Karlsruhe Institute of Technology

Algorithms for the Pagination Problem on Public Transit Networks

Bachelor Thesis of

Moritz Halm

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers	*Advisors*
Prof. Dr. Dorothea Wagner	Jonas Sauer, M. Sc.
Prof. Dr. Peter Sanders	Tobias Zündorf, M. Sc.

November 22, 2018 – March 21, 2019

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text. I also declare that I have read the Satzung zur Sicherung guter wissenschaftlicher Praxis am Karlsruher Institut für Technologie (KIT).

Karlsruhe, March 21, 2019

Abstract

We study how public transit profiles can be computed in applications featuring pagination. Pagination is the concept of splitting a profile into pieces (pages) which are computed and output subsequently. We show that the way in which journeys are ordered plays a crucial role regarding the usefulness of a paginated profile. To this end, we discuss ordering journeys by their departure time and arrival time. We further propose the earliest time after which a journey is optimal as an alternative ordering criterion. We present different approaches based on the RAPTOR algorithm in order to support computing journeys with respect to each of these orderings. Initially, we adapt the profile algorithm by Wagner and Zündorf (2017) by changing the order in which forward and backward RAPTOR searches are performed appropriately. However, faster running times can be achieved by running the rRAPTOR algorithm on a reversed network instance to obtain journeys ordered by their arrival time. For other orderings, we reached slight speed-ups combining both approaches. We experimentally evaluated the performance of our algorithms on the public transit network of Switzerland.

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Frage, wie Reiseprofile auf öffentlichen Verkehrsnetzwerken unter Einsatz von Paginierung berechnet werden können. Paginierung bedeutet hier das Aufteilen eines Profiles in Abschnitte (Seiten), die nacheinander berechnet und ausgegeben werden. Dabei hat die Sortierung der Reisen innerhalb eines Profils entscheidenden Einfluss auf die Nützlichkeit des Ergebnisses. Wir vergleichen Sortierungen von Reisen nach Ankunfts- und Abfahrtszeit, und schlagen außerdem vor, Reisen nach dem frühesten Zeitpunkt, von dem an sie optimal sind, zu sortieren. Wir stellen verschiedene auf dem RAPTOR Algorithmus basierende Ansätze vor, wie Reisen in einer jeder dieser Sortierungen entsprechenden Reihenfolge berechnet werden können. Zunächst beschreiben wir, wie der von Wagner und Zündorf (2017) vorgestellte Profilalgorithmus jede dieser Sortierungen liefert, wenn man die Reihenfolge der durchgeführten Vorwärts- und Rückwärtssuchen in geeigneter Weise anpasst. Speziell für die Sortierung nach Ankunftszeit, können allerdings kürzere Laufzeiten durch den Einsatz von rRAPTOR erreicht werden, indem man den Algorithmus auf einem umgekehrten Netzwerk ausführt. Für die anderen beiden Sortierungen erreichten wir leichte Geschwindigkeitsvorteile durch eine Kombination beider Ansätze. Wir haben die Laufzeit unserer Algorithmen anhand von Experimenten auf dem öffentlichen Verkehrsnetzwerk der Schweiz gemessen.

Contents

1. Introduction

There has been much progress in the field of route planning on public transportation networks in recent years. Optimal journeys from one stop to another in a metropolitan scale public transit network can be computed within a few milliseconds [Bas+15]. Journeys of interest are the ones which are optimal in a Pareto sense, i.e., that minimize not only the travel time but also the number of trips, the ticket price or other criteria. For instance, passengers might favor a journey with a later arrival time if it requires less switching between trips.

One is often interested in finding Pareto optimal journeys not just for a single departure time but rather for a time range. Such *profile queries* are especially relevant in end user applications such as the online timetable information offered by railway operators, e.g., bahn.de. Typical users do not have a specific minimal departure time in mind. Instead, they prefer to get an overview of different journeys over a more general period of time, e.g., throughout the morning.

While there are already well-known algorithms, such as RAPTOR, that answer profile queries efficiently, practical implementations lack explicit handling of pagination. Pagination is the concept of splitting a result, here a list of journeys, into pages. When the user enters a minimum departure time τ, the earliest journeys departing after τ are presented to them. Later ones are computed and shown only if the user demands to see more journeys. This technique has the main advantage of spreading the costly computation of the profile over multiple pages and thus decreasing the response time until a user first sees results. Furthermore, a user is less overwhelmed by too many, possibly not very relevant results. However, current algorithms do not take advantage

of pagination in their computation, but always generate complete results for the entire time range.

When using pagination, the way in which journeys are ordered can have crucial impact on the user's decision making and must thus be carefully chosen. Suppose, for instance, journeys are ordered by departure time which can lead to the first journey on page two having an earlier *arrival time* than the last journey on page one. Despite being a relevant travel option, this journey is likely not to be noticed since a user does not expect earlier arriving journeys being listed on later pages. G

Related Work. While the problem of finding fastest routes in road networks is well understood, routing in schedule-based public transportation networks poses fundamentally different challenges. Public transit networks are inherently *time-dependent*, as passengers may only travel with vehicles that depart and arrive at fixed points in time. Existing algorithms differ in the way in which they model timetables.

Common solutions model public transportation networks as graphs (see [Mül+04] for an overview). Finding fast journeys is then reduced to the problem of finding shortest paths. This approach seems promising at first glance, since the shortest-path problem can be efficiently solved by Dijkstra's algorithm [Dij59] and further optimized using various speedup techniques [DPW09]. However, Dijkstra searches benefit less from these speedup techniques when applied to public transit networks due to the different structure of a public transit graph [Bas09]. Asking for journeys that are Pareto-optimal with respect to travel time and number of transfers further complicates the problem [Ber+09]. A very fast graph-based technique are the so-called *Transfer Patterns* [Bas+10]. On the downside, they require extensive precomputation.

The relatively new RAPTOR algorithm (introduced by Delling et al. [DPW12]) employs a dynamic programming approach to compute Pareto-optimal journeys. Due to more efficient memory access patterns, RAPTOR performs significantly faster than other, graph-based multicriteria algorithms such as *Layered Dijkstra* [BJ04] or the *Multicriteria Label-Setting* algorithm [MS07]. Similarly, the *Connection Scan Algorithm* (CSA) [Dib+18], which scans all connections in a network in order, achieves running times even faster than RAPTOR. However, it optimizes the arrival time as the only criterion for single departure time queries. There are variants of both RAPTOR and CSA that answer profile queries as well. The RAPTOR based variant, called RRAPTOR, makes use of a technique called *self-pruning*. However, both algorithms have limitations regarding the incorporation of footpaths between stops. In order to answer profile queries with unrestricted walking between stops, Wagner and Zündorf [WZ17] proposed an algorithm (referred to as *Alternating RAPTOR* in this thesis) that computes profiles by alternately

running forward and backward RAPTOR searches. For a more detailed overview of public transit routing algorithms we refer the reader to an extensive survey by Bast et al. [Bas+15].

The pagination problem has not been subject to research so far.

Contributions. In this thesis, we study the problems arising from computing profiles in the context of pagination. We discuss the advantages and drawbacks of different orderings. Apart from the obvious ordering criteria, i.e., the departure and arrival time of a journey, we also propose to order journeys by the earliest time after which they are optimal. This criterion, called *earliest optimal departure time*, combines the advantages of ordering by departure time and ordering by arrival time.

Since we are interested in journeys that are Pareto optimal with regard to arrival time and number of trips, we focus on the RAPTOR algorithm and its profile variant, rRAPTOR. Our objective is to employ these well-known algorithms to obtain journeys page-wise and already ordered by each of the aforementioned criteria. To this end, Alternating RAPTOR can be adapted quite straightforwardly by changing the order in which forward and backward searches are performed. However, faster running times are achieved using rRAPTOR. We propose to run rRAPTOR on a reverted network instance which yields journeys ordered by their arrival time. To make use of rRAPTOR's performance advantage for the orderings by departure time or earliest optimal departure time as well, we combine the rRAPTOR approach with the Alternating RAPTOR approach. We evaluate the developed algorithms experimentally on data of the public transportation network in Switzerland.

Outline. The remainder of this thesis is structured as follows: In Chapter 2, we introduce the basic notation we use to describe a public transit network and journeys in it. We define the problems of finding Pareto-optimal journeys for single departure times and departure time ranges, and describe in detail how these problems can be solved using the RAPTOR algorithm. In Chapter 3, we state formal requirements for correct pagination and present a generic framework for pagination algorithms. We also define the *earliest optimal departure time* of a journey as an alternative ordering criterion and discuss the advantages and drawbacks of different orderings. Chapter 4 presents algorithms that compute profiles page-wise and with respect to the previously defined orderings. It is divided into the approaches based on Alternating RAPTOR (4.1) and those based on rRAPTOR (4.2). An experimental evaluation of the algorithms can be found in Chapter 5. Finally, we summarize our insights and give an outlook on possible future work in Chapter 6.

2. Preliminaries

In this chapter, we introduce basic concepts and notations related to public transit routing. In particular, we describe how public transit networks are modeled, define common routing problems and describe the existing RAPTOR based algorithms to solve these problems.

The models used in this thesis apply to any public transport vehicles with a fixed schedule, e.g., trains, buses, ferries or planes. Since all these vehicles are modeled in the same way, a network may combine several different transport types. For ease of reading, however, we refer to all vehicles as trains.

2.1 Public Transit Network

All algorithms presented in this thesis take a *timetable* of a public transit network as input. The formalization of timetables we present here is very similar the one originally introduced along with the RAPTOR algorithm [DPW12]. A timetable describes where and when trains operate and how passengers can enter, exit and transfer between trains. More precisely we define a timetable $\mathcal{N}$ as a tuple $\mathcal{N} = (\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$.

$\Pi \subset \mathbb{N}_0$ is the *period of operation*, the time interval in which all trains operate. Typically, we only consider an interval consisting of one or two consecutive days. $\mathcal{S}$ is the set of *stops*. A stop is a location where trains stop and passengers can board or get off a train, i.e., a train station. A certain train, i.e., a physical vehicle, traveling along a sequence of stops at a specific time is called a *trip*. $\mathcal{T}$ is the set of all trips. For every stop s, visited by a trip tr, we denote the time the train arrives at the stop as $\tau_{\text{arr}}(\text{tr}, s) \in \Pi$ and the time at which it departs as $\tau_{\text{dep}}(t, s) \in \Pi$, with $\tau_{\text{arr}}(\text{tr}, s) \leq \tau_{\text{dep}}(\text{tr}, s)$. The arrival time

at the first stop and the departure time at the last stop of a trip are undefined. A set containing all trips that visit exactly the same sequence of stops is called a *route*. Trips can not overtake one another, i.e., there is a ordering of the trips tr_i of a route r, such that for every stop s on r $\tau_{\text{dep}}(\text{tr}_i, s) \leq \tau_{\text{arr}}(\text{tr}_{i+1}, s)$ holds true. The set $\mathcal{R}$ contains all routes.

Usually there are many trips per route. For example, a subway line in a city could be a route with a trip every 10 minutes. Note that a route in our model is not necessarily equivalent to a line in the network map of a city: Some trains of the line S1 in Karlsruhe, for example, drive as far as Bad Herrenalb while others already end at Ettlingen (which lies on the same track as Bad Herrenalb). The S1 would be modeled as two different routes, since all trips on one route must have the exact same stop sequence.

Entering a train at a stop can require some time (due to long distances between different platforms for example). To take this into account, a *departure buffer time* $\tau_{\text{buf}}(s)$ is associated with every stop $s \in \mathcal{S}$. The departure buffer time $\tau_{\text{buf}}(s)$ models the time a passenger arriving at s needs to walk to a platform and to enter a train there. On big train stations the walking distance between platforms can differ noticeably. In order to allow preciser buffer times where possible, such train stations can be modeled as multiple stops with footpaths between them.

Footpaths allow passengers to move to another stops by walking. They are defined by a directed graph $\mathcal{G} = (\mathcal{S}, \mathcal{F})$, where the nodes $\mathcal{S}$ are the train stops, and the edges $\mathcal{F} \subseteq \mathcal{S} \times \mathcal{S}$ are footpaths. Each footpath $(s_1, s_2) \in \mathcal{F}$ is associated with a constant walking time $\ell(s_1, s_2) \in \mathbb{N}$. We require the transfer graph to be transitively closed and to satisfy the triangle inequality, i. e., if there is a footpath from s_1 to s_2 and from s_2 to s_3, there also must be one between s_1 and s_3 (transitively closed), and $\ell(s_1, s_3) \leq \ell(s_1, s_2) + \ell(s_2, s_3)$ must hold (triangle inequality). This requirement can result in rather large transfer-graphs, since every connected component forms a clique. Hence, it is a common restriction to initially only consider real-world footpaths with a "short" walking time in order to keep the size of the transitive closure small.

2.2 Journeys

The objective of every routing algorithm considered in this thesis is to compute one or more *journeys* between a *source* stop s and a *target* stop $t \in \mathcal{S}$. A journey describes a way of traveling from s to t. It consists of trips (traveling by train) and footpaths (walking between stops) in the order of travel. Formally, a journey can be defined as a sequence of *trip segments*. A trip segment is a tuple (tr, u, v), where tr is a trip in $\mathcal{T}$ and u and v are stops in $\mathcal{S}$ served by tr, such that u is reached before v. Every trip segment

then describes a part of the journey, in which a passenger enters tr at u and exits tr at v. A transfer between subsequent trip segments is defined implicitly as the footpath between the exiting stop of the first trip and the entering stop of the following trip. For the same reason, the entering stop of the first trip segment in a journey and the exiting stop of the last segment do not need to be s or t, respectively.

We are only interested in *valid s-t-journeys*. A journey is considered valid if it is possible to transfer between all involved trains "in time" and to get from s to t. Formally, given $s, t \in \mathcal{S}$, a journey $J = ((\mathrm{tr}_1, u_1, v_1), \ldots, (\mathrm{tr}_k, u_k, v_k))$ is valid iff the following conditions hold:

- If J contains the subsequent trip segments $(\mathrm{tr}_i, u_i, v_i)$ and $(\mathrm{tr}_{i+1}, u_{i+1}, v_{i+1})$ and
  - if $v_i \neq u_{i+1}$, it must be possible to walk from v_i to u_{i+1}, i.e., $(v_i, u_{i+1}) \in \mathcal{F}$. Furthermore, the time span between the arrival of tr_i and the departure of tr_{i+1} must be large enough with respect to the walking time from v_i to u_{i+1} and the departure buffer time at u_{i+1}, i.e., $\tau_{\mathrm{dep}}(\mathrm{tr}_{i+1}, u_{i+1}) - \tau_{\mathrm{arr}}(\mathrm{tr}_i, v_i) \geq \ell(v_i, u_{i+1}) + \tau_{\mathrm{buf}}(u_{i+1})$.
  - if $v_i = u_{i+1}$, it must be possible to change trains in time, i.e., $\tau_{\mathrm{dep}}(\mathrm{tr}_{i+1}, v_i) - \tau_{\mathrm{arr}}(\mathrm{tr}_i, v_i) \geq \tau_{\mathrm{buf}}(v_i)$.
- If $u_1 \neq s$, there must be a footpath from s to u_1, i.e., $(s, u_1) \in \mathcal{F}$.
- If $v_k \neq t$, there must be a footpath from v_k to t, i.e., $(v_k, t) \in \mathcal{F}$.
- If J is an empty sequence, there must be a footpath from s to t. We call such a journey a *pure walking journey*.

Journey Properties

Given a non-empty s-t-journey $J = ((\mathrm{tr}_1, u_1, v_1), \ldots, (\mathrm{tr}_k, u_k, v_k))$, we define several properties of J. The *departure time* $\tau_{\mathrm{dep}}(J)$ of J is the time at which a passenger has to leave at s in order to arrive at u_1 and catch the first trip there: $\tau_{\mathrm{dep}}(J) = \tau_{\mathrm{dep}}(\mathrm{tr}_1, u_1) - \tau_{\mathrm{buf}}(u_1) - \ell(s, u_1)$. If $s = u_1$, the walking time is omitted, so $\tau_{\mathrm{dep}}(J) = \tau_{\mathrm{dep}}(\mathrm{tr}_1, s) - \tau_{\mathrm{buf}}(s)$. The *arrival time* $\tau_{\mathrm{arr}}(J)$ of J is the time at which a passenger arrives at t. If they have to walk the last part to t, i.e.,$v_k \neq t$, it is defined as $\tau_{\mathrm{arr}}(J) = \tau_{\mathrm{arr}}(t_k, v_k) + \ell(v_k, t)$. Otherwise, if $t = v_k$ the arrival time of J is simply the arrival time of the last trip segment: $\tau_{\mathrm{arr}}(J) = \tau_{\mathrm{arr}}(t_k, t)$. The *travel time* $\tau_{\mathrm{t}}(J)$ of J is the difference between arrival and departure time: $\tau_t(J) = \tau_{\mathrm{arr}}(J) - \tau_{\mathrm{dep}}(J)$. The *number of trips* (or trip-count) $|J|$ is the number of trip segments k used by J.

We are often only interested in journeys that depart no earlier than a given minimal departure time τ_{min}, i.e., $\tau_{\text{dep}}(J) \geq \tau_{\text{min}}$. In that case, we call these journeys *feasible*.

2.3 Problems

This thesis' focus lies on *public transit profile queries*. Profile queries ask for optimal journeys within an interval of possible departure times. It is a generalization of the more basic problem of finding optimal journeys for a single departure time, which we introduce first. In both cases we will only consider *one-to-one* problems, i.e., finding journeys from a single source stop s to a single target stop t.

2.3.1 Single Departure Time

The simplest routing problem is to fin an s-t-journey that minimizes the arrival time at the target stop t.

2.1 Definition Earliest Arrival Problem. *Given a public transit network $\mathcal{N} = (\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, a source stop $s \in \mathcal{S}$, a target stop $t \in \mathcal{S}$, and a departure time $\tau_{\text{min}} \in \Pi$, find a journey J among all feasible journeys with the earliest arrival time $\tau_{arr}(J)$.*

There may be several journeys with this arrival time. In this case, any of them is a valid solution, since arrival time is the only criterion. Alternatively, one could define the number of trips as a secondary criterion such that the earliest arriving journey with the lowest number of trips is the unique solution. This takes into account that many passengers consider a high number of transfers as inconvenient.

However, the earliest arrival time is not the only interesting primary criterion. Consider a scenario in which the earliest arriving journey requires many transfers, but there is another journey with fewer transfers but a slightly later arrival time. Some passengers would opt for the second journey, while others would prefer the first one. In such cases users should be able to decide on their own which journey they prefer. Hence, we ask for a set of journeys, such that each journey has the earliest arrival time with respect to its number of trips. Such a set is called a *Pareto set*. Pareto sets are a commonly used tool for multicriteria optimization problems ,for instance, in the field time-independent routing [Mar84]. A Pareto set is defined as follows:

Given a set S containing possible solutions and functions $f_1, ..., f_k : S \to \mathbb{R}$ to be minimized, a solution $s \in S$ is said to *dominate* another solution s', if it is not worse than s' regarding all objectives, i.e., $f_i(s) \leq f_i(s')\ (1 \leq i \leq k)$. A solution $s \in S$ is further considered *Pareto optimal*, if any other solution in S that dominates s is equivalent

to s according to all objectives. A *Pareto set* is a minimal set $P \subseteq S$ of Pareto optimal solutions such that every $s' \in S$ is dominated by at least one solution $s \in P$.

In our scenario, the objectives are arrival time and number of trips. This leads to the following definition for domination:

2.2 Definition Bicriteria Domination. *A journey J* bicriteria dominates *a journey J', if $\tau_{arr}(J) \leq \tau_{arr}(J')$ and $|J| \leq |J'|$.*

The definitions of Pareto optimal journeys and Pareto sets of journeys follow naturally.

2.3 Definition Bicriteria Problem. *Given a public transit network $\mathcal{N} = (\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, a source stop $s \in \mathcal{S}$, a target stop $t \in stops$, and a departure time $\tau_{\min} \in \Pi$, find a Pareto set of feasible s-t-journeys, i.e., a minimal set of s-t-journeys that dominate every other feasible journey. We call these journeys* bicriteria-optimal.

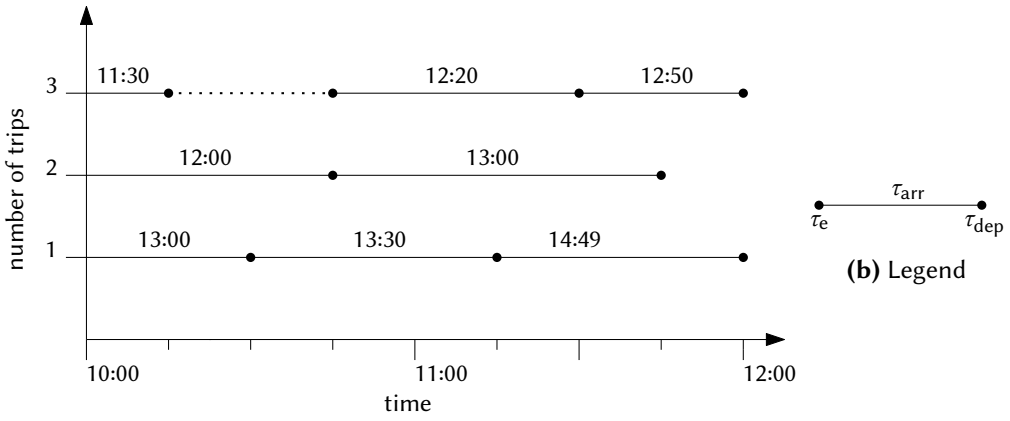
Note that according to our definition of Pareto optimality there can be multiple Pareto optimal journeys with the same arrival time and number of trips. However, only one of them is included in the Pareto set since we require it to be of minimal size. The solution to the Bicriteria Problem is therefore ambiguous.

2.3.2 Departure Time Ranges (Profile Queries)

Users often are not interested in a specific minimal departure time, but rather have a vague time period in mind during which they want to depart. We model this period as a discrete time interval I and ask for optimal journeys in the sense of the Bicriteria Problem (Definition 2.3) for any point of time in this interval. In other words: for any point of time τ_{dep} in I and any number of trips $n \in \mathbb{N}_0$ we ask for a journey with earliest possible arrival time departing no later than τ_{dep} and using no more than n trains.

2.4 Definition Range Problem. *We are given a public transit network $\mathcal{N} = (\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, a source stop $s \in \mathcal{S}$, a target stop $t \in \mathcal{S}$, and a time interval $I = [\tau_{\min}, \tau_{\max}] \subseteq \Pi$. For each $\tau_{dep} \in I$ we ask for a Pareto set containing all bicriteria-optimal s-t-journeys departing no earlier than τ_{dep}. A set of journeys P of minimal size containing such a Pareto set for every $\tau_{dep} \in I$ as a subset is called a* profile.

If there are several journeys with the same arrival time and number of trips but different departure times, the profile contains only the latest departing journey. This property follows from the minimality of the profile: Let J and J' be two journeys with $\tau_{arr}(J) = \tau_{arr}(J')$, $|J| = |J'|$ and $\tau_{dep}(J) > \tau_{dep}(J')$. Then there is a time $\tau :=$



(a) Sample Profile Diagram

(b) Legend

Figure 2.1 – Visualization of a fictional travel profile. There is one time line for each possible number of trips (a). Each time line is partitioned in intervals such that for each interval there is at most one journey solving the earliest arrival problem for this number of trips. Each interval begins with the earliest time after which that journey J is bicriteria-optimal, $\tau_e(J)$, and ends with its departure time $\tau_{dep}(J)$. The arrival time, $\tau_{arr}(J)$ of the journey is written above the respective interval (b). A dotted time line indicates that the optimal journey for this time interval involves less than n trips. It is represented by some interval in a lower time line. A dotted time interval in the time line for one trip means that it is either not possible at all to get from s to t in this interval using one trip or that walking directly would be faster.

$\tau_{dep}(J') + \varepsilon \leq \tau_{dep}(J)$ ($\varepsilon > 0$) for which J is feasible, but J' not anymore. Choosing J makes J' redundant and therefore results in a smaller profile. This observation shows that a profile can also be interpreted as a Pareto set of s-t-journeys regarding arrival time, number of trips and departure time as objective functions. Note that in contrast to the other two functions the departure time is maximized.

A profile P may contain journeys departing after τ_{max}. This is because for the departure times at the end of the interval optimal journeys often only depart after τ_{max}.

In theory, a profile could possibly contain as many journeys as there are time units in I: If there is a footpath $(s, t) \in \mathcal{F}$, then for any point of time $\tau \in I$ starting a walk from s to t at τ is an optimal journey for zero trips. In practice, if a pure walking journey exists, we therefore output it only once per profile rather than listing all $|I|$ walking journeys explicitly.

Profile Visualization

We visualize profiles using the following consideration: For every number of trips n, the profile can be partitioned into a sequence of non-empty time intervals, such that for the duration of each interval there is at most one optimal journey using at most n trips. Such an interval always ends with the departure time of the journey. This observation can be proven easily by contradiction: Assume the interval ends at a time $\tau < \tau_{\text{dep}}(J)$. Then there must be a journey J' with $|J'| < |J|$ and $\tau_{\text{dep}}(J') \geq \tau$ which dominates J, i.e., $\tau_{\text{arr}}(J') \leq \tau_{\text{arr}}(J)$. In this case J' would have already dominated J for the time before τ, so J is either not part of the profile at all or the interval for which J is optimal starts after $\tau_{\text{dep}}(J')$. With this observation, we can visualize profiles by drawing a timeline for every n, for which at least one bicriteria-optimal s-t-journey exists. For an example, see Figure 2.1.

2.4 Bicriteria Problem Algorithm: RAPTOR

In this section we present the basic variant of the RAPTOR (Round bAsed Public Transit Optimized Router) algorithm that was introduced by Delling et al. [DPW12]. The following description is analogous to the one given in the original paper.

RAPTOR solves the Bicriteria Problem (Definition 2.3) as well as the EARLIEST ARRIVAL PROBLEM (Definition 2.1) for a single departure time. In contrast to other approaches (see Chapter 1 for an overview), RAPTOR does not solve the public transit routing problem as a shortest-path problem in a graph, which models the public transit network. Instead, it uses dynamic programming to iterate over all stops that are reachable in a *round*, i.e., reachable with an increasing number of trips from the source stop s, and updates their earliest arrival time. This procedure is then repeated for several rounds. RAPTOR can solve *one-to-all queries*, i.e., compute the Pareto set for every stop in the network for a single departure stop. However, when only considering a single target stop (*one-to-one query*), *target pruning* can be used to speed up the computation.

The RAPTOR algorithm runs for multiple rounds. Round k computes the earliest arrival time $\tau_{\text{arr}}(v, k)$ for every possible target stop v using exactly k trips. A round consists of two phases: scanning routes and relaxing transfers. In every round we only consider arrivals that decrease the arrival time at the stop that was found in previous rounds. This ensures we only find Pareto-optimal journeys.

In the first phase, only routes containing an updated stop are scanned. A stop is called *updated* if its arrival time was improved in the previous round. We iterate over all stops in a route in increasing order, beginning with the first stop that was updated.

For each stop, we determine the earliest trip on the route that can be reached with the current arrival time at this stop. We then check for all subsequent stops on the route if their arrival time can be improved by using this trip and update them accordingly. In the second phase, we relax transfers. For every stop updated in the first phase, we relax all outgoing edges in the footpath graph, i.e., we update the arrival time of the stop at the other end if it is improved by taking the footpath. Since the footpath graph is transitively closed and satisfies the triangle inequality, scanning a single edge is sufficient to find a shortest walking path. Running rounds is repeated until no more stops have been updated by scanning routes in a round. This break condition is correct, since any potential bicriteria-optimal journey found in a later round would involve changing trains at one stop whose arrival time was improved in this round.

Pseudocode for RAPTOR is given in Algorithm 2.1. The main data structure is a two-dimensional array, the *round table* $R[\cdot][\cdot]$. It stores the earliest arrival time for every round and stop. More precisely, for every stop $v \in \mathcal{S}$, $R[k][v]$ denotes the earliest time at which v can be reached using *at most* k trips. U_r and U_t contain stops that were updated by a trip or a transfer, respectively. C is an associative array that maps a route to the first updated stop on it.

We initialize the round table for every new round with the value of the previous round. In the first round, we set all entries to ∞, except for $R[0][s]$, which is set to the departure time $\tau_{\min}$. We then begin by relaxing transfers, since a passenger can walk from s to another stop where they enter a train. All stops that were updated by transfer are stored in U_t. In order to scan routes, we need to determine the first stop on each route that was updated. We do this by iterating over every stop v in U_t and look up all routes r containing v. If there already is a stop u for r in C we replace it by v if v comes before u on r. Otherwise we store the mapping from r to v in C.

In the route scan phase (Algorithm 2.2), we look at all routes in C in an arbitrary order. For every route r with $v := C(r)$ we call the function ET to determine the earliest trip tr that can still be entered at v with regard to the earliest arrival time found so far and the departure buffer time. This trip can be found by scanning all trips operating on r in decreasing order. We iterate over all stops u on r. We check for each stop u whether its arrival time is improved by tr, i.e., $\tau_{\text{arr}}(\text{tr}, u) < R[k][u]$. If this is the case, we update the arrival time for this round $R[k][u]$ and insert u in U_r. For every stop, we replace tr by an even earlier trip, if possible. This may occur if a stop was reached by a different route or transfer in a previous round and thus has an earlier arrival time.

After the route phase is finished, U_r contains all updated stops. If U_r is empty, there was no progress made by taking any trip and we can terminate the algorithm. Otherwise we relax transfers (Algorithm 2.3). To do this, we iterate over every stop v in U_r and relax

Algorithm 2.1: RAPTOR

Input: Public transit network $(\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, source stop $s \in \mathcal{S}$, target stop $t \in \mathcal{S}$, departure time $\tau_{\min} \in \Pi$.
Data: Round table $R[\cdot][\cdot]$, stops updated by transfer U_t, stops updated by route U_r, routes containing updated stops C
Output: Round table $R[\cdot][\cdot]$

```
// Initialization
forall v ∈ S do
    R[0][v] ← ∞
R[0][s] ← τ_min
U_r.INSERT(s)
RELAXTRANSFERS(0)
forall k ← 1, 2, 3... do
    forall v ∈ S do
        R[k][v] ← R[k − 1][v]
    // Collect updated routes
    C.CLEAR()
    forall v ∈ U_t do
        forall routes r containing v do
            if r ∈ C then
                C(r) ← minindex_r(v, C(r))
            else
                C(r) ← v
    SCANROUTES(k)
    if U_r = ∅ then break
    RELAXTRANSFERS(k)
```

all outgoing edges $(v, u) \in \mathcal{F}$. We update u if its arrival time by transfer is improved by walking from v to u, i.e., $R[k][v] + \ell(v, u) < \tau_t(u)$.

Journey Extraction

So far our implementation only computes the arrival time, the departure time and the number of trips of a journey, but not the actual trips and transfers. However, this can easily be achieved by changing the definition of the round table. We now store

Algorithm 2.2: scanRoutes

```
1  Function scanRoutes(k):
2      U_r.clear()
3      forall (r, u) ∈ C do
4          tr ← ⊥
5          forall v on r beginning with u do
6              if tr ≠ ⊥ then
7                  if τ_arr(tr, v) < min(R[k][v], R[k][t]) then
8                      U_r.insert (v)
9                      R[k][v] ← τ_arr(tr, v)
10             τ ← R[k][v] + τ_buf(v) // earliest possible departure time
11             if tr = ⊥ ∨ τ < τ_dep(tr, v) then
12                 tr ←et(r, v, τ)
```

labels consisting of the arrival time and the trip segment that was taken in a round, i.e., $R[k][v] = (\tau_{\text{arr}}, (\text{tr}, u, w))$. If $w \neq v$, the journey also contains the footpath (w, v). In the route scanning phase, we have to keep track not only of the current trip tr of a route, but also of the stop u where tr was entered. Whenever updating the arrival time at a stop v we then set the trip segment entry of v to (tr, u, v). When arriving at a stop v' by footpath from v, we simply copy the trip segment entry of $R[k][v]$. Journeys can later be extracted by backtracking through the trip-segment entries.

Running Time

The worst case running time of RAPTOR can be bounded as follows: In every round we traverse every route $r \in \mathcal{R}$ at most once, which takes constant time per stop. This sums up to $\sum_{r \in R} |\text{stops}(r)|$, where $\text{stops}(r)$ are the stops on a route r. Note that $\sum_{r \in R} |\text{stops}(r)| \geq |\mathcal{S}|$ as one stop can be part of multiple routes. The same cost is required to determine the earliest updated stop per route, since for each updated stop v we look up all routes that contain v. If all routes are numbered consecutively, we can implement C as an array with constant access time.

The procedure to find the earliest reachable trip at a stop, ET, can be efficiently implemented by maintaining a pointer to the current trip. Note that ET is only called if the earliest possible departure time at a stop is smaller than the departure time of the current trip. Thus, the trip pointer may only decrease throughout the route and the total

Algorithm 2.3: relaxTransfers

```
1 Function relaxTransfers(k):
2     U_t.clear()
3     forall v in U_r do
4         forall (v, u) ∈ F do
5             τ_arr ← R[k][v] + ℓ(v, u)
6             if τ_arr < min(R[k][u], R[k][t]) then
7                 U_t.insert(u)
8                 R[k][u] ← τ_arr
```

running time of ET is bounded by $|r|$, the number of trips per route. Hence, we consider every trip at most once per round. Each footpath $(v, u) \in \mathcal{F}$ is relaxed at most once per round. Thus, in total the running time is bounded by $O(K(\sum_{r \in R} |\text{stops}(r)| + |\mathcal{T}| + |\mathcal{F}|))$, where K is the number of rounds.

Target Pruning

The basic RAPTOR algorithm computes Pareto sets for every stop in $\mathcal{S}$. However, since we are only interested in the arrival times at a single target stop t, we can exploit the following observation: If in any round k the arrival time $R[k][u]$ at a stop $u \neq t$ is greater than or equal to the arrival time $R[k][t]$ at the target, this arrival cannot be part of a Pareto-optimal s-t-journey. The lower arrival time at t was already reached with a lower or equal number of trips than the arrival time at u. Any journey containing this arrival at u would therefore be dominated. Hence, we can skip updating stops if they are dominated by the target arrival time. This decreases the search space and running time significantly. The pseudocode for Algorithm 2.2 and Algorithm 2.3 features target pruning.

2.5 Profile Algorithms

2.5.1 rRAPTOR

The RAPTOR algorithm as described in 2.4 can be adapted easily to answer profile queries, which has been defined in Section 2.3.2. Delling et al. [DPW12] introduced also a profile variant, rRAPTOR (*r* stands for *range*), in the same paper as single departure time RAPTOR. In this section, we describe rRAPTOR along with two optimization techniques.

Essentially, rRAPTOR runs a RAPTOR query for every possible departure time in the discrete departure time range $I = [\tau_{\min}, \tau_{\max}]$. Its efficiency results from reusing the round table from previous runs to reduce the amount of scanned trips in subsequent runs. This technique is based on the idea of the *Self-pruning Connection-Setting* algorithm [DKP12].

For simplification, we first describe rRAPTOR without considering footpaths from s. Every possible s-t-journey must then begin with a trip tr that serves s. Let Ψ be the set containing the departure times at s within I of all such trips. It then is sufficient to perform a single departure time RAPTOR query for each $\tau \in \Psi$ to find all bicriteria-optimal journeys for any time point in I. We call such a RAPTOR query a *run*. However, since an optimal journey for τ with $\tau \in \Psi$ does not necessarily have to use tr but can also start with a trip departing later on another route, these runs find many duplicate journeys, which have to be removed in a post-processing step. Furthermore, if there are two journeys J and J' with $\tau_{\text{arr}}(J) = \tau_{\text{arr}}(J')$ and $|J| = |J'|$, but $\tau_{\text{dep}}(J) > \tau_{\text{dep}}(J')$, only J may be included in the profile, so J' has to be removed as well. Due to the high degree of redundancy, this approach is rather costly.

rRAPTOR makes use of the observation that a journey departing at τ_1 can also be bicriteria-optimal for earlier departure times $\tau_2 < \tau_1$. It can thus dominate a journey departing at τ_2 but not vice-versa. rRAPTOR scans the departure times in Ψ in *decreasing* order. The round table is preserved between different RAPTOR runs. Suppose the arrival time τ_{arr} at a stop v of a s-v-journey J is greater than or equal to the one already stored in $R[k][v]$. Since $R[k][v]$ was set in a previous run there must be a later departing s-v-journey J' that dominates J. Hence, J can safely be pruned.

It remains to describe how initial footpaths are considered. Journeys can not only start with trips departing at s, but also at any other stop v that is reachable from s by a footpath. Let $\tau_s(\text{tr}, v) \coloneqq \tau_{\text{dep}}(\text{tr}, v) - \tau_{\text{buf}}(v) - \ell(s, v)$ be the time at which it is necessary to leave s in order to enter a trip tr that departs at v at $\tau_{\text{dep}}(\text{tr}, v)$. If we started a RAPTOR run for every such $\tau_s(\text{tr}, v)$ with $(s, v) \in \mathcal{F}$, all stops u reachable from s by footpath would be updated during the initial transfer phase of every run. However, we can use the fact that each stop u is scanned in any case by the RAPTOR run for $\tau_s(\text{tr}, u)$. We can thus skip the initial transfer phase. Instead, we have to store v together with $\tau_s(\text{tr}, v)$ in Ψ. To start a RAPTOR run we then set $R[0][v] = \tau_s(\text{tr}, v) + \ell(s, v)$, and insert v into U_t. The RAPTOR run then begins with scanning routes. Note that it is still important to scan departures in order of the departure time at s (not at v) to ensure correct self-pruning.

Lazy Round Table Propagation

Suppose a RAPTOR run for a certain departure time $\tau_i \in \Psi$ terminates after n rounds, whereas the subsequent RAPTOR run for $\tau_{i-1} < \tau_i$ only performs $m < n$ runs. If the round table entry $R[m][v]$ of a stop v has been updated in the second run, we have to propagate its arrival time to all higher-round entries $R[k][v]$ for $m < k \leq n$. This is necessary because further runs could require more rounds and access such entries $R[k][v]$.

However, we can avoid updating the round table in advance by using timestamps. We associate each entry in R with a timestamp, denoting the RAPTOR run in which it was updated. When accessing a round table entry $R[k][v]$, we first check if its timestamp belongs to the current run. If not, we recursively check lower rounds $R[j][v]$ $(j < k)$ until we find an entry $R[i][v]$ with a current timestamp. We then propagate its value to all entries $R[j][v]$ $(i < j \leq k)$ and update their timestamps. If the profile table entry $R[0][v]$ for round 0 has an expired timestamp, it can be set to $\tau_{\text{dep}} + \ell(s, v)$, where τ_{dep} is the departure time at s for this run, or ∞, if $(s, v) \notin \mathcal{F}$.

Trip Pruning

When determining the earliest trip $\text{ET}(r, v, \tau_{\text{arr}})$ of a route r that can be entered at a stop v with arrival time $\tau_{\text{arr}} = R[k][v]$, we iterate over all trips of this route in decreasing order. In the context of rRAPTOR we can make use of the observation that the arrival time τ_{arr} at any stop can only decrease throughout RAPTOR runs. As a consequence, the earliest trip may only decrease as well. We can thus speed up the look-up of trips by maintaining a pointer to the earliest trip per round, stop, and route and preserving it across runs.

2.5.2 Alternating RAPTOR

Another algorithm to answer profile queries was introduced by Wagner and Zündorf [WZ17]. They studied the problem of public transit queries with unrestricted walking between stops. An unrestricted walking graph contains further vertices apart from stops. It is not transitively closed and there is no restriction on the maximum length of walking paths passengers can use. Since having walking paths to almost all stops in the network results in a large number of possible departure labels rRAPTOR is impractical in this context.

The main idea of *Alternating RAPTOR* is to find an earliest arrival time τ_{arr} by running a basic RAPTOR search (see Section 2.4) and to find the journey departing latest that arrives at τ_{arr} by running a subsequent backward RAPTOR search. Given a departure

time interval $I = [\tau_{\min}, \tau_{\max}]$, Alternating RAPTOR first performs a RAPTOR search with departure time $\tau_{\min}$. This query yields a Pareto set of s-t-journeys with minimal arrival time for their respective number of trips. However, these journeys are not necessarily part of a profile, as for every journey J there could be another journey J' departing later with the same arrival time. This is because RAPTOR always takes the earliest reachable trip at every stop. For every journey J we found, we therefore run a *backward RAPTOR search* from t, starting with the arrival time $\tau_{\text{arr}}(J)$. The backward query then finds a journey J' with the latest possible departure time τ_{dep} for arrival time $\tau_{\text{arr}}(J)$ and $|J|$ trips. J' is then part of the profile. The profile is now complete for $|J'|$ trips and the interval $I' = [\tau_{\min}, \tau_{\text{dep}}]$. Figure 2.1a shows how a profile is composed of such intervals. We then perform another forward search with departure time $\tau_{\text{dep}} + \varepsilon$, followed by backward searches for all found arrival times. ε is the smallest time unit in Π, typically one second. This procedure is repeated until journeys departing at $\tau_{\max}$ or later have been found. In this case no subsequent forward search is performed.

When running RAPTOR queries, we have to explicitly exclude journeys with zero trips. If direct walking from s to t is possible, then for every $\tau \in I$ there is one Pareto-optimal journey J with zero trips and $\tau_{\text{dep}}(J) = \tau$. If a forward search found such a pure walking journey for departure time $\tau_{\min}$, the backward search would find the same journey and the next forward search would be started for $\tau_{\min} + \varepsilon$. We would in total perform two RAPTOR queries for every time unit in the departure time range I and find the same walking journey $|\Pi|$ times. To prevent this, we modify the basic RAPTOR algorithm to avoid reaching t in the initial relax transfers phase. However, if one can walk directly from s to t, the walking time $\ell(s, t)$ is used to prune longer journeys J' that are dominated by walking., i.e., $\tau_{\text{t}}(J') > \ell(s, t)$. Since a pure walking journey is time independent it is then sufficient to output only one such journey.

A backward search can be implemented by reversing the input data $(\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$. For every trip $\text{tr} \in \mathcal{T}$ serving a stop $v \in \mathcal{S}$, the departure time $\tau_{\text{dep}}(\text{tr}, v)$ and arrival time $\tau_{\text{arr}}(\text{tr}, v)$ are replaced by $-\tau_{\text{arr}}(\text{tr}, v)$ and $-\tau_{\text{dep}}(\text{tr}, v)$, respectively. Footpaths $(u, v) \in \mathcal{F}$ are inverted. In order to find the latest departing journeys for an arrival time τ_{arr} we have to run the RAPTOR algorithm on the reversed network with t as source stop, s as target stop and $-\tau_{\text{arr}}$ as departure time.

It remains to describe in which order the forward and backward searches are performed. A priority queue is used to sort arrival times that were found by forward searches in ascending order. In every step the minimal arrival time τ_{arr} is extracted from the queue. We then run a backward search for τ_{arr}. We add every journey J that was found by the backward search to the profile and immediately start a forward search for $\tau_{\text{dep}}(J) + \varepsilon$.

To avoid doing two forward searches for the same departure time, we have to keep track of the times for which we already performed a forward search.

The total running time of Alternating RAPTOR is bounded by the number of journeys in the profile. For each journey the basic RAPTOR algorithm is invoked at most twice.

Alternating RAPTOR already computes journeys ordered increasingly by their arrival time due to the priority queue. Thus, it can easily be used to compute a profile pagewise. RRAPTOR, in contrast, finds journeys from latest to earliest. However, the reuse of the round table (*self-pruning*) in RRAPTOR results in a lower degree of redundancy, which is why it performs better than Alternating RAPTOR when computing a whole profile at once.

3. Pagination Problem

Modeling the use case of end user applications (like bahn.de) as profile queries as defined in Section 2.4 is not quite realistic. The upper end of the time interval is often not known a priori. Rather, a user enters s, t and a minimum departure time τ_{min} and the application presents them some "early" journeys from the profile. If the user cannot find a journey that satisfy their needs, they can click a button such as "Later journeys" in order to compute another batch of journeys. Every such batch is called a page. The technique of splitting a result, in this case a profile, into pieces is called *pagination*. Pagination has two main advantages: The user is not overwhelmed by too many journeys, but sees the most relevant journeys first. The costly profile computation can be split and new journeys are computed on demand.

A crucial aspect when using pagination is the way journeys are ordered. Journeys on later pages are more likely not to be seen by a user, which is how ordering introduces a certain bias to their decision making.

In this chapter, we formally define ordering on profiles (Section 3.1.2) and how ordered profiles might be split into pages (Section 3.2.1). We then discuss the advantages of different ordering criteria for profiles in the context of pagination (Section 3.2.2). Additionally to the obvious criteria, i.e., ordering journeys by their arrival or departure time, we introduce an alternative criterion, the *earliest optimal departure time* of a journey (Section 3.1.1). Finally, we describe the general framework most pagination algorithms in this thesis follow.

3.1 Ordering Profiles

3.1.1 Earliest Optimal Departure Time

We define the *earliest optimal departure time* of a journey as another property of a journey. In contrast to the properties already defined in Section 2.2, this property does not depend solely on the journey itself, but rather describes how a journey is embedded in a profile.

3.1 Definition Earliest Optimal Departure Time. *Given a time interval* $I = [\tau_{\min}, \tau_{\max}] \subseteq \Pi$ *and a profile* P *over* I *and a journey* $J \in P$*, the* earliest optimal departure time $\tau_e(J)$ *is defined as the earliest time* $\tau \in I$ *for which* J *is bicriteria-optimal.*

In profile diagrams such as Figure 2.1 $\tau_e(J)$ is simply the starting point of the time interval for which J is optimal.

3.1.2 Partial Orders on Journeys

When using pagination it is crucial to define in which way journeys are ordered. We define an ordering on a profile using a *partial order*. In general, a binary relation $\preceq$ on a set S is a partial order if, and only if $\preceq$ is *reflexive*, *transitive*, and *anti-symmetric*, i.e. $\forall a, b, c \in S : a \preceq a$, $a \preceq b \wedge b \preceq c \Rightarrow a \preceq c$, and $a \preceq b \wedge b \preceq a \Rightarrow a = b$.

In this thesis we consider three partial orders. Each partial order uses a different journey property, e.g. departure time, arrival time or earliest optimal departure time, as the primary sorting criterion. To describe a partial order we use a triple of functions (f_1, f_2, f_3), each of which maps a journey to a natural number. Formally, the partial order $\preceq$ is defined as $J \preceq J' \Leftrightarrow (f_1(J), f_2(J), f_3(J)) \leq (f_1(J'), f_2(J'), f_3(J'))$. The tuples are compared lexicographically, i.e., $J \preceq J' \Leftrightarrow \forall i \, (f_i(J) > f_i(J') \Rightarrow \exists j < i : f_j(J) < f_j(J'))$.

- Ordering by *departure time* $\preceq_{\text{dep}}$:
  $J \preceq_{\text{dep}} J' \Leftrightarrow (\tau_{\text{dep}}(J), \tau_{\text{arr}}(J), |J|) \leq (\tau_{\text{dep}}(J'), \tau_{\text{arr}}(J'), |J'|)$
- Ordering by *arrival time* $\preceq_{\text{arr}}$:
  $J \preceq_{\text{arr}} J' \Leftrightarrow (\tau_{\text{arr}}(J), |J|, \tau_{\text{dep}}(J)) \leq (\tau_{\text{arr}}(J), |J|, \tau_{\text{dep}}(J))$
- Ordering by *earliest optimal departure time* $\preceq_{\text{e}}$:
  $J \preceq_{\text{e}} J' \Leftrightarrow (\tau_{\text{e}}(J), \tau_{\text{arr}}(J), |J|) \leq (\tau_{\text{e}}(J'), \tau_{\text{arr}}(J'), |J'|])$

All of the above relations naturally define a partial order on a profile. For each of the above partial orders we further define a relation $\approx$ on journeys, such that $J \approx J'$ if J and J' are equal with respect to the primary sorting criterion, i.e., $J \approx J' \Leftrightarrow f_1(J) = f_1(J')$.

To simplify notation, we also define for any partial order $\preceq$ on journeys the respective total order $\prec$ as $J \prec J' \Leftrightarrow J \preceq J' \wedge \neg(J' \preceq J)$.

The above orderings are well-defined on arbitrary journeys. However, when ordering journeys in profiles, i.e., in solutions for the Range Problem (Definition 2.4), it is not possible that a comparison of two journeys by the third tuple function occurs. If two journeys J and J' are equal with regard to the first two functions, one of them necessarily dominates the other journey: If $\tau_{\text{dep}}(J) = \tau_{\text{dep}}(J') \wedge \tau_{\text{arr}}(J) = \tau_{\text{arr}}(J')$, the journey with the lower number of trips dominates the other. Likewise, if $\tau_{\text{arr}}(J) = \tau_{\text{arr}}(J') \wedge |J| = |J'|$, the later departing journey dominates the earlier journey. If $\tau_{\text{e}}(J) = \tau_{\text{e}}(J') \wedge \tau_{\text{arr}}(J) = \tau_{\text{arr}}(J')$, the journey with the lower number of trips dominates the other journey. In any of these three cases only one of the journeys would be included in the profile at all.

3.2 Pagination

3.2.1 Formal Prerequisites

Suppose $(\preceq, \approx)$ is an ordering on a profile P. We call a finite sequence $(J_i)_{1 \leq i \leq |P|}$ containing every journey in P with $J_j \preceq J_{j+1}\,(\forall 1 \leq j \leq |P|)$ an *ordered profile*. Given the total number of pages, m, pages are defined by using indices j_k $(1 \leq k \leq m)$ such that each page P_k is a contiguous subsequence of the ordered profile: $P_k := (J_i)_{j_k \leq i < j_{k+1}} (j_1 = 1, j_{m+1} = |P| + 1)$. Given a page size n and a profile P we require a pagination with m pages to have the following properties:

- Every page but the last page contains at least n journeys, i.e., $j_{k+1} - j_k \geq n$ $(1 \leq k \leq m - 1)$.
- Journeys that are equal with respect to the primary sorting criterion are always on the same page, i.e., $J_i \approx J_{i+1} \Rightarrow \exists k : j_k \leq i < i + 1 < j_{k+1}$.
- If a page contains more than n journeys, the additional journeys on the page are all equal with respect to the primary sorting criterion, i.e., $j_{k+1} - j_k > n \Rightarrow \forall j_k + n \leq i < j_{k+1} : J_i \approx J_{j_k+n} (1 \leq k \leq m)$.

	$\tau_{\text{dep}}(J)$	$\tau_{\text{arr}}(J)$	$\lvert J\rvert$	$\tau_{\text{e}}(J)$
J_a	10:45:48	13:52:00	3	10:45:00
J_b	10:45:48	15:39:25	2	10:45:00
J_c	10:47:02	12:52:00	4	10:45:00
J_d	10:47:02	14:39:25	3	10:45:49
J_e	11:10:48	16:09:25	2	10:45:49
J_f	11:45:48	14:52:00	3	10:47:03
J_g	11:45:48	16:39:25	2	11:10:49
J_h	11:47:02	13:52:00	4	10:47:03
J_i	11:47:02	15:39:25	3	11:45:49
J_j	12:10:48	17:09:25	2	11:45:49

(a) Profile ordered by departure time.

	$\tau_{\text{dep}}(J)$	$\tau_{\text{arr}}(J)$	$\lvert J\rvert$	$\tau_{\text{e}}(J)$
J_c	10:47:02	12:52:00	4	10:45:00
J_a	10:45:48	13:52:00	3	10:45:00
J_h	11:47:02	13:52:00	4	10:47:03
J_d	10:47:02	14:39:25	3	10:45:49
J_f	11:45:48	14:52:00	3	10:47:03
J_k	12:47:02	14:52:00	4	11:47:03
J_b	10:45:48	15:39:25	2	10:45:00
J_i	11:47:02	15:39:25	3	10:47:03
J_l	12:45:48	15:52:00	3	11:48:03
J_m	13:46:03	15:52:00	4	12:47:03
J_e	11:10:48	16:09:25	2	10:45:49

(b) Profile ordered by arrival time.

	$\tau_{\text{dep}}(J)$	$\tau_{\text{arr}}(J)$	$\lvert J\rvert$	$\tau_{\text{e}}(J)$
J_c	10:47:02	12:52:00	4	10:45:00
J_a	10:45:48	13:52:00	3	10:45:00
J_b	10:45:48	15:39:25	2	10:45:00
J_d	10:47:02	14:39:25	3	10:45:49
J_e	11:10:48	16:09:25	2	10:45:49
J_h	11:47:02	13:52:00	4	10:47:03
J_f	11:45:48	14:52:00	3	10:47:03
J_g	11:45:48	16:39:25	2	11:10:49
J_i	11:47:02	15:39:25	3	11:45:49
J_j	12:10:48	17:09:25	2	11:45:49

(c) Profile ordered by earliest optimal departure time (τ_{e}).

Table 3.1 – Example profile for traveling from Meggen, Schlössli to Weisslingen, Mühle with the departure time range [10:45, 14:00]. Only the journeys on the first two pages of the profile for a page size of $n = 5$ are listed. The horizontal line indicates a page break. Each table shows the journeys for a different ordering.

3.2.2 Trade-offs between Different Orderings

Sorting journeys by specific orderings has different trade-offs, especially in the context of pagination. The following example query on the Swiss railway network gives a good illustration. Consider traveling from Meggen, Schlössli to Weisslingen, Mühle at

a minimum departure time of 10:45 a.m. Table 3.1 shows the journeys on the first two pages of a profile for the time interval from 10:45 to 14:00 with a page size of 5 and different orderings.

Ordering the pages by departure time (Table 3.1a) might seem intuitive at first glance. The main advantage is that each page corresponds to a minimum departure time. The first page contains the earliest departing journeys after 10:45. The last journey on page 1, J_e, departs at 11:10:48. Hence, the second page can be interpreted as the first page of a different profile query with 11:10:49 as minimum departure time. This allows a stateless implementation of end user applications, e.g., web interfaces: The computation of each page also yields the minimum departure time for the following page (departure time of the last journey on the page plus ε). This can be used as a URL parameter for the next page. The server does not need to store any state about the specific user's query. However, in this example J_h, which departs at 11:47:02 and arrives at 13:52:00 (using 4 trips), does not appear on the first page, even though it features the second earliest arrival time. Users will likely ignore it, as they do not expect such a journey being listed after journeys arriving as late as 16:09:25 (J_e), and hence do not look at the second page.

If journeys are ordered by their arrival time (Table 3.1b) we encounter the inverse problem: Early departing journeys might not appear on the first page, e.g., J_b, which departs at 10:45:48 and involves 2 trips. In fact, no journey with two trips is listed on the first page. However, early arriving journeys presumably have a higher relevance for users, especially, if they want to choose the latest departing journey such that the still arrive before a certain time. Note that ordering by arrival time lacks the statelessness of departure time order pagination.

We introduced the criterion of the earliest optimal departure time (Definition 3.1) to address the issue of journeys with an early departure or arrival time being pushed to rather late positions in the profile. It represents a compromise between departure time ordering and arrival time ordering, respectively. If this ordering is used, journeys are grouped by the first time for which they are part of a bicriteria-optimal solution. In other words: Every position i in a profile, that corresponds to an earliest optimal departure time τ_i, answers the question "Which journeys are worth considering if I want to depart at τ or later?", as those journeys are precisely the ones listed from position i onward.

Ordering by earliest optimal departure time is stateless in the same sense as ordering by departure time: Let τ be the earliest optimal departure time of the latest journey on a page. Then the next page is identical to first page of a profile query with $\tau_{\min} = \tau + \varepsilon$.

Algorithm 3.1: PAGINATION

Input: Public transit network $(\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, source stop $s \in \mathcal{S}$, target stop $t \in \mathcal{S}$, departure time range $I = [\tau_{\min}, \tau_{\max}] \subseteq \Pi$.

```
1 INITIALIZE()
2 while ¬PROFILECOMPLETE() do
3     P ← ∅
4     while |P| < n ∧ ¬PROFILECOMPLETE() do
5         P ← P ∪ COMPUTEJOURNEYS()
6     output P
7     if user requests next page, continue. else break
```

3.2.3 Pagination Framework

In end-user applications the use of profile algorithms is follows a certain structure. The profile algorithm must compute a single page of journeys, then stop computation and continue computing later journeys when the user requests to see the next page. Given a profile algorithm that computes journeys in ascending order with regard to an ordering $(\preceq, \approx)$ (see Section 3.1.2), pages are retrieved using the following scheme (as described by Algorithm 3.1): After the algorithm is initialized (INITIALIZE()), journeys are computed iteratively. As long as not all journeys of the profile have been computed, which is checked by PROFILECOMPLETE(), the procedure COMPUTEJOURNEYS is called repeatedly. COMPUTEJOURNEYS outputs a set S of journeys such that $\forall J, J' \in S : J \approx J'$. The journeys in S are then added to the current page P. When P is full, i.e., $|P| \geq n$ with page size n, it can be output and the computation can be continued.

Formally, a profile algorithm is considered pagination-conform for an ordering $(\preceq, \approx)$, if the following two conditions hold true. First, journeys returned when calling COMPUTEJOURNEYS precede journeys returned by later calls of COMPUTEJOURNEYS according to $\preceq$. Second, all journeys that are equal with regard to $\approx$ are returned by the same call to COMPUTEJOURNEYS. This property ensures that journeys with the same arrival, departure, or earliest optimal time are listed on the same page.

All algorithms presented in this work that rely on modified versions of Alternating RAPTOR (Section 2.5.2) as well as the rRAPTOR variant that computes journeys ordered by arrival time conform to the above scheme. However, variants, namely RR_{dep} and RR_{e}, that mix rRAPTOR with Alternating RAPTOR compute all journeys of a page at once.

4. Algorithms

Certainly, a trivial approach to achieve pagination as described in Chapter 3 is to use an arbitrary profile algorithm, e.g., rRAPTOR, to first compute the whole profile at once. The profile is then sorted according to the required criterion and split into pages afterwards. However, this would lead to a very high response time before the first page can be output. Furthermore, not all pages might eventually be requested by the user and thus journeys are computed superfluously. Instead, we strive to already compute journeys in the required order, such that the computation cost is split over the pages.

In this chapter, we show how Alternating RAPTOR and rRAPTOR can be adapted to compute profiles page-wise for each of the orderings we introduced in Section 3.1.2.

4.1 Alternating RAPTOR-based Approaches

The Alternating RAPTOR (AR) algorithm (Section 2.5.2) can be modified in a rather straightforward way in order to obtain different orderings. We present three variants, AR_{arr}, AR_{dep}, and AR_{e}, which produce the orderings $\preceq_{\text{arr}}$, $\preceq_{\text{dep}}$, and $\preceq_{\text{e}}$, respectively. Each variant uses the principle of running forward RAPTOR searches to find earliest arrival times and then running backward RAPTOR searches to find the latest departing journeys for these arrival times. The order of the returned journeys results solely from the order in which forward and backward searches are performed. Since Alternating RAPTOR uses the RAPTOR algorithm as a building block for solving the Bicriteria Problem, the concepts described hereafter work with any other basic algorithm that computes bicriteria-optimal journeys as well.

Algorithm 4.1: AR_{arr}

Input: Public transit network $(\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, source stop $s \in \mathcal{S}$, target stop $t \in \mathcal{S}$, departure time range $I = [\tau_{\min}, \tau_{\max}] \subseteq \Pi$.

Data: Priority queue Q of arrival labels (τ, k) ordered by arrival times τ

Function INITIALIZE():
  FORWARDSEARCH($s, t, \tau_{\min}, \tau_{\min}$)

Function PROFILECOMPLETE():
  return $Q = \emptyset$

Function COMPUTEJOURNEYS():
  $P \leftarrow \emptyset$
  $(\tau_{\text{arr}}, k) \leftarrow Q$.MIN()
  $B \leftarrow$BACKWARDRAPTOR(t, s, τ_{arr})
  do
    $(\tau_{\text{arr}}, k) \leftarrow Q$.POPMIN()
    $J \leftarrow$ journey in B with k trips
    $P \leftarrow P \cup \{J\}$
    FORWARDSEARCH($s, t, \tau_{\text{dep}}(J) + \varepsilon$)
  while $Q \neq \emptyset \wedge \tau_{\text{arr}} = \tau_{\text{arr}}(Q.\text{MIN}())$
  return P

Function FORWARDSEARCH($s, t, \tau_{\min}, \tau_{\text{arr}}$):
  if already performed forward search for $\tau_{\min}$ **then return**
  $\mathcal{J} \leftarrow$ FORWARDRAPTOR($s, t, \tau_{\min}$)
  for $J \in \mathcal{J}$ **do**
    if $\tau_{\text{arr}}(J) \geq \tau_{\text{arr}}$ **then** Q.INSERT($(\tau_{\text{arr}}(J), |J|)$)

4.1.1 Ordering by Arrival Time (AR_{arr})

Alternating RAPTOR orders journeys already by their arrival time, i.e., $\preceq_{\text{arr}}$. In Algorithm 4.1 we give pseudocode that shows how it can be adapted to obey the generic scheme for pagination algorithms. The algorithm maintains a priority queue that contains arrival labels, i.e., tuples (τ_{arr}, k) of arrival time τ_{arr} and number of trips k. When COMPUTEJOURNEYS is called, a backward search is performed for the arrival time τ_{arr} of the earliest arrival label in Q. Then all labels with the same arrival time τ_{arr} are extracted from Q and the respective latest departing journeys are looked-up in the backward search results. These journeys are then added to the profile. For each

such journey J, a forward search for the subsequent departure time, i.e., $\tau_{\text{dep}}(J) + \varepsilon$ is performed. Let $\mathcal{J}$ be the set of all journeys that are found by the forward search. For each journey in $\mathcal{J}$ that arrives *not before* τ_{arr} the respective arrival label is then inserted into Q. Note that the forward search, however, can find journeys that arrive exactly at τ_{arr}. In this case the respective arrival labels are processed within the same call of computeJourneys.

It remains to show why we can ignore all journeys in $\mathcal{J}$ that arrive before τ_{arr}. We do so by showing that every "new" journey $J \in \mathcal{J}$, i.e., a journey that has not previously been found by another forward search, arrives not before τ_{arr}. Since J is found for the first time, there must exist some journey J' with $\tau_{\text{arr}}(J') = \tau_{\text{arr}}$ and $\tau_{\text{dep}}(J') < \tau_{\text{dep}}(J)$ that dominates J for any point of time up to $\tau_{\text{dep}}(J')$, inclusively. Hence J' has equal or less trips than J. If $|J| = |J'|$, J must arrive later than τ_{arr}, since the backward search for τ_{arr} has found J' as the latest departing journey with $|J|$ trips arriving at τ_{arr}. If $|J| > |J'|$ and $\tau_{\text{arr}}(J) < \tau_{\text{arr}}$, J' would not have dominated J, hence J must depart at τ_{arr} or later.

4.1.2 Ordering by Departure Time (AR_{dep})

We can modify Alternating RAPTOR in order to compute journeys by their departure time. The main idea is that the priority queue holds already found journeys ordered by their departure time instead of arrival time as in AR_{arr}.

Initially, we perform a forward RAPTOR search starting at s with departure time τ_{min}. This forward search solves the Bicriteria Problem for τ_{min}. For every journey J we find we perform a backward RAPTOR search starting at t for $\tau_{\text{arr}}(J)$ to find the latest departing journey J' with $|J| = |J'|$ trips that arrives at $\tau_{\text{arr}}(J)$. Thus J' is part of the profile and we insert it into the queue.

On every call of computeJourneys, we remove all journeys with minimal departure time τ_{dep} from the queue. We add these journeys to the profile. We then again perform a forward search for $\tau_{\text{dep}} + \varepsilon$ and backward searches for the respective arrival times as we did before for τ_{min}. To achieve a correct ordering of the returned journeys, we must ensure that only journeys departing strictly later than τ_{dep} are added to the queue. Since we start the forward search with minimum departure time $\tau_{\text{dep}} + \varepsilon$, we can only find journeys departing after τ_{dep}. Note that when the forward search finds a journey J and we perform a backward search for $\tau_{\text{arr}}(J)$, we are only interested in a journey J' with $|J|$ trips, since other journeys could depart before $\tau_{\text{dep}} + \varepsilon$ or are not part of a profile.

Algorithm 4.2: AR_{dep}

Input: Public transit network $(\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, source stop $s \in \mathcal{S}$, target stop $t \in \mathcal{S}$, departure time range $I = [\tau_{\min}, \tau_{\max}] \subseteq \Pi$

Data: Priority queue Q of journeys ordered by $\preceq_{\text{dep}}$, Map M of departure time to set of journeys

```
1  Function initialize():
2      forwardBackwardSearch(τ_min)
3  Function profileComplete():
4      return Q = ∅
5  Function computeJourneys():
6      P ← ∅
7      do
8          J ← Q.min()
9          P ← P ∪ {J}
10         Q.popMin()
11     while Q ≠ ∅ ∧ τ_dep(J) = τ_dep(Q.min())
12     if τ_dep ≤ τ_max then forwardBackwardSearch(τ_dep(J) + ε)
13     return P
14 Function forwardBackwardSearch(τ_min):
15     𝒥 ←forwardRaptor(s, t, τ_min)
16     for J ∈ 𝒥 do
17         if τ_arr(J) ∉ M then
18             M[τ_arr(J)] ←backwardRaptor(t, s, τ_arr(J))
19         if M[τ_arr(J)][|J|] ≠ ⊥ then
20             J′ ← M[τ_arr(J)][|J|]
21             M[τ_arr(J)][|J|] ← ⊥
22             Q.insert(J′)
```

The algorithm as described so far can find the same journey multiple times. For example, consider a scenario, where for one trip there is a possible journey J that departs at a rather late time $\tau_{\text{dep}} \in I$. In this case all forward searches before τ_{dep} will find J. To avoid adding it to the profile multiple times, we have to keep track of journeys that have already been added to the profile. This can be done using a set data structure.

However, caching backward search results, as described in the following paragraph, allows us to mark journeys that have already been considered.

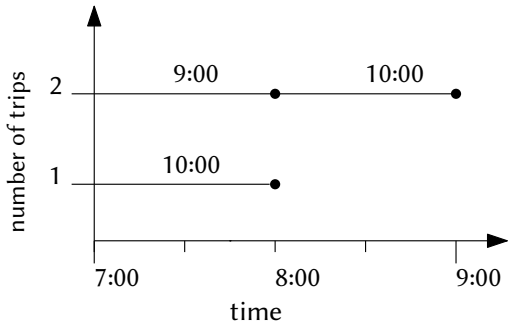


Figure 4.1 – This profile contains one journey with one trip: 08:00 – 10:00 ($J_{1,1}$), and two journeys with two trips: 08:00 – 09:00 ($J_{2,1}$), 09:00 – 10:00 ($J_{2,2}$). If backward search results are not cached, the backward search for 10:00 would be performed twice.

Caching Backward Search Results. It can occur that we perform multiple backward searches for the same arrival time. For an example, consider the profile depicted in Figure 4.1. There is an s-t-journey from 08:00 to 10:00 ($J_{1,1}$), and there are two additional s-t-journeys from 08:00 to 09:00 ($J_{2,1}$) and from 09:00 to 10:00 ($J_{2,2}$), respectively. In this example, the first forward search would find $J_{1,1}$ and $J_{2,1}$ and perform backward searches for 10:00 and 09:00, respectively. The next forward search for 08:00:01 would find $J_{2,2}$ and would then perform a backward search for 10:00 again. However, we can avoid this by caching the results of every backward search. To do this, we maintain a data structure M that maps each arrival time to a list of journeys. When performing a backward search for an arrival time τ_{arr}, the resulting journeys are stored in $M[\tau_{\text{arr}}]$. This way, we can avoid running an identical backward search later. Instead, if a journey J was found by the forward search, and $\tau_{\text{arr}}(J)$ is already stored in M, we look up the respective journey $J' \in M[\tau_{\text{arr}}]$ with $|J'| = |J|$ and add it to the queue. We then remove J' from $M[\tau_{\text{arr}}]$ to indicate that this journey was already found. Since a profile can contain at most one journey per arrival time and number of trips, we thereby ensure that J' is not added to the result twice. Furthermore, the total number of backward searches is decreased compared to AR_{arr}, since for any arrival time of a journey in the profile at most one backward search is performed.

4.1.3 Ordering by Earliest Optimal Departure Time (AR_{e})

We can adapt AR_{dep} slightly in order to output journeys according to their earliest optimal departure time, i.e., journeys that are ordered by $\preceq_{\text{e}}$. Recall that we defined the

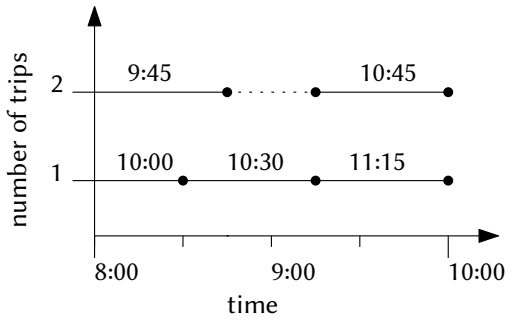


Figure 4.2 – This profile contains three journeys with one trip: 8:30 – 10:00 ($J_{1,1}$), 9:15 – 10:30 ($J_{1,2}$), 10:00 – 11:15 ($J_{1,3}$), and two journeys with two trips: 8:45 – 9:45 ($J_{2,1}$), 10:00 – 10:45 ($J_{2,2}$). Note that $J_{2,2}$ is dominated by $J_{1,2}$ before 9:15.

earliest optimal departure time $\tau_e(J)$ of a journey J as the earliest time after which on J is a bicriteria-optimal journey. If this time lies before the start of the query interval $\tau_{\min}$, $\tau_e(J)$ is defined as $\tau_{\min}$.

For an example that illustrates how τ_e results in different constellations, see Figure 4.2. The journeys that are found by a forward search for $\tau_{\min}$ have an earliest optimal departure time $\tau_e = \tau_{\min}$ per definition (e.g. $J_{1,1}$ and $J_{2,1}$ in Figure 4.2). For any other journey J in a profile, J's earliest optimal departure time $\tau_e(J)$ is the point at which the last journey J' that dominated J previously departs, i.e., $\tau_e(J) = \tau_{dep}(J') + \varepsilon$. This journey, J', can either have the same number of trips as J (such as with $J_{1,1}$ and $J_{1,2}$) or a lower number of trips, e.g., $J_{1,2}$ dominates $J_{2,2}$ for all times until 9:15. We observe that in any case the earliest optimal departure time $\tau_e(J)$ is identical to the departure time with which the first forward search that found J was performed.

AR_{dep} performs a forward search for $\tau_{dep} + \varepsilon$ when τ_{dep} is extracted from the queue. The journeys found by this forward search and the subsequent backward searches hence have an earliest optimal departure time $\tau_e = \tau_{dep} + \varepsilon$ and can immediately be added to the profile. The departure times are extracted from the queue in ascending order, which also ensures a correct ordering of the journeys by earliest optimal departure time. Note that the only difference between AR_e and AR_{dep} is that for AR_e journeys are added to the result before they are added to the priority queue, whereas for AR_{dep} journeys are added when they are taken from the queue. Hence, for AR_e it is sufficient to maintain a queue which does not store journeys, but only departure times instead.

4.2 rRAPTOR-based Approaches

As already stated, Alternating RAPTOR was originally developed to answer profile queries on public transit networks that allow unlimited walking between stops. In this scenario, any departure of a trip tr at a stop v results in a possible departure at the source stop s for the time $\tau_{\text{dep}}(\text{tr}, v) - \tau_{\text{buf}}(v) - \ell(s, v)$. Due to the huge amount of possible departures, using rRAPTOR (see Section 2.5.1), which scans all departures at s from latest to earliest, becomes inefficient. However, when applied to networks with limited walking, rRAPTOR performs faster than Alternating RAPTOR.

rRAPTOR collects possible departures at s and scans them from latest to earliest departure time. After the departure for time $\tau \in I$ is scanned, the profile is complete for the interval $[\tau, \tau_{\max}]$. In the context of pagination, however, we are interested in computing journeys from earliest to latest. The rough idea is to run rRAPTOR on a reverted route network (see Section 2.5.2 for a definition of reverted networks). In a first step, we collect all possible arrival times at the target stop t (in the original network) into a set Ψ and sort them by their arrival time in *ascending order*. Then, for every arrival time $\tau_{\text{arr}} \in \Psi$, we perform a RAPTOR search on the reverted network that finds the latest departure time at s for arrival time τ_{arr} and for each numbers of trips. All RAPTOR searches work on the same round table such that the self-pruning property just as in rRAPTOR is ensured. Here, a round table entry $R[k][v]$ denotes the *latest departure time* at which one must depart at v to reach t with k trips not after the current arrival time. After τ_{arr} has been scanned, the profile already contains all optimal journeys arriving in the interval $[\tau_{\min}, \tau_{\text{arr}}]$.

In this section, we describe in detail how the aforementioned idea can be used to obtain profiles in different orderings. While the algorithm for ordering by arrival time (RR_{arr}) follows straightforwardly, ordering profiles by their departure time or earliest optimal departure time requires some additional work.

4.2.1 Ordering by Arrival Time (RR_{arr})

Pseudocode for the RR_{arr} algorithm is given in Algorithm 4.3. Because arrivals are scanned in ascending order, the results are naturally ordered by their arrival time.

Collecting Arrivals

Given a departure time interval $I = [\tau_{\min}, \tau_{\max}]$, we start with determining the interval $I' = [\tau_{\text{arr}}^{\min}, \tau_{\text{arr}}^{\max}]$ of possible arrival times. The earliest possible arrival time $\tau_{\text{arr}}^{\min}$ at t is the earliest possible arrival time of a journey starting at s no earlier than $\tau_{\min}$. We find $\tau_{\text{arr}}^{\min}$ by performing a forward RAPTOR search from s for $\tau_{\min}$. The latest possible arrival

Algorithm 4.3: RR_{arr}

Input: Public transit network $(\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, source stop $s \in \mathcal{S}$, target stop $t \in \mathcal{S}$, departure time range $I = [\tau_{\min}, \tau_{\max}] \subseteq \Pi$

Data: sorted list of arrivals A, set $\mathcal{J}_{\text{latest}}$ bicriteria-optimal journeys for $\tau_{\max}$, RAPTOR algorithm instance backwardRAPTOR, latest scanned arrival time τ_{arr}

Function initialize():
- $F \leftarrow$ forwardRAPTOR$(s, t, \tau_{\min})$
- $\mathcal{J}_{\text{latest}} \leftarrow$ latest departing journeys for arrival times in F
- $\tau_{\text{arr}}^{\min} \leftarrow \min_{J \in F} \tau_{\text{arr}}(J)$
- $L \leftarrow$ forwardRAPTOR$(s, t, \tau_{\max})$
- $\tau_{\text{arr}}^{\max} \leftarrow \max_{J \in L} \tau_{\text{arr}}(J)$
- $A \leftarrow \left\{(\tau, s) \,\middle|\, \text{tr} \in \mathcal{T} \wedge \tau = \tau_{\text{arr}}(\text{tr}, t) \wedge \tau \in [\tau_{\text{arr}}^{\min}, \tau_{\text{arr}}^{\max}]\right\}$
- $A \leftarrow A \cup \left\{(\tau, v) \,\middle|\, v \in \mathcal{S} \wedge (v, t) \in \mathcal{F} \wedge \text{tr} \in \mathcal{T} \wedge \tau = \tau_{\text{arr}}(\text{tr}, v) + \ell(v, t) \wedge \tau \in [\tau_{\text{arr}}^{\min}, \tau_{\text{arr}}^{\max}]\right\}$
- sort A by τ

Function profileComplete():
- **return** $A = \emptyset$

Function computeJourneys():
- **do**
  - $(\tau_{\text{arr}}, v) \leftarrow A$.popMin()
  - backwardRAPTOR.setArrivalTime$(v, \tau_{\text{arr}} - \ell(v, t))$
- **while** $A \neq \emptyset \wedge \tau_{\text{arr}} = \tau_{\text{arr}}(A.\min())$
- backwardRAPTOR.setArrivalTime(t, τ_{arr})
- $\mathcal{J} \leftarrow$ backwardRAPTOR.run() `// No initial transfers, preserve round table`
- $P \leftarrow \emptyset$
- **for** $J \in \mathcal{J}$ **do**
  - **if** $\tau_{\text{dep}}(J) \geq \tau_{\min} \wedge J$ not dominated by any journey in $\mathcal{J}_{\text{latest}}$ **then**
    - $P \leftarrow P \cup \{J\}$
- $P \leftarrow P \cup \{J \in \mathcal{J}_{\text{latest}} \mid \tau_{\text{arr}}(J) = \tau_{\text{arr}}\}$
- **return** P

time $\tau_{\text{arr}}^{\max}$ at t is the arrival time of the latest arriving journey that is bicriteria-optimal for $\tau_{\max}$. This is the journey with the fewest trips that is found by a forward RAPTOR

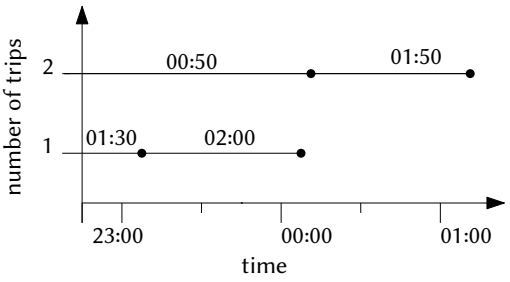


Figure 4.3 – Depicted are two journeys with one trip, $J_{1,1}$ and $J_{1,2}$, from 23:05 to 01:30 and from 00:05 to 02:00, respectively, and journeys with two trips from 00:10 to 00:50 ($J_{2,1}$) and from 1:10 to 01:50 ($J_{2,2}$). For latest departure time $\tau_{\max}$ = 00:00, $J_{1,1}$, $J_{1,2}$ and $J_{2,1}$ are part of a profile, but $J_{2,2}$ is not.

search from s for $\tau_{\max}$. We then iterate over all trips tr arriving at t and save the arrival time $\tau_{\text{arr}}(\text{tr}, t) \in I'$. To avoid relaxing initial footpaths for every backward RAPTOR search we also scan the trips that arrive at stops v with $(v, t) \in \mathcal{F}$ and collect the arrival time at t which is $\tau_{\text{arr}}(\text{tr}, v) + \ell(v, t)$ (for a more precise description, see Section 2.5). We skip any arrival time outside I'. Arrivals are stored in a list A. We sort A ascendingly by arrival time.

Alternatively, arrivals can be precomputed for every stop $v \in \mathcal{S}$ and the total time interval Π. For every profile query, we can then find the relevant range of arrivals I' by performing a binary search for $\tau_{\text{arr}}^{\min}$ and $\tau_{\text{arr}}^{\max}$. However, this approach requires $O(\sum_{v \in \mathcal{S}}(|\mathcal{T}_v| + \sum_{(u,v) \in \mathcal{F}} |\mathcal{T}_u|))$ space, where $\mathcal{T}_v$ is the set of trips that serve v. For every set of stops $C \subseteq \mathcal{S}$ that form a clique in the footpath graph ($\forall u, v \in C : (u, v) \in \mathcal{F}$), $O(\sum_{v \in C} |\mathcal{T}_v| \cdot |C|)$ arrivals must be stored.

Processing Arrivals

Scanning journeys is straightforward: On every call of COMPUTEJOURNEYS we consider all arrivals $(\tau_{\text{arr}}, v) \in A$ with the minimum arrival time τ_{arr}. For each arrival label we update the round table entry $R[0][v]$ of v with arrival time $\tau_{\text{arr}} - \ell(v, t)$ (using the function SETARRIVALTIME). This allows us to skip the initial transfer phase as we already discussed in the context of rRAPTOR in Section 2.5.1. We then perform a backward RAPTOR search with s as target. The RAPTOR search has found a new journey if it updated the round table entry $R[k][s]$ for the source stop s and some round k. If there is any such journey, we add it to the profile.

Optimal Journeys at the End of the Interval

In practice, application users typically do not have a latest departure time in mind when they are looking for a journey. Thus, $\tau_{\max}$ is commonly set to the end of the day. However, the definition of the range problem (Definition 2.4) requires that a profile contains all bicriteria-optimal journeys for every time in a finite time interval I. In order to find journeys that arrive after $\tau_{\max}$, we chose the latest arrival time of a journey which is bicriteria-optimal for $\tau_{\max}$ as the upper bound $\tau_{\text{arr}}^{\max}$ for the arrival interval. It is thus possible that multiple journeys departing *after* $\tau_{\max}$ for a certain number of trips are found. Consider the example depicted in Figure 4.3. 02:00 would be the latest arrival time $\tau_{\text{arr}}^{\max}$ and rRAPTOR would thus find $J_{2,2}$ even though it is not part of the profile. To avoid this, we ignore all journeys found by rRAPTOR with a departure time greater than or equal to $\tau_{\max}$. We compute all journeys $\mathcal{J}_{\text{latest}}$ that are bicriteria-optimal for $\tau_{\max}$ explicitly by an extra forward search for $\tau_{\max}$ and respective backward searches. We output a journey in $\mathcal{J}_{\text{latest}}$ when its respective arrival time is scanned. We must also disregard every journey arriving after $\tau_{\max}$ if it is dominated by a journey in $\mathcal{J}_{\text{latest}}$.

4.2.2 Ordering by Departure Time (RR_{dep})

Ordering journeys by arrival time is an inherent property of RR_{arr}, which was discussed in the last section. In this section, we present an approach, RR_{dep}, that augments rRAPTOR with forward and backward searches as in Alternating RAPTOR to find journeys ordered by their departure time $\preceq_{\text{dep}}$. RR_{dep} uses a heuristic approach: While RR_{dep} always computes correct solutions, its efficiency depends on properties of the profile. More precisely, we use the observation that for a *partial profile*, i.e., only for those journeys of a profile with the same number of trips k, the relative order of journeys is the same regardless whether the profile is ordered by arrival or by departure time. Let J and J' be two journeys in a partial profile. Say J departs after J'. Then J must also arrive later than the J' since it would otherwise dominate J' and they would not be both in the profile. The intuition behind RR_{dep} is to compute a variable number of journeys using RR_{arr} "on the off chance" in a first phase. In a second phase, the partial profiles, which consist of journeys found in the first phase, are merged. To do this, we take in each step the earliest departing journey among all partial profiles. If the next earlier journey of a partial profile is unknown, we determine it by forward and backward RAPTOR searches such as in Alternating RAPTOR.

Pseudocode for RR_{dep} is given in Algorithm 4.4. One can think of RR_{dep} as a wrapper for RR_{arr}. It does not satisfy the scheme for pagination algorithms we introduced in Section 3.2.3. Instead, the computation for each page is split into two phases: In the first phase,

the algorithm computes a number of journeys m by using a reversed rRAPTOR as we did for RR_{arr}. The variable $m \in \mathbb{N}$ is a tuning parameter that must be no less than the page size n. Sensible choices for m are discussed in a later paragraph. The journeys we found are stored in a priority queue Q ordered by their *departure time*. For each number of trips k we also store the departure time $\tau_{\text{dep}}^{\text{latest}}(k)$ of the latest journey with k trips we have found so far.

In the second phase, we consider the earliest departing journey J in Q as a candidate for the next earliest journey to be output. Before adding J to the page P, we check for every possible (for details, see below) number of trips k other than $|J|$ whether we have already found a journey with k trips that departs later than J, i.e., $\tau_{\text{dep}}^{\text{latest}}(k) > \tau_{\text{dep}}(J)$. If so, we can be sure that all journeys with k trips have already been output and J must now be output. If not, we perform a forward and backward RAPTOR search for $\tau_{\text{dep}}^{\text{latest}}(k)$. We repeat this until these searches either find a journey J' with k trips departing after $\tau_{\text{dep}}^{\text{latest}}(k)$ or discovered that no journey that departs earlier than J exits. If a journey J' is found, it is added to Q. If J' departs before J, we further consider J' as the new candidate. For higher number of trips $k > |J'|$ we thus have to compare $\tau_{\text{dep}}^{\text{latest}}(k)$ against $\tau_{\text{dep}}(J')$. The earliest departing journey is then added to the page and we repeat the above procedure until at least n journeys have been added to the page.

In the following paragraphs, we discuss special cases that can occur in certain steps of the algorithm as well as optimization techniques in greater detail.

Filling up Partial Profiles

To add a journey to a partial profile for k trips we perform a forward RAPTOR search starting from s for departure time $\tau_{\text{dep}}^{\text{latest}}(k) + \varepsilon$ (see Algorithm 4.5). There are several cases to consider:

- A journey J with exactly k trips is found. We then perform a backward RAPTOR search for $\tau_{\text{arr}}(J)$ to find the latest departing journey J' arriving at τ_{arr}. We add J' to Q and update $\tau_{\text{dep}}^{\text{latest}}(k)$ to $\tau_{\text{dep}}(J')$. However, there is not necessarily an optimal journey with k trips for $\tau_{\text{dep}}^{\text{latest}}(k)$ (see next case).
- A journey J with $k' < k$ trips is found. We determine the latest departing journey J' for $\tau_{\text{arr}}(J')$ and set $\tau_{\text{dep}}^{\text{latest}}(k) = \tau_{\text{dep}}(J')$. Note that we here use $\tau_{\text{dep}}^{\text{latest}}(k)$ to indicate the latest time for which we have already found an optimal journey with *at most* k trips. We repeat this procedure until either a journey with k trips was found or it is guaranteed that no journey with k trips exists that departs before τ_{dep}, i.e., until $\tau_{\text{dep}}^{\text{latest}}(k) > \tau_{\text{dep}}(\min(Q))$.

Algorithm 4.4: RR_{dep}

Input: Public transit network $(\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$, source stop $s \in \mathcal{S}$, target stop $t \in \mathcal{S}$, departure time range $I = [\tau_{\min}, \tau_{\max}] \subseteq \Pi$, page size n, parameter m

Data: Algorithm RR_{arr}, priority queue Q of journeys ordered by departure time, maximum number of trips $k_{\max}$, vector $\tau_{\text{dep}}^{\text{latest}}$ of size $k_{\max}$

```
1  Function initialize():
2      k_max ← 0
3      RR_arr.initialize()
4  Function profileComplete():
5      return RR_arr.profileComplete ∧ Q = ∅
6  Function computePage():
7      while ¬RR_arr.profileComplete() ∧|Q| < m do
8          𝒥 ← RR_arr.computeJourneys()
9          for J ∈ 𝒥 do
10             Q.insert(J)
11             if |J| > k_max then
12                 for k_max < k < |J| do τ_dep^latest(k) ← τ_min
13                 k_max ← |J|
14             τ_dep^latest(|J|) ← τ_dep(J)
15     P ← ∅
16     while Q ≠ ∅ ∧ (|P| < n ∨ τ_dep(nextJourney()) = max_{J∈P}(τ_dep(J)) do
17         P ← P ∪ {nextJourney()}
18         Q.popMin()
19     return P
20 Function nextJourney():
21     J ← Q.min()
22     k ← k_max
23     if τ_arr(J) ≤RR_arr.τ_arr then k ← |J|
24     for k' = 1, ..., k do
25         while τ_dep^latest(k) ≤ τ_max ∧ τ_dep^latest k ≤ τ_dep(J) do
26             if forwardBackwardSearch(k) then break
27         J ← Q.min()
28         if τ_arr(J) ≤RR_arr.τ_arr then k ← |J|
29     return J
```

Algorithm 4.5: forwardBackwardSearch

```
1  Function forwardBackwardSearch(k):
2      J ← forwardBackwardSearch(τ_dep^latest(k) + ε, k)
3      if ∃J ∈ J with |J| = k then
4          J' ← backwardSearch(τ_arr(J), k)
5          τ_dep^latest(k) ← τ_dep(J');
6          if τ_t(J) < ℓ(s, t) then
7              Q.insert(J')
8              return true
9      else if ∃J ∈ J with |J| < k then
10         J' ← backwardSearch(τ_arr(J), |J|)
11         τ_dep^latest(k) ← τ_dep(J');
12     else
13         τ_dep^latest(k) ← ∞
14     if ∃J ∈ J with |J| > k_max then
15         J' ← backwardSearch(τ_arr(J), k)
16         τ_dep^latest(k) ← τ_dep(J');
17         k_max ← |J|
18         if τ_t(J) < ℓ(s, t) then
19             Q.insert(J')
20     return false
```

- If the forward search finds no journey with k or less than k trips, we set $\tau_{\text{dep}}^{\text{latest}}(k)$ trip to ∞.

- Any journey found with k' trips, where $k < k' \leq k_{\max}$, can be ignored, as it cannot be optimal for k trips. If the partial profile for k' trips is empty, there will be a separate forward and backward search for k' trips with departure time $\tau_{\text{dep}}^{\text{latest}}(k') + \varepsilon$.

- However, it is possible that a journey J with $k' > k_{\max}$ trips is found. We then find the respective latest departing journey J' and add it to Q.

Optimizing Forward and Backward Searches

Note that multiple forward searches are likely to be performed for the same departure time. We thus cache the results of a forward or backward search in an associative data structure, indexed by the respective departure time. In Algorithm 4.4 the look-up and caching is implicitly done by FORWARDSEARCH. The same applies to backward searches which are cached by BACKWARDSEARCH.

As we discussed in Section 2.5.2, finding optimal journeys with forward and backward RAPTOR searches can become inefficient if one can walk directly from s to t. To avoid this, we adapt the RAPTOR search to not find journeys with zero trips. If the travel time of a journey J is longer than the direct walking time from s to t, we still update $\tau_{\text{dep}}^{\text{latest}}(|J|)$ with its departure time $\tau_{\text{dep}}(J)$, but do not add J to the page.

Checking for Earlier Journeys

To find the next earliest departing journey (see Function NEXTJOURNEY in Algorithm 4.4), we first extract the earliest departing journey J from the queue. We then check for all numbers of trips $k' < k_{\max}$ other than $k = |J|$ if there can possibly be a journey departing earlier than J we have not found yet. To do this, we test whether $\tau_{\text{dep}}^{\text{latest}}(k') > \tau_{\text{dep}}(J)$, and perform a forward and backward search otherwise. However, this can be further optimized. Let τ_{arr} be the arrival time of the latest arrival that was scanned by RR_{arr}. If J arrives before τ_{arr}, it is sufficient to consider numbers of trips $k' < k$. We prove this by showing that any optimal journey with more trips than k departing earlier than J has already been found.

Suppose there is another optimal journey J' with more than k trips that departs before J. J' must arrive before $\tau_{\text{arr}}(J)$, since otherwise it would be dominated by J and thus would not be part of the profile. If J' arrives before J, it must in particular arrive before the latest scanned arrival time τ_{arr} and thus has already been found by RAPTOR. Hence J' has been already output before J. Note that journeys that were found in the second phase, i.e., by a forward and backward search, can possibly arrive after τ_{arr}. If so, we do have to check for all number of trips k for which we have previously found journeys if $\tau_{\text{dep}}^{\text{latest}}(k) < \tau_{\text{dep}}(J)$.

Look-ahead Factor (Choosing m)

A worst case scenario for RR_{dep} is as follows: Let $X_{\text{arr}} \subset P$ be the subset of a profile P that contains the m earliest arriving journeys in P. Suppose all other journeys in the profile depart before the journeys in X_{arr} (and thus have a lower number of trips). In this scenario, RR_{dep} would find the journeys in X_{arr} in the first phase using RAPTOR

and all journeys in $P \backslash X_{\text{arr}}$ in the second phase using forward and backward searches. If $|P| \gg m$ the total cost of RR_{dep} is near the combined cost for running both RR_{arr} and Alternating RAPTOR for the complete profile.

However, in real-world profiles, the set of earliest arriving journeys X_{arr} is likely to contain some of the earliest departing journeys as well. In the aforementioned worst case scenario, the earliest arriving journeys in the profile are also the latest departing ones and thus the journeys with the shortest travel times. Real-world travel times, however, are bounded by limitations such as the average speed of trains and can thus not be arbitrarily small. Furthermore, most profiles are to some degree periodic, which makes it even more unlikely that these journeys only appear for departure times near the end of the departure time interval. Finally, the experiments discussed in Chapter 5 support our assumption.

The next earliest departing journeys, which ought to be on the first page, but are not in X_{arr}, must be found by forward and backward searches in the second phase. To decrease the number of such journeys, we run RR_{arr} in the first phase until a number of $m > n$ journeys are stored in Q. Having a larger pool of already found journeys increases the probability that the n next earliest departing journeys have already been found in the first phase. Choosing $m = \infty$ would effectively result in doing a complete RR_{arr} search on the first call of COMPUTEPAGE and then sorting all journeys by departure time, which contradicts the whole idea of pagination. The choice of m is thus a trade-off between less overhead by forward and backward searches (large values of m) and computing unnecessary journeys in the RR_{arr} phase, especially for the first page (small values of m). In our implementation, we choose m relative to the page size n. The ratio between m and n is described by a constant lf, the *look-ahead factor*, such that $m = \lfloor \text{lf} \cdot n \rfloor$. In Section 5.4 we compare the performance of RR_{dep} for different look-ahead factors.

4.2.3 Ordering by Earliest Optimal Departure Time (RR_{e})

For computing an order by earliest optimal departure time, i.e., $\preceq_{\text{e}}$, we can make use of the same observation that we used for RR_{dep}: Partial profiles are always ordered by their earliest optimal departure time. Hence, we use the same data structures as for RR_{dep}, but change the comparison function of the priority queue to $\preceq_{\text{e}}$, i.e., to order journeys ascending by their earliest optimal departure time. Furthermore, when NEXTJOURNEY is called and J is the candidate for the next journey, i.e., the minimum element in the queue, for all $k' \neq |J| \wedge k' < k_{\max}$ (or even $k' < |J|$ if $\tau_{\text{arr}}(J)$ is less than the latest arrival time that was scanned by rRAPTOR) we have to check if there might be a journey J' with k' trips and $\tau_{\text{e}}(J') < \tau_{\text{e}}(J)$ that we have not found yet. Here, it is sufficient to test

whether $\tau_{\text{dep}}^{\text{latest}}(k') > \tau_{\text{e}}(J)$. If so, the earliest optimal departure time for the next later journey with k' trips is bounded by $\tau_{\text{dep}}^{\text{latest}}(k')$. If not, the next later journey for k' trips could have an earliest optimal departure time greater than $\tau_{\text{dep}}^{\text{latest}}(k')$ and we need to determine it by a forward and backward search.

The aforementioned techniques, however, presume that we can identify the earliest optimal departure time of any journey. As we have shown in Section 4.1.3, the earliest optimal departure time of any journey J in a profile P is always equal to the departure time of the latest departing journey J' in P that has dominated J previously, increased by one time unit ε. Thus $\tau_{\text{e}}(J)$ can formally be expressed as follows: $\tau_{\text{e}}(J) = \max\left\{\tau_{\text{dep}}(J') \,\middle|\, J \in \mathcal{J}_e(J)\right\} + \varepsilon$, where $\mathcal{J}_e(J) \coloneqq \{J' \in P \mid |J'| \leq |J| \wedge \tau_{\text{arr}}(J') < \tau_{\text{arr}}(J)\}$. We can therefore compute $\tau_{\text{e}}(J)$ by iterating over the respective journeys in $\mathcal{J}_e$.

It remains to show that at any point in our algorithm where a journey J is found for the first time, all journeys $J' \in \mathcal{J}_e(J)$ have already been found. Suppose J is found by rRAPTOR in the first phase. We then know that all journeys departing before J have already been found, so the journeys in $\mathcal{J}_e(J)$ have been found in particular. Otherwise J is found by a forward and backward search in the second phase of the algorithm. If so, let τ_{e} be the earliest optimal departure time of the minimal journey in Q. The forward search that finds J is started if $\tau_{\text{dep}}^{\text{latest}}(|J|) \leq \tau_{\text{e}}$. $\tau_{\text{dep}}^{\text{latest}}(|J|)$ was either set as the departure time of the previous journey with $|J|$ trips or as the departure time of a journey J' with less than $|J|$ trips if a forward search has not found a journey for $|J|$ trips. However, all relevant numbers of trips $k' < |J|$ have been checked before such that for every such k' all journeys that depart before τ_{e} have already been found. In particular, J' must have been found, since $\tau_{\text{dep}}(J') = \tau_{\text{dep}}^{\text{latest}}(|J|) \leq \tau_{\text{e}} < \tau_{\text{dep}}^{\text{latest}}(|J'|)$.

5. Experimental Evaluation

In this chapter, we describe several experiments to evaluate the concepts and algorithms we have developed. Fist, we describe our test data based on the Swiss public transit network and the generation of random queries (Section 5.1). We then present an experiment that demonstrates the usefulness of ordering journeys by earliest optimal departure time (Section 5.2). We further report the running times of the algorithms presented in Chapter 4 we measured in different scenarios. The results are discussed in Section 5.3. Section 5.4 looks in detail at RR_{dep} and RR_{e}. In particular, it addresses the question of finding sensible values for the look-ahead factor.

5.1 Experimental Data and Setup

The experimental setup is based on the work in [WZ17] and [Sau18]. We use the public transit network of Switzerland, which is publicly available (`http://gtfs.geops.ch/`). The period of operation Π is a time interval of 48 hours. Under the simplifying assumption that the timetable is periodic, Π was obtained by duplicating the schedule data for single business day (30th of May 2017). The footpath graph $\mathcal{F}$ was constructed in several steps. Based on the footpaths and roads in the OpenStreetMap data of Switzerland, the minimum walking time – assuming a constant walking speed of 4.5 km/h – between any two stops of the public transit network was determined. As discussed in Chapter 2.1, the resulting walking graph must be transitively closed and of reasonable size. To this end, edges were only inserted between such stops with a walking time distance of no more than 15 minutes. The transitive closure of these edges was then constructed, which lead to an average vertex degree of approximately 100 in the resulting footpath-graph. For a more detailed description, we refer the reader to

Stops	Routes	Trips	Stop Events	Edges
25 426	13 934	369 534	4 740 929	215 360

Table 5.1 – Size of the Switzerland public transit network that was used in our experiments.

[Sau18]. Statistics regarding the size of the resulting public transit network instance are listed in Table 5.1.

Queries

Picking source and target stops $s, t \in \mathcal{S}$ uniformly at random leads to not quite realistic queries. Since there are many less frequented stops in rural areas, a random query is likely to involve such stops. However, there are more passengers traveling from and to such stops that are also served by many trips (e.g., Zürich HB). We thus assigned a probability proportional to the number of stop events, i.e., arriving and departing trains, to each stop. Source stop s and target stop t were then chosen at random with regard to this probability distribution.

Moreover, the running time of RAPTOR increases if s and t are far apart, because in this case the search space is typically larger and more RAPTOR rounds are required. As an approximate distance measure we use the *rank* of a target stop. The rank $r(t)$ of a stop t denotes its position in a list of all vertices in an unrestricted walking graph sorted by their walking distances to s. As walking graph we again use the graph of roads and footpaths in the OpenStreetMap data of Switzerland. To generate multiple queries, we first pick a random source stop $s \in \mathcal{S}$ and for any $e > 7$ a random target stop t with a rank between 2^e and 2^{e+1}. To indicate that a target stop was chosen within this range, we write $r(t) \approx 2^e$. Note that "picking a random stop" here means randomly choosing with regard to the number of trip events per stop as described above. Since the walking graph that we used to determine ranks has 603,910 vertices, the highest rank we consider is 18.

Unless otherwise stated, we use $I = [00{:}00, 24{:}00]$ as the departure time range for the queries. This ensures that the nighttime parts of timetables, which are typically sparse, are considered as well as daytime periods, during which trips operate more frequently.

Note that while for most algorithms the page size does not impact the number of computations that are actually performed, RR_{dep} and RR_{e} are likely to perform extra forward and backward RAPTOR searches whenever a page is returned (for more insight

on what influences the performance of these approaches, see Section 5.4). Choosing a large page size, e.g., 1000, would lead to the whole profile being contained in the first page and thus is an unrealistic scenario. For our experiments we chose a default page size of 5, which is quite a common value, e.g., is used for the online timetable information of the Swiss national railway company (`http://sbb.ch`).

Setup

We implemented all algorithms in C++17 and compiled them with GCC 7.3.1 using the optimization flag `-O3`. The experiments were conducted on a machine equipped with two quad-core Intel Xeon E5-430 processors clocked at 2.66 GHz and 32 GiB of DDR2 RAM. All queries were run on a single core.

5.2 Earliest Optimal Departure Time

In Section 3.1.1 we introduced the earliest optimal departure time $\tau_e(J)$ of a journey J as a sorting criterion for profiles. To understand the usefulness of this ordering, suppose a user scrolls through a profile ordered by earliest optimal departure time. Every position i in the profile with the respective earliest optimal departure time $\tau_e(J_i)$, where J_i is the journey at position i, then answers the question "Which journeys are worth considering if I want to depart at $\tau_e(J_i)$ or later?". These journeys are precisely the journey J_i and its successors.

In this experiment, we study to which extent this property is satisfied by the other orderings, $\preceq_{\text{arr}}$ and $\preceq_{\text{dep}}$. More precisely, if two journeys J and J' are bicriteria-optimal for the same time interval, i.e., $[\tau_e(J), \tau_{\text{dep}}(J)] \cap [\tau_e(J'), \tau_{\text{dep}}(J')] \neq \emptyset$, they should be listed "close" to each other in the profile. The further the distance between the two journeys in the sorted profile, the more likely it is that they appear on different pages. In such a case, a user could possibly not notice one of them even though they are both relevant candidates. To measure the extent of this problem, we consider a simplified variant. We are interested in the maximum distance two journeys with *the same* earliest optimal departure time can have in a profile. To this end, we define the τ_e-distance $\delta_{i,j}$ of two journeys J_i and J_j in profile with $\tau_e(J_i) = \tau_e(J_j)$ and $j > i$ as the number of journeys that are listed between J_i and J_j and have a different earliest optimal departure time, i.e., $\delta_{i,j} := |\{J_k \mid i < k < j \wedge \tau_e(J_k) \neq \tau_e(J_i)\}|$. The. τ_e-distance $\delta(P)$ of a profile P is then defined as the maximum distance among all pairs of journeys with the same earliest optimal departure time, i.e., $\delta(P) = \max\left\{\delta_{i,j} \,\middle|\, J_i \in P \wedge J_j \in P \wedge j > i \wedge \tau_e(J_i) = \tau_e(J_j)\right\}$.

We measured the τ_e-distances $\delta(P)$ for 100 profiles per rank. The profiles were computed for source-target-queries generated as described in Section 5.1. We sorted the profiles

Rank	2^7	2^9	2^{11}	2^{13}	2^{15}	2^{17}
Order by Arrival Time ($\preceq_{\text{arr}}$)	0.1	0.4	1.3	4.3	6.5	12.0
Order by Departure Time ($\preceq_{\text{dep}}$)	0.0	0.4	1.0	3.2	4.2	10.0

Table 5.2 – The average τ_e-distance $\delta(P)$ over 100 profile queries per rank.

afterwards by departure time and by arrival time. Table 5.2 shows the average τ_e-distance over all 100 profiles for each rank and for both orderings.

While for low ranks journeys with the same earliest departure time are mostly grouped together ($\delta(P)$ close to 0), on higher ranks there are large gaps between two journeys with the same earliest optimal departure time. On average, a profile for rank 2^{17} contains at least one pair of journeys J_i, J_j with the same earliest optimal departure time τ_e for which 10 journeys with earliest optimal departure times other than τ_e are listed between J_i and J_j. Since $\delta_{i,j}$ does not consider other journeys between J_i and J_j with earliest optimal departure time τ_e, the effective distance $j - i$ is even larger. Thus, it is likely, even for page sizes of 10 or 15 journeys, that J_i and J_j are listed on different pages.

There are multiple reasons why such large values for $\delta(P)$ can occur. For one thing, most trips operate periodically, e.g., hourly, which leads to a certain degree of periodicity of the profile. For instance, assume most of the journeys from a source stop s to a target stop t require at least 3 trips. If there is, however, a single train that only operates once per day, this train can make a single journey with 2 trips possible. Even though this journey might depart quite late, it is bicriteria-optimal from τ_{min} onward, since it is the only journey with 2 trips in the profile. This results in a large distance to other, much earlier departing journeys with the same earliest optimal departure time.

Moreover, many journeys throughout the day share the same earliest optimal departure time as well. This can be due to the fact that especially for large travel distances many journeys share the same first trip. For instance, one may have to take a certain bus to the main station at which there is a greater variety of possible trains to enter. All journeys that involve taking the same bus share the same departure time τ_{dep}. This departure time is then also the earliest optimal departure time of succeeding journeys even though they may have, for instance, very different arrival times. Furthermore, if one journey dominates several later departing optimal journeys, they all share the same earliest optimal departure time as well.

Algorithm	Collect Arrivals	Init	CR	SR	RE	Total
AR_{arr}	–	234.5	544.6	940.1	243.5	1956.0
AR_{dep}	–	233.8	544.5	939.4	242.3	1954.1
AR_{e}	–	233.7	544.4	939.3	242.2	1953.8
RR_{arr}	102.5	46.5	310.0	332.0	173.0	946.1
RR_{dep} 1.5	102.3	138.0	541.7	719.6	273.0	1757.2
RR_{e} 1.5	102.3	111.3	459.3	595.9	240.1	1491.6

Table 5.3 – The average running time in milliseconds of each algorithm over 1000 queries with random source and target stops. The running times of the basic RAPTOR searches used within the profile algorithms are added up for each phase of RAPTOR: initialization (init), collecting routes (CR), scanning routes (SR) and relaxing edges (RE). For rRAPTOR-based algorithms, the time required to collect arrivals when the profile algorithm is listed as well.

In total, we conclude that the intuitively more comprehensible orders by arrival time or departure time show some serious drawbacks on higher rank queries. It is very likely that multiple journeys that are optimal for the same departure time and thus solve the same bicriteria problem appear on different pages.

5.3 Performance of Profile Queries

In this section, we measure the performance of the algorithms developed in Chapter 4. To this end, we are interested in the total time it takes each algorithm to compute a profile as well as the time required to compute each page. RR_{dep} and RR_{e} were parameterized with a look-ahead factor of 1.5. Why this value is a sensible choice is discussed in Section 5.4. We conducted two different experiments. The first experiment gives an overview of the overall performance of the algorithms, whereas the second one breaks down the running time by different distance ranks.

5.3.1 Overall Performance

For the first experiment we chose 1000 source and target stops randomly with regard to the number of stop events per stop, as described in Section 5.1. We measured the running time the algorithms took to compute the entire profile. This time is split into the times required for the different phases of the RAPTOR algorithm (initialization, collecting routes, scanning journeys and relaxing transfers). For each phase the times were summed up over all runs of the basic RAPTOR algorithms that are performed within a profile algorithm.

Algorithm	Forward & Backward RAPTOR	rRAPTOR
RR_{arr}	72.6	789.0
RR_{dep}-1.5	868.7	803.6
RR_{e}-1.5	615.5	791.2

Table 5.4 – Average running times in milliseconds split into the time consumed by forward and backward RAPTOR searches and by rRAPTOR searches, respectively.

Table 5.3 shows the average running times over the 1000 queries. Since the source and target stops are randomly selected nodes, the expected distance is roughly half the size of the distance graph, which corresponds to a rank of 17. For all rRAPTOR-based algorithms 5.4 further denotes how much running time was consumed by forward and backward RAPTOR searches and by rRAPTOR searches, respectively.

We observe that the running times of the Alternating RAPTOR-based algorithms are almost identical. This was to be expected as all three algorithms perform the same number of forward and backward searches. Recall that a backward RAPTOR search is performed for every arrival time of an earliest arriving journey and a forward search for every departure time of a journey found by a backward search. The number of required RAPTOR searches are hence solely determined by the journeys in the profile itself. Since running the basic RAPTOR algorithm accounts for the vast majority of the total running time, there are few to no differences between the Alternating RAPTOR approaches.

The rRAPTOR-based algorithm RR_{arr} performs about twice as fast as the respective Alternating RAPTOR approach AR_{arr}. RR_{arr} mainly benefits from its self-pruning property: All backward RAPTOR runs operate on the same round table. The latest found departure time for every stop and number of rounds is only decreased throughout runs. Even though many RAPTOR searches are started (over 4400 on average), the majority of non-optimal journeys from s to a stop v are pruned early, if there is already a later departure time in the round table for stop v (local pruning) or for the target stop t (target pruning). By contrast, every RAPTOR search performed in Alternating RAPTOR starts "from scratch" and the same journeys may be pursued multiple times. Note that about 8 % of RR_{arr}'s running time is consumed by the forward and backward RAPTOR searches required to find journeys at the begin and the end of the departure time interval.

Apart from lacking the self-pruning property, Alternating RAPTOR also has a larger overhead for clearing and initializing data structures, most notably the round table. The round table has to be cleared every time a new forward or backward RAPTOR search

is run, whereas all RAPTOR searches in rRAPTOR operate on the same round table. This work counts among the time for the RAPTOR initialization phase, which requires about four times as much time in AR_{arr} compared to RR_{arr}.

The running time of the mixed approaches RR_{dep} and RR_{e} lies between RR_{arr} and AR_{arr} RAPTOR. The performance benefit due to rRAPTOR is outweighted by the overhead of the forward and backward searches. In fact, RR_{dep} spends more time on the second phase, i.e., performing forward and backward RAPTOR searches in the style of Alternating RAPTOR, then on the first phase. By contrast, RR_{e} requires about 25 % less running time for forward and backward searches. Reasons for this are discussed in detail in Section 5.4.

5.3.2 Performance per Rank

For this experiment, we considered 100 source and target stop pairs per distance rank. For each query, every algorithm was run five times and the running time per page was measured. We then reported the average running time per page over the five runs. Figure 5.1 shows the total time running time of different algorithms; the running time for the first pages is shown by Figure 5.2. Note that the performance we measured for AR_{dep} and AR_{e} was not plotted since the running times were nearly identical to the running times of AR_{arr}.

Again, we observe quite a significant performance difference between AR_{arr} and RR_{arr}. On queries with lower ranks the median of the running times of AR_{arr} is about three times higher than the median of the running times of RR_{arr}. On higher ranks the difference decreases to a factor of about 1.5. This decline can be explained by the overhead of AR_{arr} at the RAPTOR initialization phase. The cost of clearing the round table mainly depends on the number of stops in the network and thus accounts for a higher proportion of the running time of AR_{arr} on lower ranks, where little time is consumed by the other phases of RAPTOR.

The mixed approaches RR_{dep} and RR_{e} have higher running times than RR_{arr}. This relation is inevitable, as RR_{dep} and RR_{e} use RR_{arr} as a building block and must therefore do at least the work that RR_{arr} does. Compared to the Alternating RAPTOR-based approaches they are still about two times faster on lower ranks. However, on higher ranks, the benefit due to the use of RR_{arr} disappears as the relative advantage of RR_{arr} compared to Alternating RAPTOR decreases.

Regarding the time required to compute the first few pages, the *response time* of an algorithm is of special interest. The response time combines the running time of the initialization phase with the running time of the first page. It thus represents the time

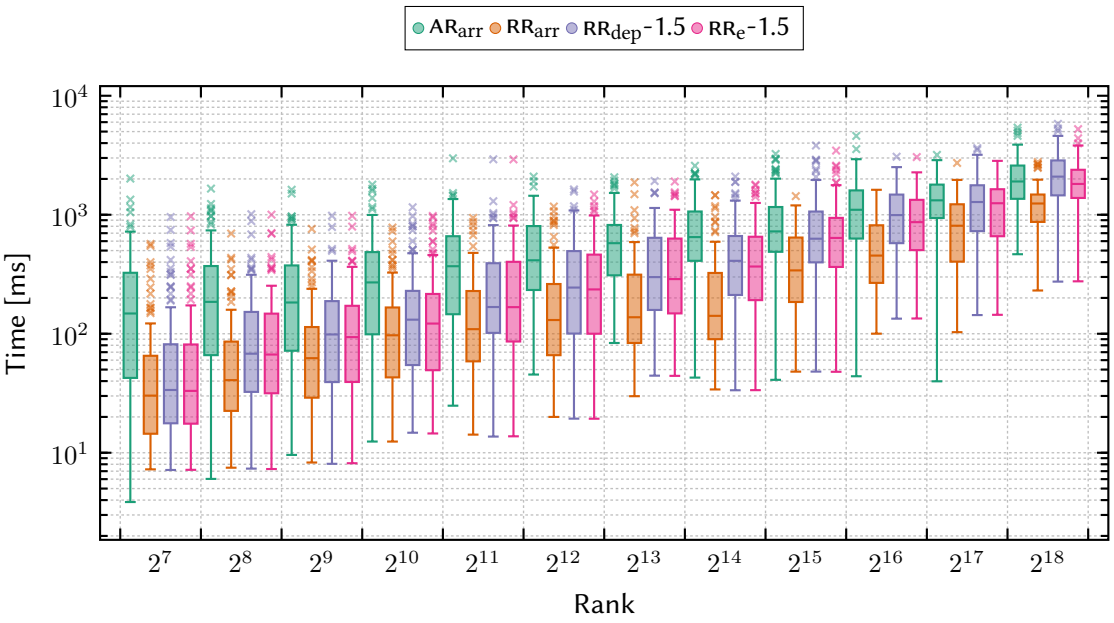


Figure 5.1 – The diagram shows the total running time different algorithms took to compute complete profiles. Each box plot shows the running times of 100 queries for an algorithm and distance rank. The running times for AR_{dep} and AR_{e} are not depicted since they are very similar to AR_{arr}.

span after which users see results of the first time. In contrast to Alternating RAPTOR, the RAPTOR-based algorithms have to collect arrivals during the initialization phase. For rank 2^9, this overhead is balanced by the shorter running time for the first page, such that both approaches have a similar response time in total. However, from the second page onward all RAPTOR based algorithms are about 2 times as fast as Alternating RAPTOR. This difference is presumably due to the reduced overhead for initializing the round table, which only occurs once during the first page.

As already discussed above, the relative difference due to the initialization overhead is reduced on higher ranks. This matches the observation that for high ranks the response time of RAPTOR-based algorithms is above the response time of AR_{arr}. However, on later pages the relative performance of the algorithms is similar to the relation of the running times for entire profiles.

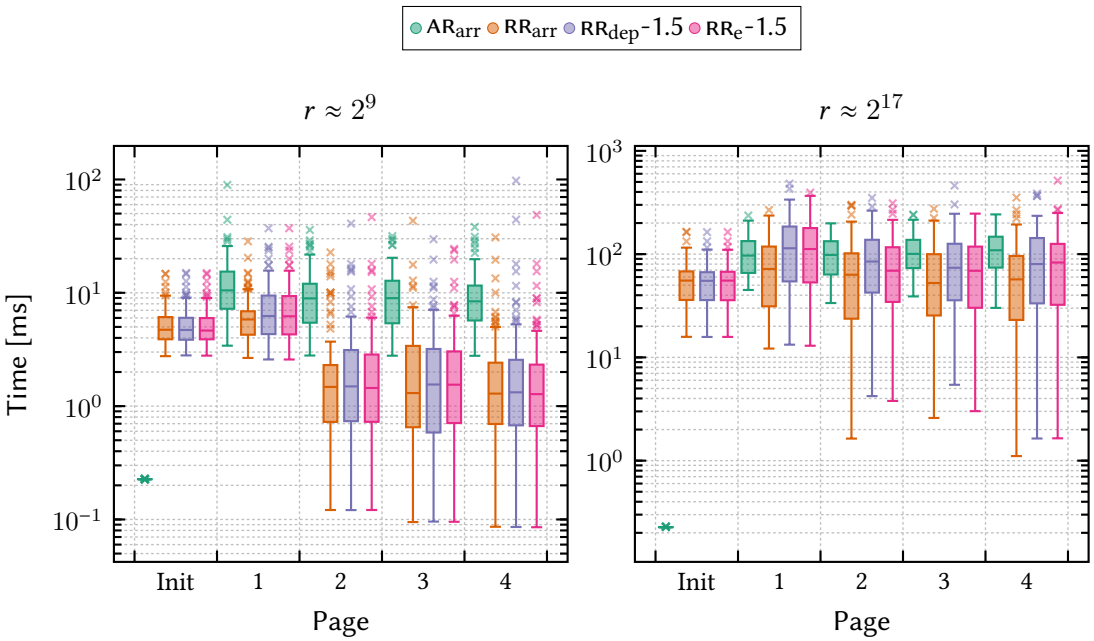


Figure 5.2 – The running times of different algorithms on 100 queries for the first four pages of a profile. The left diagram shows the running time for queries with a distance rank 2^9, the right diagram for queries with rank 2^{17}. *Init* denotes the time for initializing the profile algorithm, which manly consists of collecting arrivals in rRAPTOR based algorithms.

5.4 Parameterizing RR_{dep} and RR_{e}

In this experiment, we study the influence different parameters have on the performance of RR_{dep} and RR_{e} in detail. In particular, we are interested in finding sensible values for the look-ahead factor.

We consider different scenarios with regard to distance rank and page size. More precisely, we used 50 random queries with a rather low rank, i.e., $r \approx 2^9$, and 50 queries with a rather high rank, i.e., $r \approx 2^{17}$. Both algorithms were run for different look-ahead factors between 1.0 and 2.5 and for page sizes $n = 5$ and $n = 10$. Again, we report the average running time over 5 runs for each combination.

Figure 5.3 shows the results of our experiments. In favor of a simplified representation, only the median running times over the 50 queries for each configuration are plotted. Apart from the total running time it took the algorithm to compute a profile, we also look at the response time, i.e., time required for initialization and to compute the first page.

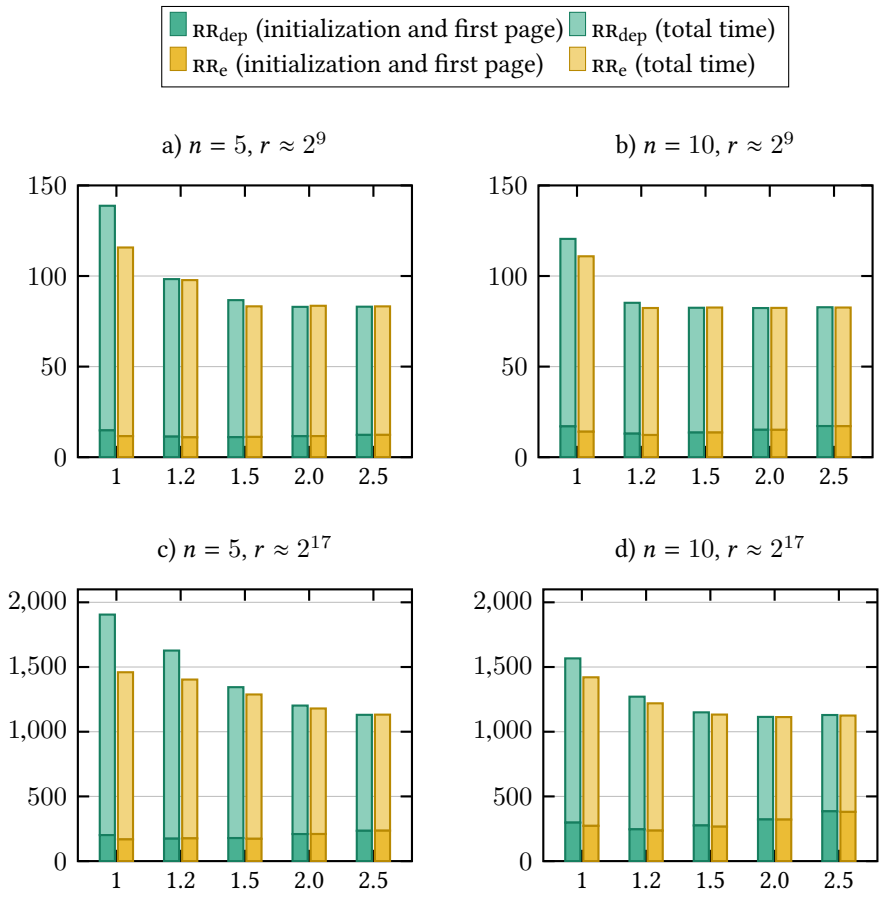


Figure 5.3 – The diagrams show the median running times of RR$_{dep}$ and RR$_{e}$ computing 50 profile queries for distance ranks r, page sizes n and look-ahead factors. The y-axis of each diagram denotes the running time in milliseconds, while the x-axis is labeled with different look-ahead factors. The green and yellow bar show the median running time required to compute the entire profile of RR$_{dep}$ and RR$_{e}$, respectively. The darker part of each bar denotes the corresponding response time. The page size used for computation of the profile and distance rank of the queries are written above the respective chart.

First, we observe that both algorithms, RR$_{dep}$ and RR$_{arr}$, have a shorter total running time when used with a page size of 10 instead of 5. The reason for this effect can be easily explained. The overhead for the forward and backward searches occurs in the second phase of the algorithm every time a page is returned. A larger page size means less pages for the same profile and thus less running time overhead.

Increasing the look-ahead factor from 1.0 to 1.5, which means that rRAPTOR is run in the first phase of the algorithm until 7 (or 15, for $n = 10$) journeys are cached, results in a decrease in the total running time of RR_{dep} of about 25%. While for low rank queries look-ahead factors greater than 1.5 do not influence the running time noticeably, for a high rank, more caching – up to a look-ahead factor of 2.5 – still improves the performance, especially for a low page size. This effect is presumably due to the different structure of the profiles for different ranks. For short travel distances, there are typically just a few possible routes that lead to bicriteria-optimal journeys. Thus, there are only a few numbers of trips for which optimal journeys exist, whereas higher rank profiles typically contain optimal journeys of a greater variety regarding their number of trips. This leads to a stronger influence of the ordering criterion on the order of the profile. Thus, more forward and backward searches have to be to be performed in the second phase of the algorithm on higher rank queries.

The response time of the algorithm, however, increases for larger look-ahead factors. For instance, for a page size of 10 and a look-ahead factor of 2.5 the algorithm has to compute 25 journeys in the first phase. While this large cache is likely to contain the 10 earliest departing journeys, it also contains many journeys that are not relevant for the current page.

RR_{e} performs consistently better than RR_{dep}, especially for small look-ahead factors. This is due to a difference in the second computation phase of RR_{e}: Let J be the candidate to be output as the next earliest journey. In the implementation of RR_{dep}, we then check for every number of trips $k \neq |J|$ the condition $\tau_{\text{dep}}^{\text{latest}}(k) > \tau_{\text{dep}}(J)$ to determine whether a forward and backward RAPTOR search is necessary. In RR_{e}, however, it is sufficient to compare $\tau_{\text{dep}}^{\text{latest}}(k)$ with $\tau_{\text{e}}(J)$ (for a proof, see Section 4.2.3). Since $\tau_{\text{e}}(J) < \tau_{\text{dep}}(J)$ the latter comparison checks a weaker condition. It is thus more likely that no forward and backward RAPTOR search needs to be performed, which leads to a lower total running time.

All in all, a look-ahead factor of 1.5 seems to be a sensible choice. In all of the considered scenarios, the total running time does hardly benefit from more caching whereas the response time increases for higher look-ahead factors.

6. Conclusion

In this thesis, we focused on a practical aspect of public transit profile queries: How can the response time of timetable information applications be decreased by computing profiles page-wise?

In Chapter 3, we formally stated the prerequisites an algorithm for pagination must fulfill. A decisive aspect is the order by which the journeys of a profile are sorted. Journeys that are listed on later pages are more likely not to be seen by a user. We discussed the advantages and drawbacks of ordering journeys by their arrival time, departure time or earliest optimal departure time. The latter is a criterion we proposed that sorts journeys by the earliest point of time after which they are bicriteria-optimal.

Ensuring that a profile algorithm computes journeys already sorted, poses a different challenge for each of the aforementioned orderings. In Chapter 4, we showed how *Alternating* RAPTOR [WZ17] can be adapted to obtain each of the orderings we consider. This can be done by rearranging the order in which forward and backward bicriteria RAPTOR searches are performed within Alternating RAPTOR appropriately.

We further pursued a different approach based on the rRAPTOR algorithm [DPW12], which inherently computes the journeys in a profile from latest to earliest departure time. By running the same algorithm on a reversed instance of the public transit network we can, however, compute journeys ordered by their arrival time. Using rRAPTOR to obtain orderings by departure and earliest optimal departure time turned out to be rather complex. To this end, we developed two algorithms, RR_{dep} and RR_{e}, featuring a mixed usage of rRAPTOR and Alternating RAPTOR: For each page, a parameterizable number of journeys, which are ordered by their arrival time, is computed using RR_{arr}.

In a second step, all missing, earlier departing journeys are determined using (a few) forward and backward RAPTOR searches for individual departure times. We expected these algorithms to benefit from rRAPTOR's performance advantage due to its self-pruning property.

We performed several experiments on the public transit network of Switzerland, which we discussed in Chapter 5. For one thing, we showed that when ordering real-world profiles by arrival or departure time, journeys that are bicriteria-optimal for the same departure time range are likely to be listed on different pages. This supports the usefulness of ordering journeys by earliest optimal departure time.

Regarding the performance of our algorithms, the rRAPTOR based approach that yields journeys ordered by their arrival time, RR_{arr}, computes average profiles about twice as fast Alternating RAPTOR. This difference comes mainly from the missing self-pruning property of Alternating RAPTOR and its bigger overhead for the initialization of round tables. The running times of the mixed approach, RR_{dep}, lie between the running times of RR_{arr} and AR_{dep}. With a carefully chosen look-ahead factor, the average running time decreases by about 10% compared to Alternating RAPTOR. When computing journeys ordered by earliest optimal departure time, an even greater speed-up of 30% is achieved.

All in all, Alternating RAPTOR seems to be a good starting point for a pagination algorithm. For profiles ordered by arrival time, we achieved significantly shorter running times by using rRAPTOR backwards. While RR_{arr} is conceptually elegant, the algorithms RR_{dep} and RR_{e} require a rather complex implementation. However, they show some performance gains compared to Alternating RAPTOR.

Future Work. Future research could focus on applying the Alternating RAPTOR approach on networks with unrestricted walking. Apart from that, it also seems promising to reduce Alternating RAPTOR's overhead for clearing and initializing round tables by using timestamps similiar to the lazy round table propagation we used in our rRAPTOR implementation (see Section 2.5.1).

The concept of running an algorithm on a reverted network, which we used for rRAPTOR, could be also applied to other algorithms which find journeys from latest to earliest. A promising candidate is the *Connection Scan Algorithm* (CSA) [Dib+18]. The profile variant of CSA scans all connections (trip segments between two consecutive stops) in decreasing order by their departure time. It thus finds the latest arriving journeys of a profile first. On a reverted network, CSA hence yields journeys increasing by their departure time. It could thus be used to decrease running times for departure time

order queries. How CSA can be adapted to obey other orderings, however, remains an open question.

List of Algorithms

List of Tables

List of Figures

Bibliography

[Bas+10] Hannah Bast et al. “Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns”. In: *Algorithms – ESA 2010*. Ed. by Mark de Berg and Ulrich Meyer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 290–301. ISBN: 978-3-642-15775-2.

[Bas+15] Hannah Bast et al. “Route Planning in Transportation Networks”. In: *CoRR* abs/1504.05140 (2015). arXiv: 1504.05140. URL: http://arxiv.org/abs/1504.05140.

[Bas09] Hannah Bast. “Car or Public Transport–Two Worlds”. In: *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*. Ed. by Susanne Albers, Helmut Alt, and Stefan Näher. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 355–367. ISBN: 978-3-642-03456-5. DOI: 10.1007/978-3-642-03456-5_24. URL: https://doi.org/10.1007/978-3-642-03456-5_24.

[Ber+09] Annabell Berger et al. “Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected”. In: *9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*. Ed. by Jens Clausen and Gabriele Di Stefano. Vol. 12. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009. ISBN: 978-3-939897-11-8. DOI: 10.4230/OASIcs.ATMOS.2009.2148. URL: http://drops.dagstuhl.de/opus/volltexte/2009/2148.

[BJ04] Gerth Stølting Brodal and Riko Jacob. "Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries". In: *Electr. Notes Theor. Comput. Sci.* 92 (2004), pp. 3–15. DOI: 10.1016/j.entcs.2003.12.019. URL: https://doi.org/10.1016/j.entcs.2003.12.019.

[Dib+18] Julian Dibbelt et al. "Connection Scan Algorithm". In: *J. Exp. Algorithmics* 23 (Oct. 2018), 1.7:1–1.7:56. ISSN: 1084-6654. DOI: 10.1145/3274661. URL: http://doi.acm.org/10.1145/3274661.

[Dij59] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: http://dx.doi.org/10.1007/BF01386390.

[DKP12] Daniel Delling, Bastian Katz, and Thomas Pajor. "Parallel Computation of Best Connections in Public Transportation Networks". In: *J. Exp. Algorithmics* 17 (Oct. 2012), 4.4:4.1–4.4:4.26. ISSN: 1084-6654. DOI: 10.1145/2133803.2345678. URL: http://doi.acm.org/10.1145/2133803.2345678.

[DPW09] Daniel Delling, Thomas Pajor, and Dorothea Wagner. "Engineering Time-Expanded Graphs for Faster Timetable Information". In: *Robust and Online Large-Scale Optimization: Models and Techniques for Transportation Systems*. Ed. by Ravindra K. Ahuja, Rolf H. Möhring, and Christos D. Zaroliagis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 182–206. ISBN: 978-3-642-05465-5. DOI: 10.1007/978-3-642-05465-5_7. URL: https://doi.org/10.1007/978-3-642-05465-5_7.

[DPW12] Daniel Delling, Thomas Pajor, and Renato Werneck. "Round-Based Public Transit Routing". In: Society for Industrial and Applied Mathematics, Jan. 2012. URL: https://www.microsoft.com/en-us/research/publication/round-based-public-transit-routing/.

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

[MS07] Matthias Müller-Hannemann and Mathias Schnee. "Finding All Attractive Train Connections by Multi-criteria Pareto Search". In: *Algorithmic Methods for Railway Optimization*. Ed. by Frank Geraets et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 246–263. ISBN: 978-3-540-74247-0.

[Mül+04] Matthias Müller-Hannemann et al. "Timetable Information: Models and Algorithms". In: vol. 4359. Jan. 2004, pp. 67–90. DOI: 10.1007/978-3-540-74247-0_3.

[Sau18] Jonas Sauer. “Faster Public Transit Routing with Unrestricted Walking”. MA thesis. Karlsruhe Institute of Technology, Apr. 2018.

[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)*. Ed. by Gianlorenzo D’Angelo and Twan Dollevoet. Vol. 59. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 7:1–7:14. ISBN: 978-3-95977-042-2. DOI: 10.4230/OASIcs.ATMOS.2017.7. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7891.