Karlsruhe Institute of Technology

UnLimited TRAnsfer Shortcuts with Delay Tolerance for Multi-Modal Journey Planning

Bachelor Thesis of

Dominik Bez

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	PD Dr. Torsten Ueckerdt
	Prof. Dr. Peter Sanders
Advisors:	Jonas Sauer, M.Sc.
	Tobias Zündorf, M.Sc.

Time Period: 1st June 2020 – 30th September 2020

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 30. September 2020

Abstract

Some routing algorithms for public transit assume that all vehicles (buses, trains, planes, etc.) depart and arrive on time, which is not a realistic scenario. This is especially true for algorithms which precompute data relying on the timetable. One such algorithm is ULTRA [BBS+19]. ULTRA precomputes transfer shortcuts between stops of the public transit network to prepare public transit routing algorithms like RAPTOR [DPW12] for multi-modal scenarios which include a secondary mode of transportation (e.g., walking or cycling). These shortcuts represent a transfer between leaving a public transit vehicle and entering another. However, if the departure or arrival of a public transit vehicle is delayed, other shortcuts not precomputed by ULTRA may become necessary. The purpose of this thesis is to adapt ULTRA to a more realistic model of public transit that includes delays. The result is called DB-ULTRA and precomputes shortcuts for all possible delay scenarios within given bounds. These shortcuts can then be used by public transit routing algorithms to find optimal journeys even in those delay scenarios. We evaluate our new algorithm on the public transit networks of Switzerland and Germany and conclude that the query times are similar to ULTRA, as long as the maximum allowed delay is not too large. If both departures and arrivals can be delayed, a maximum delay of ten minutes yields half the query times of comparable multi-modal routing algorithms, whereas if only arrivals can be delayed, even a maximum delay of 30 minutes yields similar results.

Deutsche Zusammenfassung

Einige Routenplanungsalgorithmen für öffentliche Verkehrsnetzwerke erwarten, dass alle öffentliche Verkehrsmittel (Busse, Bahnen, Flugzeuge etc.) pünktlich abfahren und ankommen, was kein realistisches Szenario darstellt. Dies gilt insbesondere für Algorithmen, die Daten vorberechnen, die von den Fahrplänen abhängen. Ein solcher Algorithmus ist ULTRA [BBS+19]. ULTRA berechnet Umstiege zwischen Haltestellen des Netzwerks für andere Routenplanungsalgorithmen für öffentliche Verkehrsnetzwerke, wie zum Beispiel RAPTOR [DPW12], vor. Diese Algorithmen können diese Umstiege dann für multimodale Szenarien, die einen sekundären Transportmodus (z. B. Laufen oder Radfahren) enthalten, verwenden. Wenn ein öffentliches Verkehrsmittel jedoch verspätet ist, könnten andere Umstiege, die nicht von ULTRA vorberechnet wurden, benötigt werden. Diese Arbeit ist darauf ausgerichtet, ULTRA für ein realistischeres Modell von öffentlichen Verkehrsnetzwerken mit Verspätungen anzupassen. Das Ergebnis heißt DB-ULTRA und berechnet für alle möglichen Verspätungsszenarien innerhalb gewisser Grenzen Umstiege vor. Diese Umstiege können dann von Routenplanungsalgorithmen für öffentliche Verkehrsnetzwerke verwendet werden, um optimale Reisen zu berechnen, selbst wenn Verspätungen eintreten. Wir evaluieren unseren neuen Algorithmus auf den öffentlichen Verkehrsnetzwerken der Schweiz und Deutschlands und schlussfolgern, dass die Anfragezeit ähnlich wie die von ULTRA ist, solange die maximal erlaubte Verspätung nicht zu groß ist. Wenn Abfahrten und Ankünfte verspätet sein können, liefert eine maximale Verspätung von zehn Minuten die halbe Anfragezeit von vergleichbaren multimodalen Routenplanungsalgorithmen. Wenn nur Ankünfte verspätet sein können, liefert sogar eine maximale Verspätung von 30 Minuten ähnliche Ergebnisse.

Contents

1. Introduction

In the past few decades, research on fast route planning algorithms has progressed rapidly. One interesting subject is a multi-modal routing problem which combines schedule-based and non-schedule-based (such as walking and driving) modes of transportation. The task is to answer queries from a source to a target at a specific departure time with a Pareto set of solutions using the criteria arrival time and number of transfers. This means a set of journeys such that for every possible journey from source to target there is a journey in the Pareto set that is at least as good for both criteria. For this, public transit and one non-schedule-based mode of transportation are allowed to be used. We often assume walking for the non-schedule-based mode of transportation. The general idea is to alternate queries of both modes. For example, the algorithm MCR [DDP+13] uses this approach. For the schedule-based part, most current algorithms work directly on the timetable as opposed to graph-based models. To solve the non-schedule-based part, a transfer graph is used. It models transfers between leaving a public transit vehicle and entering another by foot, bicycle or any other mode of transportation without a schedule. Furthermore, it can be used for an initial transfer from the source to the first trip and a final transfer from the last trip to the target or even a direct transfer from source to target. Efficient solutions rely on preprocessing of the timetable and transfer graph. Unfortunately, the precomputed data becomes invalid if public transit vehicles depart or arrive delayed. This thesis aims at extending a preprocessing technique named ULTRA [BBS+19] to handle delays.

1.1 Related Work

For road networks, routing is a well-known problem that can be solved by modeling the network as a graph and using Dijkstra's algorithm [Dij59] to find the shortest path. This algorithm is often combined with various speedup techniques that exploit attributes typical for road networks, such as hierarchies of more or less important roads using Contraction Hierarchies (CH) [GSSD08, GSSV12]. Some speedup techniques are presented in the overview article [BDG+16]. Especially in the past, these techniques were also used for journey planning in public transit networks by modeling the network as a graph as well [PSWZ08]. Since most of these speedup techniques were developed specifically for road networks, they did not work as well for public transit networks, as discussed in [Bas09]. Many modern algorithms do not model the network as a graph anymore and operate directly on the timetable. Simple data structures ensure good data locality for fast queries. Examples are RAPTOR [DPW12] and CSA [DPSW13]. RAPTOR operates on trips —

a series of stops in the network that are visited by the same vehicle at specific times — and routes, agglomerations of trips that serve the same stop sequence. CSA, on the other hand, operates on the smaller entities of connections, the parts of a trip which connect two consecutive stops.

Some of these algorithms have delay-aware modifications. One is McRAPTOR (which stands for "more criteria RAPTOR") [DPW12] with reliability as a third criterion besides arrival time and number of transfers. Reliability was defined by Disser et al. [DMS08] as the probability of catching all trips of the journey. It is higher the more *buffer time* there is between leaving a trip and entering the next one. Another approach is the MEAT (minimum expected arrival time) problem [DPSW13], which can be solved with a modified version of CSA. The goal is to find viable alternatives at every intermediate transfer where the next connection could be missed. Instead of minimizing the arrival time at the target of a single journey, the minimum expected arrival time of all alternatives is minimized, incorporating stochastic delays for all public transit vehicles. This prevents the algorithm from computing fast journeys with very low reliability and no good alternatives. Note that these algorithms do not solve the problem we consider in this thesis. In general, they do not even support a multi-modal scenario. The purpose of these algorithms is to anticipate delays at query time and computing reliable or alternative journeys. We, on the other hand, anticipate delays already at preprocessing time but expect them to be exactly known at query time.

Existing routing algorithms often impose restrictions on the transfer graph. RAPTOR and CSA, for example, can be used in the multi-modal scenario but require a transitively closed transfer graph. This has the advantage that every possible destination of a footpath can be reached by scanning a single edge. However, to achieve performance characteristics similar to uni-modal routing, the transfer graph must be limited somehow, otherwise a transitively closed transfer graph would be too large. A natural choice to limit the transfer graph is establishing a maximum walking distance between two stops of the public transit network before computing the transitively closed graph. It could be argued that long transfers are seldom useful and thus a walking limit of, e.g., 15 minutes should not harm the quality of the result by much. As Wagner and Zündorf [WZ17] showed, this is not true and these limitations indeed often lead to suboptimal routes. Lifting these restrictions without hurting performance too much is therefore a major concern.

A multi-modal extension of McRAPTOR without any restrictions is MCR (multimodal multicriteria RAPTOR) [DDP+13]. The variant which only optimizes arrival time and number of trips is called MR-∞. Basically, the idea is to alternate queries of RAPTOR and Dijkstra's algorithm, using the output of one algorithm as the input for the other. RAPTOR finds the trips to use, whereas Dijkstra's algorithm explores the transfer graph to find the transfers between trips. While MR-∞ already solves the stated problem, it is much slower than standard RAPTOR and therefore not applicable for some real-time applications.

Recently, a new approach named ULTRA [BBS+19] was proposed by Baum et al. It is a preprocessing technique aimed at finding all transfers between stops of the public transit network that are necessary to always find a Pareto set of journeys. The found journeys are stored as shortcuts between the two affected stops. A shortcut is a single edge from an origin stop to a destination stop. Any algorithm that usually required a transitively closed transfer graph can then use the resulting shortcut graph. The experiments Baum et al. performed showed that RAPTOR and CSA with ULTRA shortcuts are as fast as with a restricted transfer graph, while allowing unrestricted transfers.

1.2 Contribution

ULTRA expects all departures and arrivals to happen as scheduled without any delays. This, however, is not a valid assumption. For example, the German railway company Deutsche Bahn has reported a delay of at least six minutes for 6.1% of their train arrivals in 2019 [Deu19, Deu20]. This figure does not account for canceled trips and rises to 24.1% if only long-distance traffic is considered. When a public transit vehicle arrives at or leaves a stop later than scheduled, some shortcuts may be not needed anymore. Moreover, some other shortcuts could be required that are not in the shortcut graph, because they would be superfluous if all trips were on time. The purpose of this thesis is to adjust ULTRA such that queries yield optimal solutions even if delays happen. For this, DB-ULTRA was developed. Just like ULTRA, it computes transfer shortcuts between stops. But unlike ULTRA, it does so for every possible delay scenario with a maximum delay for all departures and arrivals, the *delay buffer*. A public transit algorithm like RAPTOR or CSA can compute all Pareto-optimal journeys correctly using these shortcuts, as long as no trip has a delay beyond the delay buffer.

1.3 Outline

The next chapter defines the necessary notation used throughout this thesis. Furthermore, we present the algorithms on which DB-ULTRA is based, especially ULTRA. In Chapter 3, two delay models are presented to incorporate possible delays. A detailed description on how to modify ULTRA to handle these new models follows in Chapter 4. The result is DB-ULTRA, a preprocessing technique to enhance public transit routers like RAPTOR for multi-modal scenarios with an unrestricted transfer graph and expected, but limited, delays. To further increase the performance of DB-ULTRA and the quality of the result, several optimizations are presented. DB-ULTRA is evaluated experimentally in Chapter 5 to verify the usefulness of this approach. Finally, the results of this thesis are summarized in Chapter 6. Moreover, an outlook on what should be investigated further on this subject is given.

2. Preliminaries

This chapter provides the necessary foundations of the thesis. The notation and terminology is inherited from the original ULTRA paper [BBS+19]. Furthermore, the problem definitions and algorithms that are important for this thesis are explained. This includes Dijkstra's algorithm [Dij59], RAPTOR [DPW12], MCR [DDP+13] and ULTRA [BBS+19] since DB-ULTRA is based on these algorithms.

2.1 Notation

A public transit network is given by a set of *stops* $\mathcal{S}$, a set of *trips* $\mathcal{T}$, a set of *routes* $\mathcal{R}$ and a directed, weighted *transfer graph* $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{S} \subseteq \mathcal{V}$. The set $\mathcal{S}$ contains the *vertices* of the graph and the set $\mathcal{E}$ the *edges*. A stop $v \in \mathcal{S}$ is a place where passengers can enter or leave a public transit vehicle. These vehicles operate in trips which are sequences of stops that are visited by the same individual vehicle at a particular point in time. For each trip $T \in \mathcal{T}$, every stop v of T has an associated arrival time $\tau_{\text{arr}}(T, v)$ and an associated departure time $\tau_{\text{dep}}(T, v)$. We may refer to these times as *stop events*. The i-th stop of the trip T is written as $T[i]$. Trips that share the same sequence of stops without overtaking each other are part of the same route $r \in \mathcal{R}$. Formally, a trip $T_a \in \mathcal{T}$ overtakes the trip $T_b \in \mathcal{T}$ if there exist two stops $u, v \in \mathcal{S}$ such that T_a arrives at or departs from u before T_b, but arrives at or departs from v later than T_b. Therefore, $\mathcal{R}$ defines a partition of $\mathcal{T}$. Associated with each edge $e = (u, v) \in \mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a *transfer time* $\tau_\theta(e)$ which is the time required to transfer from u to v. Every stop v has a *departure buffer time* $\tau_{\text{buf}}(v)$. At least this amount of time must pass after the arrival at v before a trip can be entered. It is used to model the time needed to find the right track and board the vehicle. Figure 2.1 shows an exemplary public transit network.

We define a *trip leg* T^{ij} as a subsequence of a trip T, where i is the index of the stop the trip was boarded by a passenger and j is the index where the trip was left. Naturally, $i < j$ must be true. The departure time of a trip leg is defined as the departure time of its first stop: $\tau_{\text{dep}}(T^{ij}) := \tau_{\text{dep}}(T, T[i])$. Similarly, the arrival time is defined as $\tau_{\text{arr}}(T^{ij}) := \tau_{\text{arr}}(T, T[j])$. A *transfer* $\vartheta = (v_1, v_2, \ldots, v_k)$ is a path in G which begins or ends at a trip leg. The *transfer time* of ϑ is the sum of the transfer times of its edges: $\tau_\theta(\vartheta) = \sum_{i=1}^{k-1} (v_i, v_{i+1})$. There are three types of transfers. The first is an *initial transfer*. It begins at the source (where the journey starts) and ends at the first stop of a trip leg. An *intermediate transfer* ϑ between trip legs T_a^{ij} and T_b^{mn} begins at $T_a[j]$ and ends at $T_b[m]$. We call the intermediate transfer *valid* if the transfer time is sufficiently small such that T_b^{mn} can be reached. Formally, this

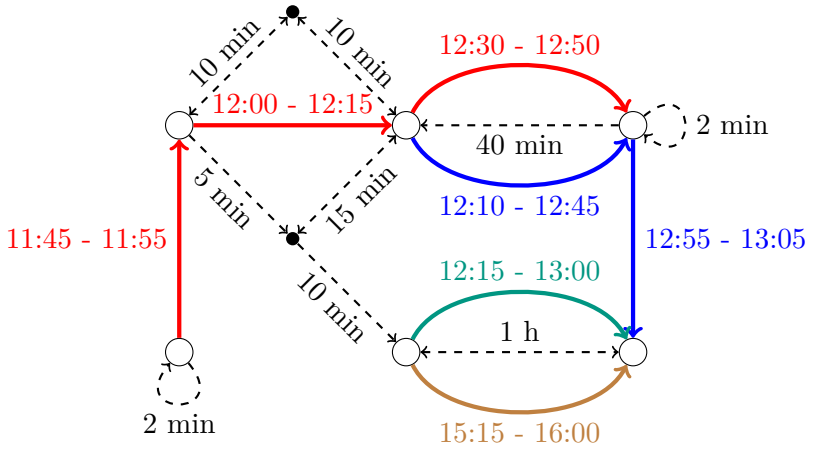

Figure 2.1: A public transit network. Big circles represent stops, small black circles represent vertices, which are not stops. Dashed lines represent edges of the transfer graph. They are labeled with their associated transfer time. A trip is portrayed as a sequence of one or more connections of the same color between stops. The shown times are the departure time at the first stop of the connection and the arrival time at the second stop. Since the green and the brown trip at the bottom do not overtake each other and visit the same stop sequence, they are part of the same route. If relevant, departure buffer times are pictured as dashed loops.

is equivalent to $\tau_{\text{arr}}(T_a^{ij}) + \tau_\theta(\vartheta) + \tau_{\text{buf}}(T_b[m]) \leq \tau_{\text{dep}}(T_b^{mn})$. The trip T_b would otherwise be missed at the stop $T_b[m]$. Finally, a *final transfer* begins at the last stop of a trip leg and ends at the target, the vertex where the journey ends. All types of transfers may be empty.

A journey $J = \langle \vartheta_0, T_0^{ij}, \ldots, T_{k-1}^{mn}, \vartheta_k \rangle$ is constituted by alternating transfers and trip legs. It begins with an initial transfer and ends with a final transfer. A journey is called *valid* if all its intermediate transfers are valid. We normally only consider valid journeys. The departure time of J is defined as follows: $\tau_{\text{dep}}(J) := \tau_{\text{dep}}(T_0^{ij}) - \tau_{\text{buf}}(T_0[i]) - \tau_\theta(\vartheta_0)$. On the other hand, the arrival time is $\tau_{\text{arr}}(J) := \tau_{\text{arr}}(T_{k-1}^{mn}) + \tau_\theta(\vartheta_k)$. The number of trips used by J is k.

2.2 Problem Definitions

A criterion is a specific attribute of a journey which can be better or worse. Departure time, arrival time and number of trips (or, equivalently, number of transfers) are all examples of criteria. For departure time, a higher value is considered better, whereas for arrival time a smaller value is better. A traveler probably wants to reach the target as fast as possible but also depart as late as possible since this means the travel time is smaller. Moreover, the traveler might not be able to depart earlier because of personal time constraints. A smaller number of trips is also better since every transfer is inconvenient and has the risk of failing, i.e., missing the next trip.

Say we have a set of criteria. Then a journey J *dominates* another journey J' given these criteria if all criteria are at least as good for J than for J'. Furthermore, both trips must start and end at the same vertex. If at least one criterion is strictly better, then J *strictly* dominates J'. A journey that is not strictly dominated by another journey is considered *Pareto-optimal*. A *Pareto set* is a set of Pareto-optimal journeys with the same source and target such that every possible journey from source to target is dominated by a journey in the set.

Different routing algorithms solve different problems. Probably the easiest one is the *one-to-one earliest arrival* problem. Given a source stop s, a target stop t and a departure time τ_{dep}, we ask for a journey with minimum arrival time that departs not earlier than τ_{dep}. We mainly focus on the *bicriteria* problem in this thesis: instead of a single journey with minimum arrival time, we ask for a Pareto set given the criteria arrival time and number of trips. This is the problem solved by RAPTOR [DPW12]. Note that both problems do not consider the departure time as an explicit criterion but restrict it.

A traveler might want to travel from one place to another not at a specific departure time but rather in a departure time interval, e.g., between 1 p.m. and 3 p.m. at one specific day. In that case, the traveler may not only be interested in the journey with the minimum arrival time. Another journey that arrives later but also departs later could be more interesting, for example, if the travel time is shorter. This leads to the *range* problem. Given a source stop s, a target stop t and a departure time range $[\tau_{\min}, \tau_{\max}]$, we ask for a Pareto set given the criteria departure time, arrival time and number of trips such that all journeys in this set depart in the departure time range. Effectively, we get all optimal journeys (regarding arrival time and number of trips) for each departure time in the range.

The range problem is crucial for some preprocessing techniques, namely ULTRA [BBS+19] and therefore DB-ULTRA which is based on ULTRA. The reason for this is that ULTRA computes shortcuts for all optimal journeys. This includes all possible departure times. Explicitly formulated, the range problem uses the following definition of dominance: a journey J dominates another journey J' if it does not use more trips and $\tau_{\text{dep}}(J) \geq \tau_{\text{dep}}(J')$ and $\tau_{\text{arr}}(J) \leq \tau_{\text{arr}}(J')$ are fulfilled. Since DB-ULTRA essentially solves the range problem, this is the definition of dominance we usually refer to.

2.3 Dijkstra's Algorithm

Dijkstra's algorithm was introduced by Edsger W. Dijkstra in 1959 [Dij59]. It solves the shortest path problem on graphs and serves as a base or building block for many other routing algorithms. We use it to explore the transfer graph. Given a graph $G = (\mathcal{V}, \mathcal{E})$ with an edge weight function $\omega : \mathcal{E} \to \mathbb{R}^+$ and a source vertex $s \in \mathcal{V}$, it computes all distances $\mathsf{dist}(s, v)$ from s to all vertices $v \in \mathcal{V}$ as well as shortest paths from s to every other vertex. For this, the algorithm maintains two arrays: $\mathsf{d}(\cdot)$ and $\mathsf{parent}(\cdot)$. The first contains the currently minimum known distance $\mathsf{d}(v)$ from s to v for every $v \in \mathcal{V}$ at any point in time during the execution of the algorithm. These tentative distances may decrease from step to step but are the correct distances $\mathsf{dist}(s, v)$ after the algorithm terminates. The second array stores the parent vertex $\mathsf{parent}(v)$ from which v was reached in the shortest currently known path. After the algorithm has terminated, it can be used to retrieve the shortest path to a vertex v by tracing back the parent pointers until s is reached. Moreover, Dijkstra's algorithm uses a priority queue Q which contains the next vertices to be scanned. The first vertex in the queue is always the minimum vertex u, meaning the vertex with the minimum tentative distance $\mathsf{d}(u)$ from the source. The priority queue can be implemented with a binary heap. There are other possible implementations, for example, a Fibonacci heap [FT87]. However, binary heaps are simpler and often faster in practical applications due to their good data locality.

The $\mathsf{d}(\cdot)$ array is initialized with ∞ for every vertex except s, which has a distance of 0. Then, the source vertex s is inserted into the priority queue Q. As long as the queue is not empty, we extract the minimum vertex u and scan its outgoing edges. For every such edge $e = (u, v)$ we *relax* the edge: If the distance from s to v using e $(\mathsf{d}(u) + \omega(u, v))$ is shorter than the current known minimum distance $\mathsf{d}(v)$, the distance of v is updated to $\mathsf{d}(u) + \omega(u, v)$ and the parent pointer of v is set to u. Afterwards, the vertex v is added to the Dijkstra queue Q if it is not present in the queue yet. Otherwise, the tentative distance

Algorithm 2.1: DIJKSTRA

Input: Graph $G = (\mathcal{V}, \mathcal{E})$, edge weight function $\omega : \mathcal{E} \to \mathbb{R}$, source vertex s
Data: Priority queue Q
Output: Distances $\mathsf{d}(v)$ for all $v \in \mathcal{V}$, shortest-path tree of s given by $\mathsf{parent}(\cdot)$

```
// Initialization
forall v ∈ V do
    d(v) ← ∞
    parent(v) ← ⊥
Q.INSERT(s, 0)
d(s) ← 0
// Main loop
while Q is not empty do
    u ← Q.DELETEMIN()
    forall (u, v) ∈ E do
        if d(u) + ω(u, v) < d(v) then
            d(v) ← d(u) + ω(u, v)
            parent(v) ← u
            if Q.CONTAINS(v) then
                Q.DECREASEKEY(v, d(v))
            else
                Q.INSERT(v, d(v))
```

of v in Q is updated with `decreaseKey`, which may cause the position in the queue to be decreased. Pseudocode is given in Algorithm 2.1.

After a vertex v is taken out of the Dijkstra queue, it is called *settled*. A settled vertex is guaranteed to not be re-inserted into the queue. This is because there can be no shorter path from s to v than the already found path. Therefore, every vertex is settled at most once, which means the main loop has no more than $|\mathcal{V}|$ iterations. Moreover, every edge is considered at most once as well — when its first vertex is settled. With a binary heap implementation of the priority queue, the work done per vertex and per edge is logarithmic in the number of vertices since the running times of all used priority queue functions are logarithmic. We can conclude the running time of Dijkstra's algorithm with a binary heap is in $\mathcal{O}((|\mathcal{V}| + |\mathcal{E}|) \cdot \log |\mathcal{V}|)$.

2.4 RAPTOR

The round-based public transit router RAPTOR [DPW12] operates directly on the timetable to find a Pareto set of journeys from a given source stop s to every other stop with the criteria arrival time and number of transfers. It does only support the use of a transitive transfer graph. RAPTOR can be extended to McRAPTOR, which can handle an arbitrary amount of criteria, and rRAPTOR, which solves the range problem. We first explain the basic version of RAPTOR before we present some optimizations and the two extension mentioned.

RAPTOR works in rounds. For every round k, every stop v has an associated tentative earliest arrival time $\tau_k(v)$ of all journeys with up to k trips. All these labels are initialized with ∞, except $\tau_0(s)$, which is initialized with the departure time. Round k computes $\tau_k(v)$ for every $v \in \mathcal{S}$ in three steps. At first, $\tau_k(v)$ is set to $\tau_{k-1}(v)$, since if v can be reached by that time with at most $k-1$ trips, it can also be reached at the same time

with at most k trips. The second step is the route scanning. We process each route r exactly once, beginning at the first stop v_0 of r. The earliest trip $\text{et}(r, v_0)$ of r that can be reached at v_0 is computed. It is called the *current* trip. This is the first trip T that fulfills $\tau_{\text{dep}}(T, v_0) \geq \tau_{k-1}(v_0) + \tau_{\text{buf}}(v_0)$ and is computed by iterating through the trips of r from last to first until this property is broken. If no such trip exists, we proceed to the next stop and try finding a suitable trip there until we find one. If we find a trip, we can use it at the following stops to update the minimum arrival times $\tau_k(\cdot)$. We still have to check for an earlier trip at every stop v_i after the current trip was found. There may be a quicker path to v_i found in a previous round such that an earlier trip can be caught. The last step relaxes transfers. Since RAPTOR requires a transitively closed transfer graph, this is just a matter of relaxing every edge. This is because in a transitively closed graph every shortest path is an edge or there is an edge with the same length as the shortest path. For every edge $(u, v) \in \mathcal{E}$ we set $\tau_k(v) = \min\{\tau_k(v), \tau_k(u) + \tau_\theta(u, v)\}$. The algorithm terminates in round k if no label $\tau_k(v)$ for $v \in \mathcal{S}$ was improved.

Some improvements can be made to speed up the algorithm. An important one is to avoid scanning all routes in every round. Note that a route must be scanned in round k only if the minimum arrival time of one of its stops was decreased in round $k-1$. Otherwise, scanning the route cannot improve any label in this round. For this, every stop v is *marked* in round k if $\tau_k(v)$ was improved by the route scanning or the relaxation of transfers. Before every round, the routes whose stops were marked in the previous round are collected in a set Q. Only these routes need to be scanned. Moreover, the routes only have to be scanned beginning at their first marked stop. We can implement Q as a map which maps routes to their first marked stop. For the first round, only the source stop is marked before the collecting. Similarly, only the edges $(u, v) \in \mathcal{E}$ where u was marked in the current round must be relaxed. Another optimization is *local pruning*. For every stop v we keep another label $\tau^*(v)$ that keeps track of the minimum known arrival time at v. Each time we improve $\tau_k(v)$ for some round k, we also improve $\tau^*(v)$. This means we can skip the first step of each round and do not have to copy the labels from the previous round. The label $\tau^*(v)$ already keeps track of the earliest arrival time at v. Pseudocode implementing RAPTOR with these optimizations is given by Algorithm 2.2.

2.4.1 McRAPTOR

To consider more criteria besides arrival time and number of trips (e.g., reliability or monetary costs) McRAPTOR was developed as an extension of RAPTOR. Instead of using a single value $\tau_k(v)$, the minimum arrival time, for every round k and stop v, a *bag* $B_k(v)$ of labels is used. This bag contains a label for every Pareto-optimal journey. Each label consists of a value for the minimum arrival time and a value for every additional criterion. Remember that the number of trips is handled implicitly via the rounds and therefore does not have to be included in the label. To scan a route r, we first create a *route bag* B_r. Instead of finding just one current trip as in RAPTOR, the route bag contains several labels, each associated with a trip. For every stop v of r, we first merge the route bag into the bag of v in the current round and discard dominated labels from the bag of v. For the arrival times, the arrival times at v of the associated trips are used since the route bag labels do not contain arrival times explicitly. Then, the bag $B_{k-1}(v)$ of the previous round is merged into the route bag B_r, again discarding dominated labels from B_r. For every new label in B_r a current trip must be found as usual using the arrival time at v of the label in $B_{k-1}(v)$. Finally, we proceed to the next stop. The relaxation of transfers is modified similarly. When relaxing an edge $e = (u, v) \in \mathcal{E}$, a temporary copy of $B_k(u)$ is created, the transfer time $\tau_\theta(e)$ is added to the arrival time of every label in the copy and this bag is merged into $B_k(v)$.

Algorithm 2.2: RAPTOR

Input: Stops $\mathcal{S}$, trips $\mathcal{T}$, routes $\mathcal{R}$, weighted transfer graph $G = (\mathcal{V}, \mathcal{E})$, source $s \in \mathcal{S}$, departure time τ
Data: Set of routes serving updated stops Q
Output: $\tau_k(v)$ for every round k and stop $v \in \mathcal{S}$

```
1  τ_0(·) ← ∞                                                    // Initialization
2  τ*(·) ← ∞
3  τ_0(s) ← τ
4  τ*(s) ← τ
5  mark s
6  forall k ∈ ℕ do
7      Q.CLEAR()
8      forall marked stops v do          // Collect routes serving updated stops
9          forall routes r serving v do
10             if (r, v') ∈ Q for some stop v' then
11                 if v comes before v' in r then
12                     Q.DELETE((r, v'))
13                     Q.ADD((r, v))
14             else
15                 Q.ADD((r, v))
16         unmark v
17     forall (r, v) ∈ Q do                                       // Scan routes
18         T ← ⊥                                                  // The current trip
19         forall stops v_i of r beginning at v do
20             if T ≠ ⊥ and τ_arr(T, v_i) < τ*(v_i) then
21                 τ_k(v_i) ← τ_arr(T, v_i)
22                 τ*(v_i) ← τ_arr(T, v_i)
23                 Mark v_i
24             if T = ⊥ or τ_{k-1}(v_i) + τ_buf(v_i) ≤ τ_dep(T, v_i) then
25                 T ← et(r, v_i)                  // Calculate earliest reachable trip
26     forall marked stops u do                                   // Relax transfers
27         forall (u, v) ∈ E do
28             if τ_k(u) + τ_θ(u, v) < τ*(v) then
29                 τ_k(v) ← τ_k(u) + τ_θ(u, v)
30                 τ*(v) ← τ_k(u) + τ_θ(u, v)
31                 mark v
32     if No stop is marked then
33         Return
34     τ_{k+1}(·) ← ∞
```

2.4.2 rRAPTOR

We already discussed the importance of the range problem for ULTRA in Chapter 2.2. As a reminder, a journey J dominates another journey J' if it does not use more trips and $\tau_{\text{dep}}(J) \geq \tau_{\text{dep}}(J')$ and $\tau_{\text{arr}}(J) \leq \tau_{\text{arr}}(J')$ are fulfilled. We are given a source stop s, a

target stop t and a departure time range $[\tau_{\min}, \tau_{\max}]$ and ask for a Pareto set using this notion of dominance such that all journeys in the Pareto set depart in the departure time range. Such queries could be solved by using McRAPTOR with the departure time as an additional criterion. However, there is a faster alternative called rRAPTOR.

For rRAPTOR, all departure times of trips at the source stop s in the departure time range are collected in a set D. RAPTOR is executed for every departure time in D in descending order. The idea for fast range queries is called *self-pruning*: The data structures of RAPTOR (i.e., the labels $\tau_k(v)$ for round k and stop v) are not cleared between the executions. Therefore, journeys departing later can implicitly dominate journeys in the current execution without explicitly saving and comparing departure times. The resulting Pareto sets are constituted by the union of the Pareto sets for every execution. Unfortunately, local pruning does not work with rRAPTOR. The minimum known arrival times $\tau^*(v)$ do not carry over to earlier departure times since they are for any number of trips and cannot be used for earlier rounds. To fix this, the labels of the previous round must be copied over to the current round k if they improve the labels. Formally, we set $\tau_k(v) = \min\{\tau_k(v), \tau_{k-1}(v)\}$ before round k for every stop v.

2.5 MCR and MR-∞

Remember that we consider the multi-modal scenario which combines public transit and an unrestricted transfer graph. There is already an algorithm available that solves this problem: MCR (multimodal multicriteria RAPTOR) [DDP+13]. As the name suggests, it is an extension of McRAPTOR. Since we are only interested in the criteria arrival time and number of transfers, we regard MR-∞, the version of MCR which optimizes these criteria. The idea is to alternate queries of Dijkstra's algorithm and RAPTOR. First, Dijkstra's algorithm is performed on the transfer graph to calculate the arrival times by direct transfer for every vertex. Note that unlike in RAPTOR, each vertex can be the source and not only stops. The direct transfer arrival times can be used to calculate which trips are reached in the first round of RAPTOR. The transfer phase of RAPTOR cannot relax single edges anymore since the transfer graph is not required to be transitively closed. Instead, it is replaced with Dijkstra's algorithm which is initialized with the arrival times that were improved in the route scanning to propagate these through the transfer graph. The newly improved labels can then be used for the next round of RAPTOR.

2.5.1 Transfer Graph Contraction

The problem with MCR and MR-∞ is their performance. Dijkstra's algorithm is much more costly than relaxing single edges. As the authors of MCR showed in [DDP+13], the transfer graph relaxation is the bottleneck of MCR. For the intermediate transfers, MCR only uses the transfer graph to find paths between stops. This means any *overlay* graph of the transfer graph which preserves shortest path distances between stops can be used for the relaxation of transfers. A smaller graph can speed up the transfer phase. To achieve this, the graph can be partially contracted similar to a Contraction Hierarchy (CH) [GSSD08, GSSV12] where only vertices are contracted which are not stops. When a vertex v is contracted, it and its incident edges are removed from the graph. To preserve shortest distances, shortcuts between the neighbors of v are inserted. Precisely, a shortcut (u, w) with the transfer time $\tau_\theta((u, w)) = \tau_\theta((u, v)) + \tau_\theta((v, w))$ is added if (u, v, w) is the shortest path from u to w. If there already is an edge between u and w, it cannot have a smaller transfer time and can therefore be discarded to prevent multi-edges.

The order in which these contractions occur is crucial. Contraction Hierarchies rely on heuristics to find a good contraction order. Less "important" vertices are contracted first.

Unfortunately, the order is suboptimal in our case because we prevent stops from being contracted. The transfer graph is usually (almost) fully connected. If the contraction is completed, we end up with a transitively closed graph which contains all stops. Such a graph is too big, and we could use RAPTOR in this case anyway. To resolve this issue, the contraction is stopped when the average vertex degree of the uncontracted remaining *core* graph surpasses a fixed threshold. This *Core-CH* is used as the transfer graph for the relaxation of transfers. The source vertex s and the target vertex t may not be stops. In that case, the core graph cannot trivially be used for initial and final transfers. These must be handled separately by constructing a new graph by adding the contracted vertices and their original edges again. A forward search from s and a backward search (using reversed edges) from t is executed on this graph where only edges that lead to vertices of higher importance are used. The computed times can be used to initialize the labels for the first RAPTOR round and updating the arrival time at t whenever a label of a stop is improved.

2.6 ULTRA

Even with a contracted transfer graph, MR-∞ is still significantly slower than RAPTOR. The goal of ULTRA [BBS+19] is to find a set of shortcuts between stops that are sufficient to answer all point-to-point queries in public transit networks with a transfer graph and the criteria arrival time and number of transfers correctly without restricting the transfer graph or using costly Dijkstra searches. The resulting set can be used, for example, for the transfer phase of RAPTOR. This set shall be sufficiently small to ensure fast query times. These shortcuts are only used for intermediate transfers since they only connect stops and not vertices in general. As Sauer showed [Sau18], intermediate transfers usually have a smaller impact than initial and final transfers. This suggests that the number of necessary shortcuts is low.

A shortcut set satisfies the stated requirement if there is a journey J that only uses intermediate transfers from the set for every Pareto-optimal journey J', such that J is not strictly dominated by J'. Before describing the algorithm, we observe the following. Given a journey J' with a subjourney j' that consists of exactly two trips with an intermediate transfer, let j be a journey that dominates j'. Then the journey J that arises from J' by replacing j' with j dominates J'. To see this, notice that J is valid. The intermediate transfer of j is valid since j itself is valid. The intermediate transfer in J immediately before j — if there is any — is valid because it is valid in J' and the time to reach the next trip is not smaller since $\tau_{\text{dep}}(j) \geq \tau_{\text{dep}}(j')$. Analogously, the intermediate transfer in J after j is valid as well. All other intermediate transfers in J are not affected and therefore still valid. If the first trip of j' is also the first trip of J', then $\tau_{\text{dep}}(J) = \tau_{\text{dep}}(j) \geq \tau_{\text{dep}}(j') = \tau_{\text{dep}}(J')$. If not, the departure time of J and J' is equal. Following the same argument for the last trip of J', $\tau_{\text{arr}}(J) \leq \tau_{\text{arr}}(J')$ is true. Furthermore, J does not use more trips than J' since j does not use more than j'. Consequently, J dominates J'.

This knowledge is used by ULTRA. The idea is to examine every journey that consists of exactly two trips with an intermediate transfer. These journeys are called *candidate journeys*. The intermediate transfers are considered *shortcut candidates*, since these are all shortcuts that could possibly be needed. For every candidate journey dominating journeys are searched for, the *witnesses*. Witnesses can use an initial and final transfer in contrast to candidate journeys. If no witness is found for a candidate journey, the corresponding candidate shortcut is necessary. Note that the resulting shortcut graph is in general not transitively closed. Nevertheless, it can be used for RAPTOR and other algorithms that have this restriction, since all intermediate transfers that could ever be used are in this graph as a shortcut. Recall that this restriction is only required to ensure that edges can be relaxed individually and relaxing whole paths does not help. This is the case for ULTRA shortcuts.

Algorithm 2.3: ULTRA

Input: Stops $\mathcal{S}$, trips $\mathcal{T}$, routes $\mathcal{R}$, weighted transfer graph $G = (\mathcal{V}, \mathcal{E})$
Data: Distance $\mathsf{dist}(a,b)$ from a to b in G, arrival time $\tau_{\mathsf{dir}}(a)$ at a by direct transfer
Output: Shortcut graph $G' = (\mathcal{S}, \mathcal{E}')$

```
1  forall s ∈ S do
2      Clear arrival times and Dijkstra queues
3      Compute dist(s, ·) for every stop with Dijkstra
4      D ← Collect departure times of trips at s with corresponding reachable routes
5      forall τ_dep ∈ D in descending order do
6          forall v ∈ S do
7              τ_dir(v) ← τ_dep + dist(s,v)
8          scanRoutes()  // all departing at s or coll. in line 4 for τ_dep
9          relaxTransfers()                                          // Dijkstra
10         collectRoutesServingUpdatedStops()
11         scanRoutes()
12         E' ← E' ∪ relaxTransfers()  // collect non-dominated candidates
```

For ULTRA to be efficient, it is impossible to search for witnesses for each candidate journey individually, because there are too many candidate journeys. Instead, ULTRA essentially calls rRAPTOR once for every stop in the network. However, the transfer relaxation consists not only of relaxing each edge, but involves a call to Dijkstra's algorithm to propagate arrival labels through the transfer graph. Stated differently, the call to RAPTOR for each departure time in rRAPTOR is replaced by a call to MR-∞. Therefore, the transfer graph does not have to be transitively closed or be restricted in any other way. Each rRAPTOR invocation is restricted to the first two rounds. This way, all candidate journeys are analyzed without doing unnecessary work. Moreover, self-pruning eliminates many dominated candidate journeys before they are even fully computed. Pseudocode is given by Algorithm 2.3.

The shortcuts are found and added to the shortcut graph in line 12. When a stop is settled there, the best journey to reach that stop will not change anymore. Therefore, this is a suitable moment to check if this journey is a candidate journey (i.e., does not use an initial or final transfer) by tracing parent pointers which are set every time a label is improved. If it is indeed a candidate journey, the intermediate transfer is added to the shortcut graph.

2.6.1 Implementation Details

There are more key differences to rRAPTOR, which are presented in this section.

Initial Transfers

Since witnesses can have an initial transfer, the transfer graph must be explored before routes are scanned for the first round. However, an invocation of Dijkstra's algorithm is not necessary for every departure time. Instead, Dijkstra's algorithm is called once for every source stop in line 3 to compute the distances from the source stop to every other stop. These distances are then used in line 7 to set the arrival times by direct transfer for every stop.

For the first trip, a witness could use every route it can reach by direct transfer. This can be every route in the network because the transfer graph is not restricted in any way.

Scanning every route for every departure time is expensive. Luckily, this can be avoided. ULTRA finds all routes that need to be scanned in line 8 already in line 4. To do this, all departure events $(r, v, \tau_{\text{dep}})$ of route r, stop v visited by r and departure time τ_{dep} are collected in a list. This list must be sorted descending by the departure time at the source stop $\tau_{\text{dep}} - \tau_{\text{buf}}(v) - \mathsf{dist}(s, v)$. Now, all routes that need to be scanned for a departure time τ_{dep} at s are exactly the routes between this departure time and the previous departure time in the sorted list. This means we can iterate over this list, collecting routes until a departure event at the source stop is found. The collected routes are stored in a set and labeled with the departure time of the just found departure event at the source. Then, we proceed with iterating through the list. At the end of the list, all relevant departure times are found in the right order and with their corresponding routes that need to be scanned in line 8.

Stopping Criteria

The final relaxation of transfers in line 12 can be stopped once every target of any candidate journey has been settled, because shortcuts are only found settling such vertices. For this, the number of stops that were reached by a candidate journey in line 11 is counted. Each time such a stop is settled during Dijkstra's algorithm, the counter is decreased by one. After the counter reaches zero, the execution of Dijkstra's algorithm can be aborted. The same idea can be applied to the intermediate relaxation of transfers in line 9 by counting the number of stops reached by a trip without an initial transfer. However, this can lead to more shortcuts in the result since witnesses could be pruned if Dijkstra's algorithm is stopped too soon. This can be mitigated by stopping Dijkstra's algorithm not immediately. Instead, a parameter $\overline{\tau}$ is introduced. Dijkstra's algorithm is continued until the first element of the Dijkstra queue has an arrival time greater than $\tau_{\text{arr}} + \overline{\tau}$, where τ_{arr} is the arrival time of the last settled stop which was reached by a trip without an initial transfer. For this thesis, $\overline{\tau}$ is always set to 15 minutes.

All three Dijkstra invocations in line 3, 9 and 12 use separate Dijkstra queues. As the data structures of RAPTOR, the queues of the latter two are not cleared between different source departure times of the same source. Otherwise, the stopping criteria would undermine self-pruning. Consider the first stopping criterion. When it is applied, all candidate journeys for the current source departure time are already examined. This means we do not need to search for witnesses anymore for this source departure time. However, through self-pruning, further witnesses may be necessary for an earlier source departure time that cannot be found for the earlier source departure time since not all routes are scanned. Similar considerations are true for the second stopping criterion which affects the intermediate transfers. A label can be removed from one of those queues if it is improved in another phase of the algorithm before the corresponding Dijkstra.

Transfer Graph Contraction

We have already seen the usefulness of a partially contracted transfer graph for MCR in Chapter 2.5.1. Since ULTRA is based on MR-∞, the same idea can be applied as well. Furthermore, ULTRA only uses the transfer graph to find paths between stops even for the initial and final transfers of witnesses. Therefore, only the core graph is necessary for the preprocessing. This is contrary to MCR, where the original graph is necessary for initial and final transfers.

Cyclic Witnessing

Witnesses are only required to dominate candidate journeys weakly. Therefore, two journeys $J = \langle \vartheta_0, T_0^{ij}, \vartheta_1, T_1^{mn}, \vartheta_2 \rangle$ and $J' = \langle \vartheta'_0, U_0^{kl}, \vartheta'_1, U_1^{pq}, \vartheta'_2 \rangle$ which dominate each

other may exist. Consider the case $\tau_\theta(\vartheta_0) > 0$. The journey J without the initial transfer ($J_0 = \langle T_0^{ij}, \vartheta_1, T_1^{mn}, \vartheta_2 \rangle$) is not dominated by the journey J' extended by the reverse initial transfer ($J'_0 = \langle \vartheta_0^{-1}, \vartheta'_0, U_0^{kl}, \vartheta'_1, U_1^{pq}, \vartheta'_2 \rangle$) since $\tau_{\text{dep}}(J'_0) < \tau_{\text{dep}}(J') = \tau_{\text{dep}}(J) < \tau_{\text{dep}}(J_0)$. Consequently, the shortcut for the intermediate transfer ϑ_1 of J is added to the shortcut graph and cyclic witnessing is only a problem between journeys with initial transfers of length 0. It can be solved by contracting groups of stops with transfer distance 0 before the preprocessing. These contractions are only temporary and must be reversed after the preprocessing.

Parallelization

ULTRA calls a variant of rRAPTOR once for every stop in the network. These calls are almost independent. The only shared data between different invocations is the network, but it is only accessed to read except for the resulting shortcut graph. This means ULTRA can easily be parallelized by distributing the source stops among several threads. Each thread maintains its own RAPTOR data structures and shortcut graph. After all source stops are processed, the shortcut graphs are joined by uniting their edge sets.

Prune with Existing Shortcuts

A candidate journey whose intermediate transfer is already represented in the shortcut graph does not have to be considered anymore. This can be exploited to enhance the first stopping criterion as introduced in "Stopping Criteria" by only counting the stops for which the corresponding candidate shortcut is not already in the shortcut graph. In a parallel context, the thread-local shortcut graph is used for this. Note that the speedup by parallelization could be decreased since this optimization works better the more shortcuts are already in the thread-local shortcut graph. More threads induce fewer shortcuts per thread.

2.6.2 Query Algorithm

The algorithm explained so far computes shortcuts between two trips. However, there can also be an initial and a final transfer from the source vertex and to the target vertex. These transfers are not represented in the shortcut graph. Therefore, using the shortcut graph as a substitute for the original transfer graph directly without modifying the query algorithm (e.g., RAPTOR) is not enough. The easiest way to incorporate initial and final transfers is to add a shortcut from the source to every other stop and from every stop to the target before running the query. The travel times for these shortcuts can be computed with a one-to-many algorithm where the queries are performed in reverse for the final transfers.

For this, Bucket-CH [KSS+07, GSSD08, GSSV12] is used. It uses a Contraction Hierarchy (CH), but not a partial one like the preprocessing. For Bucket-CH, every vertex has an associated bucket containing distances to the stops. These distances are computed by executing a backward search from every stop in the CH to vertices of higher importance. Afterwards, a forward search from the source to vertices of higher importance is performed. For each such vertex v the distances from s to every stop u in v's bucket can be updated with the sum of $\mathsf{dist}(s, v)$ and the distance from v to u in the bucket.

The query algorithm can be improved by adding fewer shortcuts for initial and final transfers to the transfer graph. Note that for a stop v with $\mathsf{dist}(s, v) \geq \mathsf{dist}(s, t)$ (i.e., the distance from the source to v is greater than the distance from the source to the target), the shortcut from s to v is not necessary since every journey that uses this initial transfer is dominated by the direct transfer from s to t. Analogously, the shortcut from v to the target t is not needed if $\mathsf{dist}(v, t) \geq \mathsf{dist}(s, t)$.

This knowledge can also be used to accelerate Bucket-CH by not computing the distances to such stops v if it is already clear that the distance from s (or to t, respectively) is too large. To do this, a standard CH query from source to target is performed. This works bidirectional: forward from the source and backward from the target, both leading to vertices of higher importance. This query can be stopped for both directions once the current distance is greater than the tentative distance from the source to the target. Afterwards, the Bucket-CH query is run only considering the vertices found in the standard CH query. The entries in the buckets are kept sorted by the distance to their target stop. Therefore, scanning through the bucket of a vertex v can be stopped once a stop u in the bucket with $\mathsf{dist}(s, v) + \mathsf{dist}(v, u) \geq \mathsf{dist}(s, t)$ (or $\mathsf{dist}(u, v) + \mathsf{dist}(v, t) \geq \mathsf{dist}(s, t)$, respectively) is reached. These optimizations improve local queries since only stops which are close to the source or target must be considered.

If we are allowed to alter the query algorithm, we can avoid adding shortcuts for the initial and final transfers to the shortcut graph altogether. Instead, the arrival times by direct transfer are set directly with the distances computed with the one-to-many algorithm. In the case of RAPTOR, the routes of the updated stops must be collected accordingly. The shortcuts from s to other stops are not needed anymore. To remove the shortcuts for final transfers, the minimum arrival time at the target is updated every time a stop v is updated by setting the arrival time $\tau_{\text{arr}}(t)$ to $\min\{\tau_{\text{arr}}(t), \tau_{\text{arr}}(v) + \mathsf{dist}(v, t)\}$. The resulting algorithms for RAPTOR and CSA are called ULTRA-RAPTOR and ULTRA-CSA.

3. Delay Buffer

This chapter introduces two delay models for public transit networks. The definitions established in the last chapter are extended to these new models. Furthermore, we show on a theoretical level how to use these models to calculate shortcuts for public transit networks with delay.

3.1 Modeling Delays

As defined in Chapter 2.1, a stop v of the trip $T \in \mathcal{T}$ has a scalar *arrival time* $\tau_{\text{arr}}(T, v)$ and a scalar *departure time* $\tau_{\text{dep}}(T, v)$. However, if the trip is delayed, these times may become invalid and the ULTRA query algorithm (e.g., ULTRA-RAPTOR) may produce suboptimal results. For example, if a trip that was used by a candidate journey arrives delayed, the candidate journey may miss the next trip or arrive later at the target. In both cases, there may be a new optimal journey with the same number of trips. Unfortunately, the ULTRA preprocessing cannot guarantee that the intermediate transfer of this journey is in the shortcut graph and therefore this journey can be found by the query algorithm. On the other hand, if the departure of a trip is delayed, this may produce new optimal valid journeys whose shortcuts are not necessarily in the shortcut graph. Note, this only affects the intermediate transfers, as these are preprocessed while everything else is computed at query time without any preprocessing. To accommodate for delays in the preprocessing of ULTRA, we introduce two new models for a public transit network: the *Delay-All* model and the *Delay-Arrivals* model. These are meant to model all possible delay scenarios within given bounds. Our goal is to adapt the ULTRA preprocessing to these new models, such that the query algorithms can answer all queries correctly, as long as the occurred delays do not exceed the given bounds.

For the Delay-All model the notion of departure and arrival times are relaxed from being a *point* in time to instead being a time *interval*, i.e., a minimum and a maximum departure and arrival time

$$\tau^{\text{int}}_{\text{dep}}(T, v) := [\tau^{\min}_{\text{dep}}(T, v), \tau^{\max}_{\text{dep}}(T, v)] \quad \text{and} \quad \tau^{\text{int}}_{\text{arr}}(T, v) := [\tau^{\min}_{\text{arr}}(T, v), \tau^{\max}_{\text{arr}}(T, v)].$$

Of course, $\tau^{\min}_{\text{arr}}(T, v) \leq \tau^{\min}_{\text{dep}}(T, v)$ and $\tau^{\max}_{\text{arr}}(T, v) \leq \tau^{\max}_{\text{dep}}(T, v)$ must always be true. This definition is forwarded to the departure time of a trip leg, i.e.,

$$\tau^{\text{int}}_{\text{dep}}(T^{ij}) := [\tau^{\min}_{\text{dep}}(T^{ij}), \tau^{\max}_{\text{dep}}(T^{ij})] := \tau^{\text{int}}_{\text{dep}}(T, T[i])$$

and equivalently for the arrival time

$$\tau^{\text{int}}_{\text{arr}}(T^{ij}) := [\tau^{\min}_{\text{arr}}(T^{ij}), \tau^{\max}_{\text{arr}}(T^{ij})] := \tau^{\text{int}}_{\text{arr}}(T, T[j]).$$

For a journey $J = \langle \vartheta_0, T_0^{ij}, \ldots, \vartheta_{k-1}, T_{k-1}^{mn}, \vartheta_k \rangle$ this naturally translates to

$$\tau^{\text{int}}_{\text{dep}}(J) := [\tau^{\min}_{\text{dep}}(J), \tau^{\max}_{\text{dep}}(J)] := \tau^{\text{int}}_{\text{dep}}(T_0^{ij}) - \tau_{\text{buf}}(T_0[i]) - \tau_\theta(\vartheta_0)$$

for the departure time and to

$$\tau^{\text{int}}_{\text{arr}}(J) := [\tau^{\min}_{\text{arr}}(J), \tau^{\max}_{\text{arr}}(J)] := \tau^{\text{int}}_{\text{arr}}(T_{k-1}^{mn}) + \tau_\theta(\vartheta_k)$$

for the arrival time. The notation $[a, b] + c := [a + c, b + c]$ was used for this. Two journeys J and J' are called *independent* if they do not share a stop event. This is true, among other things, if they do not share a trip.

The Delay-Arrivals model is similar, except that all departures are expected to be on time. For this, we can just require $\tau^{\min}_{\text{dep}}(T, v) = \tau^{\max}_{\text{dep}}(T, v)$ for all trips T with corresponding stops v in the Delay-All model. While the Delay-All model is more realistic, the Delay-Arrivals model is easier and hence used, for example, in MEAT [DPSW13]. The Delay-Arrivals model can also be interpreted in such a way that departures may be delayed, but a journey cannot make use of the delay. This means the traveler must be at the stop already when the vehicle would depart regularly.

In the following, we only consider intervals of the same fixed length d (e.g., ten minutes) for all arrival times, which we call the *delay buffer*. Hence, $\tau^{\text{int}}_{\text{arr}}(T, v) := [\tau^{\min}_{\text{arr}}(T, v), \tau^{\min}_{\text{arr}}(T, v) + d]$ holds for every trip $T \in \mathcal{T}$ and every stop v of T. The same is true for departure times in the Delay-All model, where we use the same delay buffer. Consequently, all arrival and departure times can be described by a scalar value again.

We define a *delay scenario* as a concrete instantiation of arrival and departure times from the given intervals. To be specific, for every trip $T \in \mathcal{T}$ and every stop $v \in \mathcal{S}$ of T, a delay scenario a has an arrival time $\tau^a_{\text{arr}}(T, v) \in \tau^{\text{int}}_{\text{arr}}(T, v)$ and a departure time $\tau^a_{\text{dep}}(T, v) \in \tau^{\text{int}}_{\text{dep}}(T, v)$. The delay scenario is part of the input for the query algorithms. ULTRA only guarantees optimal answers for the delay scenario where all stop events are on time. Our goal is to extend ULTRA such that it can guarantee optimal answers for all possible delay scenarios.

There must be at least one delay scenario in which $J = \langle \vartheta_0, T_0^{ij}, \ldots, \vartheta_{k-1}, T_{k-1}^{mn}, \vartheta_k \rangle$ is a valid journey for J to be considered a journey in our new models. For this, all intermediate transfers ϑ_l ($l = 1, \ldots, k-1$) of J must meet a new condition:

$$\tau^{\min}_{\text{arr}}(T_{l-1}^{ij}) + \tau_\theta(\vartheta_l) + \tau_{\text{buf}}(T_l^{mn}[m]) \leq \tau^{\max}_{\text{dep}}(T_l^{mn}).$$

Note that in the Delay-All model this condition is weaker than its scalar counterpart, since the departure of T_l^{mn} could be delayed. However, such a journey may not be a valid journey for a concrete delay scenario, if T_{l-1}^{ij} arrives delayed and/or T_l^{mn} departs rather punctually.

A delay scenario where no trips of the same route overtake each other is called *non-overtaking*. The delay scenario where all departure and arrival times are the minimum departure and arrival times is required to be non-overtaking. This ensures compatibility with the original model for a delay buffer of 0. Note that RAPTOR delivers correct results only for non-overtaking scenarios. However, this is not a big problem since routes where trips overtake each other can be split into several routes such that no trips overtake each other. The same is true for MCR. Apart from that, MCR is capable of handling any change of departure and arrival times since it does not use any preprocessing that relies on those times.

3.2 Delay-Domination

The notion of dominance, as introduced in the Preliminaries, is only applicable for concrete delay scenarios. In this section, we extend it to our new delay models. A set of journeys $\mathcal{X}$ *delay-dominates* a journey J' in this model if the following condition holds true: for every possible delay scenario (i.e., for every scalar time in the corresponding time intervals) in which J' is valid, there is a valid journey $J \in \mathcal{X}$ which dominates J'. We also define a concept of domination for single journeys. A journey J *worst-case-dominates* J' if all following conditions are fulfilled:

- J must be valid in all delay scenarios.
- J and J' start and end at the same vertex.
- J does not depart earlier than J' even if it departs on time and J' departs with maximum delay, i.e., $\tau_{\text{dep}}^{\min}(J) \geq \tau_{\text{dep}}^{\max}(J')$.
- J does not arrive later than J' even if it arrives with maximum delay and J' arrives on time, i.e., $\tau_{\text{arr}}^{\max}(J) \leq \tau_{\text{arr}}^{\min}(J')$.
- J does not use more trips than J'

In other words, J worst-case-dominates J' even if we assume the worst case for every arrival and departure time in J and the best case for J'. We want to compare these concepts of dominance.

Theorem 3.1. *Let J and J' be journeys such that J worst-case-dominates J'. Then $\{J\}$ delay-dominates J'.*

Proof. Consider an arbitrary delay scenario in which J' is valid. By definition of worst-case-dominance, J is valid and does not use more trips than J'. Moreover, $\tau_{\text{dep}}(J) \geq \tau_{\text{dep}}^{\min}(J) \geq \tau_{\text{dep}}^{\max}(J') \geq \tau_{\text{dep}}(J')$ is true. Similarly, $\tau_{\text{arr}}(J) \leq \tau_{\text{arr}}^{\max}(J) \leq \tau_{\text{arr}}^{\min}(J') \leq \tau_{\text{dep}}(J')$ applies. Therefore, J dominates J' in every delay scenario and consequently $\{J\}$ delay-dominates J'. □

This means that if J worst-case-dominates J', then J dominates J' for every possible delay scenario.

Theorem 3.2. *Let J and J' be independent journeys. Then the following is true:*

$$J \textit{ worst-case-dominates } J' \Leftrightarrow \{J\} \textit{ delay-dominates } J'.$$

Proof. "⇒": Follows directly from Theorem 3.1.

"⇐": We prove the contraposition. Let J not delay-dominate J'.

- Case 1: There is a delay scenario in which J' is valid, but J is not. For this scenario, there is no journey in $\{J\}$ that dominates J'.
- Case 2: J uses more trips than J'. Therefore, J does not dominate J' for any delay scenario.
- Case 3: $\tau_{\text{dep}}^{\min}(J) < \tau_{\text{dep}}^{\max}(J')$. Choose any delay scenario in which J' is valid, but set $\tau_{\text{dep}}(J)$ to $\tau_{\text{dep}}^{\min}(J)$ and $\tau_{\text{dep}}(J')$ to $\tau_{\text{dep}}^{\max}(J')$ by adjusting the first stop event of both journeys. This yields a valid delay scenario since both journeys are independent. Note that this does not change the validity of the journeys, since the intermediate transfers are unaffected. In this delay scenario, J does not dominate J'.

- Case 4: $\tau_{\text{arr}}^{\max}(J) > \tau_{\text{arr}}^{\min}(J')$. Choose any delay scenario, in which J' is valid, but set $\tau_{\text{arr}}(J)$ to $\tau_{\text{arr}}^{\max}(J)$ and $\tau_{\text{arr}}(J')$ to $\tau_{\text{arr}}^{\min}(J')$ by adjusting the last stop event of both journeys. This yields a valid delay scenario since both journeys are independent. Note that this does not change the validity of the journeys, since the intermediate transfers are unaffected. In this delay scenario, J does not dominate J'.

□

According to Theorem 3.2, both concepts of dominance are equivalent if we restrict the journey sets to a cardinality of one and assume the journeys to be independent. Therefore, J dominates J' for every possible delay scenario if and only if J worst-case-dominates J'. Note that if the journeys are not independent, they could share the same first or last stop event. In this case, our proof of cases 3 and 4 no longer works, since two contradicting values would be assigned to the same stop event.

As explained for ULTRA, a candidate journey is a journey with exactly two trips and without an initial and final transfer. To find exactly the set of necessary shortcuts for any possible delay scenario, we must enumerate all candidate journeys which are not delay-dominated by any journey set $\mathcal{X}$. In contrast to ULTRA, this means it is not sufficient to search for individual witnesses but rather witness sets. Unfortunately, this is much more complicated. Instead, we only consider worst-case-dominance. This allows us to search for individual witnesses again and leads to the following definition: a witness of the candidate journey J_{cand} is a journey J_{wit} which worst-case-dominates J_{cand}.

To find all shortcuts which are needed in any possible delay scenario, we have to enumerate all candidate journeys and discard those which are worst-case-dominated by another journey. Theorem 3.1 guarantees us that the discarded journeys are delay-dominated, which means they are dominated in every delay scenario. Therefore, we do not discard necessary candidate journeys and may only produce too many shortcuts but not too few. Assume there is a candidate journey J_{cand} and a set $\mathcal{X}$ of journeys which delay-dominates J_{cand}, but $|\mathcal{X}| > 1$ and there is no other set $\mathcal{X}'$ which delay-dominates J_{cand} and has only one element. Then we do not find a witness, despite J_{cand} being delay-dominated and therefore dominated in every possible delay scenario. This means we may produce unnecessary shortcuts in this situation. We want to investigate what situations can lead to superfluous shortcuts.

Theorem 3.3. *Let J' be a journey and $\mathcal{X}$ a set of journeys which delay-dominates J'. All journeys in $\mathcal{X}$ are independent of J'. Then there is a journey $J \in \mathcal{X}$, which worst-case-dominates J'.*

Proof. We look at the delay scenario a where the best case is assumed for J' and the worst case for every journey in $\mathcal{X}$. Concretely, the departures of J' and the arrivals of $\mathcal{X}$ are delayed as much as possible, whereas the arrivals of J' and the departures of $\mathcal{X}$ are on time. This is possible since all journeys in $\mathcal{X}$ are independent of J'. Consequently, $\tau_{\text{dep}}^{a}(J') = \tau_{\text{dep}}^{\max}(J')$, $\tau_{\text{arr}}^{a}(J') = \tau_{\text{arr}}^{\min}(J')$, $\tau_{\text{dep}}^{a}(J) = \tau_{\text{dep}}^{\min}(J)$ and $\tau_{\text{arr}}^{a}(J) = \tau_{\text{arr}}^{\max}(J)$ are true for every journey $J \in \mathcal{X}$. In this scenario, J' is valid because there is at least one scenario in which J' is valid (we would not call it a journey otherwise) and the times to reach the next trip in a is at least as good for every intermediate transfer since all vehicles arrive on time but depart as late as possible. By definition of delay-dominance, there is a journey $J \in \mathcal{X}$ which dominates J' for the scenario a.

Let b be any delay scenario in which J' is valid. The situation can only be better for J and worse for J', meaning the arrivals are earlier for J and later for J' and the departures are later for J and earlier for J' than in a. Therefore, J is still valid and

$\tau^b_{\text{dep}}(J) \geq \tau^a_{\text{dep}}(J) \geq \tau^a_{\text{dep}}(J') \geq \tau^b_{\text{dep}}(J')$. Similarly, $\tau^b_{\text{arr}}(J) \leq \tau^a_{\text{arr}}(J) \leq \tau^a_{\text{arr}}(J') \leq \tau^b_{\text{arr}}(J')$ holds true and since J does not use more trips than J', it dominates J' for every delay scenario in which J' is valid. Stated differently, $\{J\}$ delay-dominates J'. According to Theorem 3.2, J worst-case-dominates J'. □

If a shortcut is found which is not needed for any possible delay scenario, it means there is a candidate journey J_{cand} with this shortcut for which no witness (i.e., a worst-case-dominating journey) was found but which has a delay-dominating journey set $\mathcal{X}$. As Theorem 3.3 shows, there must be a journey $J \in \mathcal{X}$ which is not independent of J_{cand}. Otherwise, there would be a witness which would have been found. Therefore, the use of worst-case-dominance as opposed to delay-dominance can only lead to superfluous shortcuts in situations where candidate journeys and potential witnesses are not independent.

3.3 Optimizations

Before we explain how to use these new models to make ULTRA delay-robust, a few things are worth mentioning. It is actually not necessary to enumerate all candidate journeys. Consider two candidate journeys J and J' with the same source, target, intermediate transfer (the shortcut candidate) and departure time. Let $\tau_{\text{arr}}(J) \leq \tau_{\text{arr}}(J')$. If there is a witness J_{wit} of J, then J_{wit} is a witness of J' too, since all metrics (number of trips, departure time, arrival time) of J are better or equal than of J'. Therefore, we only need to consider J. If we do not need the shortcut for J, we do not need it for J' either.

Recall that the algorithm should yield as few shortcuts as possible, while still computing all shortcuts which are necessary to produce optimal results in the query algorithms. As seen above, if a candidate journey is delay-dominated by journeys which are not independent of the candidate journey, there may be no witness. In this case, we add the shortcut candidate of this candidate journey to the shortcut graph, although it might be not really necessary. To overcome this issue, special measures are engineered in the next chapter if candidate journeys and potential witnesses are not independent.

Another opportunity to reduce the number of shortcuts is restricting the possible delay scenarios. Note that some delay scenarios are physically not possible in real-life settings. For example, let $T \in \mathcal{T}$ be a trip with the stops $u, v \in \mathcal{S}$ such that u is visited before v. The stop event intervals $\tau^{\text{int}}_{\text{dep}}(T, u)$ and $\tau^{\text{int}}_{\text{arr}}(T, v)$ may overlap, leaving the possibility of a delay scenario a with $\tau^a_{\text{dep}}(T, u) > \tau^a_{\text{arr}}(T, v)$. In words, the trip T arrives at v before it even departs at u. Since no public transit vehicle is capable of traveling through time (yet), it seems plausible to ignore such delay scenarios. This means shortcuts which are only necessary in those scenarios can be discarded.

We can also ignore delay scenarios where trips of the same route overtake each other as justified by the following theorem:

Theorem 3.4. *Let c be a delay scenario and J_{cand} a candidate journey which is not dominated by any journey for c. Then there is a non-overtaking delay scenario b where J_{cand} is not dominated by any journey.*

Proof. Set the stop events in b of the two trips $T_1, T_2 \in \mathcal{T}$ used by J_{cand} to exactly the times in c. This ensures that the candidate journey is unchanged and still valid in b. All other stop events are set to the worst case: departures are punctual and arrivals are delayed as much as possible. Therefore, potential witnesses cannot get any better compared to scenario c. This delay scenario is not non-overtaking yet. However, the only remaining problem are the trips of the routes $r_1, r_2 \in \mathcal{R}$ which were used by J_{cand} since they may overtake the trips T_1 and T_2. All other trips do not overtake each other because the delay

scenario where all departure and arrival times are the minimum departure and arrival times is non-overtaking and a fixed delay buffer is used. For the trips of r_1 and r_2 we may have to increase departure times of later trips and decrease arrival times of earlier trips just enough to prevent overtaking. Concretely, consider a trip $T \in r_1$ earlier than T_1. For each stop v in T we change $\tau^b_{\text{arr}}(T, v)$ to $\min\{\tau^b_{\text{arr}}(T, v), \tau^b_{\text{arr}}(T_1, v)\}$. Similarly, if T is later than T_1, we set $\tau^b_{\text{dep}}(T, v)$ to $\max\{\tau^b_{\text{dep}}(T, v), \tau^b_{\text{dep}}(T_1, v)\}$. The same is done for trips in r_2 earlier or later than T_2. These modifications ensure that b is non-overtaking but could improve stop events in b compared to c leading to witnesses of J_{cand}. Assume J_{wit} is a witness dominating J_{cand} in b. The witness must have a trip leg T^{ij}_{wit} which was improved in b compared to c since otherwise J_{wit} would have been a witness in c as well. The trip leg T^{ij}_{wit} must use a trip T_{wit} of r_1 or r_2 because the arrival and departure times of all other trips did not improve in b compared to c. Let $r_{\text{cand}} \in \{r_1, r_2\}$ be the route of T_{wit} and T^{ij}_{cand} be the trip leg with the same stops of the trip T_{cand} in r_{cand} which was used by the candidate journey. Two cases remain:

1. $\tau^b_{\text{arr}}(T^{ij}_{\text{wit}}) < \tau^c_{\text{arr}}(T^{ij}_{\text{wit}})$: T_{wit} is an earlier trip than T_{cand}. Since we only decreased the arrival time as little as possible, it is actually the same as the arrival time of the candidate trip: $\tau^b_{\text{arr}}(T^{ij}_{\text{wit}}) = \tau^b_{\text{arr}}(T^{ij}_{\text{cand}})$. Therefore, T^{ij}_{cand} could be used just as well and that trip leg is equal in c.
2. $\tau^b_{\text{dep}}(T^{ij}_{\text{wit}}) > \tau^c_{\text{dep}}(T^{ij}_{\text{wit}})$: T_{wit} is a later trip than T_{cand}. Since we only increased the departure time as little as possible, it is actually the same as the departure time of the candidate trip: $\tau^b_{\text{dep}}(T^{ij}_{\text{wit}}) = \tau^b_{\text{dep}}(T^{ij}_{\text{cand}})$. Therefore, T^{ij}_{cand} could be used just as well and that trip leg is equal in c.

□

Theorem 3.4 allows candidate journeys to always greedily take the first reachable trip of a given route. This is important for the RAPTOR used by ULTRA and therefore for the correctness of our algorithm. Note that for a trip T and a stop v visited by T the departure of T at v is punctual in b but the arrival is not. Consequently, $\tau^b_{\text{dep}}(T, v) < \tau^b_{\text{arr}}(T, v)$ could be true in which case b includes some kind of time travel since T departs from v before it even arrived there. However, as long as we do not prevent this kind of time travel, this does not lead to missing shortcuts. This means we cannot ignore the corresponding delay scenarios.

4. DB-ULTRA

By using the newly introduced notion of worst-case-domination in ULTRA, a journey is only considered a witness of a candidate journey if it is at least as good even in the worst case. This means the arrival times of the witness and the departure times of the candidate journey (only in the Delay-All model and if needed to catch the trip) are delayed. The core idea of DB-ULTRA (*Delay Buffer* ULTRA) is the separation of candidate journeys and witnesses. This way we can focus on the best case for candidate journeys and on the worst case for potential witnesses. Note, however, that candidate journeys are also potential witnesses for other candidate journeys. Therefore, we must consider both cases for candidate journeys. By using the appropriate arrival and departure times, we can use the original definition of dominance once again.

The query algorithms established for ULTRA also work with DB-ULTRA shortcuts since initial and final transfers are handled at query time. To compute the shortcuts, a few changes have to be made to the original ULTRA algorithm. Pseudocode is given by Algorithm 4.1. The following explanations apply to both delay models if not declared otherwise.

4.1 Implementation Details

As seen in the pseudocode, the rough structure of DB-ULTRA is very similar to ULTRA. The differences are mainly how exactly the routes are scanned and the transfers are relaxed. This section describes the necessary changes in the order in which they appear in the pseudocode.

Collecting Departure Times

For the Delay-All model, the first difference to the original ULTRA algorithm is in line 4. Here we need to collect the maximum departure times for the candidate journeys since one could arrive at the stop at exactly this time, which may be too late for witnesses, but just right for this trip as the beginning of candidate journeys. An example for this is given by Figure 4.1. If we were to collect the minimum departure times, the only regarded departure time would be 12:00. In that case, the red trip at the top dominates the candidate journey even if we assume maximum delay for both red trips. However, one could arrive at s at 12:10 and have the luck to catch the first green trip just in time. If the first red trip departs on time, it is missed and the red journey cannot be used as a witness anymore. This shows

Algorithm 4.1: DB-ULTRA

Input: Stops $\mathcal{S}$, trips $\mathcal{T}$, routes $\mathcal{R}$, weighted transfer graph $G = (\mathcal{V}, \mathcal{E})$
Data: Distance $\mathsf{d}(a,b)$ from a to b in G, arrival time $\tau_{\mathsf{dir}}(a)$ of a by direct transfer
Output: Shortcut graph $G' = (\mathcal{S}, \mathcal{E}')$

```
1  forall s ∈ S do
2      Clear arrival times and Dijkstra queues
3      Compute d(s, ·) for every stop with Dijkstra
4      D ← Collect departure times of trips at s with corresponding reachable routes
5      forall τ_dep ∈ D in descending order do
6          forall v ∈ S do
7              τ_dir(v) ← τ_dep + d(s,v)
8          scanRoutesWithInitialTransfers()
9          collectRoutesServingUpdatedStops1()  // departing from source
10         scanRoutesWithoutInitialTransfers()
11         relaxWitnessTransfers()                  // relax witness labels
12         relaxCandidateTransfers()              // relax candidate labels
13         collectRoutesServingUpdatedStops2()
14         scanRoutes2()                             // similar to McRAPTOR
15         E' ← E' ∪ relaxTransfers2()  // collect non-dominated candidates
```

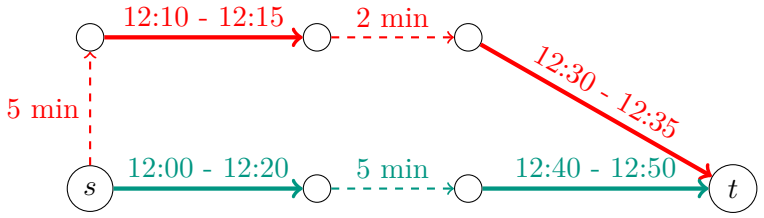

Figure 4.1: An example network to prove the need of using the maximum departure times in the Delay-All model while collecting them in line 4. The given departure and arrival times of the trips are the minimum departure or arrival times, respectively. All departure buffer times shall be zero. The stop s is the source, t the target. Let the green journey at the bottom be the candidate journey and set a delay buffer of ten minutes.

that the green shortcut is indeed needed to answer all queries correctly and is only found if we collect the maximum departure times. For witnesses, the minimum departure times are collected. In the Delay-Arrivals model, the procedure can be used unaltered as in ULTRA.

Scanning Routes with Initial Transfers

As opposed to ULTRA, we only scan routes we reached via an initial transfer in this route scanning step. These were collected in line 4. Because of the initial transfer, no candidate journey is built in this process, i.e., the considered journeys are only potential witnesses, and we can concentrate on the worst case. In this case, this means adding the delay buffer to every arrival time.

Scanning Routes without Initial Transfers

In line 10, only routes reachable without an initial transfer are scanned. These are beginnings of candidate journeys and are collected in line 9 by marking the source stop and collecting routes as described for RAPTOR in Chapter 2.4. In order to differentiate

between candidate journeys and potential witnesses (a candidate journey can be a witness for other candidate journeys), every stop has two arrival labels: a *candidate label* and a *witness label*. These labels contain tentative arrival times of the fastest known first trip of candidate journeys or witnesses, respectively. We can compare labels by comparing their respective arrival times. The witness labels are initialized with the arrival times computed in line 8. The candidate labels are initialized with ∞ once for every source stop. Both labels are preserved for different source departure times of the same source as usual.

When scanning a route at the stop v, we have to keep track of two *current* trips. The *candidate trip* is the trip T_{cand} with the earliest minimum departure time that fulfills $\tau_{\text{dep}}^{\max}(T_{\text{cand}}, v) \geq \tau_{\text{dir}}(v) + \tau_{\text{buf}}(v)$. Note that such a trip always exists because otherwise the route would not have been collected in line 9. This trip is used for the candidate labels. The *witness trip* is the earliest trip T_{wit} with $\tau_{\text{dep}}^{\min}(T_{\text{wit}}, v) \geq \tau_{\text{dir}}(v) + \tau_{\text{buf}}(v)$, if such exists. In that case we use it to update the witness labels, otherwise we do not update these labels for this route. In the Delay-Arrivals model, both trips are the same. Nevertheless, we distinguish them in the following for the sake of consistency between both delay models.

When visiting a stop u, the *current candidate arrival time* is computed as the minimum arrival time of the candidate trip at u. Similarly, the *current witness arrival time* is computed as the maximum arrival time of the witness trip at u. The candidate and witness labels of u are compared with these two, respectively, and updated if they are improved. The stop u is marked as an origin (the first stop of a shortcut) if the candidate label was improved. If the stop was already marked as an origin, all candidate journeys starting with the trip that formerly led to u and left the trip there are formally delay-dominated by a candidate journey starting with T_{cand} and leaving it at the origin. This is not a problem here since all shortcuts of the dominated candidate journeys are considered too by the dominating journeys. Stated differently, we can only enhance candidate journeys which never leads to missing necessary shortcuts (see Chapter 3.3). Finally, we recompute the witness trip as described above using the direct transfer labels τ_{dir}, possibly taking an earlier trip. Bear in mind that a trip could be found at u even if none was found previously. The candidate trip cannot be changed since a candidate journey is not allowed to use an initial transfer. Also, if the witness trip is now the same as the candidate trip or even an earlier trip, we do not need to update the following candidate labels since these will be dominated by the witness labels anyway.

Relaxing Intermediate Transfers

The propagation of arrival labels through the transfer graph happens in two distinct Dijkstra invocations. In the first (line 11), only the witness labels are relaxed. We start by putting all witness labels that were improved in the route scanning steps into the Dijkstra queue. Relaxing an edge (u, v) means improving the witness label of the vertex v if the new arrival time is smaller. If the vertex v is a stop and the new witness label is less than or equal to the candidate label, the stop is unmarked as an origin. The witness labels are preserved for different source departure times of the same source as usual.

The second Dijkstra invocation (line 12) relaxes the candidate labels. For this, the Dijkstra queue is initialized with the candidate labels of all origin stops. Every label has to store the origin stop now since multiple shortcuts with different origins can end in the same stop. That is because candidate journeys cannot dominate each other in general. Therefore, labels with different origin stops cannot dominate each other and every vertex of the transfer graph has to keep a bag of candidate labels. However, as seen in Chapter 3.3, labels with the same origin stop can indeed dominate each other. We can view this relaxation of transfers as a multi-criteria Dijkstra with the criteria arrival time and origin stop, where the notion of dominance for the origin stop is equality, i.e., a label can only dominate

another if they both share the same origin. When relaxing an edge (u, v), we first check if the current candidate label is dominated by the witness label of v. If not, we search the bag of v for a candidate label of the same origin stop and improve it if possible. If no such label was found, the current candidate label must be added to the bag of v. The candidate labels are not preserved for different source departure times of the same source since they are not witnesses and therefore not necessary for self-pruning.

Scanning Second Routes

The processing of a route for the second trip of the journeys in line 14 is inspired by McRAPTOR [DPW12]. After relaxing the intermediate transfers, every stop v has a witness label $w(v)$ and a bag of candidate labels $B(v)$. For each label, we have to find a current trip. The arrival time of a label ℓ is denoted as $\tau_{\text{arr}}(\ell)$. Once again, we concentrate on the worst case for the witness label. This means the witness trip T_{wit} is the first trip that fulfills $\tau^{\min}_{\text{dep}}(T_{\text{wit}}, v) \geq \tau_{\text{arr}}(w(v)) + \tau_{\text{buf}}(v)$ since it departs on time. We assume maximum delay of T_{wit} at all subsequent stops.

Unlike in the first route scanning step, it is now enough to concentrate only on the best case for the candidate labels. This is because for them we already assumed the best case for the first taken route. Therefore, they cannot be used as witnesses — which assume the worst case for both routes — anymore. The candidate trip $T_{\text{cand}}(\ell)$ of a label $\ell \in B(v)$ is the first trip which fulfills $\tau^{\max}_{\text{dep}}(T_{\text{cand}}(\ell), v) \geq \tau_{\text{arr}}(\ell) + \tau_{\text{buf}}(v)$ since it can depart delayed in the Delay-All model. We assume the trip is on time for all subsequent stops. Only if this candidate trip $T_{\text{cand}}(\ell)$ is earlier than the witness trip T_{wit}, we need to consider it in the following since otherwise, the candidate journey is dominated for all succeeding stops as targets. Formally, we did not find a worst-case-dominating witness if the candidate and witness trip are the same because we assume maximum delay at the target for the witness trip. However, the candidate journey and the potential witness share the same last stop event (i.e., they are not independent), which cannot be delayed and on time at the same time. In fact, if both use the same trip, they will always arrive at the target at the same time, and therefore the candidate journey is dominated.

All these candidate labels are stored with their current trip in a route bag. We also have to include the associated shortcut candidate for each candidate label since a target stop (the target of a candidate journey) can have several candidate labels with different shortcut candidates. Moreover, the shortcut candidate is needed later to possibly add the shortcut to the shortcut graph. For every stop u, we take three steps. The first is updating the witness label of u with the current witness trip. In the second step, we merge the route bag into the bag of u by adding all labels and only keeping one label per shortcut candidate. The last step updates the route bag by merging u's candidate labels created during the preceding transfer relaxation into the route bag, as well as updating the witness trip. After that, all current trips must be recomputed. If the witness trip has changed, all other labels with later trips can be removed from the route bag. Like for the relaxation of intermediate transfers, only the witness labels are preserved for different source departure times of the same source.

Final Relaxation of Transfers

Collecting the shortcuts of candidate journeys that are not dominated takes place in line 15. As in line 11, we only propagate witness labels through the transfer graph. The witness label does not change anymore after a stop is taken out of the Dijkstra queue. Thus, this is the moment to add all shortcuts of the non-dominated candidate labels at this stop to the resulting shortcut graph. A shortcut is only added to the shortcut graph if the arrival time at the target of the corresponding candidate journey is less than the witness label since that could have been changed in this phase.

4.2 Optimizations

As we have seen in Chapter 3, only considering worst-case-dominance could lead to superfluous shortcuts. In the following, some optimizations and implementation details are presented to decrease the number of shortcuts and the preprocessing time.

Stopping Criteria

The original ULTRA algorithm employs two stopping criteria, one of which we adapted to DB-ULTRA. For the final relaxation of transfers in line 15, we count the number of stops reached by a candidate journey in line 14. Each time such a stop is taken out of the Dijkstra queue, the counter is decreased by one. When it reaches zero, the relaxation can be aborted since no candidate journeys can be found in this round anymore. The other stopping criterion of ULTRA affects the intermediate relaxation of transfers. It could be adapted to DB-ULTRA in line 11. However, in contrast to ULTRA, there is no clear sensible moment when to stop the relaxation since ULTRA stops it once every candidate journey is processed, but in DB-ULTRA candidate journeys are processed after the witnesses. In fact, it could be stopped at any time since only witnesses are considered in line 11, and therefore a premature stop may only lead to superfluous shortcuts. We decided not to implement this stopping criterion. The profiling in Chapter 5.7 supports this decision.

Faster Transfer Relaxation of Candidate Labels

We can accelerate the relaxation of candidate labels in line 12 by splitting the single Dijkstra invocation into several independent Dijkstra searches, one for each origin. This is possible since candidate labels with different origin stops cannot dominate each other anyway. For three reasons this is much faster than a single call of Dijkstra's algorithm:

- The Dijkstra queue stays smaller, allowing for faster access and modifications.
- We do not need to search through the whole bag to find a label with the same origin. If such a label exists, it is always the last element in the bag since we only add labels at the end, and it could only have been added in this invocation of Dijkstra's algorithm.
- In our implementation, independent Dijkstra searches allow the use of a simpler heap for the Dijkstra queue because the labels for each origin are guaranteed to stay at the same position in memory for the duration of Dijkstra's algorithm. Therefore, memory pointers can be used to implement a fast and cache-friendly heap.

Starting with the Same First Trip

Until now, we mostly only used the notion of worst-case-dominance, i.e., the best case was assumed for candidate journeys and the worst case for witnesses. As seen in the previous chapter, this is fine as long as the candidate journey and witnesses are independent. Otherwise, we may produce unnecessary shortcuts. We already considered the case where the last stop event of candidate journey and witness is shared in "Scanning Second Routes". Now we want to consider the case where the first stop event is shared.

At the source, we assume the worst case for all witnesses, which means all trips depart on time. However, for the Delay-All model, the departure times collected in line 4 are the maximum departure times. That is, no witness can use the same first trip as a currently examined candidate journey. This unfairly weakens witnesses because a specific trip cannot depart on time and delayed at once. A witness that uses the same first trip as a candidate journey would not be found by the basic form of DB-ULTRA. Formally speaking, if a

journey J for which the worst case is assumed for all stop events except the first (where the best case, i.e., maximum delay is assumed) dominates a candidate journey J' that uses the same first stop event, then J' is delay-dominated. This is true since, as usual, the candidate journey J' can only get worse in a specific delay scenario, whereas J can only get better except for the first stop event. However, this stop event is shared with the candidate journey and therefore J will never depart earlier than the candidate journey.

To find such witnesses, we take several measures. The first takes place after the intermediate relaxation of transfers in line 12. In this situation, some stops are shortcut destinations and have candidate labels with corresponding origins. For every destination, we group the labels by the trip used to reach the origin. In every group, we only keep the earliest label ℓ_e and every label ℓ that is less than the delay buffer d later, i.e., $\tau_{\text{arr}}(\ell) < \tau_{\text{arr}}(\ell_e) + d$. Every other label is not necessary since it is dominated by ℓ_e even if the first trip arrived with a delay at ℓ_e's origin. We refer to this measure as SFT1.

Another measure we call SFT2 is implemented in line 14 when adding candidate labels to the route bag. Consider the stop v with the bag $B(v)$ of candidate labels and the label $\ell \in B(v)$. Instead of only finding the trip $T_{\text{cand}}(\ell)$ for the candidate journey (i.e., the first trip which fulfills $\tau^{\max}_{\text{dep}}(T_{\text{cand}}(\ell), v) \geq \tau_{\text{arr}}(\ell) + \tau_{\text{buf}}(v)$), we also find two additional trips to dominate other candidate journeys which use the same first trip. The trip $T_{\text{sft}}(\ell)$ represents a witness which uses the same first trip as the candidate journeys it can possibly dominate. The witness may leave the first trip at another stop, i.e., it can have another shortcut origin than the candidate journeys it is allowed to dominate. Let d be the delay buffer. The trip $T_{\text{sft}}(\ell)$ is calculated as the first trip which fulfills $\tau^{\min}_{\text{dep}}(T_{\text{sft}}(\ell), v) \geq \tau_{\text{arr}}(\ell) + \tau_{\text{buf}}(v) + d$ since we have to assume the worst case at all stop events except for the first, i.e., the witness arrives delayed at the shortcut origin and consequently at the shortcut destination (hence the addition of d) and the second trip departs on time. This trip can be used to dominate other candidate labels in the route bag which use a later trip than $T_{\text{sft}}(\ell)$ and to prevent the addition of further candidate labels later if they use the same or a later trip.

Similarly, we compute the trip $T_{\text{sftao}}(\ell)$ that represents a witness which uses not only the same first trip but also leaves it at the same shortcut origin as the candidate journeys it can possibly dominate. This means $T_{\text{sftao}}(\ell)$ is the first trip which fulfills $\tau^{\min}_{\text{dep}}(T_{\text{sftao}}(\ell), v) \geq \tau_{\text{arr}}(\ell) + \tau_{\text{buf}}(v)$ since it arrives at the shortcut origin and consequently at the shortcut destination at the same time as the candidate journeys it may dominate, which is on time. When using this trip to dominate other candidate labels with the same first trip and origin, caution must be taken. There are some details that must be minded. Consider a trip T_{cand} of a candidate journey J_{cand} which uses the same first trip and origin as $T_{\text{sftao}}(\ell)$. There may be some spare time between the arrival at the shortcut destination and the departure of the second trip, which we call the *enter-buffer*. The first trip taken by J_{cand} could arrive that much later at the origin without changing J_{cand} since T_{cand} is reached, but $T_{\text{sftao}}(\ell)$, which assumed punctual arrival at the shortcut destination, could be missed. To solve this problem, the enter-buffer for the candidate journey trip T_{cand} and the witness trip $T_{\text{sftao}}(\ell)$ can be stored in the route bag. We now only let the latter dominate the former if its enter-buffer is equal or bigger.

Moreover, the delay at the origin could be even greater than the enter-buffers of both trips. To cover this case, we propose an approximation: We assume maximum delay at the origin of the first trip used by J_{cand} and the candidate journey of ℓ. The journey J_{cand} is allowed to take the next trip $T_{\text{cand},2}$ after T_{cand}, although it might miss it. This may only improve J_{cand} and therefore does not lead to missing shortcuts. Fortunately, we already calculated the witness trip that assumed maximum delay at the source: $T_{\text{sft}}(\ell)$. If it is the same or an earlier trip than $T_{\text{cand},2}$, then the candidate journey J_{cand} can finally be discarded, i.e., the corresponding label can be removed from the route bag or be prevented from being added

to the route bag. Note that this approximation allows us to only regard maximum delay at the source. Otherwise, we would have to consider all trips between T_{cand} and the one that must be used if the first trip of J_{cand} is delayed by the delay buffer at the origin, which might not be the next trip.

Similarly, this idea of manipulating candidate journeys to dominate other candidate journeys can be used when arriving at the target in line 14. First, the current label ℓ_c could be dominated by an intermediate candidate label ℓ_i that was formed in the relaxation of transfers in line 12. These labels represent journeys with only one trip. If ℓ_i used the same origin, it can be used directly to possibly dominate ℓ_c. If it only used the same first trip, the delay buffer must be added to ℓ_i since we have to assume maximum delay at its origin. This measure is called SFT3. Later, ℓ_c can dominate or be dominated by other labels of the same round (SFT4). This works essentially the same as SFT2 (including the enter-buffer and the mentioned approximation) but compares the arrival times at the target of the candidate journeys and the witnesses which used the same first trip $(T_{\text{sft}}, T_{\text{sftao}})$ instead of comparing those trips directly. We have to store the same-first-trip arrival times and the enter-buffers in the arrival labels for this to work. In contrast to SFT2, the candidate journey and the witness may used a different route for the second trip.

Finally, the same-first-trip arrival times can be used after line 15 in Algorithm 4.1 to dominate remaining labels by relaxing transfers. This means the shortcuts must be added to the shortcut graph afterward. We may refer to this optimization as SFT final transfers. For each route used by a candidate journey for the first trip, one Dijkstra search is invoked to propagate the same-first-trip labels with the first trip in this route (as computed with trips such as $T_{\text{sft}}(\ell)$ above) from the targets through the transfer graph. An edge (u, v) is only relaxed if the new arrival time at v is smaller than the arrival time computed in line 15. This restricts these searches without adding an explicit stopping criterion. If u is a stop, all labels at u with the same first trip can be erased if their arrival time is greater than the just computed one.

Likewise, the same-first-trip arrival times that also assumed the same origin (as computed with trips such as $T_{\text{sftao}}(\ell)$ above) could be propagated through the transfer graph. However, as for SFT2 and SFT4, the enter-buffers must be compared and the approximation applied. This is possible by invoking Dijkstra's algorithm once for every same-first-trip label and using its enter-buffer for comparison with the candidate labels. Unfortunately, the number of same-first-trip labels can be quite high. Consequently, this optimization is very slow and leads to very long preprocessing times, even for small networks. Therefore, we do not activate this optimization.

Note that the whole idea of this section can only be applied to the first trip because at the source the arrival time is fixed. At the shortcut destination, every candidate label is strictly earlier than the witness label. Assume there is a trip T and a witness label $w(v)$ at a shortcut destination v with $\tau_{\text{dep}}^{\min}(T, v) < w(v) \leq \tau_{\text{dep}}^{\max}(T, v)$. At query time, the departure time $\tau_{\text{dep}}(T, v)$ could be a second before the witness arrives at v. In that case, the candidate can reach this trip, but the witness does not. This means we cannot assume the trip is reached by the witness to dominate other labels that used the same trip.

Pruning with Existing Shortcuts

As in ULTRA (see Chapter 2.6.1), candidate journeys whose shortcuts are already in the shortcut graph do not have to be computed anymore. This can be used by deleting all labels that correspond with such a shortcut after the intermediate relaxation of candidate labels in line 12. In contrast to ULTRA, this optimization not only leverages the stopping criterion for the final relaxation of transfers, but also saves a lot of work in the following route scanning step in line 14. This is because perhaps fewer routes need to be scanned

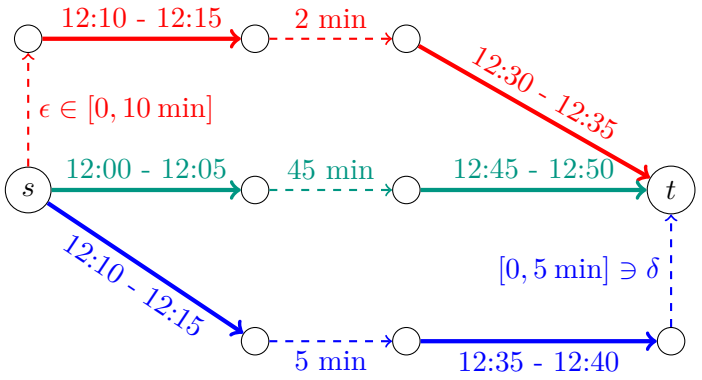

Figure 4.2: A network to underline the stated problem that time travel cannot be easily prevented at the first trip. The given departure and arrival times of the trips are the minimum departure or arrival times, respectively. All departure buffer times shall be zero. The stop s is the source, t the target. Let the green journey in the middle be the candidate journey and set a delay buffer of ten minutes. The blue journey below and the red journey above are potential witnesses.

and fewer trips be found per route. Unfortunately, this optimization collides with the ones described in "Starting with the Same First Trip" since the labels that are deleted could be used to dominate other candidate journeys. This leads to many superfluous shortcuts, rendering the same-first-trip-optimizations much less useful. Therefore, one must decide whether to decrease the number of shortcuts (i.e., using the same-first-trip-optimizations) or the preprocessing time (i.e., prune with existing shortcuts). If not stated otherwise, we use the same-first-trip-optimizations and deactivate pruning with existing shortcuts.

Preventing Time Travel

In Chapter 3.3, we already discussed the possibility of restricting the allowed delay scenarios. This paragraph is dedicated to investigating how to prohibit delay scenarios with time travel in our algorithm. For candidate journeys, we assume maximum delay for the departure and no delay for the arrival in the Delay-All model. However, if the two stop events are close (one or two minutes are common in urban networks) and the delay buffer large enough, the arrival can actually be before the departure. This is obviously physically not possible and therefore leads to superfluous shortcuts since many candidate journeys are evaluated better by DB-ULTRA than they truly are. To avoid this at the second trip, we save the arrival time at the stop where the trip is entered for every candidate label in the route bag. For every subsequent stop, the arrival time is calculated as the maximum of this time and the arrival time of the trip. On a theoretical level, we do not use the minimum arrival time of the candidate journey at the target among all delay scenarios for the worst-case-domination, but only of those delay scenarios without such time travel. This prevents any time travel, but still allows teleportation and superluminal (faster-than-light) travel, which are both impossible too [Ein05]. For this, the minimum travel time between the stops could be calculated with their physical distance and the maximum speed of the vehicle. However, approximations showed the number of saved shortcuts are probably not worth the effort.

One could think the same idea could be used for the first trip too. In line 4, we used the maximum departure times, so time travel is even more apparent here. The difficulty is the traveler could be at the source — and consequently the first trip departing — earlier than the maximum departure time. In that case, the trip could arrive earlier too for the subsequent stops. If we prevent time travel as above, the candidate journeys would be considered worse than they truly are, leading to missing shortcuts and therefore to incorrect

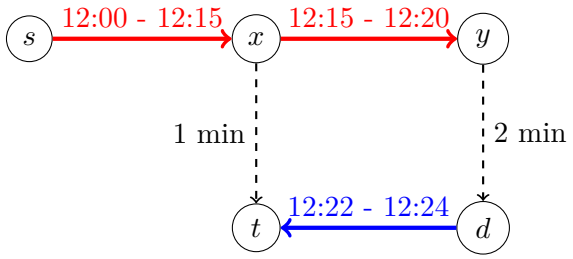

Figure 4.3: An example of the second kind of time travel. The given departure and arrival times of the trips are the minimum departure or arrival times, respectively. All departure buffer times shall be zero. A delay buffer of ten minutes is used.

queries. Stated differently, it is not necessarily the best case for a candidate journey if the first departure is delayed when time travel is prohibited this way. Note that this is not a problem for the second trip since we assume the second trip to only be delayed as much as necessary for the candidate journey to reach the trip.

Unfortunately, it is not enough to collect both minimum and maximum departure times either. Consider Figure 4.2. Right now, $\epsilon = 5\,\text{min}$ and $\delta = 4\,\text{min}$ are fixed. In the following, we assume time travel is prevented for all trips as described. For a departure time of 12:00 at s, the red journey above dominates the candidate journey even if its trips are delayed by ten minutes. Now assume a departure time of 12:10. Then the candidate journey arrives at 12:55 at t. Notice that for this the candidate journey used teleportation two times. It arrives at the origin at 12:10, the destination at 12:55 and then the target at 12:55. This means the blue journey below now dominates the candidate journey. However, for a departure time of 12:06 at s, the candidate journey arrives at 12:51 at t. This means the blue journey is not a witness. Moreover, the first trip of the red journey is missed, so the red journey is no witness either. Therefore, the green shortcut is indeed needed to answer all queries correctly on this network but is not found by only considering the minimum and the maximum departure time while collecting departures. In fact, only for a departure time in the interval $(12{:}10 - \epsilon, 12{:}05 + \delta)$ at s the green shortcut is found by DB-ULTRA if time travel is prevented for all trips. This means the departure times that need to be collected are inherently dependent on the network — even on data far away from the source such as δ — and this problem cannot be solved this way. It is unclear whether this problem can be solved efficiently and therefore we must accept the superfluous shortcuts for now.

Another kind of time travel that even affects the Delay-Arrivals model occurs at consecutive stops of the same route in combination with the solution to the problem described above in “Starting with the Same First Trip”. Let r be a route with stops x and y where x is visited before y in r. Say T is a trip in r for which the difference between the arrival times of x and y is less than the delay buffer. To dominate a candidate journey with y as the shortcut origin, we already figured a journey that uses T and disembarks at x could be used. Until now, T was assumed to have maximum delay at x. However, if the delay is that large, T must be delayed at y too.

Figure 4.3 provides an example for this. The candidate journey starts at the source s, uses the red trip T to the origin y and walks from there to the destination d. It takes the blue trip there to the target stop t where it arrives at 12:24. A potential witness using the same first trip could take the red trip to x and walk directly to t. However, we assumed the trip T to be delayed by ten minutes at x, whereas it arrives punctually at y. Therefore, the witness arrives at 12:26 at the target, two minutes later than the candidate journey. The candidate journey is not dominated and the shortcut from y to d added to the shortcut graph. In real life, this is not possible since the trip T cannot arrive at 12:25 at x and

already five minutes earlier at y. We have to consider two cases to cover all delay scenarios without such time travel:

1. The delay at x is less than eight minutes: The witness arrives before 12:24 at t and dominates the candidate journey. The shortcut from y to d is not necessary for these delay scenarios.
2. The delay at x is $\delta \in [8\text{ min}, 10\text{ min}]$: The witness arrives at $12{:}16 + \delta$ at t. The candidate journey will not arrive before $12{:}15 + \delta$ at the origin y, even if T teleports from x to y. Consequently, it will not arrive before $12{:}17+\delta$ at d. In the Delay-Arrivals model, it will miss the blue trip and never arrive at the target. In the Delay-All model, the candidate journey may catch the blue trip, but even with teleportation, it will not arrive at the target t before $12{:}17 + \delta$, which is later than the witness. The shortcut from y to d is not necessary for these delay scenarios either.

This means we can assume a delay of not more than the difference of the arrival times because for every extra second of delay T must be delayed at y by at least this amount. That knowledge can be used at every measure described above. For example, at the first measure (SFT1) we do not keep every label per group anymore that is less than the delay buffer later than the best label. For a label ℓ whose origin comes after the best label's origin in the first trip, we can assume the best label ℓ_e to only be delayed the difference of the origin arrival times. This means we only keep ℓ if $\tau_{\text{arr}}(\ell) < \tau_{\text{arr}}(\ell_e) + \min\{d, \tau_{\text{arr@o}}(\ell) - \tau_{\text{arr@o}}(\ell_e)\}$, where d is the delay buffer and $\tau_{\text{arr@o}}(\ell)$ and $\tau_{\text{arr@o}}(\ell_e)$ denote the arrival time at the origin of ℓ or ℓ_e, respectively. For other labels whose origins come before ℓ_e's origin, the delay buffer is added as usual.

SFT3 can be adapted similarly for the case where the shortcut origin is different and the witness' origin comes before the candidate's origin of the same trip. Instead of adding the delay buffer to the intermediate label ℓ_i, we only add the difference of the origin arrival times plus the enter-buffer of the candidate journey (if this number is smaller than the delay buffer). The enter-buffer is necessary since the first trip can be delayed this much at the origin of the candidate journey without changing it.

For SFT2 and SFT4, we try to recompute the same-first-trip trips T_{sft} on the fly before each domination check. Like in the last paragraph, we do not assume maximum delay at the shortcut destination anymore and instead only add the difference of the origin arrival times plus the enter-buffer of the candidate journey. This may lead to earlier same-first-trip trips. Note that SFT4 normally compares arrival times and not trips. We do not store trips in the arrival labels but only in the route bag label. Therefore, we use this optimization only to prune arrival labels with the current route bag label and not to prune the current label with existing arrival labels. For the same reason, we do not prevent this kind of time travel at the SFT final transfers measure.

The third kind of time travel concerns the enter-buffer. For candidate journeys in the Delay-All model, the enter-buffer is calculated as the difference of the maximum departure time $\tau^{\max}_{\text{arr}}(T, v)$ of the used trip T and the arrival time of the candidate journey J at the destination v (capped by the delay buffer). However, the minimum arrival time at succeeding stops may be earlier than $\tau^{\max}_{\text{arr}}(T, v)$. This means that a delay of T at v slightly smaller than the enter-buffer indeed changes the candidate journey J since the delay is propagated to the next stops even with teleportation. We can try to calculate a smaller enter-buffer for each target t as the difference of the arrival time $\tau_{\text{arr}}(J)$ at t and the arrival time at the shortcut destination v. For every second additional delay at v, the arrival at t must be delayed at least this much as well. We store this new enter-buffer in the arrival label at t instead of the original enter-buffer as calculated for SFT2 and use it for SFT4 and the time travel measure described in the last two paragraphs for SFT3 and SFT4. For

these, a smaller enter-buffer of the candidate journey can lead to fewer shortcuts. Bear in mind that we do not know the target in SFT2 and therefore cannot use this smaller enter-buffer there.

Preferring Existing Shortcuts

Sometimes when a route bag label dominates another as described above in "Starting with the Same First Trip", the two labels are actually coequal. In that case, it does not matter for the correctness of the algorithm which label is chosen. However, if one is associated with a shortcut that has already been added to the shortcut graph, we prefer it, meaning it dominates coequal labels. This way, we avoid producing shortcuts that have an alternative in the resulting shortcut graph. Since the shortcut graphs are thread-local, this optimization works better the fewer threads are used.

5. Experiments

This chapter is dedicated to evaluating the quality and performance of our newly developed preprocessing technique DB-ULTRA. If not stated otherwise, all experiments that require measurements of running times were run on a machine with two 8-core Intel Xeon Skylake SP Gold 6144 CPUs clocked at 3.5 GHz with 24.75 MiB of L3 cache and 192 GiB of DDR4-2666 RAM. This is the same machine as was used in the original ULTRA paper [BBS+19]. All algorithms were implemented in C++17 compiled with GCC version 9.3.1 and optimization flags `-O3` and `-march=native`. Except for the preprocessing algorithm DB-ULTRA, all algorithm implementations were provided by the authors of ULTRA. Some implementations may have changed since the experiments were performed for ULTRA, which could lead to different running times than those reported in the ULTRA paper.

5.1 Networks

Like the implementations, the networks were provided by the authors of ULTRA [BBS+19]. Most experiments were conducted on the public transit network of Switzerland. It was downloaded from a public GTFS feed[1] and contains the two successive business days 30th and 31st of May 2017. To test the scalability of our algorithm, some experiments use the public transit network of Germany. It was gathered from `bahn.de` and consists of two successive identical days from Winter 2011/2012. Stops outside of Switzerland or Germany, respectively, were removed. The transfer graphs for both networks were obtained from OpenStreetMap[2]. All vertices that are not stops and have a degree of one or two were contracted. The transfer mode is walking with a speed of 4.5 km/h on each edge. The transitively closed transfer graphs which are needed for standard RAPTOR and CSA are computed by inserting an edge between all stops with a distance lower than 15 minutes for Switzerland and 8 minutes for Germany before computing the transitive closure. Table 5.1 gives an overview of the described networks.

For the preprocessing and some query algorithms, a contracted transfer graph where stops are not contracted (Core-CH) is useful. As explained in Chapter 2.5.1, this can be achieved with a partial CH. An average vertex degree of 20 was chosen for both networks. This is the same for Germany as in the original ULTRA paper. For Switzerland, the authors used a degree of 14 to minimize the preprocessing time, including the time to calculate the partial CH. However, DB-ULTRA spends more time with Dijkstra searches than ULTRA.

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

Table 5.1: The important figures regarding the public transit networks with their transfer graphs used to evaluate DB-ULTRA. Notice that the Switzerland network is not exactly the same as the one used in the ULTRA paper.

Network	Stops	Routes	Trips	Stop events	Vertices	Full edges	Tran. edges
Switzerland	25 125	13 785	350 006	4 686 865	603 691	1 853 260	4 671 514
Germany	244 055	231 089	2 387 297	48 495 169	6 872 105	21 372 360	22 645 480

These get faster with a greater average vertex degree. Therefore, the longer contraction time is made up for. The contraction took 4:57 minutes for the Switzerland network and 24:56 minutes for Germany on a single CPU core. For all following running times, the transfer graph is expected to already be contracted.

5.2 ULTRA Shortcuts and Delays

Before we evaluate our new algorithm, it is interesting to see how good the results of the original ULTRA-RAPTOR algorithm with ULTRA shortcuts are if the trips are delayed. After all, the goal of this thesis is to augment ULTRA such that algorithms that use the shortcuts (e.g., ULTRA-RAPTOR) yield optimal results even if trips are delayed. To assess what happens if trips are delayed, we compare the results of ULTRA-RAPTOR with ULTRA shortcuts and DB-ULTRA shortcuts. For both delay models and several delay buffers, 100 000 random ULTRA-RAPTOR queries were run on the Switzerland network. Before running the queries, the arrival times (and departure times for Delay-All) were delayed by a random amount not larger than the delay buffer, following a linear distribution with the highest probability at zero and the lowest at the delay buffer.

The delays are calculated route-wise, always beginning at the first trip and stop. The delays are propagated to following stops and trips if necessary to prevent time travel and ensure the optimality of ULTRA-RAPTOR queries with DB-ULTRA shortcuts. In the Delay-Arrivals model, all departures are on time. This means only arrival times are affected by this propagation, and we do not change a departure time even if the arrival time for the same trip and stop is greater. The delays are recalculated every 500 queries, yielding 200 different delay scenarios overall per delay model and buffer. The results are shown in Table 5.2.

Most queries are answered correctly with ULTRA shortcuts. However, for some queries the difference of the earliest arrival time is large. The number of queries which have different results are greater in the Delay-All model and for higher delay buffers. Note that only the fastest journey of each result is considered for the earliest arrival time. Journeys with fewer trips are ignored. As Wagner and Zündorf pointed out, this underestimates the importance of walking [WZ17]. For example, the fastest journey could be equal for both shortcut graphs, but with DB-ULTRA shortcuts a journey with fewer trips and an only slightly later arrival time could be found which is not found with ULTRA shortcuts.

To measure the influence of the *number* of delayed stop events, another experiment was conducted. Each stop event was delayed with a given probability p. Afterwards, the delays were propagated as explained above. For different probabilities, 100 000 random ULTRA-RAPTOR queries with 200 delay scenarios were run each. As Figure 5.1 shows, the highest difference with ULTRA and DB-ULTRA shortcuts is at a probability of 25%. For higher probabilities, the number of queries with different results decreases and reaches 0 for a probability of 100% since this is the same delay scenario as without any delays, only that every stop event is shifted by ten minutes.

Table 5.2: Journey quality comparison of ULTRA and DB-ULTRA. Reported are for various delay buffers (DB) the percentages of queries for which the results with ULTRA shortcuts and DB-ULTRA shortcuts differ (#Diff.). This is the same as the number of queries for which the queries are not optimal anymore with ULTRA shortcuts. The average difference of the earliest arrival time of all queries (Avg. diff. all) is shown in the HH:MM:SS format. Moreover, the average (Avg. diff.) and maximum (Max. diff.) differences of the earliest arrival time of the queries with different results for ULTRA and DB-ULTRA are shown. Only the fastest journey is considered and journeys with fewer trips are not relevant for those statistics. For the Delay-Arrivals model and a delay buffer of 30 minutes, a query was found which has no solution with ULTRA shortcuts, but has one with DB-ULTRA shortcuts (#ULTRA fails). This means there is no path from the source to the target stop in the unrestricted transfer graph, as this path would always be in the resulting Pareto set. The failed query is not represented in the average and maximum difference.

Delay model	DB [min]	#Diff.	Avg. diff. all	Avg. diff.	Max. diff.	#ULTRA fail
Delay-All	5	1.42%	00:00:10	00:11:06	06:01:06	0
	10	3.57%	00:00:22	00:10:06	07:55:03	0
	30	12.58%	00:01:21	00:10:40	16:11:37	0
Delay-Arrivals	5	0.56%	00:00:05	00:15:46	05:25:58	0
	10	0.90%	00:00:11	00:19:51	09:32:02	0
	30	1.16%	00:00:16	00:23:12	12:12:49	1

To conclude, standard ULTRA shortcuts are indeed not sufficient if delays are possible. While most queries are answered correctly, especially for small delays or the Delay-Arrivals model, some have an immensely later arrival time of several hours. This problem is solved by DB-ULTRA.

5.3 Switzerland Preprocessing

In this section, the DB-ULTRA preprocessing is evaluated. The algorithm is implemented as explained in Chapter 4. Different configurations where certain optimizations are deactivated are investigated in Chapter 5.6. DB-ULTRA has two important parameters: the delay buffer and the delay model. The most interesting result is how many shortcuts are computed for a given set of parameters since that is the only influence the preprocessing has on the query times of algorithms as ULTRA-RAPTOR. The fewer shortcuts are computed, the better. Figure 5.2 depicts the number of computed shortcuts for delay buffers between 1 and 30 minutes and both delay models. For comparison, original ULTRA, which outputs 135 738 shortcuts, is also shown. For this, the same contracted graph with average vertex degree 20 was used. The plot clearly shows that while the number of shortcuts remains bearable even for fairly high delay buffers in the Delay-Arrivals model, the number of shortcuts increases drastically in the Delay-All model for high delay buffers, rising to over twelve million shortcuts for a delay buffer of 30 minutes.

The running times shown in Figure 5.3 exhibit a similar pattern as the number of shortcuts. While the running time only slowly increases in the Delay-Arrivals model for higher delay buffers, it rises rapidly in the Delay-All model, reaching 2:31:12 hours for a delay buffer of 30 minutes. Compared to the 8:01 minutes the original ULTRA algorithm took, this is long. However, DB-ULTRA is a preprocessing technique and depending on the context, the long preprocessing time may be worth it, for example, if the shortcuts are used for a real-time application which would not be possible otherwise.

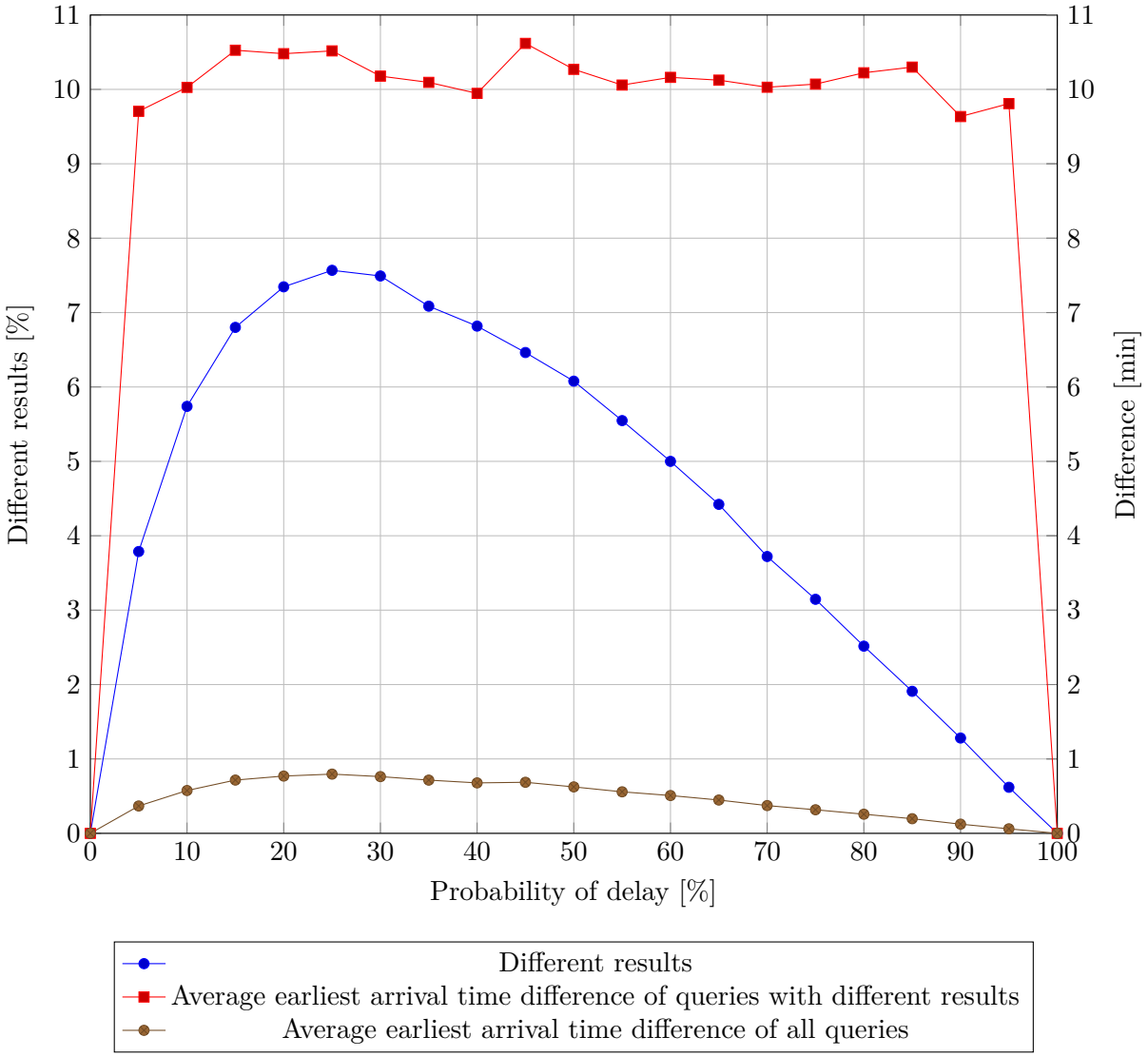

Figure 5.1: Journey quality comparison of ULTRA and DB-ULTRA. A delay buffer of ten minutes and the Delay-All model were used. The x-axis shows the probability that a stop event is delayed by ten minutes before the delays are propagated to the following stops. The blue line shows the percentage of queries with different results using the left axis. The red and brown lines show the average difference of the minimum arrival time of all queries or only those with a different result with ULTRA and DB-ULTRA shortcuts, respectively. They use the right axis with the unit minutes.

5.4 Switzerland Query Times

The reason why so much effort was put into decreasing the number of shortcuts is to keep the running times of ULTRA-RAPTOR and ULTRA-CSA with DB-ULTRA shortcuts as low as possible. The success of this effort shall be measured in this section. For each algorithm, or in the case of the ULTRA algorithms, for each delay buffer, all 10 000 queries were run in sequence. This means parts of the network may still be in the processor cache and different algorithms do not influence each other. The algorithms in this section were run on a single core.

At first, ULTRA-RAPTOR is compared to RAPTOR with a transitively closed transfer graph and MR-∞ with the Core-CH which was used by the preprocessing. Keep in mind

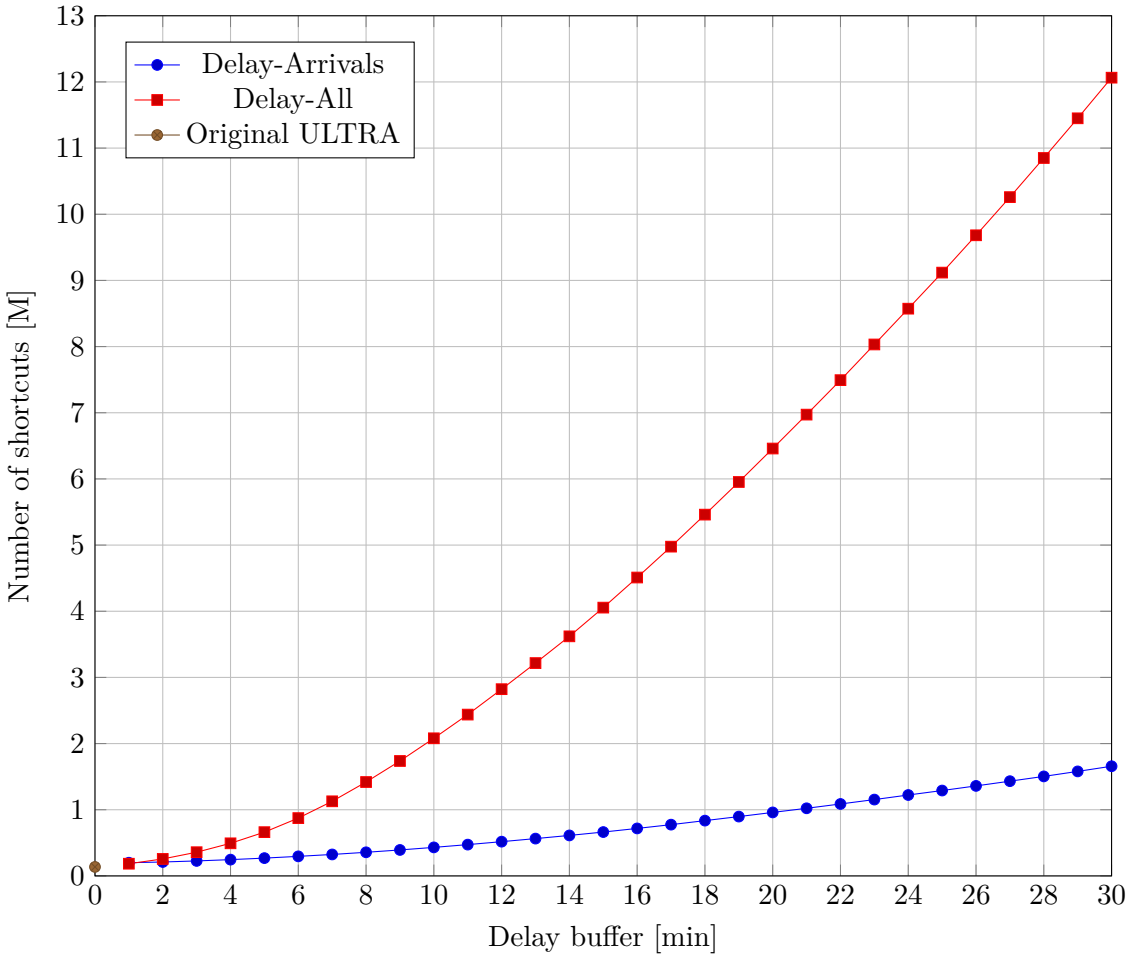

Figure 5.2: Shows the number of computed shortcuts for delay buffers from one minute to 30 minutes and both delay models. The Delay-Arrivals model is depicted with the blue line, the Delay-All model with the red line. For comparison, the number of shortcuts of the original ULTRA algorithm is depicted in brown at the delay buffer 0.

that RAPTOR does not produce optimal results since the transitively closed transfer graph is restricted. Moreover, it only supports stop-to-stop queries. The running times for various delay buffers are presented in Table 5.3. For a delay buffer of up to five minutes in the Delay-All model and 15 minutes in the Delay-Arrivals model, ULTRA-RAPTOR is even faster than RAPTOR. In the Delay-Arrivals model, the running time of MR-∞ is beaten significantly even for a delay buffer of 30 minutes. The same can be said in the Delay-All model at least for a delay buffer of ten minutes. Unfortunately, for a delay buffer greater than 20 minutes in the Delay-All model, ULTRA-RAPTOR is slower than MR-∞.

ULTRA-CSA is compared to CSA with the transitively closed transfer graph. As RAPTOR, CSA does not produce optimal results and only supports stop-to-stop queries. Therefore, a multi-modal variant of CSA is necessary as a baseline to compare ULTRA-CSA with. The authors of ULTRA proposed MCSA for this purpose. Similar to MR-∞, MCSA alternates connection scans with Dijkstra searches on the Core-CH. Since the main performance benefit of CSA is its good memory locality, MCSA is significantly slower than CSA. Fortunately, in ULTRA-CSA only a few edges must be relaxed after scanning a connection. This maintains the good memory locality. Table 5.4 shows the running times for various delay buffers and both delay models. Recall that CSA, MCSA and ULTRA-CSA only minimize the arrival time and do not consider further criteria. Therefore, it is not surprising that these running

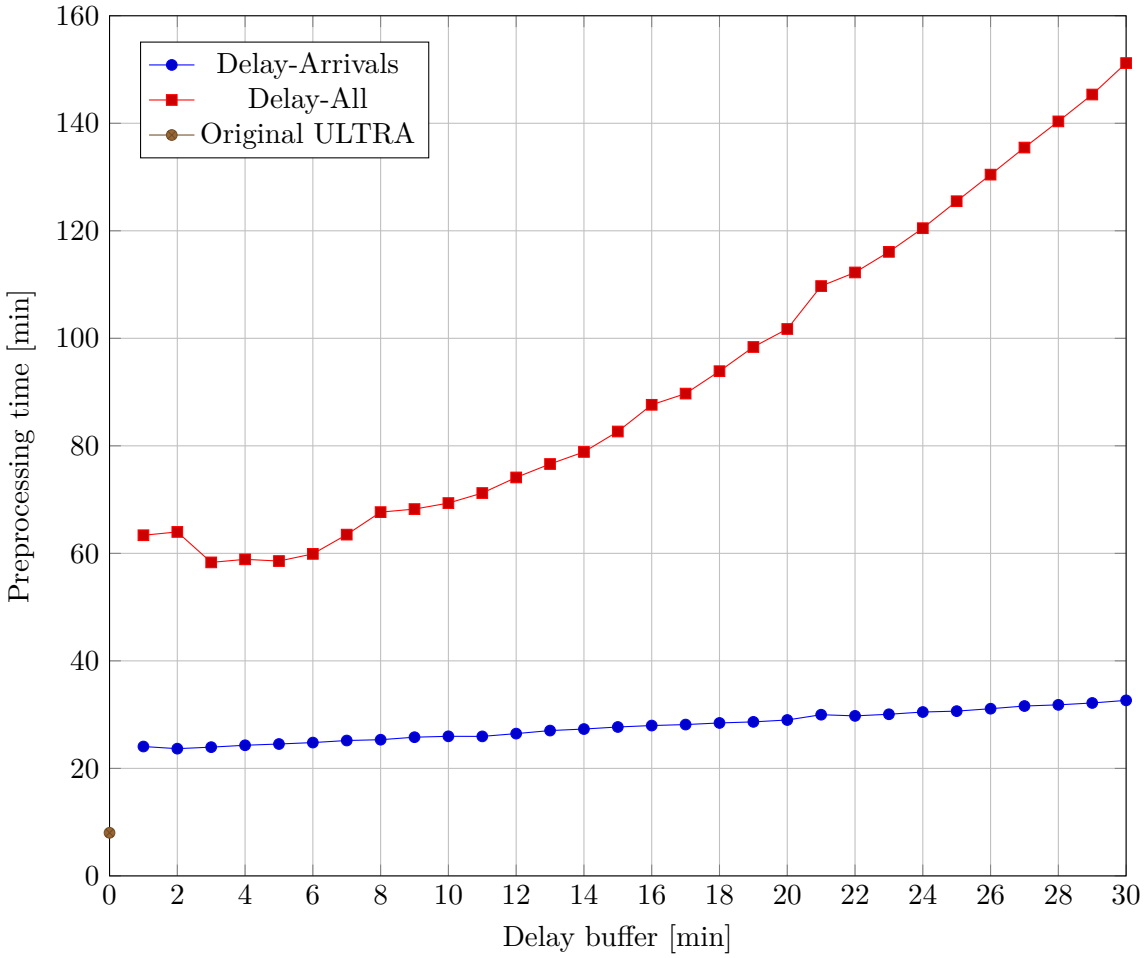

Figure 5.3: Shows the running times of the DB-ULTRA preprocessing for delay buffers from 1 minute to 30 minutes and both delay models. The Delay-Arrivals model is depicted with the blue line, the Delay-All model with the red line. For comparison, the running time of the original ULTRA algorithm is depicted in brown at the delay buffer 0. Both algorithms were run with 16 threads. The time to compute the contracted transfer graph is not included.

times are smaller than those of the corresponding RAPTOR algorithms. In contrast to ULTRA-RAPTOR and MR-∞, ULTRA-CSA is faster than MCSA even for a delay buffer of 30 minutes in the Delay-All model. In the Delay-Arrivals model, ULTRA-CSA outperforms even CSA for a delay buffer up to 20 minutes.

5.5 Germany

As seen in Table 5.1, the Germany network is approximately ten times larger than the Swiss network. The preprocessing time of the original ULTRA algorithm as reported in [BBS+19] was 10:53:35 hours. This suggests a running time of more than one day for DB-ULTRA even for the Delay-Arrivals model and a small delay buffer. To prevent this, the preprocessing of the German network was executed on another machine. This machine has two 64-core AMD EPYC 7742 CPUs clocked at 2.25 GHz with 256 MiB of L3 cache and 1024 GiB of DDR4-3200 RAM. The Delay-Arrivals model and a delay buffer of ten minutes were used. The preprocessing took 17:38:19 hours and produced 5 752 805 shortcuts. For comparison, ULTRA computed 2 077 374 shortcuts.

Table 5.3: Running time in milliseconds for the Switzerland network, averaged over 10 000 random queries. RAPTOR (marked with *) only supports stop-to-stop queries and was therefore run on a different set of random queries. The shortcuts for a delay buffer of 0 were computed with the original ULTRA algorithm.

		Delay buffer [min]						
Delay model	Algorithm	0	5	10	15	20	25	30
Delay-All	RAPTOR*	17.0	17.0	17.0	17.0	17.0	17.0	17.0
	MR-∞	38.1	38.1	38.1	38.1	38.1	38.1	38.1
	ULTRA-RAPTOR	13.9	16.0	20.8	26.3	32.0	38.7	45.0
Delay-Arrivals	RAPTOR*	17.0	17.0	17.0	17.0	17.0	17.0	17.0
	MR-∞	38.1	38.1	38.1	38.1	38.1	38.1	38.1
	ULTRA-RAPTOR	13.9	14.6	15.5	15.5	17.5	18.9	20.2

Table 5.4: Running time in milliseconds for the Switzerland network, averaged over 10 000 random queries. CSA (marked with *) only supports stop-to-stop queries and was therefore run on a different set of random queries. The shortcuts for a delay buffer of 0 were computed with the original ULTRA algorithm.

		Delay buffer [min]						
Delay model	Algorithm	0	5	10	15	20	25	30
Delay-All	CSA*	5.9	5.9	5.9	5.9	5.9	5.9	5.9
	MCSA	17.6	17.6	17.6	17.6	17.6	17.6	17.6
	ULTRA-CSA	4.8	5.4	6.9	9.0	11.2	13.6	15.9
Delay-Arrivals	CSA*	5.9	5.9	5.9	5.9	5.9	5.9	5.9
	MCSA	17.6	17.6	17.6	17.6	17.6	17.6	17.6
	ULTRA-CSA	4.8	4.9	5.2	5.4	5.8	6.1	6.5

The query times are presented in Table 5.5. ULTRA-RAPTOR clearly outperforms MR-∞ but is slower than RAPTOR. This is not quite as good as the result for Switzerland since there ULTRA-RAPTOR was faster than RAPTOR for a delay buffer of ten minutes in the Delay-Arrivals model. For the CSA algorithms, the results are better. ULTRA-CSA is almost as fast as CSA and more than twice as fast than MCSA.

5.6 Optimizations

Chapter 4.2 introduced many optimizations. Some of those are evaluated in this section. Table 5.6 compares the original DB-ULTRA algorithm as introduced in Chapter 4 with

Algorithm	Running time [ms]
RAPTOR*	317.5
MR-∞	774.8
ULTRA-RAPTOR	402.3
CSA*	144.6
MCSA	344.4
ULTRA-CSA	148.1

Table 5.5: Query running time in milliseconds for the German network, averaged over 10 000 random queries. A delay buffer of ten minutes and the Delay-Arrivals model were used. RAPTOR and CSA (marked with *) only support stop-to-stop queries and were therefore run on a different set of random queries.

Table 5.6: Number of shortcuts and running time of various modifications of DB-ULTRA, where some optimizations are turned on or off. A delay buffer of ten minutes was always used. The modifications are: deactivation of the stop criterion (No stop criterion), no propagation of same-first-trip labels after relaxing final transfers (No SFT final transfers), no same-first-trip-optimizations (No SFT), no SFT but prune with existing shortcuts (No SFT & PWES), prune with existing shortcuts (PWES) and allow time travel at the second trip (Time travel). For comparison, the original algorithm as described in Chapter 4 is listed (Original). Remember that the second kind of time travel which occurs on nearby stops is solved by leveraging same-first-trip-optimizations and is therefore turned off if these optimizations are turned off. The same is true for preferring existing shortcuts. No values are reported for "No SFT final transfers" in the Delay-Arrivals model because this optimization has no use if all trips depart on time and is therefore always disabled.

	Delay-All		Delay-Arrivals	
Modification	#Shortcuts	Preprocessing	#Shortcuts	Preprocessing
Original	2 079 977	01:09:21	431 219	00:25:58
No stop criterion	2 079 626	01:10:39	431 218	00:26:15
No SFT final transfers	2 594 731	00:39:31	—	—
No SFT	12 657 528	00:31:36	755 006	00:20:59
No SFT & PWES	12 657 835	00:26:17	755 011	00:18:36
PWES	5 096 403	01:01:51	632 152	00:23:39
Time travel	2 172 730	01:10:24	431 254	00:26:29

variations where certain optimizations are disabled or, in the case of "prune with existing shortcuts", enabled. As the results clearly show, the same-first-trip-optimizations are by far the most important ones, reducing the number of shortcuts from more than twelve million to just about two million in the Delay-All model. The influence in the Delay-Arrivals model is smaller since the departures are on time. The main purpose of these optimizations is to give witnesses a chance to reach the same first trips as the candidates, which is always possible in the Delay-Arrivals model. Nevertheless, since these optimizations also incorporate the use of the same shortcut origin, the preference of existing shortcuts and the mitigation of the second kind of time travel, they are still useful for the Delay-Arrivals model. Unfortunately, these optimizations are also the most expensive ones, more than doubling the preprocessing time in the Delay-All model. Most of this time is spent on the SFT final transfers.

As predicted, enabling pruning with existing shortcuts increases the number of shortcuts drastically while reducing the preprocessing time only marginally. If the same-first-trip-optimizations are deactivated, enabling it reduces the preprocessing time without increasing the number of shortcuts. The prevention of time travel at the second trip does not seem to reduce the number of shortcuts by a significant amount. However, on an urban network the influence of this optimization could be higher since stops are closer. Similarly, time travel becomes more frequent for larger delay buffers. The stopping criterion for the final relaxation of transfers does not have a big influence on the running time.

5.7 Profiling

DB-ULTRA is significantly slower than ULTRA. To improve the performance in the future, it is important to know where the most time is spent. Table 5.7 shows the share the different

Table 5.7: Proportions of the running time for the DB-ULTRA preprocessing of the major phases as presented in the pseudocode of Algorithm 4.1. The last two phases are not represented in the pseudocode since they are only necessary for the SFT final transfers optimization. The phase "Add shortcuts" adds the shortcuts for the current source departure time to the shortcut graph.

	Delay buffer [min]			
	Delay-All		Delay-Arrivals	
Phase	5	30	5	30
Clear	0.08%	0.03%	0.18%	0.13%
Initial Dijkstra	0.32%	0.12%	0.66%	0.49%
Collect departures	3.03%	1.16%	6.93%	5.25%
Set $\tau_{\text{dir}}(\cdot)$	0.39%	0.15%	0.67%	0.53%
Scan routes with initial transfers	2.73%	1.00%	3.68%	2.79%
Collect routes departing from source	0.01%	0.00%	0.01%	0.01%
Scan routes without initial transfers	0.03%	0.01%	0.04%	0.03%
Relax witness transfers	12.19%	4.47%	24.49%	17.91%
Relax candidate transfers	18.57%	17.53%	23.66%	33.47%
Collect routes 2	5.66%	2.12%	9.03%	6.70%
Scan routes 2	7.95%	49.93%	10.51%	18.03%
Relax transfers 2	11.03%	3.67%	20.01%	13.64%
SFT final transfers	37.70%	14.97%	0.00%	0.00%
Add shortcuts	0.32%	4.85%	0.16%	1.03%

phases have of the preprocessing time. Consider the Delay-All model first. For a small delay buffer of five minutes, the running time is mainly dominated by the intermediate and final transfers. Especially the SFT final transfers are a large part of the running time. For a higher delay buffer of 30 minutes, the relaxations of transfers become less relevant and the running time is dominated by the second route scanning step which is responsible for half the running time. This is not a surprise since most of the same-first-trip-optimizations take place in this phase.

In the Delay-Arrivals model and for a delay buffer of five minutes, the running time is split fairly evenly between the relaxation of witness transfers, candidate transfers, the second route scanning step (including the collection of the routes) and the final relaxation of transfers. Remember that SFT final transfers is useless in the Delay-Arrivals model and therefore deactivated. For a delay buffer of 30 minutes, the relaxation of candidate labels and the second route scanning step have a higher impact on the running time. This is because fewer candidate journeys are pruned early, leading to more origin stops.

Although the intermediate relaxation of witness transfers is unrestricted in contrast to the final relaxation of transfers which is restricted by the stopping criterion, it does not have a significantly larger share of the overall running time. This matches the finding in Table 5.6 which showed a small influence of the stopping criterion on the running time. Therefore, employing a stopping criterion for the intermediate transfers relaxation probably does not decrease the preprocessing time by much.

6. Conclusion

In this thesis, we defined two different delay models for public transit, the Delay-All and the Delay-Arrivals model. For these models, new notions of dominance were introduced in Chapter 3. We showed on a theoretical level how these notions can be used to solve the problem of this thesis, calculating public transit shortcuts for a range of possible delay scenarios. We used these ideas in Chapter 4 to extend ULTRA to DB-ULTRA. Furthermore, we proposed many optimizations to speed up DB-ULTRA and decrease the number of computed shortcuts. The success of those optimizations was confirmed in Chapter 5. For this, we used the algorithms ULTRA-RAPTOR and ULTRA-CSA which were developed for the original ULTRA algorithm but also work with DB-ULTRA shortcuts. The experiments showed good results for the Delay-Arrivals model even for high delay buffers. A delay buffer of 30 minutes yielded query times from 35% to 45% higher than with ULTRA shortcuts, which is still twice as fast as the baseline algorithms MR-∞ and MCSA. Similar results were achieved in the Delay-All model for a delay buffer of 10 minutes.

6.1 Future Work

There are many possibilities for improving or extending DB-ULTRA. One possibility is to use statistical data that indicates which trips are usually delayed and by how much. In Chapter 3, we modeled departure and arrival times as arbitrary time intervals before restricting those to a fixed length, the delay buffer. If this restriction is lifted, every stop event can have its own delay buffer based on statistical data. This could lead to fewer shortcuts while increasing the percentage of covered delay scenarios that are likely to occur in real life. There may not be sufficiently detailed data available or the algorithm to handle arbitrary intervals is too complex. In this case, the intervals could be restricted on another level, for example, one delay buffer per trip or route. The German rail company Deutsche Bahn reported much higher delays for long-distance traffic compared to other traffic [Deu20]. This could be exploited by using a larger delay buffer for long-distance routes than for others without further data.

A similar idea is using separate delay buffers for the departures and arrivals. Technically, the DB-ULTRA implementation already uses separate delay buffers to support both delay models. Therefore, this idea is already implemented and must only be evaluated. Since the Delay-All model produces much more shortcuts than the Delay-Arrivals model for high delay buffers, the combination of a high *arrival buffer* and a small *departure buffer* seems

promising. This would allow a passenger to exploit only small delays of departures while still being able to handle high delays of arrival.

An approach worthy of investigation is annotating shortcuts with the delay scenarios for which the shortcut is necessary. Then the shortcut must only be considered for a query if it is annotated with the current delay scenario. For any given delay scenario, the number of necessary shortcuts is expected to be similar to the number of shortcuts computed by ULTRA. If all shortcuts are annotated with exactly the delay scenarios for which they are necessary and not more, performance characteristics similar to ULTRA shortcuts could be achieved even with very high delay buffers in the Delay-All model. However, it is unclear how to annotate shortcuts. Maybe some optimizations need to be deactivated for this to work.

In this thesis, we focused on RAPTOR and CSA to evaluate our algorithm. It could be interesting to incorporate DB-ULTRA shortcuts into other public transit routing algorithms. One such algorithm is Trip-Based Routing [Wit15], which uses transfers between trips instead of stops. Adapting DB-ULTRA to this could be a challenge. As for the annotation of shortcuts, maybe some optimizations must be deactivated.

Until now, we always assumed the concrete delay scenario is completely known at query time. However, this is not always true, especially if the journey is planned days in advance. Moreover, if a trip is delayed but this fact is not known before it is boarded, the journey may even be impossible to complete. We already presented two approaches which try to solve such problems: McRAPTOR [DPW12] with reliability as a third criterion and MEAT [DPSW13]. Instead of relying on precise delay information at query time, they try to favor reliable journeys or provide good alternatives. Using these approaches together with DB-ULTRA shortcuts could yield multi-modal variants of McRAPTOR and MEAT. Therefore, evaluating them with DB-ULTRA shortcuts would be interesting.

Bibliography

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

[BBS+19] Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution. In Michael A. Bender, Ola Svensson, and Grzegorz Herman, editors, *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BDG+16] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. *Route Planning in Transportation Networks*, pages 19–80. Springer International Publishing, Cham, 2016.

[DDP+13] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing Multimodal Journeys in Practice. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2013.

[Deu19] Deutsche Bahn AG. Fragen und Antworten zur Pünktlichkeit. `https://www.deutschebahn.com/resource/blob/1187698/4ec4a0d0470d6389556725b1775e563d/fragenantworten-data.pdf`, 2019. Accessed: 26 July 2020.

[Deu20] Deutsche Bahn AG. Kennzahlen 2019. `https://www.deutschebahn.com/resource/blob/5058446/d9863eb3ef6c365c76c7759078dea156/20190328_pbk_2020_kennzahlen-data.pdf`, 2020. Accessed: 26 July 2020.

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

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

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

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

[Ein05] A. Einstein. Zur Elektrodynamik bewegter Körper. *Annalen der Physik*, 322(10):891–921, 1905.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

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

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, August 2012.

[KSS+07] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 36–45. SIAM, 2007.

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

[Sau18] Jonas Sauer. Faster Public Transit Routing with Unrestricted Walking. Master's thesis, Karlsruhe Institute of Technology, April 2018.

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

[WZ17] Dorothea Wagner and Tobias Zündorf. Public Transit Routing with Unrestricted Walking. In Gianlorenzo D'Angelo and Twan Dollevoet, editors, *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, volume 59 of *OpenAccess Series in Informatics (OASIcs)*, pages 7:1–7:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.