last leg of either journey. Due to space constraints, we abbreviate the arrival time $\tau_\mathrm{arr}$ as $\tau_\mathrm{a}$ and the minimum change time $\tau_\mathrm{ch}$ as $\tau_\mathrm{c}$.

The case for $J_1$ ending in a trip and $J_2$ ending in a footpath includes the step of finding the earliest departing trip $\mathrm{tr}' \neq \mathrm{tr}_1$ with $\tau_\mathrm{dep}(\mathrm{tr}', t) \geq \tau_\mathrm{arr}(J_2)$. In practice, this search is too costly. To avoid it, we strengthen the criteria for target-switch-dominance by assuming that a trip $\mathrm{tr}' \neq \mathrm{tr}_1$ with $\tau_\mathrm{dep}(\mathrm{tr}', t) = \tau_\mathrm{arr}(J_2)$ exists. Under this assumption, $J_1$ target-switch-dominates $J_2$ if $\tau_\mathrm{arr}(J_1) + \tau_\mathrm{ch}(t) \leq \tau_\mathrm{arr}(J_2)$. In order to simplify the dominance test even further, we also use this condition for the case that $J_2$ ends with a trip $\mathrm{tr}_2 \neq \mathrm{tr}_1$. This condition can be checked in constant time and is stricter than the original conditions. As a consequence of making target-switch-dominance stricter than necessary, we now compute an overapproximation of the Pareto set of target-switch-optimal journeys.

A further optimization is based on the following two observations about $\mathcal{P}[k][t]$:

- $\mathcal{P}[k][t]$ can include at most one journey $J$ that ends with a footpath, since any journey that is dominated by $J$ is also target-switch-dominated by it.
- $\mathcal{P}[k][t]$ can include at most one journey that ends with a trip that ends at $t$. We call such a trip a *non-continuing trip*, as opposed to a *continuing* trip. Assume that $\mathcal{P}[k][t]$ contained two journeys $J_1$ and $J_2$ ending in non-continuing trips, and, without loss of generality, $\tau_\mathrm{arr}(J_1) \leq \tau_\mathrm{arr}(J_2)$. Then $J_1$ would target-switch-dominate $J_2$ because passengers exiting either journey need to obey the minimum change time to enter any other trip.

We can exploit these observations by splitting $\mathcal{P}[k][t]$ into three components: $\mathcal{P}_f[k][t]$ is the only target-switch-optimal journey that ends with a footpath (or $\perp$ if none exists), $\mathcal{P}_n[k][t]$ in the only target-switch-optimal journey that ends with a non-continuing trip (or $\perp$ if none exists), and $\mathcal{P}_t[k][t]$ is a Pareto set of target-switch-optimal journeys that end in continuing trips, sorted by their arrival time. The conditions that need to be checked to determine if a journey $J_1$ target-switch-dominates another journey $J_2$ are now dependent on which components the two journeys belong to; Table 3.1 lists these conditions.

When a new $s$-$t$-journey $J$ is found during round $k$, the following steps need to be taken to update $\mathcal{P}[k][t]$:

- Check if $\mathcal{P}_f[k][t]$, $\mathcal{P}_n[k][t]$ or $\mathcal{P}_t[k][t]$ contains a journey that target-switch-dominates $J$. If so, discard $J$.
- If $J$ is not target-switch-dominated, delete all entries that are dominated by it. In the case of $\mathcal{P}_f[k][t]$ or $\mathcal{P}_n[k][t]$, if the respective journey is dominated, replace it with $\perp$.
- Add $J$ to its respective component, depending on whether it ends with a transfer, a non-continuing trip or a continuing trip. In the case of $\mathcal{P}_f[k][t]$ or $\mathcal{P}_n[k][t]$, the

journey can simply be overwritten. In the case of $\mathcal{P}_t[k][t]$, $J$ must be inserted at the right index so that it is still sorted by arrival time.

The fact that $\mathcal{P}_t[k][t]$ is sorted by arrival time allows us to implement the dominance checks efficiently. Except for the condition $\mathrm{tr}_1 = \mathrm{tr}_2$ in the case that both journeys end with a continuing trip, all conditions depend only on the arrival times of the two journeys as well as the minimum change time at $t$, which is independent of the journeys. When checking if a new journey $J$ is target-switch-dominated by any journey in $\mathcal{P}_t[k][t]$, it suffices to check if it is dominated by the journey with the smallest arrival time among $\mathcal{P}_t[k][t]$, which can be found in constant time. To delete journeys that are target-switch-dominated by $J$, it suffices to iterate over $\mathcal{P}_t[k][t]$ in descending order of arrival time until we find a journey that is not target-switch-dominated. The only exception is if $J_1$ ends with a continuing trip $\mathrm{tr}_1$ and there is a journey $J_2 \in \mathcal{P}_t[k][t]$ that also ends with $\mathrm{tr}_1$. Since $\tau_{\mathrm{arr}}(J_1) = \tau_{\mathrm{arr}}(J_2)$, it does not matter which of the two journeys we preserve. When iterating over $\mathcal{P}_t[k][t]$ to find target-switch-dominated journeys, we will eventually encounter $J_2$. Since it is target-switch-dominated by $J_1$, we can simply delete it as we do with other target-switch-dominated journeys.

## 3.4 Computing Source-Switch-Optimal Journeys

In the target-switch-optimal case, we exploited the fact that target-switch-optimal journeys can be constructed out of journeys found by a regular RAPTOR algorithm, followed by a single trip. This allowed us to keep the round table data structure unchanged and handle target-switch-dominance via separate Pareto set at the target stops. Unfortunately, this approach does not work for source-switch-optimal journeys. While it would be possible to show an analogous result to Theorem 3.7 which involves separating the first trip from the remaining journey, this has no practical use for the forward-search version of the RAPTOR algorithm.

In order to represent all source-switch-optimal journey, we need to replace the single entries in the round table with Pareto sets. As with the target-switch-optimal case, we will first discuss a naive approach that maintains full Pareto sets and subsequently introduce optimizations. The naive approach is to maintain a Pareto set in each round table entry $R[k][t]$. When finding a new journey, it is compared to every journey in the Pareto set, which requires a routine for determining source-switch-dominance.

Given two $s$-$t$-journeys $J_1$ and $J_2$ for the departure time $\tau_{\mathrm{dep}}$ with $\leq k$ trips, the following steps need to be performed to determine if $J_1$ source-switch-dominates $J_2$:

- If $J_1$ does not dominate $J_2$, i.e., $\tau_{\mathrm{arr}}(J_1) \leq \tau_{\mathrm{arr}}(J_2)$ or $|J_1| \leq |J_2|$ does not hold, $J_1$ does not source-switch-dominate $J_2$.
- Assume now that $J_1$ dominates $J_2$. If $J_1$ starts with a footpath, passengers can enter $J_1$ from any trip tr that arrives at $s$ with an arrival time $\tau_{\mathrm{arr}}(\mathrm{tr}, t) \leq \tau_{\mathrm{dep}}$, regardless of $J_2$. Thus, $J_1$ source-switch-dominates $J_2$.
- If $J_1$ does not start with a footpath, let $\mathrm{tr}_1$ be the first trip of $J_1$. If $J_2$ starts with a footpath, passengers arriving via any trip tr that arrives at $s$ with an arrival time $\tau_{\mathrm{arr}}(\mathrm{tr}, t) \leq \tau_{\mathrm{dep}}$ can enter $J_2$, while they must obey the minimum change time to enter $J_1$ unless they arrived via $\mathrm{tr}_1$. $J_1$ source-switch-dominates $J_2$ if $\mathrm{tr}_1$ is the only arriving trip within the minimum departure time. To check this, find the latest arriving trip $\mathrm{tr}' \neq \mathrm{tr}_1$ with $\tau_{\mathrm{arr}}(\mathrm{tr}', s) \leq \tau_{\mathrm{dep}}$. If no such trip exists, $J_1$ source-switch-dominates $J_2$. Otherwise, $J_1$ source-switch-dominates $J_2$ if $\tau_{\mathrm{dep}}(J_1) \geq \tau_{\mathrm{arr}}(\mathrm{tr}', s) + \tau_{\mathrm{ch}}(s)$.
- If $J_2$ starts with $\mathrm{tr}_1$ at $s$, any arriving passengers that can enter $J_2$ can obviously enter $J_1$ as well, so $J_1$ source-switch-dominates $J_2$.

| $J_1$ | $J_2$ | |
|---|---|---|
| | Footpath | Trip $\mathrm{tr}_2$ |
| Footpath | $\tau_\mathrm{a}(J_1) \leq \tau_\mathrm{a}(J_2)$ | $\tau_\mathrm{a}(J_1) \leq \tau_\mathrm{a}(J_2)$ |
| Trip $\mathrm{tr}_1 = \mathrm{tr}_2$ | $\tau_\mathrm{a}(J_1) \leq \tau_\mathrm{a}(J_2) \wedge$ $\tau_\mathrm{d}(J_1) \geq \tau_\mathrm{a}(\mathrm{tr}, s) + \tau_\mathrm{c}(s)$ | $\tau_\mathrm{a}(J_1) \leq \tau_\mathrm{a}(J_2)$ |
| Trip $\mathrm{tr}_1 \neq \mathrm{tr}_2$ | $\tau_\mathrm{a}(J_1) \leq \tau_\mathrm{a}(J_2) \wedge$ $\tau_\mathrm{d}(J_1) \geq \tau_\mathrm{a}(\mathrm{tr}, s) + \tau_\mathrm{c}(s)$ | $\tau_\mathrm{a}(J_1) \leq \tau_\mathrm{a}(J_2) \wedge$ $\tau_\mathrm{d}(J_1) \geq \tau_\mathrm{a}(\mathrm{tr}, s) + \tau_\mathrm{c}(s)$ |

Table 3.2: Conditions for simplified source-switch-dominance. Each cell lists the conditions under which $J_1$ source-switch-dominates $J_2$, depending on the first leg of either journey. Due to space constraints, we abbreviate the arrival time $\tau_\mathrm{arr}$ as $\tau_\mathrm{a}$, the departure time $\tau_\mathrm{dep}$ as $\tau_\mathrm{d}$ and the minimum change time $\tau_\mathrm{ch}$ as $\tau_\mathrm{c}$.

- If $J_2$ starts with a trip $\mathrm{tr}_2 \neq \mathrm{tr}_1$ at $s$, arriving passengers arriving via trip must obey the minimum change time to enter either journey, except for those arriving via $\mathrm{tr}_1$ in the case of $J_1$ and $\mathrm{tr}_2$ in the case of $J_2$. $J_1$ source-switch-dominates $J_2$ if for each trip tr that arrives at $s$ with an arrival time $\tau_\mathrm{arr}(\mathrm{tr}, t) \leq \tau_\mathrm{dep}$, passengers arriving via tr that can enter $J_2$ can enter $J_1$ as well. Since passengers arriving via $\mathrm{tr}_1$ can always enter $J_1$, it does not need to be considered. If $\tau_\mathrm{arr}(\mathrm{tr}_2, s) \leq \tau_\mathrm{dep}$, then $\tau_\mathrm{dep}(J_1) \geq \tau_\mathrm{arr}(\mathrm{tr}_2, s) + \tau_\mathrm{ch}(s)$ must hold. Among all other trips, only those that arrive at $s$ before $\tau_\mathrm{dep}(J_2) - \tau_\mathrm{ch}(s)$ can enter $J_2$. Find the latest arriving trip $\mathrm{tr}' \notin \{\mathrm{tr}_1, \mathrm{tr}_2\}$ with $\tau_\mathrm{arr}(\mathrm{tr}', s) \leq \min(\tau_\mathrm{dep}, \tau_\mathrm{dep}(J_2) - \tau_\mathrm{ch}(s))$. If no such trip exists, $J_1$ source-switch-dominates $J_2$. Otherwise, $J_1$ source-switch-dominates $J_2$ if $\tau_\mathrm{dep}(J_1) \geq \tau_\mathrm{arr}(\mathrm{tr}', s) + \tau_\mathrm{ch}(s)$. It is not necessary to check the conditions for both $\mathrm{tr}'$ and $\mathrm{tr}_2$, only for the one with the later arrival time.

As in the target-switch-optimal case, some of the steps that need to be performed are too complicated, so we use a stricter version of source-switch-dominance that is easier to compute:

- The case for $J_1$ starting with a trip and $J_2$ starting with a footpath includes the step of finding the latest arriving trip $\mathrm{tr}' \neq \mathrm{tr}_1$ with $\tau_\mathrm{arr}(\mathrm{tr}', s) \leq \tau_\mathrm{dep}$. We simplify this slightly by omitting the condition $\mathrm{tr}' \neq \mathrm{tr}_1$. As a consequence, there is only one $\mathrm{tr}'$ for each departure time $\tau_\mathrm{dep}$. This allows us to compute $\mathrm{tr}'$ at the start of the algorithm, when the possible departure times are collected.

- The case for $J_2$ starting in a trip $\mathrm{tr}_2 \neq \mathrm{tr}_1$ requires us to find the latest arriving trip $\mathrm{tr}' \notin \{\mathrm{tr}_1, \mathrm{tr}_2\}$ with $\tau_\mathrm{arr}(\mathrm{tr}', s) \leq \min(\tau_\mathrm{dep}, \tau_\mathrm{dep}(J_2) - \tau_\mathrm{ch}(s))$. Instead, we use the same trip $\mathrm{tr}'$ that we precomputed for the previous case and simplify the condition to $\tau_\mathrm{arr}(\mathrm{tr}', s) \leq \tau_\mathrm{dep}$, which is a stricter condition.

Similarly to the target-switch-optimal case, we observe that $R[k][t]$ can include at most one journey $J$ that starts with a footpath, since any journey that is dominated by $J$ is also source-switch-dominated by it. This allows us to split $R[k][t]$ into two components: $R_f[k][t]$ is the only source-switch-optimal journey that starts with a footpath (or $\perp$ if none exists) and $R_c[k][t]$ is a Pareto set of source-switch-optimal journeys that start with trips, sorted by their arrival time at $t$. The conditions that need to be checked to determine if a journey $J_1$ source-switch-dominates another journey $J_2$ are now dependent on which components the two journeys belong to. Table 3.2 lists these conditions, using $\tau_\mathrm{dep}$ to denote the departure time of the current RAPTOR execution and tr to denote the latest arriving trip with $\tau_\mathrm{arr}(\mathrm{tr}, s) \leq \tau_\mathrm{dep}$.

When a new $s$-$t$-journey $J$ is found during phase $k$, the following steps need to be taken to update $R[k][t]$:

- Check if $R_f[k][t]$ or $R_c[k][t]$ contains a journey that source-switch-dominates $J$. If so, discard $J$.
- If $J$ is not source-switch-dominated, delete all entries that are dominated by it. In the case of $R_f[k][t]$, this means replacing it with $\perp$ if it is dominated.
- Add $J$ to its respective component, depending on whether it ends with a transfer or a trip. In the case of $R_f[k][t]$, the journey can simply be overwritten. In the case of $R_c[k][t]$, $J$ must be inserted at the right index so that it is still sorted by arrival time.

As in the target-switch-optimal case, we can exploit the fact that $R_c[k][t]$ is sorted by arrival time to implement the dominance checks more efficiently:

- A necessary condition for any journey $J_2$ source-switch-dominating $J_1$ is $\tau_{\text{arr}}(J_2) \leq \tau_{\text{arr}}(J_1)$. If $J_1$ and $J_2$ both start with the same trip, this condition is also sufficient. Otherwise, the additional condition $\tau_{\text{dep}}(J_2) \geq \tau_{\text{arr}}(\text{tr}, s) + \tau_{\text{ch}}(s)$ must hold. We can thus check if $J_1$ is source-switch-dominated by iterating over $R_c[k][t]$ in ascending order of arrival time until we exceed $\tau_{\text{arr}}(J_1)$. For each journey $J_2$ that is found, we check if $\text{tr}_1 = \text{tr}_2$ or $\tau_{\text{dep}}(J_2) \geq \tau_{\text{arr}}(\text{tr}, s) + \tau_{\text{ch}}(s)$.
- A necessary condition for $J_1$ source-switch-dominating any journey $J_2$ is $\tau_{\text{arr}}(J_1) \leq \tau_{\text{arr}}(J_2)$. This is also sufficient unless $J_1$ and $J_2$ start with different trip, in which case $\tau_{\text{dep}}(J_1) \geq \tau_{\text{arr}}(\text{tr}, s) + \tau_{\text{ch}}(s)$. Note that this condition is independent of $J_2$. We can thus find source-switch-dominated journeys by iterating over $R_c[k][t]$ in descending order of arrival time until we reach below $\tau_{\text{arr}}(J_1)$. For each journey $J_2$ that is found, we delete $J_2$ if $\tau_{\text{dep}}(J_1) \geq \tau_{\text{arr}}(\text{tr}, s) + \tau_{\text{ch}}(s)$ or $\text{tr}_1 = \text{tr}_2$.

### 3.4.1 Range Queries

So far we have discussed how target-switch-optimal journeys can be found by maintaining an entire Pareto set in every round table entry. Unfortunately, this makes journey retrieval more complicated: In the normal RAPTOR algorithm, it is sufficient for each round table entry $R[u][k]$ to maintain a reference to the parent stop $v$, which allows us to find the parent entry $R[v][k]$. If the round table entries are Pareto sets representing multiple journeys, we additionally need to identify the correct parent journey in Pareto set of the parent entry. This requires us to index the individual journeys in the Pareto set. For the fixed departure version of RAPTOR, this is fairly easy since the round table entries for round $k$ are not changed anymore after the round has finished. This allows us assign indices to the journeys afterwards and reference them in the next round.

However, this does not work for the Range Problem and the rRAPTOR algorithm. Here the round table is reused across multiple RAPTOR execution. During round $k$, journeys in the round table for round $k$ may already referenced as parents by journeys found in earlier iterations. If journeys are added to or removed from a round table entry, the indexing may become inconsistent and may need to be updated.

We avoid these problems by taking a different approach: rRAPTOR already performs one RAPTOR execution for each possible departure time. We divide these executions further so that each execution can find at most one source-switch-optimal journey per round table entry $R[k][t]$.

For a given departure time $\tau_{\text{dep}}$, a target $t$ and a RAPTOR phase $k$, the following is true about $R[k][t]$:

- As already mentioned, $R[k][t]$ can include at most one journey $J$ with departure time $\tau_{\text{dep}}$ that ends with a footpath. Therefore, we can group all departures via a footpath at $\tau_{\text{dep}}$ into a single RAPTOR execution.
- If there are multiple trips that depart from $s$ at $\tau_{\text{dep}}$, it is possible that each of them contributes a source-switch-optimal journey to $R[k][t]$. Consider two journeys $J_1$ and $J_2$ with initial trips $\text{tr}_1$ and $\text{tr}_2$ that do not begin at $s$ and with $\tau_{\text{dep}}(\text{tr}_1, s) = \tau_{\text{dep}}(\text{tr}_2, s) = \tau_{\text{dep}}$. This implies that $\tau_{\text{arr}}(\text{tr}_1, s) \leq \tau_{\text{dep}}$ and $\tau_{\text{arr}}(\text{tr}_2, s)$. If $\tau_{\text{arr}}(\text{tr}_1, s) + \tau_{\text{ch}}(s) > \tau_{\text{dep}}$, $J_2$ cannot be entered by passengers arriving via $\text{tr}_1$, so $J_1$ cannot source-switch-dominate $J_2$. Likewise, if $\tau_{\text{arr}}(\text{tr}_2, s) + \tau_{\text{ch}}(s) > \tau_{\text{dep}}$, $J_1$ cannot be entered by passengers arriving via $\text{tr}_2$, so $J_2$ cannot source-switch-dominate $J_1$. If both are true, both journeys must be represented in $R[k][t]$. To achieve this, we perform a separate RAPTOR execution for each trip that departs at $s$ at $\tau_{\text{dep}}$.

For each departure time $\tau_{\text{dep}}$, we therefore perform one RAPTOR execution for all departures via footpath, and one execution for each departure via trip. To prevent the algorithm from finding journeys that should be found in a different RAPTOR execution, we add the following restriction: During the route phase of round 1, do not enter any trip tr at $s$ unless the current RAPTOR execution is specifically dedicated to tr. This means that each trip departing at $s$ is scanned in only one RAPTOR execution and the resulting journeys are added to the Pareto sets only in that execution.

Because each RAPTOR execution can now only contribute one source-switch-optimal journey per round table entry $R[k][t]$, we do not have to store Pareto sets in that entry and can instead store the single source-switch-optimal entry. However, we still need to maintain journeys from previous RAPTOR executions to determine if found journeys are source-switch-optimal. We keep these journeys in a Pareto set $\mathcal{P}[k][t]$. Note that we do not have to retain all source-switch-optimal solutions, but only those that can potentially source-switch-dominate solutions found in later executions.

A journey $J_1$ source-switch-dominates another journey $J_2$ if it fulfills two conditions: its arrival time may not be worse ($\tau_{\text{arr}}(J_1) \leq \tau_{\text{arr}}(J_2)$) and arriving passengers who are able to enter $J_2$ must be able to enter $J_1$. If $J_1$ starts with a footpath or both journeys start with the same trip $\text{tr}_1$, the latter condition is automatically fulfilled. Otherwise, $\tau_{\text{dep}}(J_1) \geq \tau_{\text{arr}}(\text{tr}, s) + \tau_{\text{ch}}(s)$ must be fulfilled. Due to the way the RAPTOR executions are separated, the second condition can already be evaluated once the RAPTOR execution that finds $J_2$ has started, even if $J_2$ has not been found yet. This allows us to determine at the start of the RAPTOR execution which journeys from $\mathcal{P}[k][t]$ fulfill the second condition and can therefore potentially dominate journeys found during the RAPTOR execution. Note that journeys from different RAPTOR executions can never start with the same trip at $s$, so we can omit the check if both trips start with the same trip.

At the start of each RAPTOR execution, we initialize $R[k][t]$ by selecting all journeys from $\mathcal{P}[k][t]$ that fulfill the second condition. Among these, we select a journey $J$ with minimal arrival time and initialize $R[k][t]$ with it. A journey found during this RAPTOR execution is source-switch-dominated by $\mathcal{P}[k][t]$ if and only if it is dominated by $J$. This means that we can use regular dominance when determining whether $R[k][t]$ must be updated, just as in the regular RAPTOR algorithm. Whenever a new journey $J$ is written into $R[k][t]$, we know that it is source-switch-optimal and therefore has to be inserted into $\mathcal{P}[k][t]$. Additionally, we must remove all journeys from $\mathcal{P}[k][t]$ that are dominated by $J$.

As mentioned earlier, it is not necessary to keep all source-switch-optimal journeys in $\mathcal{P}[k][t]$, but only those that we may need to source-switch-dominate journeys found in future RAPTOR executions. When initializing $R[k][t]$, we select a journey with minimal arrival time among those in $\mathcal{P}[k][t]$ that can possibly source-switch-dominate journeys found during the current RAPTOR execution. If we can find a set of journeys which will fulfill this condition for all future RAPTOR executions, we only have to retain the journey with minimal arrival time among this set.

A Pareto set $\mathcal{P}[k][t]$ consists of one source-switch-optimal journey that starts with a footpath, $R_f[k][t]$, as well as a set of source-switch-optimal journeys that end with a trip, $R_c[k][t]$. The journey in $R_f[k][t]$ can be entered from any earlier arrival at $s$ and therefore always fulfills the condition. A journey $J \in R_c[k][t]$ fulfills the condition if $\tau_{\text{dep}}(J) \geq \tau_{\text{arr}}(\text{tr}, s) + \tau_{\text{ch}}(s)$ holds. Since tr is the latest arriving trip at $s$ for which $\tau_{\text{arr}}(\text{tr}, s) \leq \tau_{\text{dep}}$ holds, and since $\tau_{\text{dep}}$ cannot decrease between RAPTOR executions, we know that $\tau_{\text{arr}}(\text{tr}, s)$ also cannot decrease. Therefore, once a journey fulfills the condition, it will continue to fulfill it for all future RAPTOR executions.

At the start of each RAPTOR execution, we check if $\tau_{\text{arr}}(\text{tr}, s)$ has increased. If so, then for each Pareto set $R_c[k][t]$, we delete all journeys $J$ with $\tau_{\text{dep}}(J) \geq \tau_{\text{arr}}(\text{tr}, s) + \tau_{\text{ch}}(s)$. Let $J'$ be the journey with minimal arrival time among the deleted journeys. We set $R_f[k][t]$ to the minimum of $R_f[k][t]$ and $\tau_{\text{arr}}(J')$. This means that $R_f[k][t]$ now contains the journey with minimal arrival time among all those that can potentially source-switch-dominate journeys found in future RAPTOR executions.

## 3.5 Computing Switch-Optimal Journeys

Since switch-optimality is the combination of source-switch-optimality and target-switch-optimality, we can make a similar observation as in the target-switch-optimal case: A full Pareto set of switch-optimal journeys can be found by taking journeys found by a RAPTOR algorithm for source-switch-optimal journeys and extending them by at most one trip. This means we can compute source-switch-optimal journeys and maintain an additional Pareto set $\mathcal{P}'[k][t]$ per potential target stop $t$ and number of trips $k$ which contains switch-optimal journeys.

The Pareto set $\mathcal{P}'[k][t]$ can be maintained in a straightforward way: Whenever a new journey $J$ is found, it is compared to all journeys in $\mathcal{P}'[k][t]$. If $J$ is switch-dominated, it is discarded. Otherwise, it is inserted into $\mathcal{P}'[k][t]$ and all journeys that are switch-dominated by it are removed. To determine switch-dominance, we use the simplified versions of source- and target-switch-dominance as given by Tables 3.2 and 3.1.

# 4. Speeding Up the Transfer Phase

The approach for public transit routing with unrestricted walking discussed in Chapter 2.5 involves adding a transfer phase to each round of RAPTOR's algorithm in which the entire footpath graph is explored via a Dijkstra search. The problem with this approach is that the footpath graph is typically much larger than the public transit network, and therefore this transfer phase slows down the algorithm considerably and dominates the overall running time. This can be mitigated somewhat by contracting the footpath graph, but the performance remains significantly worse than that of algorithms that restrict walking. The goal of this chapter is therefore to develop a preprocessing technique that reduces the running time of the transfer phase during queries, ideally to the point where it is not much larger than the running time of the route phase, as is the case in algorithms with restricted walking.

As discussed in the introduction, the experiments conducted by Wagner and Zündorf which we recreated in Figure 1.1 show that unrestricted walking is frequently necessary for finding optimal journeys. When investigating the optimal journeys that involve long footpaths, it becomes apparent that most of these long footpaths occur as the first or last leg of the journeys. Optimal journeys requiring footpaths between two trip segments that are longer than a few minutes are rare, but do occur occasionally.

These findings motivate the following approach: Since long walking legs before the first trip and after the last trip are fairly common, footpaths for the first and last leg of a journey are explored on the full, unrestricted footpath graph $\mathcal{F}$. For intermediate walking legs connecting two trip segments, we use a restricted footpath graph $\mathcal{F}' = (\mathcal{S}, E')$. For each optimal journey that includes an intermediate walking leg connecting two stops $u$ and $v$, $\mathcal{F}'$ contains a shortcut edge $(u, v)$ that represents that walking leg. This allows us to drop the Dijkstra search and return to simply relaxing the outgoing edges of reached stops as in the original RAPTOR algorithm, which required a transitively closed footpath graph. While $\mathcal{F}'$ is not necessarily transitively closed, all necessary footpaths are represented by direct edges in $\mathcal{F}'$ and therefore can be found by only relaxing the outgoing edges.

This approach requires a preprocessing phase that identifies all needed intermediate footpaths and build the restricted footpath graph $\mathcal{F}'$ from them. To ensure correct query results, $\mathcal{F}'$ must represent all intermediate footpaths that are part of at least one optimal journey. In order to limit the preprocessing time, we may also include edges in $\mathcal{F}'$ that are not needed for any optimal journey. This does not impact the correctness of the approach, but including too many unnecessary edges will lead to worse query times.

## 4.1 Query Algorithm

Assuming we have precomputed a restricted footpath graph $\mathcal{F}'$, we can solve a public transit query for a given departure time $\tau_{\text{dep}}$ and source and target stops $s, t \in \mathcal{S}$ with a slightly modified RAPTOR algorithm:

Before starting the actual RAPTOR search, we perform a forward Dijkstra search from $s$ in the full footpath graph $\mathcal{F}$. Let $S$ be the set of stops reached by this search. For each stop $v \in S$, we store the walking time $\tau_w(P_{sv})$ from $s$ so that it can be used by the RAPTOR search. Similarly, we perform a backward Dijkstra search from $t$ in $\mathcal{F}$. Let $T$ be the set of stops reached by this search. For each stop $v \in T$, we store the walking time $\tau_w(P_{vt})$ from $t$.

We then perform a RAPTOR search with restricted walking, using $\mathcal{F}'$ as the footpath graph. To incorporate the unrestricted initial and final walking legs, we make the following changes:

- During the initial transfer phase, instead of relaxing the edges in $\mathcal{F}'$, we call ARRIVALBYTRANSFER for each stop $v \in S$ with the arrival time $\tau_{\text{dep}} + \tau_w(P_{sv})$.
- During the transfer phase in round $k$, when scanning a stop $v$ with arrival time $R[2k][v]$, we relax all outgoing edges from $v$ in $\mathcal{F}'$ as usual. Additionally, if $v \in T$, we call ARRIVALBYTRANSFER for $t$ with the arrival time $R[2k][v] + \tau_w(P_{vt})$.

To solve the Range Problem, we can use rRAPTOR or the profile RAPTOR algorithm presented in Chapter 2.6.2, using this modified RAPTOR algorithm for the individual RAPTOR executions. As with the initial transfer phase in rRAPTOR, the initial Dijkstra searches from $s$ and $t$ only have to be performed once at the beginning of the algorithm, as the results of the search can be reused for each individual RAPTOR execution.

## 4.2 Preprocessing Phase

A naive approach to precomputing $\mathcal{F}'$ would involve solving the all-to-all Range Problem for a departure time interval $I$ that includes all departure times that are possible in the public transit network. For each optimal journey $J$ found by this search and each intermediate walking leg between stops $u$ and $v$ in $J$, we could then insert an edge $e = (u, v)$ with weight $\tau_w(e) = \tau_w(P_{uv})$ into $\mathcal{F}'$ that represents the walking leg.

In fact, it is not necessary to consider and unpack every optimal journey. As proven by Theorem 3.2, every optimal journey can be constructed out of optimal proper subjourneys. The smallest proper subjourneys that still include an intermediate walking leg consist of two trip segments connected by a walking leg, with no initial or final walking leg. If we precompute all journeys of this type, they will include all intermediate walking legs that are part of any optimal journey.

This motivates the following approach: We initially contract the footpath graph $\mathcal{F}$ to speed up the RAPTOR queries that are performed during the preprocessing phase. We then perform a one-to-all rRAPTOR query from each stop $s \in \mathcal{S}$. For each stop $t \in \mathcal{S}$ reached by such a query, we extract the full Pareto set $\mathcal{P}_{st}$ of optimal $s$-$t$-journeys. We then identify all journeys in $\mathcal{P}_{st}$ that consist of three legs, with the first and third leg being trip segments and the second being a walking leg, i.e., all journeys of the form $J = ((\text{tr}_1, s, u), (\text{tr}_2, v, t))$ with $u \neq v$. For each such journey $J$, we insert a shortcut edge $e = (u, v)$ into $\mathcal{F}'$ with the weight $\tau_w(e) = \tau_w(P_{uv})$, if it does not already exist. The individual one-to-all queries are independent of each other and can therefore be parallelized.

This approach still solves the all-to-all Range Problem, which is expensive, but it no longer unpacks all the optimal journeys. It is sufficient to unpack journeys from the round table entries in $R[4]$, which are guaranteed to include exactly two trip segments and no final walking leg. However, in order to decide if the journeys have an intermediate walking leg and no initial walking leg, and in order to compute the walking time of the intermediate walking leg, it is still necessary to unpack the journeys. Unpacking journeys after the rRAPTOR query has finished and the full profile has been constructed is difficult, since the round table at the end of the query only includes entries that are optimal for the last scanned departure time. Round table entries that are optimal for later departure times may have already been dominated and removed from the round table. This could be solved by maintaining profiles that contain all optimal entries that were ever entered into the round table during the rRAPTOR query, but it is much easier and more convenient to unpack optimal journeys “on-the-fly”, as soon as they are found by the rRAPTOR algorithm. Whether a journey is optimal or not depends only on the journeys already found for later departure times, so we can determine whether a journey is optimal or not immediately when it is found. If it is optimal, we can unpack it using the round table entries from the current RAPTOR execution.

In this thesis, we are only interested in solving queries for departure times in the interval $I = [0h, 24h]$. It would therefore seem natural to restrict our preprocessing rRAPTOR query to this departure interval as well. However, this leads to problems if our preprocessing phase does not compute full journeys, but only proper subjourneys of optimal journeys. An optimal journey that begins before $24h$ may well include proper subjourneys that begin after $24h$ and thus lie outside of our departure interval. As outlined in Chapter 2.1, our public transit network includes a copy of every trip that is shifted forward by 24 hours. Therefore, if a journey that begins after $24h$ only includes trips that themselves begin after $24h$, the network also includes an analogous journey that takes place 24 hours earlier and thus lies inside our departure interval.

Unfortunately, this does not account for trips that start before $24h$ but include stop events after $24h$. If a journey enters one of these trips at a stop where it departs after $24h$, there will be no corresponding copy of the trip 24 hours earlier. To remedy this, we identify such trips and insert copies of them into the network that are shifted backward by 24 hours. This will lead to some stop events with departure and arrival times before $0h$, but these will be ignored by our algorithm because they lie outside the departure interval.

**Approximations.** So far we have discussed how to compute an exact solution, in which $\mathcal{F}$ contains only shortcuts that are needed to construct an optimal journey. The following optimizations can be made in order to improve the preprocessing time, at the expense of computing an overapproximation of $\mathcal{F}$ that contains unnecessary footpaths:

The first optimization is to limit the number of rounds performed by the RAPTOR queries to some value $k \geq 2$. The journeys we are interested in contain exactly two trip segments, so at least two rounds need to be performed in order to find them. Any optimal journeys with more than $k$ trips will not be found. For the Bicriteria Problem, we can safely set $k = 2$ while still computing an exact solution, since journeys with two trips cannot be dominated by journeys with more than two trips. For the Earliest Arrival Problem, however, this may cause us to misidentify suboptimal journeys as optimal if they are only dominated by journeys with more than $k$ trips. In this case, unnecessary shortcuts will be inserted for the walking legs in these journeys.

The second optimization is to limit walking distances in the initial transfer phase to some value $x$, e.g., 15 minutes. This improves the performance of the rRAPTOR algorithm by limiting the amount of stops that can be reached from $s$ with initial walking, and thereby the number of possible departures times that are found and scanned. However, this also means that the algorithm will not find optimal journeys that start with walking legs that are longer than $x$. Since the journeys we are interested in do not include initial walking legs, they will all still be found. However, we may again misidentify suboptimal journeys as optimal and inserted unnecessary shortcuts for them if they are only dominated by journeys with initial walking legs that are longer than $x$.

Note that the main performance improvement comes not from shortening the Dijkstra search in the initial transfer phase, but from reducing the number of scanned departure times. Instead of pruning the initial Dijkstra search at distance $x$, we can let it finish and simply not add stops with distances above $x$ to the set $S$ of reached stops. This allows us to compute optimal pure walking journeys to all stops in the network, which makes them available for dominating journeys that the algorithm might otherwise consider optimal.

**Optimization for Earliest Arrival Problem.** rRAPTOR collects departure times for all stops that can be reached by direct walking. When using an unrestricted footpath graph, this typically includes all (or almost all) stops. As a consequence, the algorithm collects departures based on initial walking legs that are clearly unrealistic, e.g., taking a three-day walk to another city and then entering a trip there. When using bicriteria optimality, these departures are nevertheless necessary because pure walking journeys can never be dominated by journeys using at least one trip, and therefore the pure walking journeys found by the initial transfer phase are always Pareto-optimal even if they are extremely long.

When using earliest arrival optimality, however, extremely long initial walking legs are likely to be dominated by journeys that use trips. Let $v \neq s$ be a stop reached by initial walking from the source stop $s$ and let $\tau_{\text{dep}}$ be the departure time of a trip at $v$. The rRAPTOR algorithm will then include $\tau_{\text{dep}} - \tau_w(P_{sv})$ as a departure time at $s$ that must be scanned. However, this is not necessary if there is an $s$-$v$-journey $J$ with the following properties:

- If $J$ ends with a footpath, $\tau_{\text{arr}}(J) \leq \tau_{\text{dep}}$.
- If $J$ ends with a trip, $\tau_{\text{arr}}(J) + \tau_{\text{ch}}(v) \leq \tau_{\text{dep}}$.

In this case, any journey that begins with direct walking to $v$ will be dominated by the corresponding journey that replaces the initial walking leg with $J$.

We can exploit this fact to prune the set of scanned departure times: For a scanned departure time $\tau_{\text{dep}}$, let $S'$ be the set of stops $v$ that are reachable from $s$ and are visited by a trip tr with the departure time $\tau_{\text{dep}} + \tau_w(P_{sv})$. These are the stops that will be inserted into the set of updated stops $U_t$ when $\tau_{\text{dep}}$ is scanned. Before starting the RAPTOR execution for $\tau_{\text{dep}}$, we check for each stop $v \in S'$ if the algorithm has already found an $s$-$v$-journey with the properties described above by searching the round table entries for $v$. If so, we do not insert $v$ into $U_t$. If we found such a journey for every stop in $S'$, we can skip the RAPTOR execution for $\tau_{\text{dep}}$ entirely.

# 5. Speeding Up the Route Phase

The technique for precomputing intermediate footpaths introduced in Chapter 4 reduces the impact of RAPTOR's transfer phase on the overall running time, bringing it closer to that of restricted walking approaches. However, even with restricted walking, the RAPTOR algorithm is too slow on country-sized networks to allow for truly interactive queries, especially for the Range Problem. While speedup techniques have been developed for RAPTOR and other public transit algorithms, they were typically designed for restricted walking scenarios. Applying them directly to unrestricted walking is often either impossible because it conflicts with their design or causes them to lose their efficiency. This chapter is therefore aimed at adapting existing speedup techniques to unrestricted walking in order to speed up the entire algorithm, including the route scans.

## 5.1 Overview of Preprocessing Techniques

This section gives an overview of some speedup techniques that are candidates for being adapted to public transit routing with unrestricted walking.

One possible approach is to model the public transit network as a time-dependent graph and then directly use preprocessing techniques that work on time-dependent graphs. The time-dependent graph model for public transit networks uses vertices to represent stops and edges to represent the public transit connections between stops. The edge weights are time-dependent piecewise linear functions, which have a similar form to the profiles introduced in Chapter 2.3. This model can easily incorporate footpaths by simply modeling them as edges with a constant weight. While previous research suggests that direct adaptations of speedup techniques to the graph-based model typically offer smaller speedups than for road networks [BDGM09] [BDG$^+$16], they have often only been evaluated for restricted walking.

A speedup technique that was already adapted to graph-based public transit routing with restricted walking is Contraction Hierarchies (CH) [Gei08], which uses the contraction technique described in Chapter 2.5.1 to augment the graph with a hierarchy of shortcut edges. CH was adapted to time-dependent road networks in [BDSV09]. Wirth adapted it further to public transit routing with restricted walking [Wir15] and suggested that allowing unrestricted walking would lead to too many shortcuts which would exceed reasonable space limitations. We conducted preliminary experiments that support this finding.

Another candidate for adaptation to graph-based public transit models is Customizable Route Planning (CRP) [DGPW11], which has also been extended to time-dependent edge weights [BDPW16]. CRP is based on recursively partitioning the graph into multiple cells to create a hierarchical multi-level partition. In each cell, shortcuts between all boundary vertices are precomputed that can be used to quickly bypass the cell during queries. As with CH, we conducted preliminary experiments to determine whether adapting CRP to graph-based public transit networks is feasible. We observed that the number of boundary vertices is significantly higher than in road networks, possibly due to the weaker natural hierarchy in public transit networks. This leads to a higher number of shortcuts per cell. Combined with the high complexity of the edge weight functions, which can increase further when linked together to represent longer paths, this leads to very high memory consumption on the higher levels of the partition, making the approach infeasible in practice.

An alternative to graph-based approaches is to consider speedup techniques that were developed specifically for public transit routing with restricted walking and adapt them to unrestricted walking. One such technique is HypRAPTOR, a speedup technique for RAPTOR based on hypergraph partitioning [DDPZ17]. Instead of partitioning the stops, this approach partitions the routes of the network, which is more suited to RAPTOR's trip-based network model. This is done by constructing a hypergraph where the vertices represent routes and hyperedges connect routes that share stops. This hypergraph is then partitioned with a hypergraph partitioning tool.

An $s$-$t$-query can be restricted mostly to the cells containing $s$ and $t$. While routes from other cells may be needed, especially if the two cells do not overlap, these are expected to be fairly few and can be identified in a preprocessing phase. The set of all routes that are used in optimal journeys where they are part of neither the source cell nor the target cell is called the *fill-in*. The precomputation of the fill-in is based on the observation that entering and exiting the fill-in must always occur via a trip switch at a cut stop, i.e., a stop that is visited by routes from different cells. This is no longer true when allowing unrestricted walking, since the initial (or final) walking leg may lead outside of the source (or target) cell and therefore enter the fill-in without a trip switch. Therefore, the hypergraph partitioning approach cannot be adapted in a straightforward manner.

In the following, we will describe and evaluate two feasible approaches in more detail: The first is an adaptation of Arc-Flags, an established preprocessing technique for the shortest path problem that partitions the graph into cells and flags edges that are part of an optimal path into a cell. The second is an adaptation of Connection Scan Accelerated, a speedup technique for CSA that is in itself an adaptation of Customizable Route Planning to CSA's network model.

## 5.2 Adapting Arc-Flags

This section discusses how to adapt the Arc-Flags algorithm to RAPTOR's trip-based public transit model. First we outline the original algorithm and then we discuss our adaptation, which we call *Route-Flags*.

### 5.2.1 Arc-Flags

The Arc-Flags algorithm [HKMS09] is a preprocessing technique for the shortest problem on a weighted graph $G = (V, E)$. It is based on a *partition* of $V$ into a disjoint set of cells $\mathcal{Z} = \{z_1, \dots, z_k\}$. For a vertex $v \in V$, we denote the cell that $v$ belongs to by $z(v)$. We call a vertex $u$ a *boundary vertex* of $z(u)$ if $G$ contains an edge $(u, v)$ with $z(v) \neq z(u)$.

For each edge $e \in E$ and each cell $z \in \mathcal{Z}$, the algorithm maintains a *flag* $f_z(e)$, which is a boolean value that indicates whether $e$ is part of any optimal path whose target vertex lies in $z$. During an $s$-$t$-query, it is therefore only necessary to relax an edge $e$ if $f_{z(t)}(e)$ is set to true.

A naive approach to computing the flags would be to solve the all-to-all shortest path problem, i.e., compute a shortest path $P = (e_1, \dots, e_k)$ for each pair of source and target vertex $s, t \in V$. For $1 \leq i \leq k$, the flag $f_{z(t)}(e_i)$ could then be set to true. However, it is not necessary to compute shortest paths between all pairs of vertices. Note that every path that starts outside a cell $z$ and ends inside it must at one point visit a boundary vertex of $z$. Therefore, it is sufficient to only consider boundary vertices as target vertices.

For each cell $z$, we perform a backward Dijkstra search from each boundary vertex $v$ of $z$. This results in a backward shortest path tree rooted in $v$. For each edge $e$ in this tree, we set $f_z(e)$ to true. Since these backward searches are independent of each other, they can be parallelized. Additionally, for every intra-cell edge in the graph, i.e., every edge $e = (u, v)$ with $z(u) = z(v)$, we set $f_{z(u)}(e)$ to true. Unless $e$ is dominated by a $u$-$v$-path with more than one edge (in which case it is superfluous), $e$ constitutes the shortest $u$-$v$-path.

Overall, the Arc-Flags algorithm can achieve a high speedup of more than a factor of 1 000 on country-size graph, but at the cost of fairly high preprocessing time and memory usage. Furthermore, intra-cell queries are not sped up at all, since all intra-cell edges are flagged. To remedy this, Arc-Flags can be combined with contraction into a multi-level partitioning approach called SHARC [BD09].

### 5.2.2 Route-Flags

In order to partition the public transit network, we must transform it into a graph. We construct an unweighted partitioning graph $G_P = (V, E_P)$ that includes the entire footpath graph $\mathcal{F}$ as well as edges representing the public transit connections between stops. For each trip $\mathrm{tr} = (v_1, \dots, v_k)$ and $1 \leq i < k$, we add an edge $(v_i, v_{i+1})$ to $G_P$ that represents the elementary connection between $v_i$ and $v_{i+1}$. Multiple edges between any pair of stops are merged into a single edge. We then partition this graph using the Inertial Flow [SS15] partitioning algorithm, which exploits the geographical embedding of the graph and is known to produce good results on road and footpath graphs while having a fairly low running time.

The original Arc-Flags algorithm performs backward searches from all boundary vertices in order to precompute the edge flags, which are then used to speed up forward queries. A backward query would require edge flags that were computed via forward searches from the boundary vertices. In our public transit scenario, the profile RAPTOR algorithm presented in Chapter 2.6.2 requires both forward and backward searches. In order to adapt Arc-Flags to it, we would therefore need two sets of flags: *forward flags* that were computed via backward searches and speed up forward queries, and *backward flags* that were computed via forward searches and speed up backward queries. This is not necessary for rRAPTOR, which only uses forward searches. For ease of exposition, we will only discuss how to compute backward flags via forward searches; forward flags can be computed in an analogous manner.

The backward search from the boundary vertices in the original Arc-Flags algorithm relies on the fact that subpaths of optimal paths are optimal themselves. The equivalent of a subpath in public transit networks is a subjourney. Since a journey may cross cell boundaries midway through a trip or a footpath, subjourneys that are cut off at cell boundaries may be improper subjourneys. As shown in Chapter 3.2, improper subjourneys of optimal journeys may not be optimal. In order to find all subjourneys that may be part of an optimal journey, we must compute all switch-optimal journeys. This requires one-to-all rRAPTOR queries in the interval $[0h, 24h]$. In order to find journeys involving trips that begin before $24h$ but include stop events after $24h$, we insert copies of these trips that are shifted backward by 24 hours, as discussed in Chapter 4.2. The flags found for

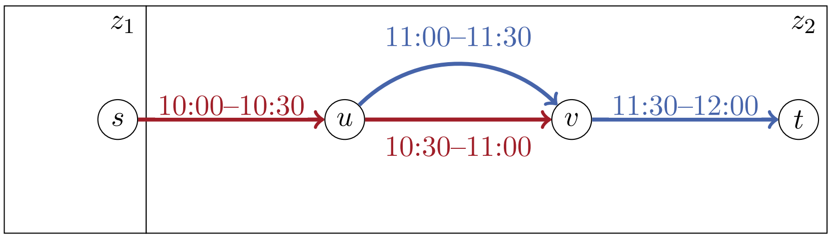


Figure 5.1: In this example, a forward profile search from $s$ to $t$ would flag the stop event for the trip departing at 10:30 at $u$, but not the one for the trip departing at 11:00. When performing a backward query with the target stop $t$ for the arrival time 12:00, the latest arriving trip at $v$ found by an unmodified RAPTOR query would be the one departing at 11:00 and arriving at 11:30, which is not flagged for the cell $z(s) = z_1$.

trips in the interval $[0h, 24h]$ can then be copied to the corresponding trips in $[24h, 48h]$ at the end of the preprocessing phase.

In our case, we need to compute switch-optimal journeys from each boundary vertex $u$ of each cell $z$ to every stop $t \in \mathcal{S}$ in the network. Note that $u$ may not be a stop. So far we have only discussed how to solve public transit queries between stops, but the algorithms presented in Chapter 2 can be easily adapted to handle queries between vertices as well. The $u$-$t$-journeys that we compute need to be sufficient to construct optimal journeys from any other vertex $v \in z$ to $t$. Since the target stops remains the same, the journeys we compute are suffixes of the optimal journeys. Because the optimal journeys end at $t$, we do not need to consider target-switch-dominance there; it suffices to compute source-switch-optimal journeys. Furthermore, if $u$ is a stop, we only need to consider arrivals from inside $z$ when determining source-switch-dominance (see Definition 3.4).

What remains to be discussed is how the flags themselves are adapted to public transit connections. The footpath edges of a found journey can be flagged in the same way as in the original Arc-Flags algorithm. For the trips, we flag individual stop events. However, it is not necessary to flag every stop event that is part of a found journey. We only flag stop events directly adjacent to a transfer: Consider a forward search from a boundary vertex $v$ of cell $z$ that finds a source-switch-optimal journey $J$. For each transfer from $\mathrm{tr}_1$ at $u$ to $\mathrm{tr}_2$ at $v$ in $J$, the stop events directly adjacent to the transfer are $e_1 = (\mathrm{tr}_1, u, \tau_{\mathrm{arr}}(\mathrm{tr}_1, u), \tau_{\mathrm{dep}}(\mathrm{tr}_1, u))$ and $e_2 = (\mathrm{tr}_2, v, \tau_{\mathrm{arr}}(\mathrm{tr}_1, u), \tau_{\mathrm{dep}}(\mathrm{tr}_1, u))$. We set $f_z(e_1)$ and $f_z(e_2)$ to true.

For a backward $s$-$t$ query, we use the precomputed flags as follows: During the transfer phase, we only relax an edge $e$ if $f_{z(s)}(e)$ is set to true. When collecting the routes serving updated stops, we only collect a route $r$ visiting a stop $v$ whose arrival time is $\tau_{\mathrm{arr}}(v)$ if $r$ contains at least one trip tr with $\tau_{\mathrm{dep}}(\mathrm{tr}, v) \geq \tau_{\mathrm{arr}}(v)$ whose corresponding stop event $e = (\mathrm{tr}, v, \tau_{\mathrm{arr}}(\mathrm{tr}, v), \tau_{\mathrm{dep}}(\mathrm{tr}, v))$ at $v$ is flagged, i.e., and $f_{z(s)}(e)$ is set to true. During the route phase, we only enter a trip tr if its corresponding stop event $e = (\mathrm{tr}, v, \tau_{\mathrm{arr}}(\mathrm{tr}, v), \tau_{\mathrm{dep}}(\mathrm{tr}, v))$ at $v$ is flagged, i.e., $f_{z(s)}(e)$ is set to true. This can be achieved by skipping unflagged stop events when calling the function LT that determines the latest trip that can be exited. If a stop event is not flagged for $z(s)$, that means there is no optimal $s$-$t$-journey that enters or exits the associated trip at the associated stop, and therefore it can be ignored.

Note that it is not sufficient to simply call LT without skipping unflagged stop events and then only scan the trip that is found if its stop event is flagged. Consider the example in Figure 5.1. A forward profile search from the boundary vertex $s$ to $t$ would only find the journey using the trip departing at 10:30 at $u$. The trip departing at 11:00 would therefore remain unflagged for $z(s)$. However, when performing a backward query with the source stop $s$ and target stop $t$ for the arrival time 12:00, the latest arriving trip found by an unmodified LT would be the one departing at 11:00 at $u$ and arriving at 11:30 at $v$. Since the stop event of this trip at $v$ is unflagged, the route scan would abort here and no journey would be found, even though an earlier trip exists that is flagged and could be scanned.

When implementing our Route-Flags algorithm, an important detail is how the flags for the stop events and edges are stored. A simple way to store them is as a nested array $f[\cdot][\cdot]$ of boolean values. The flag data can be laid out in two ways: For a given cell $z$ and edge $e$, the flag $f_z(e)$ can be stored either as $f[z][e]$ or as $f[e][z]$. In the former layout, flags belonging to the same cell are stored consecutively, while flags belonging to the same edge are scattered across the array. The latter layout inverts this; flags belonging to the same edge are stored consecutively, while flags belonging to the same cell are scattered. We use the former layout since it is more cache-efficient: During an $s$-$t$-query, only the flags belonging to $z(s)$ are relevant. It therefore makes sense to group flags belonging to the same cell together so that they have a greater chance of being cached together.

## 5.3 Adapting Connection Scan Accelerated

In this section, we describe how to adapt the ideas behind Connection Scan Accelerated (ACSA), a speedup technique for the CSA algorithm, to our problem setting. First we briefly recapitulate the most important aspects of CSA, the Customized Route Planning algorithm (which ACSA is based on) and ACSA itself. Then we discuss our adaptation, which we call *ML-RAPTOR* (Multilevel-RAPTOR).

### 5.3.1 Connection Scan Algorithm

The Connection Scan Algorithm (CSA) [DPSW13] is a very memory-efficient algorithm for public transit routing that uses a model of the public transit network that is based on *connections*. A connection is a subunit of a trip that represents the journey of the trip's vehicle from one stop to another without intermediate stops. In RAPTOR's model, the equivalent to a connection is a trip segment in which the trip visits no further stops between the entering and exiting stop. A trip can therefore be expressed as a sequence of connections, whereas RAPTOR's model would express a trip as a sequence of stop events. Apart from introducing the new subunit of the connection, CSA's model is similar to RAPTOR's, using equivalent definitions for routes, trips, stops and the footpath graph. Like RAPTOR, CSA requires that the footpath graph is transitively closed.

Formally, a connection $c$ is modeled as a 5-tuple $(c_{\text{dep\_stop}}, c_{\text{arr\_stop}}, c_{\text{dep\_time}}, c_{\text{arr\_time}}, c_{\text{trip}})$ consisting of a departure stop $c_{\text{dep\_stop}}$, an arrival stop $c_{\text{arr\_stop}}$, a departure time $c_{\text{dep\_time}}$, an arrival time $c_{\text{arr\_time}}$ and a trip $c_{\text{trip}}$. We require that $c_{\text{dep\_stop}} \neq c_{\text{arr\_stop}}$ and $c_{\text{dep\_time}} < c_{\text{arr\_time}}$.

The original formulation of CSA, which will be explained here, solves the Earliest Arrival Problem with the number of trips as a secondary criterion, i.e., if two journeys have the same arrival time but a different number of trips, the journey with fewer trips is considered optimal. The basic approach of CSA is to sort all connections in the network according to the departure time and then scan each connection individually in ascending order. The algorithm maintains two data structures: For each stop $v$, $S[v]$ stores the earliest arrival time at $v$ found so far. For each tr, $T[\text{tr}]$ is a boolean value that indicates whether at least one connection in tr has been reached from the source stop $s$ so far.

When scanning a connection $c$, we must first determine if $c$ can be entered. If $S[c_{\text{dep\_stop}}] \leq c_{\text{dep\_time}}$, the algorithm has already found an arrival that allows passengers to enter $c$ at $c_{\text{dep\_stop}}$. Furthermore, if $T[c_{\text{trip}}]$ is true, $c_{\text{trip}}$ can be entered on an earlier connection that was previously scanned, so passengers can enter the trip there and stay in it to reach $c$.

Thus, if either of these two conditions holds, $c$ can be entered and is therefore scanned by the algorithm. This involves setting $T[c_{\text{trip}}]$ to true and checking if taking $c$ improves the earliest arrival time at $c_{\text{arr\_stop}}$, i.e., $c_{\text{arr\_time}} < S[c_{\text{arr\_stop}}]$. If this is the case, we set $S[c_{\text{arr\_stop}}]$ to $c_{\text{arr\_time}}$ and relax all outgoing footpath edges of $c_{\text{arr\_stop}}$, in a similar manner to the RAPTOR algorithm.

The main advantage of CSA is that it does not use a priority queue and only involves a linear scan over all connections. This makes it very memory-efficient, allowing it to solve queries on city-sized networks very quickly. However, like RAPTOR, it is too slow on country-sized networks to allow for interactive queries and therefore preprocessing techniques are needed to speed it up.

### 5.3.2 Customizable Route Planning

Customizable Route Planning (CRP) [DGPW11] is a speedup technique that was originally developed for route planning scenarios on road networks where it must be possible to frequently change the metric used for the edge weights, for example to incorporate traffic information. This is achieved by using two separate preprocessing phases: a metric-independent preprocessing phase that only considers the graph topology and a *customization* phase that considers the metric as well. The customization phase is designed to be fast so it can be executed frequently.

The metric-independent preprocessing phase computes a *multi-level partition* of the graph, which is a family of partitions $\{\mathcal{Z}^1, \dots, \mathcal{Z}^L\}$. We require that the partitions are nested, i.e., for each cell $z_i$ on a level $\ell < L$, there exists a cell $z_j$ on level $\ell + 1$ such that $z_i \subseteq z_j$. This means that the partition $\mathcal{Z}^\ell$ is obtained by further partitioning each cell in $\mathcal{Z}^{\ell+1}$. For ease of notation, we define $\mathcal{Z}^{numlevels+1}$ as a partition with only one cell, which contains all vertices in the graph.

For each level $\ell$ and each cell $z \in \mathcal{Z}^\ell$, the customization phase computes direct shortcut edges between all pairs of boundary vertices of $z$. These shortcuts can then be used to quickly traverse the cell during queries without relaxing the interior edges. Because the boundary vertices of a cell form a clique after computing the shortcuts, the shortcuts can be efficiently stored in a matrices that can already be set up in the metric-independent preprocessing phase. The customization phase then only has to fill these matrices. Boundary edges, i.e., edges that connect vertices from different cells, are marked with the highest level on which they are boundary edges.

During an $s$-$t$-query, the *query level* $q(v)$ of a vertex $v$ is the highest level on which $v$ is not in the same cell as either $s$ or $t$. When $v$ is reached during a query, it is only necessary to relax the outgoing shortcuts and boundary edges belonging to the query level $q(v)$. This ensures that regions of the graph that are far away from $s$ and $t$ are traversed quickly via high-level shortcut edges. As the query gets closer to $s$ or $t$, it descends to lower levels and scans more edges.

During the customization, shortcuts are computed in a bottom-up fashion, starting with the lowest level. This allows the customization on each level to benefit from the shortcuts that were already computed for lower levels.

### 5.3.3 Connection Scan Accelerated

Connection Scan Accelerated (ACSA) [SW14] is a speedup technique for CSA that adapts the ideas of CRP to public transit networks. Like the original version of CSA, it solves the Earliest Arrival Problem with the number of transfers as a secondary criterion and requires that the footpath graph is transitively closed. On the network of Germany, ACSA achieves a speedup of up to 49 compared to the original CSA.

In order to compute the multi-level partition, the public transit network is transformed into a weighted graph in the following manner: Each connected component of stops in the footpath graph is mapped onto one vertex in the partitioning graph. This prevents footpaths from crossing cell borders. An edge exists between vertices $u$ and $v$ if the network contains at least one connection between a stop in the component represented by $u$ and a stop in the component represented by $v$. The weight of the edge is the number of such connections.

A connection $c$ is called an *interior (exterior) connection* of a $z$ if it departs inside (outside) of $z$. An *entry connection* of a cell $z$ is a connection that departs outside of $z$ but arrives inside of it. Accordingly, an *exit connection* of $z$ departs inside of $z$ but arrives outside of it. Entry and exit connections have a similar role to boundary edges in the original CRP. Due to the way the partition is constructed, footpath edges are always entirely contained within a cell. For a connection $c$ and a level $\ell$, we denote the cell where $c$ departs on level $\ell$ by $z_\ell(c)$.

Instead of precomputing shortcuts as in CRP, ACSA collects for each cell $z$ a set of interior connections that are needed to traverse the cell, the *transit connection set* $T(z)$. The transit connection set is defined as follows: For each optimal journey $J$ that visits $z$, it is possible to construct an equivalent journey $J'$ such that for each transfer between trips in $J'$ that occurs inside of $z$, the two connections immediately adjacent to the transfer are contained in $T(z)$ (unless they are exterior connections of $z$). Any other connections belonging to $J'$ do not need to be contained in $T(z)$. This is similar to our adaptation of Arc-Flags, where we marked stop events that are directly adjacent to a transfer.

The transit connection sets can be used to prune the set of connections that are scanned by the CSA algorithm: In order to determine if a connection $c$ belonging to the trip $c_{\text{trip}}$ can be reached, it is not necessary to have scanned all preceding connections in $c_{\text{trip}}$; it is sufficient to have scanned one preceding connection where $c_{\text{trip}}$ was entered, since this sets $T[c_{\text{trip}}]$ to true. Therefore, if the CSA algorithm only scans the connections in $T(z)$, it will still be able to find Pareto-optimal sets of journeys that traverse $z$.

The *long distance connection set* $D(z)$ of a cell $z$ on level $\ell > 1$ is the union of the transit connection sets of all children cells of $z$ on level $\ell - 1$. If $z$ is a cell on the lowest level 1, $D(z)$ is the set of all interior connections of $z$. An $s$-$t$-query can be performed by merging the long distance connection sets of all cells containing $s$ or $t$ and scanning these connections with CSA. For local queries where $s$ and $t$ are part of the same cell on some level $\ell < L$, this can be optimized by only merging long distance conection sets up to the lowest cell $z$ that contains both $s$ and $t$. However, this only works as long as optimal journey does not leave $z$. ACSA handles this by computing an additional *loop connection set* that contains all connections that are necessary to find optimal journeys that leave and reenter $z$. However, this will not be discussed in detail here.

The customization phase of ACSA involves computing the transit connection sets and long distance connection sets of all cells. Similarly to CRP, this can be done in a bottom-up fashion, starting with the lowest level: For a cell $z$ on level $\ell > 1$, once the transit connection sets of all its direct children cells on $\ell - 1$ have been computed, $D(z)$ can be computed as the union of these sets. If a connection $c$ is included in $T(z_\ell(c))$ on $\ell$, then for each lower level $\ell' < \ell$, it is also included in the corresponding transit connection set $T(z_{\ell'}(c))$ on level $\ell'$. This implies $T(z) \subseteq D(z)$, so only connections in $D(z)$ have to be scanned in order to compute $T(z)$.

The original paper on ACSA proposed two alternative versions of the transit connection set $T(z)$, with different algorithms for computing it: The first is the *arrival time transit set* $T_a(z)$, which is only sufficient for the Earliest Arrival problem and makes no guarantees with respect to number of transfers. Furthermore, it does not support transfer edges that are loops, which is how ACSA models minimum change times. Since minimum change times are an integral part of our model, this set is not suitable for adaptation. Instead we consider the *transfer transit set* $T_t(z)$. It can be computed by using an extended CSA algorithm for the Range Problem to compute profiles between each pair of entry and exit connection of $z$. For each optimal journey that is found, the first and last connection of each trip in the journey is added to $T_t(z)$ if it is an interior connection of $z$.

### 5.3.4 ML-RAPTOR

When constructing the partitioning graph, ACSA maps each connected component in the footpath graph onto the same vertex. If unrestricted walking is allowed, the entire graph is connected, so this is no longer possible. Instead, we construct the partitioning graph in the same manner as for the Route-Flags algorithm in Chapter 5.2.2 and again partition it with Inertial Flow. As a consequence, it is now possible for footpaths to cross cell borders.

For a stop event $e$ and a level $\ell$, we denote the cell containing the stop associated with $e$ on level $\ell$ by $z_\ell(e)$. The definitions for the transit and long distance connection sets can be adapted in a fairly straightforward manner if we collect stop events instead of connections. We define the *transit stop event set* $T(z)$ of a cell $z$ as follows: For each optimal journey $J$ that visits $z$, it is possible to construct an equivalent journey $J'$ such that for each stop $v \in z$ where $J'$ enters or exits a trip, the stop event of that trip at $v$ is contained in $T(z)$. The *long distance stop event set* $D(z)$ of a cell $z$ on level $\ell$ is the union of the transit stop event sets of all direct children cells of $z$ on level $\ell - 1$ or, if $\ell = 1$, the set of all stop events belonging to stops in $z$.

In practice, instead of maintaining the actual sets of stop events, we maintain markings for the stop events. Similarly to connections in ACSA, if a stop event $e$ is contained in $T(z_\ell(e))$ on level $\ell$, then for each lower level $\ell' < \ell$, it is also contained in $T(z_{\ell'}(e))$. Thus, for each stop event $e$, $M[e]$ stores the highest level $\ell$ on which $e$ is contained in $D(z_\ell(e))$. Note that this also means that $\ell + 1$ is the highest level on which $e$ is contained in $T(z_{\ell+1}(e))$.

In ACSA, connections that departs in a cell $z_1$ and arrive in another cell $z_2$ are considered part of $z_1$. In RAPTOR's model, there are edges and trip segments that connect vertices from different cells. For the purpose of computing the transit stop event sets, we consider these to belong to the cell where they begin. When computing the transit stop event set $T(z)$ for a cell $z$, we therefore compute journeys that start at a boundary vertex of $z$ and end at the first vertex outside of $z$ that is reached, which can be achieved by pruning the RAPTOR search immediately after it leaves $z$. We must compute a set of journeys $\mathcal{J}$ such that for each optimal journey that visits $z$, the subjourney restricted to $z$ can be replaced by a journey in $\mathcal{J}$. Subjourneys restricted to $z$ may begin or end midway through a trip or a footpath, so they may be improper subjourneys. To ensure that the journeys in $\mathcal{J}$ can be improper subjourneys of optimal journeys, they must be switch-optimal. We can therefore compute $T(z)$ by running a one-to-all rRAPTOR search from each boundary vertex $v$ of $z$ and pruning it once it leaves $z$. For all source-switch-optimal journeys that are found, we identify the required stop events and mark them accordingly. As with Route-Flags and the footpath shortcut preprocessing algorithm presented in Chapter 4, we compute queries in the interval $[0h, 24h]$. For each trip that begins before $24h$ but continues after it, we insert a copy that is shifted backward by 24 hours. The markings for stop events in the interval $[0h, 24h]$ are then copied to the corresponding stop events in $[24h, 48h]$ at the end of the customization.

When computing $T(z)$, we are only interested in subjourneys that start and end outside of $z$ but visit it intermittently. ACSA took advantage of this by starting the customization searches on entry connections and ending them on exit connections. In our adaptation, we similarly end our customization searches once the first vertices outside of $z$ that are reached. Furthermore, when determining source-switch-dominance (see Definition 3.4), we only need to consider arrivals from outside of $z$. Furthermore, if a boundary vertex has no incoming edges or trips from outside the cell, we do not need to start a search there.

A problem arises during the customization when scanning a route $r$ that exits the current cell $z$ and later re-enters it. We do not want to scan the parts that lie outside of $z$, but we do want to scan the part after $r$ has re-entered $z$. If we simply aborted the route scan once $z$ is left, the part after the re-entry would be ignored. To prevent this, once $z$ is left, we skip all further stops until $z$ is re-entered. At the skipped stops, we neither attempt to update the arrival time nor attempt to switch to an earlier trip. After re-entering $z$, we reset the currently scanned trip to $\perp$, essentially treating the part of $r$ after the re-entry as a separate route. Otherwise, the algorithm would find arrivals that involve entering a trip of $r$ before it leaves $z$ and then staying in the trip while it leaves and re-enters $z$. As an additional optimization, we can precompute for each $r$ on each level $\ell$ if $r$ leaves and re-enters any cell on level $\ell$. During the customization for a cell $z$, if the currently scanned route $r$ does not leave and re-enter any cells on the current level, we can simply stop scanning $r$ once it leaves $z$.

So far we have only discussed how to handle the stop events during the customization. To handle the footpaths, we insert a shortcut edge into $\mathcal{F}$ for each walking leg that is part of a journey found during the customization. Similarly to stop events, we store for each edge $e$ a marking $M[e] = \ell + 1$, where $\ell$ is the highest level on which $e$ is part of a switch-optimal journey through the cell that contains $e$ on level $\ell$. As an alternative to inserting shortcuts to walking legs, we could also unpack the walking legs and mark the traversed edges directly.

An $s$-$t$-query can utilize the stop event and edge markings computed in the customization phase in a similar manner as the flags used by our adaption of Arc-Flags: As in CRP, the query level $q(v)$ of a vertex is defined as the lowest level on which $v$ is not contained in the same cell as either $s$ or $t$. During the transfer phase, we only relax an edge $e = (u, v)$ if $M[e] \geq q(u)$. Note that even though the customization inserted shortcuts for the walking legs, we still need to use a Dijkstra search since walking legs may span multiple cells, in which case they will be divided into multiple shortcuts.

When collecting the routes serving updated stops, we only collect a route $r$ visiting a stop $v$ whose arrival time is $\tau_{\text{arr}}(v)$ if $r$ contains at least one trip tr with $\tau_{\text{dep}}(\text{tr}, v) \geq \tau_{\text{arr}}(v)$ whose corresponding stop event $e = (\text{tr}, v, \tau_{\text{arr}}(\text{tr}, v), \tau_{\text{dep}}(\text{tr}, v))$ is relevant on the query level of $v$, i.e., $M[e] \geq q(v)$. During the route phase, a trip tr is only entered at a stop $v$ if $M[e] \geq q(v)$ for the corresponding stop event $e = (\text{tr}, v, \tau_{\text{arr}}(\text{tr}, v), \tau_{\text{dep}}(\text{tr}, v))$.

As with CRP and ACSA, the stop event and edge markings as well as the footpath shortcuts can be computed in a bottom-up fashion, starting with the lowest level. This way, the customization on level $\ell$ can make use of the marking and shortcuts already computed for the lower levels. On each level, the rRAPTOR searches for each source boundary vertex are independent of each other and can therefore be parallelized.

Overall, our adaptation of ACSA avoids the problems that made a direct adaptation of CRP to a graph-based model impossible. While we still compute shortcuts for the walking legs, the trips are retained as they are and are marked with the levels on which they are relevant. This means that the shortcuts can use scalar weights and therefore consume little memory even on higher levels. However, this also means we cannot hope to achieve a speedup comparable to that of CRP itself, since the queries still need to scan the original trips in each cell instead of bypassing the cells via shortcuts.

# 6. Experiments

In this chapter, we evaluate the performance of the algorithms developed in Chapters 4 and 5 on a real public transit network.

## 6.1 Network Data and Experimental Setup

To ensure comparability of our results, our experimental setup is for the most part based on that used in [WZ17]. All our experiments were performed on the public transit network of Switzerland, which was obtained from a publicly available GTFS feed [1]. Stops and connections that lie outside of the country borders were removed. We extracted the public transit data for a single business day (30th of May, 2017), including trips that start on that day and extend into the next day. As discussed in Chapter 2.1, we make the simplified assumption that the timetable is periodic and repeats daily. To reflect this, we created copies of all trips in the network and shifted them forward by 24 hours, creating an identical second day. Furthermore, as discussed in Chapter 4.2, trips that start on the first day and continue into the second day were copied and shifted back by 24 hours. This ensures that the data for the two included days is completely equivalent.

The original network already contains a few footpaths, which we retained. To create an unrestricted footpath graph, we extracted the road network of Switzerland, including pedestrian zones and stairs, from the OpenStreetMap project [2]. This footpath graph was connected with the public transit network in the following manner: For each stop, we identified the nearest vertex in the footpath graph. If the two were less than 5 meters apart, we merged them into a single vertex. If their distance was between 5 and 100 meters, we inserted a new vertex for the stop and connected the two vertices with an edge. Since the OpenStreetMap data is mainly used for map rendering, it includes many vertices with degree one or two that exist only to display curved roads in a realistic manner. Since these vertices are irrelevant for routing, we used the contraction operation to remove all such vertices unless they coincided with a stop in the public transit network. We computed travel times for the edges by assuming a direct walking path between the two endpoints of each edge and a constant walking speed of 4.5 km/h. As discussed in Chapter 2.5.2, in order to prevent the occurrence of cyclical paths that circumvent the minimum change time at a stop, we increased the walking times of edges adjacent to stops if necessary.

[1]http://gtfs.geops.ch/
[2]http://download.geofabrik.de/

| Network | Stops | Routes | Trips | Stop events | Vertices | Edges | Max. deg. |
|---|---|---|---|---|---|---|---|
| Original | 25 394 | 13 480 | 183 365 | 2 351 875 | 25 394 | 5 582 | 25 |
| Partial | 25 394 | 13 480 | 183 365 | 2 351 875 | 25 394 | 4 268 486 | 1 223 |
| Complete | 25 394 | 13 480 | 183 365 | 2 351 875 | 604 188 | 1 844 200 | 25 |

Table 6.1: Overview of the network instances used in our experiments. All instances are based on the public transit network of Switzerland. The Original instance includes only the footpaths contained in the original network. The Complete instance adds an unrestricted footpath graph based on OpenStreetMap data. The Partial instance contains a restricted, transitively closed footpath graph based on a selection of footpaths from the Complete instance.

For comparison with the original RAPTOR algorithm, we also created a network instance with a transitively closed footpath graph. Computing the transitive closure of the unrestricted footpath graph would result in a graph that is much too large. Instead, we created a restricted footpath graph by inserting direct edges between all stops whose walking distance was below a certain threshold. We chose the threshold so that the resulting transitively closed graph has an average vertex degree of approximately 100, which led to a threshold of 15 minutes. An overview of the three network instances used is given in Table 6.1.

When generating random queries for evaluating the performance of our algorithms, we want to choose queries that are as representative as possible of queries that would be requested by users of a real routing application. Stops in metropolitan areas are typically frequented by more passengers than stops in rural areas, and would therefore be chosen as the source or target stop of a query more often. To reflect this, instead of choosing the source and target stops uniformly at random, we choose them with a probability proportional to the number of trips serving them, as in [WZ17]. This results in a more realistic query distribution, since the number of trips serving a stop is a good indicator of how many passengers typically depart or arrive at this stop. Choosing the stops uniformly at random would overestimate stops in rural areas, which are less frequently used by real passengers. This would have an impact on the query results, especially with regard to initial walking, since stops served by fewer trips tend to require walking more frequently.

All algorithms were implemented in C++ and compiled with GCC 5.3.1 using the optimization flag -O3. Experiments were conducted on a machine with two 8-core Intel Xeon E5-2670 processors clocked at 2.6 GHz and 64 GiB of DDR3-1600 RAM. For the parallelizable portions of the preprocessing phases of our algorithms, all 16 cores were used. Everything else, including all queries, was run sequentially on a single core.

## 6.2 Analysis of Initial and Intermediate Walking

To analyze the impact that unrestricted walking has on the quality of the found journeys, we recreated an experiment originally conducted in [WZ17], which compared the travel times of optimal journeys in the Partial and Complete network instances. This experiment involved performing 100 one-to-one Earliest Arrival profile queries for the departure time interval $[0h, 24h]$. The source stops were chosen randomly, while the target stops were chosen according to their *distance rank*, which is a measurement of distance: Given a graph $G = (V, E)$ and a source vertex $s \in V$, the vertices in $V$ can be ranked in the order in which they are discovered by a Dijkstra search from $s$. The rank of a vertex $v \in V$ in that order is called its distance rank. For each source stop $s$, we chose a target stop whose distance rank from $s$ in the unrestricted footpath graph of the Complete network instance was between $2^{15}$ and $2^{16}$. This corresponds to an average travel time of roughly two hours.

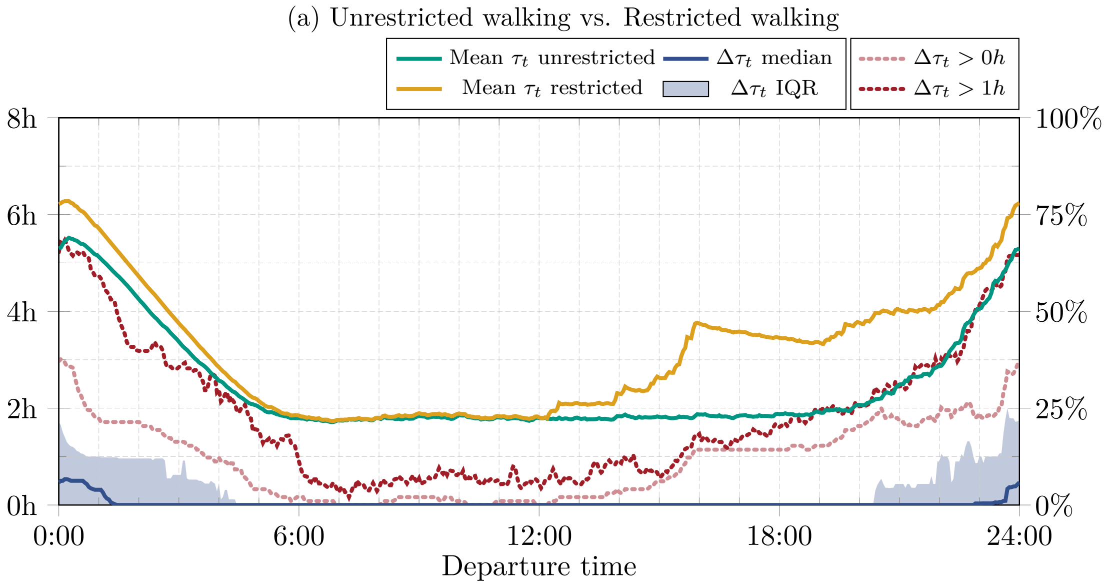
For each of the 100 queries, we ran a RAPTOR algorithm with unrestricted walking on the Complete instance and a regular RAPTOR algorithm with restricted walking on the Partial instance. We then compared the average travel times of the optimal journeys found in the two network instances, depending on the departure time. The results are shown in the upper plot of Figure 6.1. The green curve indicates the average travel time for the Complete instance, while the average travel time for the Partial instance is shown in yellow. It can be seen that the difference in the average travel times is very small in the morning, but significantly larger in the afternoon, evening and night, peaking at a departure time of approximately 16:00 with a travel time difference of almost two hours.

The dark red dotted curve shows the percentage of journeys whose travel time is not identical in both network instances. It averages around 10% during the daytime, but is significantly higher during the evening and night, reaching over 50% at some points. The percentage of journeys whose travel time difference is greater than one hour is indicated by the light red dotted curve. This percentage is very small during the morning, but during the evening and early night, it makes up the majority of all journeys whose travel times are not identical. Overall, the experiment shows that unrestricted walking is frequently needed during the evening and night, when fewer public transit connections are available than during the day.

In the introduction, we hypothesized that long walking legs mostly occur at the start or end of a journey, but only rarely between trips. To confirm this, we conducted a variation of the previous experiment, for which the results are shown in the lower plot of Figure 6.1. Here we replaced the regular RAPTOR algorithm on the Partial network instance with our algorithm from Chapter 4.1, which uses Dijkstra searches on the unrestricted footpath graph for the initial and final walking legs, but a restricted, transitively closed footpath graph for the intermediate walking legs. While this algorithm was originally developed with the intention of using a precomputed shortcut graph for the intermediate walking legs, it can be used with any transitively closed graph. Here we used the footpath graph of the Partial instance, resulting in an algorithm that can find journey with unrestricted initial and final walking legs, but only limited intermediate walking legs.

As before, the green curve indicates the average travel time for the Complete instance, while the average travel time for our algorithm with partially restricted footpaths is shown in yellow. The two curves almost overlap, indicating that the differences in the average travel times are extremely small. The percentage of journeys whose travel time is not identical in both network instances, which is indicated by the dark red dotted curve, is now below 10% during all times of the day. The percentage of journeys which a travel time difference of more than one hour, which is shown by the light red dotted curve is close to 0% for most departure times.

Overall, the experiment confirms our hypothesis that unrestricted walking legs beyond the 15 minutes permitted by the Partial instance are mostly needed for initial and final walking legs. While long intermediate walking legs do occur, they are very rare. This justifies the approach taken in Chapter 4, where we use Dijkstra searches on the unrestricted footpath graph for the initial and final walking legs and precompute shortcuts for all needed intermediate walking legs in a preprocessing phase. Unlike the intermediate transfer phases, the Dijkstra searches for initial and final walking only need to be performed once during a profile query, since the results can be reused by the individual RAPTOR executions for each departure time. Therefore, speeding up the intermediate transfer phases is much more important than speeding up the Dijkstra searches. While it would be possible to remove the need for the Dijkstra searches by precomputing shortcuts for initial and final walking as well, our experiments suggest that this would vastly increase the number of needed shortcuts and thus negatively impact the performance of the intermediate transfer phases.



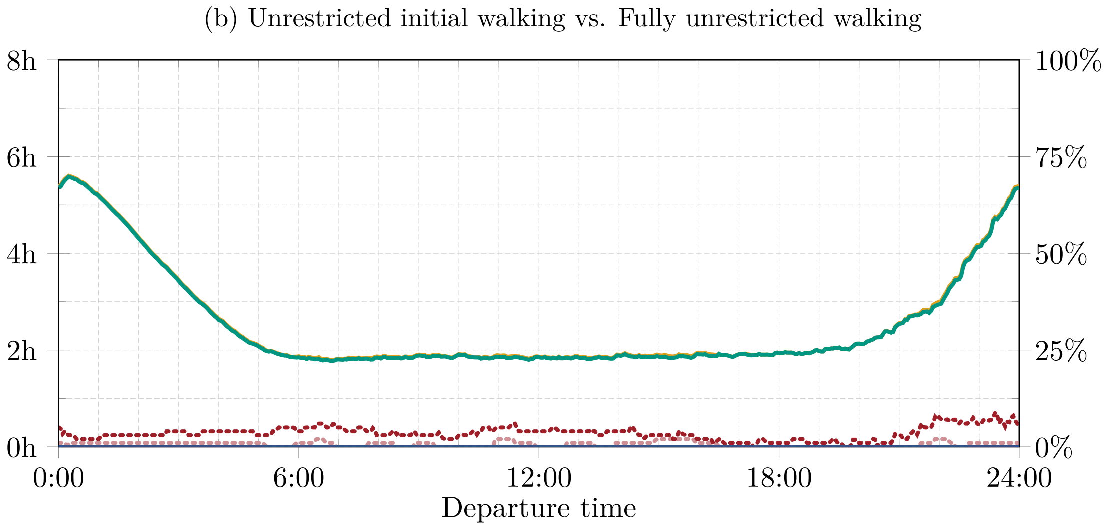


Figure 6.1: Comparison of the optimal travel times throughout the day on the Switzerland network, based on an experiment from [WZ17]. 100 random queries with distance rank $2^{16}$ were evaluated. In both plots, the green curve depicts the average travel time on the Complete network instance, where walking is unrestricted. The yellow curve indicates the average travel time with restricted walking. In the upper plot, the transitively closed footpath graph of the Partial instance is used for all walking. In the lower plot, only intermediate walking legs between trips are restricted, while initial and final walking legs may use the unrestricted footpath graph of the Complete instance. The blue curve shows the median of the travel time difference, while the light blue shaded area depicts the interquartile range (IQR). The two red dotted curves use the right y-axis instead of the left one. The light red dotted curve depicts the percentage of queries where the travel times found on both network instances were not identical. The percentage of queries where the travel time difference was greater than one hour is indicated by the dark red dotted curve.

Finally, we conducted an experiment to study how many different stops are actually relevant as destinations of initial walking legs or starting points of final walking legs. As discussed in Chapter 2.6.1, the main drawback of allowing unrestricted walking in the rRAPTOR algorithm is that it allows all stops to be reached from the source stop by direct walking. As a result, all possible departure times in the entire network need to be scanned, significantly

| Walking | Avg. #stops | Max. #stops | Avg. length |
|---|---|---|---|
| Initial | 10.5 | 33 | 2:20:39 |
| Final | 7.8 | 19 | 2:48:54 |

Table 6.2: Analysis of the initial and final walking legs of optimal journeys found for 1 000 random one-to-one profile queries. Listed are the average and maximum number of stops that serve as endpoints of the walking legs among the journeys for each query, as well as the average length of the walking legs.

slowing down the algorithm. Intuitively, it seems clear that the majority of these departure times are not actually needed since they involve extremely long initial walking legs. To confirm this, we performed 1 000 random one-to-one profile queries for the departure time interval $[0h, 24h]$, using bicriteria optimality. For each $s$-$t$-query, we evaluated all optimal journeys in the profile and counted how many different stops were reached via an initial walking leg from $s$ or were used to reach $t$ via a final walking leg. The results are listed in Table 6.2.

The experiment shows that on average, the journeys of an $s$-$t$-profile use only 10.5 different stops as endpoints of initial walking legs and only 7.8 as starting points of final walking legs. Among the 1 000 queries performed, the highest numbers of stops recorded were 33 for initial walking and 19 for final walking. This shows that only the departure times of a few stops actually need to be scanned during an rRAPTOR query. If it were possible to identify these stops beforehand or at least prune the set of stops whose departure times need to be scanned, this could severely reduce the running time of rRAPTOR. Both ML-RAPTOR and Route-Flags achieve this at least partially by not relaxing footpaths that are not marked or flagged, respectively, during the initial transfer phase.

## 6.3 Footpath Shortcut Preprocessing

In this section, we evaluate the performance of the speedup technique developed in Chapter 4, which precomputes shortcuts for intermediate walking. To this end, we compared the performance of our speedup technique in various configurations to that of the original profile RAPTOR algorithm presented in Chapter 2.6.2 for 100 random one-to-one profile queries. This was done on the Partial and Complete network instances, for both earliest arrival and bicriteria optimality. The results are reported in Tables 6.3 (for the Complete instance) and 6.4 (for the Partial instance).

To speed up the preprocessing phase for the Complete instance, we contracted the footpath graph beforehand, as described in Chapter 4.2, using an average vertex degree of 20 as the termination criterion. The contraction took 7:09 minutes and produced a footpath graph with 33 090 vertices (5.5% of the original vertices) and 661 904 edges (35.9% of the original edges). In the contracted graph, 76.7% of all vertices are stops, compared to only 4.2% in the original graph. Running the profile RAPTOR algorithm with this contracted graph instead of the original footpath graph already reduces the average query time from 38.0 seconds to 6.2 seconds, a speedup of 6.2. The share of the transfer phases in the overall query time is reduced from 94% to 72%. In the Partial instance, the footpath graph contains no additional vertices besides the stops, so it cannot be contracted.

We compared these existing approaches to our new speedup technique. On the Complete instance, the preprocessing phase used the contracted footpath graph, while the queries used the contracted footpath graph for the initial Dijkstra searches and the precomputed shortcut graph for all intermediate transfer phases. On the Partial instance, the original

| Alg. | Limits | | Prepr. | #Shtct. | E.Sc. | Query time [ms] | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Rnd. | Wlk. | | | | Total | Rou. | Tr. | Sp. |
| Orig. | — | — | — | — | 537M | 37 975 | 2 010 | 35 541 | — |
| Contr. | — | — | — | — | 283M | 6 151 | 1 476 | 4 436 | 6.2 |
| EA | ∞ | ∞ | 2:11:26 | 134 921 | 39M | 2 407 | 959 | 1 364 | 15.8 |
| | ∞ | 15 | 1:32:25 | 808 575 | 219M | 3 144 | 1 056 | 2 009 | 12.1 |
| | 3 | 15 | 0:37:54 | 849 559 | 229M | 3 481 | 1 073 | 2 324 | 10.9 |
| | 3 | 60 | 0:37:12 | 460 863 | 129M | 2 797 | 1 013 | 1 699 | 13.6 |
| | 3 | ∞ | 0:51:49 | 146 287 | 41M | 2 499 | 967 | 1 437 | 15.2 |
| Bicrit. | 2 | ∞ | 3:23:56 | 1 040 316 | 287M | 3 347 | 1 101 | 2 166 | 11.3 |
| | 2 | 15 | 0:26:19 | 14 027 448 | 3 844M | 11 997 | 1 367 | 10 559 | 3.2 |
| | 2 | 30 | 0:25:32 | 10 115 601 | 2 747M | 9 416 | 1 336 | 8 010 | 4.0 |
| | 2 | 60 | 0:28:09 | 6 486 966 | 1 777M | 7 198 | 1 319 | 5 806 | 5.3 |
| | 2 | 120 | 0:34:52 | 4 145 820 | 1 172M | 5 633 | 1 247 | 4 313 | 6.7 |
| | 2 | 240 | 0:49:20 | 3 142 079 | 920M | 5 007 | 1 217 | 3 718 | 7.6 |

Table 6.3: Performance of the footpath shortcut preprocessing technique on the Complete network instance for 100 random profile queries. We compared the following algorithms: original rRAPTOR (Orig.), original RAPTOR with a contracted footpath graph (Contr.) and various configurations of our preprocessing technique for the Earliest Arrival Problem (EA) and the Bicriteria Problem (Bicrit.) with different limits for the number of rounds (Rnd.) and initial walking (Wlk., in minutes). Reported are the preprocessing time in hours (Prepr.), the number of computed shortcuts (#Shtct.), the number of edge relaxations performed by the intermediate transfer phases of the queries (E.Sc.) and the query times in milliseconds. Query times include the total running time, the time spent by the route phases (Rou.) and transfer phases (Tr.), and the achieved speedup (Sp.).

footpath graph was used in place of the contracted footpath graph. As discussed in Chapter 4.2, there are two simple ways to speed up the preprocessing phase: limiting the number of rounds performed by the individual RAPTOR executions and limiting the walking distances of the initial transfer phase. Doing so causes the preprocessing phase to compute an overapproximation of the shortcut graph, which can negatively impact the query performance. An exception is the Bicriteria Problem, where the number of rounds can be set to 2 while still computing a minimal solution. We evaluated our algorithm for various combinations of round and walking limits. On the Partial instance, limiting the walking distances is superfluous since the transitive footpath graph is already based on limited walking distances.

For the Earliest Arrival Problem on the Complete instance, an unlimited preprocessing phase takes slightly over 2 hours and produces 134 921 shortcut edges. The resulting query algorithm achieves a speedup of 15.8 compared to the original profile RAPTOR algorithm and 2.6 compared to the profile RAPTOR with contraction. The transfer phases now only constitute 57% of the overall running time and the number of relaxed edges during the intermediate transfer phases is reduced to less than 10% of the original edge relaxations.

Limiting the initial walking to 15 minutes reduces the preprocessing time by slightly less than 30%. At first glance, a more significant reduction might be expected since the number of scanned departure times is drastically reduced. However, this does not take into account the optimization proposed in Chapter 4.2 for the Earliest Arrival Problem, which discards departure times if their corresponding initial walking leg is already dominated by a previously found journey. This optimization already prunes many departure times which require walking legs that are longer than 15 minutes. Furthermore, the ones that are retained are likely to lead to optimal journeys, since they are not dominated by previously found journeys. If we set a strict limit of 15 minutes, these optimal journeys are no longer found and therefore are not available for dominating other journeys, which may be suboptimal. As a consequence, the computed shortcut graph is six times larger, the running time of the transfer phases increases by almost 50% and the overall speedup drops to 12.1.

By comparison, restricting the number of rounds to 3 is much more successful, both in terms of decreasing the preprocessing time and keeping the size of the shortcut graph low. A configuration with 3 rounds and 60 minutes of initial walking requires only 37 minutes of preprocessing time and achieves a speedup of 13.6. A configuration with 3 rounds and unlimited walking requires 52 minutes of preprocessing time and achieves a speedup of 15.2, which is only slightly worse than that achieved with unlimited preprocessing.

For the Bicriteria Problem, a preprocessing phase with unlimited initial walking takes over 3 hours and produces over 1 million shortcuts, significantly more than for the Earliest Arrival Problem. As a result, the transfer phases of the query now take up 65% of the overall running time and the overall speedup drops to 11.3. This shows that unrestricted walking is needed much more frequently when minimizing the number of transfers as well. This is not surprising, since one can always reduce the number of transfers in a journey by replacing trip segments with walking legs. The resulting journey will not be dominated by the original journey, regardless of how long the walking legs are.

The reason for the increase in preprocessing time compared to the Earliest Arrival Problem is that the optimization during the collection of the departure times is no longer possible. As a result, all departure times must be scanned, which significantly increases the running time of each profile RAPTOR search. A further consequence is that restricting the initial walking distances can now significantly reduce the number of scanned departure times and thereby the overall preprocessing time. Limiting initial walking to 15 minutes lowers the preprocessing time to 26 minutes. However, the downside to this is that the number of shortcuts increases dramatically, to 14 million. The resulting footpath graph is so dense that the query performance becomes worse than that of the original profile RAPTOR using the contracted footpath graph. A more conservative limit of 4 hours yields a preprocessing time of 49 minutes and a query speedup of 7.6, which is superior to using only the contracted footpath graph.

On the Partial instance, the achieved speedups are predictably poorer, since the running time of the transfer phases is much lower to begin with. An unlimited preprocessing phase for the Earliest Arrival Problem takes one hour and produces 70 309 shortcuts. This yields a query speedup of 3.6 compared to the original profile RAPTOR algorithm. The share of the transfer phases in the overall running time is reduced from 73% to 49%. Restricting the number of rounds to 3 reduces the preprocessing time to only 9 minutes while barely impacting the size of the shortcut graph or the query performance. For the Bicriteria Problem, 12 minutes of preprocessing are necessary and the resulting query speedup is 3.5.

Overall, we observe that our speedup technique improves on previous approaches for public transit routing with unrestricted walking. Compared to the best previously known approach, which involved contracting the footpath graph, our algorithm achieves a speedup of 2.6 for the Earliest Arrival Problem with a preprocessing time of 2 hours. This can be improved to less than one hour with only slight performance losses. The share of the transfer phases in the overall query time is reduced to under 60%. For the Bicriteria Problem, we only achieve a speedup of 1.8 with a preprocessing time of 3.5 hours. This can be reduced to 49 minutes, but at the cost of lowering the speedup to 1.2.

| Alg. | Rounds | Prepr. | #Shtct. | E.Sc. | Query time [ms] | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | Total | Rou. | Tr. | Sp. |
| Orig. | — | — | — | 665M | 3 649 | 821 | 2 677 | — |
| EA | $\infty$ | 1:00:17 | 70 309 | 10M | 1 007 | 449 | 492 | 3.6 |
| | 3 | 0:09:22 | 72 116 | 11M | 1 002 | 439 | 491 | 3.6 |
| Bicrit. | 2 | 0:12:12 | 108 266 | 17M | 1 041 | 483 | 510 | 3.5 |

Table 6.4: Performance of the footpath shortcut preprocessing technique on the Partial network instance for 100 random profile queries. We compared the original rRAPTOR (Orig.) to various configurations of our preprocessing technique for the Earliest Arrival Problem (EA) and the Bicriteria Problem (Bicrit.) with different limits for the number of rounds. Reported are the preprocessing time in hours (Prepr.), the number of computed shortcuts (#Shtct.), the number of edge relaxations performed by the intermediate transfer phases of the queries (E.Sc.) and the query times in milliseconds. Query times include the total running time, the time spent by the route phases (Rou.) and transfer phases (Tr.), and the achieved speedup (Sp.).

## 6.4 ML-RAPTOR

We continue by evaluating the performance of the ML-RAPTOR algorithm proposed in Chapter 5.3.4. We compared the performance of ML-RAPTOR to that of the original RAPTOR for 1 000 random one-to-one Earliest Arrival queries. The departure times were chosen uniformly at random.

The ACSA algorithm, which ML-RAPTOR is based on, was evaluated on a network instance that contains European long distance trains, German trains and many German buses [SW14]. For this network, a 5-level partition was computed where each cell was subdivided into exactly 3 children cells for the level directly below. The Inertial Flow algorithm, which we used to partition our networks, is typically not configured to exactly produce a given number of cells per level; instead, it recursively partitions the cells until none of them exceeds a given maximum cell size. For the Complete instance, which has 604 188 vertices, we computed a 5-level partition and chose the maximum cell sizes for each level so that the number of cells would be roughly similar to the partition used to evaluate ACSA. We chose the maximum cell sizes $[2^9, 2^{11}, 2^{13}, 2^{15}, 2^{17}]$, which yielded a partition with 8 top-level cells and approximately 4 direct children cells per cell on average. For the Partial instance, which only contains the 25 394 stops as vertices, we computed a 3-level partition with the maximum cell sizes $[2^9, 2^{11}, 2^{13}]$. We decided against using additional lower levels with even smaller cells, since this would lead to an overly high proportion of boundary vertices on the lowest levels. Statistics for the individual levels of the two partitions are reported in Table 6.5.

The running times for the customization phase, per level and in total, are reported in Table 6.6. On both network instances, the customization phase requires approximately 10 minutes. On the Partial instance, most of that time is spent on level 0, where the customization is much more expensive than on level 0 in the Complete instance despite containing fewer vertices in total. The reason for this is that in the Complete instance, the level 0 cells constitute very small geographical regions and contain few stops. As a consequence, the rRAPTOR searches performed between the boundary vertices of these cells only need to scan very few departure times and explore only very short route segments. In the Partial instance, all vertices are stops and the level 0 cells represent much larger regions. Therefore, the rRAPTOR searches need to scan more departure times and explore more

| Network | Level | Cells | Vertices | | Boundary vertices | |
|---|---|---|---|---|---|---|
| | | | Max. | Avg. | Total | Avg. |
| Complete | 0 | 1 927 | $2^9$ | 314 | 48 762 | 25 |
| | 1 | 478 | $2^{11}$ | 1 264 | 20 987 | 44 |
| | 2 | 123 | $2^{13}$ | 4 912 | 9 270 | 75 |
| | 3 | 32 | $2^{15}$ | 18 881 | 3 844 | 120 |
| | 4 | 8 | $2^{17}$ | 75 524 | 1 482 | 185 |
| Partial | 0 | 88 | $2^9$ | 289 | 5 799 | 66 |
| | 1 | 20 | $2^{11}$ | 1 270 | 632 | 32 |
| | 2 | 6 | $2^{13}$ | 4 232 | 226 | 38 |

Table 6.5: Statistics for our ML-RAPTOR partitions of the Complete and Partial network instances. Reported are the number of cells, maximum and average number of vertices per cell, total number of boundary vertices and average number of boundary vertices per cell.

| | Complete | | Partial | |
|---|---|---|---|---|
| Level | Time [s] | Subopt. journeys | Time [s] | Subopt. journeys |
| 0 | 88.9 | 1.33% | 591.3 | 0.45% |
| 1 | 91.4 | 1.00% | 14.8 | 1.03% |
| 2 | 125.7 | 0.81% | 3.8 | 0.56% |
| 3 | 170.4 | 0.47% | — | — |
| 4 | 152.9 | 0.48% | — | — |
| Total | 630.7 | 0.93% | 610.5 | 0.45% |

Table 6.6: Performance of the ML-RAPTOR customization on the Complete and Partial network instances, including the time spent per level and in total and the percentage of found switch-optimal journeys that are not optimal.

routes, making them much slower. Since the customization phase must find switch-optimal journeys, we also measured and reported the share of found journeys that were not optimal. In both cases, the percentage of suboptimal journeys was less than 1%, which suggests that their impact on the preprocessing time and the precomputed stop event and edge markings is negligible.

Table 6.7 gives an overview of the data that was precomputed by the customization, including the inserted shortcut edges and the markings for the stop events and edges. For each level $\ell$, we report the number of stop events and edges whose level marking is $\ell$, as well as the number of routes for which no stop events are marked above $\ell$. Note that the total number of stop events is slightly more than twice as high as that reported in Table 6.1; this is because we also count the duplicated stop events that were inserted to ensure that the public transit data for both days represented in the network is identical. For the Complete instance, the customization inserted 2 857 773 shortcut edges in addition to the 1 844 200 existing footpath edges. No shortcuts were inserted for the Partial instance, since its footpath graph is already transitively closed and therefore each connected component is already fully connected.

In the Partial instance, we observe that the number of marked stop events and edges decreases dramatically as the level increases, as is typical and desirable for CRP-based approaches. This is crucial for achieving a high query speedup, since it ensures that much fewer stop events and edges have to be considered on higher query levels. In the Complete instance, this effect is much less pronounced. 6.2% of all edges are marked on the highest level, compared to just 0.1% in the Partial instance. This can be explained by the fact that the inserted shortcut edges cover much greater distances than the original edges in the footpath graph. As seen in Section 6.2, long walking legs are not uncommon in optimal journeys. The shortcuts inserted for these walking legs typically span across multiple cells even on higher levels and therefore have a high level marking. While each such shortcut is typically only used by a few journeys, there are many such journeys in total, leading to a large number of shortcuts with high level markings.

The distribution of the stop event markings is even more atypical. The levels with the most marked stop events are actually level 2 and 3, while 9.1% of all stop events are still marked on level 5. As with the shortcuts, this is partially explained by the fact that public transit connections tend to cover greater distances than footpath edges. The cells on the lower levels, which are very small and contain mostly non-stop vertices, can often be traversed with pure walking. Public transit connections start to become more important for the higher levels, where the distances between boundary vertices become too large for pure walking.

Another possible reason for the high number of marked stop events on the higher levels is how the multi-level partition is constructed. When constructing the partitioning graph, we merge all public transit connections that connect the same pair of stops into a single, unweighted edge. As a result, the partitioning graph contains much fewer route-derived edges than footpath edges. The Inertial Flow partitioner attempts to minimize the number of cut edges between cells, but it makes no distinction between footpath-derived and route-derived edges. Since there are fewer route-derived edges in total, the partitioner tends to place a greater focus on minimizing the number of cuts among the footpath edges. This leads to a high number of public transit connections that cross cell borders, which are very likely to be marked during the customization. This could be remedied by using a partitioning algorithm that allows for weighted edges and assigning larger weights to the route-derived edges. However, this would come at the expense of creating more cuts among the footpath edges.

The Partial instance is easier to partition by comparison. Although it contains even more footpath edges than the Complete instance, these edges cover short distances and can be decomposed into fairly small fully connected components. Separating these connected components leads to a lot of cut edges, so the partitioning algorithm is likely to leave them intact whenever possible. The connected components are separated by large regions which contain no footpaths at all. In these regions, minimizing the total number of cut edges in the partitioning graph automatically minimizes the number of cuts among the trips as well.

For the routes in the Complete instance, the distribution across the levels is the opposite of what would be desirable: Instead of decreasing with each level, the number of routes with at least one marked stop event actually increases, and 42.0% of all routes contain at least one stop event that is marked on the highest level. On the Partial instance, the distribution is less extreme, but still does not have the desired form. The reason for this is that a route covers an even greater distance than its individual connections and is therefore even more likely to cross a cell border at least once. Routes that span large portions of the country, such as long-distance trains, are very likely to connect different top-level cells and therefore contain stop events marked on the highest level. Since Switzerland is a fairly small country, comparable in size to the larger German states, many routes representing regional trains will also cross cell borders on the higher levels. On the network of a larger country or a continental network, it would be possible to introduce higher levels with cells that cover even larger regions, making it less likely for regional routes to cross cell borders on the highest levels.

| Network | Level | Stop events | Routes | Edges |
|---|---|---|---|---|
| Complete | Total | 4 723 039 | 13 480 | 4 701 973 |
| | 0 | 585 682 (12.4%) | 206 (1.5%) | 1 795 230 (38.2%) |
| | 1 | 791 652 (16.8%) | 452 (3.4%) | 911 630 (19.4%) |
| | 2 | 1 201 077 (25.4%) | 1 341 (9.9%) | 703 434 (15.0%) |
| | 3 | 1 155 919 (24.5%) | 2 723 (20.2%) | 559 283 (11.9%) |
| | 4 | 559 942 (11.9%) | 3 093 (22.9%) | 442 926 (9.4%) |
| | 5 | 428 767 (9.1%) | 5 665 (42.0%) | 289 470 (6.2%) |
| Partial | Total | 4 723 039 | 13 480 | 4 268 486 |
| | 0 | 2 773 254 (58.7%) | 5 246 (38.9%) | 3 795 656 (89.0%) |
| | 1 | 1 720 456 (36.4%) | 3 630 (26.9%) | 461 612 (10.8%) |
| | 2 | 147 624 (3.1%) | 2 046 (15.2%) | 6 983 (0.2%) |
| | 3 | 81 705 (1.7%) | 2 558 (19.0%) | 4 235 (0.1%) |

Table 6.7: Overview of the data precomputed by the ML-RAPTOR customization on the Complete and Partial network instances. For each level $\ell$ of the multi-level partition, we report the number of stop events and edges whose level marking is $\ell$, as well as the number of routes which have no stop events marked above $\ell$.

Table 6.8 compares the performance of ML-RAPTOR to the original RAPTOR for 1 000 random queries. On the Complete instance, ML-RAPTOR achieves a speedup of 7.3. However, almost all of that speedup comes from the transfer phases, which are sped up by a factor of 10.3. This is not as strong as the speedup achieved by our footpath shortcut preprocessing technique, but slightly better than using a contracted footpath graph for the original RAPTOR algorithm, which speeds up the transfer phases by a factor of 8.1. The route phases are only sped up by a small factor of 1.2.

The most important factor determining the running time of the route phases is the number of scanned routes. A route is always scanned from its first updated stop to its last stop. Any intermediate stops cannot be skipped, even if none of their stop events are marked. Therefore, the number of marked stop events on the route has little impact on the running time of the route scan. Unfortunately, as observed above, most routes contain at least one stop event that is marked on a high level. If the transfer phase reaches the stop containing that stop event and the current query level is not higher than the stop event marking, the route phase must scan the entire route from that stop onward. As a consequence, the number of route scans performed by ML-RAPTOR is only cut in half compared to RAPTOR. Additionally, ML-RAPTOR must determine the query level and look up the stop event markings at every stop reached during the route phase, which further slows down the route scans. Together, this explains the low speedup of 1.2. On the Partial instance, the speedup for the route phase is slightly better since the distribution of the routes across the levels is more even.

Overall, ML-RAPTOR does not achieve the goal of significantly speeding up the route phases. This is mainly due to the way routes are explored by RAPTOR: Once a route has been entered, all following stops must be scanned, regardless of whether they have marked stop events on their query level. This is a significant difference to ACSA, which ML-RAPTOR is based on. Here, connections that are not contained in the long distance connection sets can be ignored entirely and do not have to be scanned. This allows ACSA to skip intermediate parts of routes that are irrelevant on higher levels, which ML-RAPTOR is unable to do.

| Network | Alg. | Route phases | | | | Transfer phases | | | | Total | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | R.Sc. | St.Sc. | T. | Sp. | V.Sc. | E.Sc. | T. | Sp. | T. | Sp. |
| Complete | Orig. | 29 683 | 335 299 | 8.3 | — | 1 187 899 | 3 693 540 | 200.0 | — | 212.0 | — |
| | Contr. | 29 833 | 337 025 | 7.1 | 1.2 | 61 102 | 1 809 600 | 24.6 | 8.1 | 34.2 | 6.2 |
| | ML | 16 112 | 83 170 | 6.9 | 1.2 | 17 747 | 1 288 702 | 19.5 | 10.3 | 28.9 | 7.3 |
| Partial | Orig. | 21 314 | 234 422 | 5.7 | — | 24 099 | 4 861 984 | 18.2 | — | 26.2 | — |
| | ML | 7 280 | 38 458 | 2.6 | 2.2 | 3 696 | 411 700 | 4.2 | 4.3 | 8.6 | 3.0 |

Table 6.8: Query performance of ML-RAPTOR for 1 000 random queries on the Complete and Partial network instances. We compared the original RAPTOR (Orig.) to our ML-RAPTOR (ML). For the Complete instance, we additionally evaluated the performance of RAPTOR with a contracted footpath graph (Contr.). For the route phases, we report the number of scanned routes (R.Sc.) and stops (St.Sc.). For the transfer phases, we report the number of scanned vertices (V.Sc.) and relaxed edges (E.Sc.). For both types of phases and overall, we report the running time (T., in milliseconds) and the achieved speedup (Sp.).

## 6.5 Route-Flags

We close the chapter with an evaluation of the Route-Flags algorithm proposed in Chapter 5.2.2. Similarly to ML-RAPTOR, we compared the performance of Route-Flags to that of the original RAPTOR for 1 000 random one-to-one Earliest Arrival queries. The departure times were chosen uniformly at random. Because Route-Flags does not speed up queries where the source and target stop are in the same cell, we only chose queries where the source and target are in different cells. While these may not be representative of the overall average performance of the algorithm, they give a clearer picture of the speedup that is achieved for long-distance queries.

We evaluated the algorithm on the Complete and Partial network instances. For the partitions, we used the highest level of the respective multi-level partitions we computed for ML-RAPTOR. For the Complete instance, the resulting partition has 8 cells. This is the same number of cells that exhibited the best performance for HypRAPTOR, another partition-based speedup technique which was also evaluated on the Switzerland network [DDPZ17]. The partition for the Partial instance has 6 cells.

Precomputing the flags took 3:12:27 hours on the Complete instance, but only 1:49 minutes on the Partial instance. The large discrepancy is mainly due to the fact that the Partial instance has much fewer vertices and the preprocessing phase therefore needs to compute much fewer journeys. Since the preprocessing phase must find source-switch-optimal journeys, we measured the share of found journeys that were not optimal. The percentage is 0.04% for the Complete instance and 0.08% for the Partial instance, which is even lower than the percentage of suboptimal journeys found during the customization for ML-RAPTOR.

Table 6.9 reports the total number and percentage of flagged stop events and edges for each cell on both network instances. Since intra-cell stop events and edges are always flagged for their respective cell, we also reported the numbers excluding those stop events and edges. On the Partial instance, only approximately 1% of the edges are flagged per cell. This is consistent with our findings in Section 6.3, where we observed that the vast majority of edges in the transitively closed footpath graph are not actually needed for any optimal journeys. The number of flagged stop events is much higher, ranging between 11% and 17%. On the Complete instance, around 20% of stop events and edges are flagged.

| Cell | Stop events | | Edges | |
|---|---|---|---|---|
| | Total | w/o intra-cell | Total | w/o intra-cell |
| | | Complete | | |
| 0 | 1 486 612 (31.5%) | 924 928 (19.6%) | 526 239 (28.5%) | 360 883 (19.6%) |
| 1 | 1 834 808 (38.8%) | 1 086 411 (23.0%) | 680 689 (36.9%) | 394 839 (21.4%) |
| 2 | 1 258 228 (26.6%) | 860 707 (18.2%) | 473 632 (25.7%) | 345 168 (18.7%) |
| 3 | 1 785 491 (37.8%) | 1 253 906 (26.5%) | 665 801 (36.1%) | 439 939 (23.9%) |
| 4 | 1 904 615 (40.3%) | 1 046 881 (22.2%) | 751 103 (40.7%) | 369 349 (20.0%) |
| 5 | 1 505 686 (31.9%) | 1 029 229 (21.8%) | 664 715 (36.0%) | 386 521 (21.0%) |
| 6 | 1 432 488 (30.3%) | 989 298 (20.9%) | 596 159 (32.3%) | 421 533 (22.9%) |
| 7 | 1 739 025 (36.8%) | 1 038 241 (22.0%) | 646 658 (35.1%) | 443 686 (24.1%) |
| | | Partial | | |
| 0 | 830 281 (17.6%) | 585 657 (12.4%) | 217 394 (5.1%) | 52 466 (1.2%) |
| 1 | 892 389 (18.9%) | 537 075 (11.4%) | 321 493 (7.5%) | 49 979 (1.2%) |
| 2 | 2 115 805 (44.8%) | 533 610 (11.3%) | 1 364 229 (32.0%) | 42 023 (1.0%) |
| 3 | 1 227 663 (26.0%) | 792 225 (16.8%) | 323 810 (7.6%) | 63 434 (1.5%) |
| 4 | 1 084 484 (23.0%) | 676 294 (14.3%) | 237 510 (5.6%) | 51 312 (1.2%) |
| 5 | 2 268 684 (48.0%) | 572 884 (12.1%) | 2 110 991 (49.45%) | 47 731 (1.1%) |

Table 6.9: Overview of the stop event and edge flags computed by the Route-Flags preprocessing phase on the Complete and Partial network instances. For each cell of the partition, we report the total number of flagged stop events and edges. Since intra-cell stop events and edges are always flagged, we also report the number of flagged stop events and edges excluding intra-cell stop events and edges.

Table 6.10 compares the performance of Route-Flags to the original RAPTOR for 1 000 random queries where the source and target stop are contained in different cells. Given the percentage of flagged stop events and edges, the speedup should not be expected to exceed a factor of 4–5 on the Complete instance. Indeed, the transfer phases are sped up by a factor of 3.7, which appears to be consistent with the number of flagged edges. For the route phases, however, we only achieve a speedup of 2.0. This is due to the same problems that affect ML-RAPTOR's route phases: Due to the way RAPTOR's route scans operate, the performance of the route phase depends mainly on how many routes are scanned in total and not on how many stop events are marked on the scanned routes. Since the proportion of routes that have at least one flagged stop event is significantly higher than the proportion of flagged stop events overall, the number of scanned routes is only reduced by a factor of 2.4. On the Partial instance, the speedup in the route phases is slightly higher than on the Complete instance while the speedup in the transfer phases is lower, just as with ML-RAPTOR.

Since the speedup achieved in the transfer phases is worse than that achieved by contracting the footpath graph, it makes sense to discard the edge flags altogether and instead use the contracted footpath graph for the transfer phases. This way, the algorithm still benefits from the slight improvements gained in the route phases without sacrificing performance in the transfer phases. However, since the transfer phases still take up the majority of the running time, this only constitutes a very slight improvement over merely using the contracted footpath graph. Unfortunately, it is not feasible to run the preprocessing phase directly on the contracted footpath graph, since it is very dense. This would lead to an extremely high number of boundary vertices. Overall, we find that neither ML-RAPTOR nor Route-Flags are able to dramatically reduce the running time of the route phases,

| Netw. | Alg. | Contr. | Route phases | | | | Transfer phases | | | | Total | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | R.Sc. | St.Sc. | T. | Sp. | V.Sc. | E.Sc. | T. | Sp. | T. | Sp. |
| Comp. | Orig. | ○ | 12 751 | 146 198 | 3.0 | — | 1 177 695 | 3 638 227 | 177.2 | — | 183.2 | — |
| | | ● | 12 760 | 146 308 | 2.3 | 1.3 | 60 434 | 1 544 936 | 17.9 | 9.9 | 21.9 | 8.4 |
| | RF | ○ | 5 420 | 59 033 | 1.5 | 2.0 | 268 241 | 607 147 | 48.5 | 3.7 | 52.1 | 3.5 |
| | | ● | 6 615 | 73 696 | 1.5 | 2.0 | 45 009 | 1 119 264 | 17.2 | 10.3 | 20.2 | 9.1 |
| Part. | Orig. | — | 11 152 | 126 018 | 2.4 | — | 19 482 | 3 557 843 | 13.0 | — | 16.1 | — |
| | RF | — | 4 223 | 45 532 | 1.0 | 2.4 | 5 902 | 728 727 | 7.4 | 1.8 | 9.1 | 1.8 |

Table 6.10: Query performance of Route-Flags for 1 000 random queries spanning different cells on the Complete and Partial network instances. We compared the original RAPTOR (Orig.) to our Route-Flags algorithm (RF). For the Complete instance, we additionally evaluated the performance of both algorithms when using a contracted footpath graph (Contr.). When using the contracted footpath graph with Route-Flags, the computed edge flags for the original footpath graph are ignored. For the route phases, we report the number of scanned routes (R.Sc.) and stops (St.Sc.). For the transfer phases, we report the number of scanned vertices (V.Sc.) and relaxed edges (E.Sc.). For both types of phases and overall, we report the running time (T., in milliseconds) and the achieved speedup (Sp.).

partially due to the quality of the partitions and partially due to the way RAPTOR performs the route scans.

# 7. Conclusion

In this thesis we developed and evaluated several techniques aimed at speeding up public transit routing with unrestricted walking. A crucial ingredient of many speedup techniques for Dijkstra's algorithm is the subpath property, which states that subpaths of optimal paths are optimal themselves. In Chapter 3, we showed that an analogous subjourney property does not hold for all journeys in a public transit network, but that it holds when restricting the subjourneys to the special class of proper subjourneys. To characterize which of the remaining improper subjourneys are needed to construct optimal journeys, we defined the property of switch-optimality. We developed an extension of RAPTOR that finds all switch-optimal journeys, which allowed us to consider speedup techniques that are based on precomputing subjourneys of optimal journeys.

In previous public transit routing algorithms with unrestricted walking, exploring the footpath graph was the main performance bottleneck, since it is typically much larger than the public transit network. We observed that the vast majority of long footpaths which are not contained in restricted footpath graphs occur as the first or last walking leg of a journey, whereas long intermediate walking legs are rare. This motivated the preprocessing technique for RAPTOR presented in Chapter 4, which involves precomputing shortcut edges for all necessary intermediate walking legs. This removes the need for Dijkstra searches during the intermediate transfer phases. This is especially useful for profile queries, since the remaining Dijkstra searches for the initial and final walking legs only need to be performed once, whereas the more frequent intermediate transfer phases are sped up significantly.

While precomputing shortcuts for the footpath graph reduces the impact of the transfer phases on the overall running time, it does not speed up the route phases. In Chapter 5, we therefore considered several existing speedup techniques and discussed whether they can be adapted for RAPTOR to create an algorithm that speeds up both the route phases and the transfer phases. We observed that some speedup techniques that produced good results in restricted walking scenarios cannot be adapted to unrestricted walking in a straightforward manner. Two approaches that seemed promising were Arc-Flags and Connection Scan Accelerated. We adapted these techniques to RAPTOR, creating the algorithms Route-Flags and ML-RAPTOR, respectively.

We evaluated the performance of our algorithms on the public transit network of Switzerland in Chapter 6. For the Earliest Arrival Problem, our footpath preprocessing technique achieves a speedup of 15.8 compared to the original RAPTOR and 2.6 compared to using a contracted footpath graph, which was the best previously known approach. The share of the transfer phases in the overall running time is reduced to less than 60%. For the Bicriteria Problem, our technique only achieves a speedup of 1.8 compared to using a contracted footpath graph, reflecting the fact that the number of relevant footpaths increases drastically if minimizing the number of transfers is added as a second criterion.

The results for Route-Flags and ML-RAPTOR were less promising. While ML-RAPTOR achieves a speedup of 7.3 compared to the original RAPTOR, it is only slightly faster than using a contracted footpath graph. Route-Flags achieves a speedup of 3.5 compared to the original RAPTOR. While it is slower than using a contracted footpath graph, we can combine the advantages of both by discarding the edge flags and using the contracted footpath graph for the transfer phases instead. For both Route-Flags and ML-RAPTOR, the majority of the performance gains compared to the original RAPTOR is due to speeding up the transfer phases. The route phases are only sped up slightly, by a factor of 1.2 for ML-RAPTOR and 2.0 for Route-Flags. This is mainly due to the manner in which RAPTOR scans routes, which does not allow for skipping irrelevant stops or stop events.

**Future Work.** Future research on the topic of speeding up public transit routing with unrestricted walking should be focused mainly on achieving larger speedups for the route phases, while still being able to handle unrestricted footpaths in such a manner that they do not dominate the running time. This could be achieved by combining our footpath shortcut preprocessing technique with other speedup techniques. Many existing speedup techniques for road networks or public transit networks with restricted walking cannot be adapted to unrestricted walking in a straightforward manner; it may be necessary to change them significantly or develop entirely new approaches. Speeding up the route phases of RAPTOR with precomputed data has proven to be problematic because RAPTOR's route scans cannot easily skip irrelevant stop events. Other algorithms that offer more fine-grained control over which data is processed by queries, such as CSA, may be better candidates for integration with techniques that involve filtering the amount of relevant public transit connections.

The performance of our Route-Flags algorithm may be improved by combining it with a multi-level partition, as done for the original Arc-Flags algorithm with SHARC [BD09]. It may be possible to reduce the comparatively high preprocessing time for Route-Flags by adapting ideas commonly used to speed up the computation of the shortest path trees in the original Arc-Flags algorithm, such as Centralized Shortest Path Search [HKMS09] or PHAST [DGNW13]. Another possible optimization is to partition the departure time interval into smaller intervals and compute time-dependent flags for these intervals, which was previously done for an adaptation of Arc-Flags to a graph-based public transit network model [BDGM09]. This may reduce the number of flagged stop events and edges during queries, albeit at the expense of a higher memory consumption.

Part of the reason why the speedups achieved by Route-Flags and ML-RAPTOR are fairly low is that the number of flagged or marked stop events and edges is very high. Future research could focus on examining whether this is due to intrinsic properties of public transit networks or the interaction between the public transit network and the footpath graph, which have different structures. Different partitioning approaches, such as the hypergraph partitioning approach used for HypRAPTOR [DDPZ17], may produce partitions that are more suitable to public transit networks. Another possible factor may be that the public transit network of Switzerland, on which we evaluated our algorithms, is fairly small. Networks of larger countries or entire continents may exhibit stronger hierarchical features that make it easier to find good partitions.

Finally, we argued in Chapter 2.5.2 that the common approach for modeling minimum change times is not realistic and observed that it causes several problems when allowing unrestricted walking. For the future, we propose using minimum entering and exiting times instead, which are applied every time a vehicle is entered or exited.

# Bibliography

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

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

[BD09] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, August 2009. Special Section on Selected Papers from ALENEX 2008.

[BDG$^+$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*, volume 9220, pages 19–80. Springer International Publishing, 11 2016.

[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 Modeling, Optimization, and Systems (ATMOS'09)*, OpenAccess Series in Informatics (OASIcs), 2009.

[BDPW16] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Dynamic Time-Dependent Route Planning in Road Networks with User Preferences. In *Experimental Algorithms*, pages 33–49. Springer International Publishing, 2016.

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

[BHS16] Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable transfer patterns. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 15–29, 2016.

[BJ04] Gerth Brodal and Riko Jacob. Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. In *Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03)*, volume 92 of *Electronic Notes in Theoretical Computer Science*, pages 3–15, 2004.

[DDP$^+$13] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing Multimodal Journeys in Practice. In *Proceedings of the*

*12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2013.

[DDPZ17] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Tobias Zündorf. Faster Transit Routing by Hyper Partitioning. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, volume 59 of *OpenAccess Series in Informatics (OASIcs)*, pages 8:1–8:14, Dagstuhl, Germany, 2017.

[DGNW13] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DKP09] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel Computation of Best Connections in Public Transportation Networks. Technical Report 16, Faculty of Informatics, Karlsruhe Institute of Technology, 2009.

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

[DPW09] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Accelerating Multi-Modal Route Planning by Access-Nodes. In *Proceedings of the 17th Annual European Symposium on Algorithms (ESA'09)*, volume 5757 of *Lecture Notes in Computer Science*, pages 587–598. Springer, September 2009.

[DPW12] Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. User-Constrained Multi-Modal Route Planning. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 118–129. SIAM, 2012.

[DPW15] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. *Transportation Science*, 49(3):591–604, 2015.

[Gei08] Robert Geisberger. Contraction Hierarchies. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008. `http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/geisberger_dipl.pdf`.

[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 Book*, pages 41–72. American Mathematical Society, 2009.

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

[Mar84] Ernesto Queiros Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 26(3):236–245, 1984.

[PS98] Stefano Pallottino and Maria Grazia Scutellà. Shortest Path Algorithms in Transportation Models: Classical and Innovative Aspects. In *Equilibrium and Advanced Transportation Modelling*, pages 245–281. Kluwer Academic Publishers Group, 1998.

[PSWZ08] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2008.

[SS15] Aaron Schild and Christian Sommer. On Balanced Separators in Road Networks. In *Experimental Algorithms*, pages 286–297. Springer International Publishing, 2015.

[SW14] Ben Strasser and Dorothea Wagner. Connection scan accelerated. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 125–137, 2014.

[Wir15] Alexander Wirth. Algorithms for Contraction Hierarchies on Public Transit Networks. master thesis, Karlsruhe Institute of Technology, May 2015.

[Wit15] Sascha Witt. Trip-based public transit routing. In *Algorithms - ESA 2015*, pages 1025–1036. Springer Berlin Heidelberg, 2015.

[WZ17] Dorothea Wagner and Tobias Zündorf. Public Transit Routing with Unrestricted Walking. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, volume 59 of *OpenAccess Series in Informatics (OASIcs)*, pages 7:1–7:14, Dagstuhl, Germany, 2017.