Karlsruhe Institute of Technology

Combining Multiple Criteria in Multi-Modal Route Planning using ULTRA

Master Thesis of

Manuel Schweigert

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: April 1st, 2020 – November 3rd, 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, November 3rd, 2020

Abstract

We study the problem of finding multimodal, multicriteria journeys in transportation networks, including a single mode of unrestricted individual transport like walking, cycling or driven, and a schedule-based public transportation system. We use the ideas of the ULTRA preprocessing technique (UnLimited TRAnsfers) and combine them with a multicriteria search based on RAPTOR to produce full Pareto sets of optimal journeys.

We examine the performance of this technique to optimize for three criteria: arrival time, walking distance and number of rides on public transportation vehicles utilized and show that this approach leads to a significant speed up for queries using these criteria for transfer speeds up to driving-speeds compared to other algorithms. Furthermore, we also examine different heuristics and their impact on both the preprocessing and the queries in speed and quality. Using a combination of techniques, we manage to produce journeys with good quality in 63 milliseconds for a metropolitan area, showing that this technique is ready for use in consumer applications.

Deutsche Zusammenfassung

Wir untersuchen multimodale, multikriterielle Routenplanung in öffentlichen Verkehrsnetzwerken mit einer Form von Transfer von und zu den Verkehrsmitteln wie Laufen, Fahrradfahren oder Autofahren. Das öffentliche Verkehrsnetzwerk ist dabei unbeschränkt in der Anzahl der Verkehrsmittel, welche von Zügen zu Flugzeugen und Fähren reichen können, solange sie auf einem Fahrplan basieren.

Hierzu benutzen wir die Beschleunigungstechnik UnLimited TRAnsfers, oder ULTRA, und erweitern diese Technik um eine multikriterielle Optimierung. Dies ermöglicht eine größere Wahl an Routen, je nach Geschmack des Benutzers: schnellere Ankunftszeit, kürzere Laufwege oder weniger Umstiege.

Auf dieser Vorberechnungsbasis benutzen wir einen für uns angepassten, multimodalen und multikriteriellen RAPTOR Algorithmus um Anfragen in weniger als der halben Zeit zu berechnen als es bisher üblich war. Wir zeigen verschiedene Optimierungsparameter und deren Auswirkungen sowie Heuristiken um sowohl die Anfragen, als auch die Vorberechnung weiter zu beschleunigen. Schließlich ermitteln wir die Qualität dieser Heuristiken und zeigen, dass Anfragen in Metropolregionen mit guter Qualität in nur 63 Millisekunden beantwortet können, was die Algorithmen reif für den Betrieb in Applikationen für Endnutzer macht.

Contents

1. Introduction **1**
1.1. Related Work . . . 2
1.2. Contribution . . . 4
1.3. Outline . . . 5

2. Preliminaries **7**
2.1. Problem Statement . . . 7
2.2. Definitions . . . 7
2.3. Algorithms . . . 10
2.3.1. RAPTOR . . . 10
2.3.2. MR-∞ . . . 12
2.3.3. Contraction . . . 12
2.3.4. Contraction Hierarchies . . . 13
2.3.5. rRAPTOR . . . 13
2.3.6. ULTRA . . . 14
2.3.7. McRAPTOR . . . 15
2.3.8. MCR . . . 16
2.4. Heuristics . . . 17
2.4.1. Fuzzy Dominance . . . 17
2.4.2. Discretized Dominance . . . 18
2.4.3. Scoring and Quality . . . 18
2.5. ULTRA . . . 19

3. Query Algorithm **21**
3.1. McRAPTOR + BucketCH . . . 21
3.2. Implementation Details . . . 22

4. Preprocessing using McULTRA **25**
4.1. Optimizations . . . 26
4.1.1. Implementation Details . . . 28

5. Experiments **29**
5.1. Data . . . 29
5.1.1. Computer Specs . . . 29
5.2. Pre-Processing . . . 30
5.2.1. Key Composition . . . 30
5.2.2. Prune After Last Candidate . . . 30
5.2.3. Prune Existing Shortcuts . . . 32
5.2.4. Transfer Speed . . . 33
5.3. Queries . . . 34
5.4. Heuristics . . . 36
5.4.1. Overdomination . . . 36

5.4.2. Fuzzy Domination 36
5.4.3. Discretization 37

6. Conclusion **39**

Bibliography **41**

Appendix **45**
A. Real World Example 45

1. Introduction

The diversity of transportation methods available to people has increased dramatically in the last years and while online services for journey planning have started to adopt these, a lot of these services as well as recent research either focus on road-based journeys like driving, cycling and walking, or on schedule-based public transit journeys like buses, trains and flights. However there is a need for integrated journey planning, returning the best journeys given all the transportation methods available to the user. This journey planning problem is called multi-modal route planning.

What makes a journey the best journey though? For one user, having a slightly longer journey may be favorable to having to switch to a different train mid-journey. One might not mind having to walk for a significant portion of the journey if it improves the arrival time dramatically, while that may be impossible for others. This means that different criteria can usually not be compared to each other and algorithms were developed which return a set of journeys where no journey is better in every criterion than any other journey. This result is called a Pareto set, and the journey planning problem is called multiple-criteria route planning. Most algorithms only optimize for one or two criteria, though, as starting with four criteria, the amount of possible solutions becomes impractical.

Timetable-based public transit planning is also not very useful by itself, as most journeys never start or end exactly at a stop covered by the network; they rather usually involve a trip to a station, as well as a trip from a station to the destination. To accommodate for this, time-dependent algorithms usually employ a road-based route planning algorithm before and after their own computations, and most algorithms then also allow for transits between stops mid-journey, making these algorithms partly multi-modal.

However most algorithms are only able to use a specific set of transfers between stops as defined in the database, but being able to walk between any stop arbitrarily improves the quality of journeys consistently [WZ17]. For this reason, we use combine a technique that allows unlimited walking with a multi-criteria optimization.

1.1. Related Work

Route planning is a widely researched topic in computer science, mainly because of its direct impact to people in the real world. The problem of route planning for a street network can be modeled by finding a shortest path on a directed weighted graph. Almost no route planning work gets around mentioning Dijkstra's algorithm [Dij59] which solves this problem in theoretically optimal time complexity. Other notable early works include Bellmann and Ford [For56, Bel58] for their network-flow based algorithm and Hart, Nilsson and Raphael [HNR68] for their work on heuristics for route planning, resulting in the A* algorithm.

Even though Dijkstra's algorithm is theoretically optimal, on large networks, it will still take a long time to compute even on today's computers. Improving this query time has been a research topic since its inception. Since its publication in 1959, developing speed-up techniques for and based on it have been an on-going effort, however only recently, advancements in computing power have made it feasible to pre-process large networks for vastly accelerated query times by reducing the search space.

Routing for Road Networks

Most modern speed-up techniques converged to use a common set of techniques [Sch08a, Sch08b]: Bi-directional search, goal directed search and contraction. As an intuition, the *bi-directional search* not only searches from a starting point to a target point, but it simultaneously also searches from the target point to the starting point in a backwards search [Dan63, GH04]. When these two searches encounter each other, their respective paths are merged for the end result. While elegant, this technique is only easy to implement on time-independent networks like road networks. As soon as time-dependent factors are introduced to the network, for example public transport, the implementation of this technique becomes non-trivial [NDLS08].

The *goal directed search* is a technique that uses precomputed data on the graph to guide the search towards the target, thereby reducing the search space. Two commonly used approaches exist to accomplish this. First, by constructing a robust heuristic for the A* algorithm by Nilsson and Raphael [HNR68] based on the triangle inequality property on graphs, Goldberg et al. created the landmark-approach which uses potential functions to guide the search [GH05, GW05] and called the technique *ALT*.

The second approach augments edges in the graph with labels which guide the search. These labels respresent a subset of nodes of the graph for which a shortest path starts from the given vertex, which lets the query algorithm prune nodes which do not contain a label for the target node [WW03]. This technique was later refined by Lauther and is now known as *Arc-Flags* [Lau04]. There, the graph is partitioned into cells, with every node receiving a label for each graph cell, which indicates whether a shortest path into at least one target node of the respective cell starts at this node. This approach relies on a good *graph partitioning*, which means that a graph should be split in a number of parts with the least amount of connections between the parts as possible. This is an independent field of research which is useful in many applications besides route planning.

Contraction is a technique to reduce the size of the graph in the pre-processing. The *Highway Hierarchies* technique by Sanders and Schultes [SS05] uses the explicit hierarchical structure of road networks given by different road categories, e.g., highways versus city streets, to determine when and where to do contractions. We give a short explanation of the basic principle of vertex-contraction in Chapter 2. A technique yielding very fast query speeds is called *Contraction Hierarchies* or *CH* presented by Geisberger et al [Gei08, GSSD08], which also relies on the hierarchy in road networks. However, it uses a graph-based approach to approximate these hierarchies on a per-node basis, yielding impressive results.

Another approach worth mentioning is the *table lookup* method. Obviously being able to simply look up pre-computed distances results in a constant query time, but saving the lookup tables for whole graphs is impractical as the size of the table would be immense. For this reason, the technique commonly relies on selecting a good subset of nodes for which to calculate these lookup tables beforehand, which is utilized for example in *Transit Node Routing* [SS06]. Approaches to select these nodes are numerous: separators, partitioning and nodes which turn out to be important from other techniques, e.g., CH.

A plethora of combinations of these techniques exist today with very good results, some reducing the time for continental-scale queries down to just two microseconds [BDS+10].

Routing in Public Transit Networks

Unfortunately, the nature of public transport is that transportation vehicles like trains rarely depart exactly when someone arrives at the departure location. Rather, they depart at specific times or intervals, which introduces a key complication in route planning: waiting [OR89]. Modeling these schedules in a way that Dijkstra-based techniques can be used is also non-trivial.

Generally, there are two types of approaches to solving the route planning problem in time-dependent graphs like a public transit network: The time-expanded and the time-dependent approach.

The *time-expanded graph* approach models the graph as nodes of *events*. An event may be an arrival event or a departure event and edges are added based on the trips of vehicles and the transfers which are possible from arrival events to other departure events on the same station. On this kind of graph, a query based on Dijkstra's algorithm can be performed, however as the graphs become bigger due to this modeling, the query times become slower. Speed-up techniques mentioned previously can be applied, but it is challenging and they have limited success as the hierarchy of train networks, which are the most researched types of public transit networks, is not as pronounced as it is for road networks. However, a combination of ALT and Arc-Flags has shown promise [DPW09].

The other way to approach this problem is by utilizing a *time-dependent graph*. In the time-dependent graph, a node represents a stop in the network and an edge represents a connection between the stops using a public transit vehicle. The edge weights now correspond to time-tables, modeling when this connection departs and when it arrives at the next stop. This results in query algorithms having to include the time factors, which makes them more complex, but it also reduces the graph structure compared to the time-expanded graphs and for that reason, queries on them tend to be faster.

One algorithm presented by Dibbelt et al. which solves the earliest arrival time problem for this kind of network is the *Connection Scan Algorithm* (CSA) [DPSW18]. By using simple data structures and a simple algorithm, it achieves great efficiency on modern computers at the cost of lesser extensibility for more criteria.

The importance of incorporating and optimizing for more than just the earliest arrival time however is more important in a public transit setting than in route planning for road networks, as the wish of people to be able to balance the price and the convenience against the duration of the journey is higher. Taking a longer journey for more convenience is enticing to many, but not to all users, so the importance of algorithms to optimize multiple criteria is greater than for road networks. While CSA can be adapted to optimize for a multiple criteria scenario, this involves introducing dynamic data structures, thereby greatly reducing its advantage in efficiency.

There exists a different algorithm which, in addition to solving for earliest arrival time, also solves for the number of trips taken, while remaining efficient by not needing dynamic data structures. This is the *RAPTOR* algorithm, presented by Delling et al. [DPW12]. RAPTOR can also be extended to optimize for even more criteria using the *McRAPTOR* variant, of which we will speak more in Chapter 3.

One common flaw with the mentioned public transit routing algorithms is that the only use public transit. Usually, a desirable journey starts and ends at locations which are not stops and stations. Desirable journeys rather include these parts and also allow for transferring between different stops using other modes of transportation, e.g., walking, bicycling or riding a taxi. An algorithm using the time-expanded graph to solve for multiple criteria which is based on Dijkstra's algorithm is the *multi-label-correcting* algorithm (MLC) [PSWZ08]. RAPTOR and CSA allow for a number of foot-paths between stops to be added to the graph explicitly, as long as they are transitively closed, which means that when defining a transfer from stop a to stop b, and from b to stop c, there must also be a transfer defined between stop a and c. Modeling transfers in this way allows the algorithms to remain efficient. To allow for more useful transfers between stops, Delling et al. introduce the *multimodal multicriteria RAPTOR* (MCR) [DDP+12], which uses both a time-table based graph for RAPTOR, and a contracted road graph to compute journeys which are not restricted by the duration or number of the transfers and optimize multiple criteria, including the amount of walking involved. This added complexity comes with a cost, though, which is paid in longer query times compared to RAPTOR and CSA, but it is faster than MLC.

To reduce this query time while keeping the option of choosing journeys with unrestricted transfer phases by means of walking, cycling, etc., Baum et al. introduce *UnLimited TRAnsfers for Multi-Modal Route Planning* (ULTRA) [BBS+19]. This pre-processing technique allows CSA, RAPTOR and any other public transit algorithm, to find all optimal journeys they optimize for without a limit to a transfer distance while in some cases even improving their respective query times slightly.

For completeness sake, it is also fair to mention that for many route-planning applications, it is not only sufficient to compute only the first optimal journey, but to compute a range of optimal journeys in a time span. Both CSA and RAPTOR provide such *range* variants, and ULTRA makes use of this *rRAPTOR* variant for its pre-processing.

1.2. Contribution

We introduce the *McULTRA* algorithm, which enables the multi-criteria variant of RAPTOR to utilize unlimited transfers while optimizing for three criteria. This enables it to compute the same multi-modal journeys as MCR [DDP+12] in less than half the time. The three criteria we optimize are: earliest arrival, number of rides taken and walking duration. By creating multi-criteria variants of speed-up techniques used in the ULTRA algorithm, we pre-compute all shortcuts which would be necessary in a multi-modal scenario. Additionally, we test combinations of heuristics for both the pre-processing, as well as the queries, to further improve the query speeds while maintaining good quality, to just 63 milliseconds for the London network, enabling wide-spread use of this technique in consumer software.

1.3. Outline

In Section 1.1 we give a concise overview of the state of route planning research with a focus on giving a rough overview of the commonly used concepts in the field and a focus on the techniques important to this work.

A brief summary of our contribution to the field can be found in Section 1.2.

We get formal in Chapter 2 where we first define the basic building blocks required to formulate the problem to be solved. Then we introduce a number of algorithms and explain how they work so that an understanding of the contribution of this work is possible.

Then we introduce our work, which is split in two parts: the changes to the query algorithm can be found in Chapter 3, while most of the work happens in Chapter 4 where we take apart the ULTRA preprocessing technique in detail and explain how we incorporated multiple criteria in the algorithm.

In Chapter 5, we present our experiments and conclusions, as well as give explanations to different findings. We compare the performance of our work with related work and on graphs of different sizes. Then, we introduce the parameters we chose for the heuristics explained in Chapter 2 and explain how we measure the quality of the resulting journeys. At the end of the chapter, we discuss the effects of the combinations of heuristics.

Finally, we give our conclusion, a short glance into ongoing work in the field and a look at future research options.

2. Preliminaries

In this section we define the basic notation and terminology used in this work. Furthermore, we will explain the RAPTOR, ULTRA and MCR algorithms, which are essential to understand this work.

2.1. Problem Statement

We want to find *journeys* which are Pareto-optimal for the *arrival time*, the *walking duration* and the number of *rides* taken. These journeys are to be unrestricted in the walking duration their of their transfers as long as they remain optimal. They start at a given *source vertex*, which can but need not be a stop, and finish at a given *target vertex* which may also be a stop but need not be. Additionally, the journey must start after a given *source departure time*. We aim to solve this in a faster manner than was previously possible using an acceptable amount of pre-processing.

As many more useful criteria are imaginable and potentially useful, our goal is to present a robust pre-processing technique that can use different or more criteria than the ones presented. As such, we do not introduce specific optimizations towards the walking distance criterion, even though it stands in relation to arrival time in some phases of the algorithm and therefore, some specific optimizations may be applied. Instead, we engineer the algorithm to be extendable with any criteria the data can supply. As the complexity of four criteria increases dramatically according to Dibbelt et al. though [DDP+12], we focus fully on the three-criteria problem and trying to decrease the time to compute all such journeys to a level which makes it appealing for wide-spread usage. Filtering the journeys using heuristics becomes necessary when using more criteria, which is not part of this work.

2.2. Definitions

Public Transit Network

A public transit network $(\mathcal{S}, \mathcal{R}, \mathcal{T}, \mathcal{G})$ consists of a set of stops $\mathcal{S}$, a set of routes $\mathcal{R}$ between stops, a set of trips $\mathcal{T}$, and a directed, weighted transfer graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. This transfer graph is a representation of a street network consisting of a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. The weight $\tau_\Delta(e)$ associated with each edge $e \in \mathcal{E}$ defines the travel time between the connected vertices for transfers. Note that the travel speed can be limited by the choice of the mode of transport, for example walking or cycling, which may result in longer travel times for edges the lower the transfer speed limit is defined.

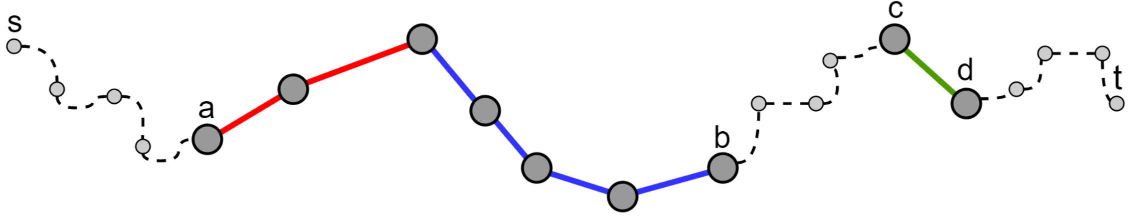

Figure 2.1.: An example journey from s to t containing an initial transfer from s to a, an intermediate transfer from b to c and a final transfer from d to t, as well as three rides.

All vertices in $\mathcal{V}$ from which a passenger can enter a public transit vehicle or exit from one are called a stop $s \in \mathcal{S}$, so $\mathcal{S} \subseteq \mathcal{V}$.

A route is a sequence of stops $(s_1, \ldots, s_i), i \geq 2$ that can be traversed using a single transit vehicle in the given order.

Since many vehicles may traverse the same route each day, we define trips associated to routes. A trip is therefore a function defined for all stops of a route, returning a pair of arrival and departure times $(\tau_{\text{arr}}, \tau_{\text{dep}})$ for every stop representing when the vehicle arrives at the stop and when it departs again.

Note that the departure times may be augmented with a non-negative departure buffer time $\tau_{buf}(s), s \in \mathcal{S}$, which intends to add a delay before being able to board a vehicle after arriving at a stop s, modeling the time required to reach the platform and the vehicle.

Criteria

To define which journeys are better than others, we need to first take a look at criteria. The most common criteria that users are interested in usually are the arrival time, price and convenience. While arrival time is the most common of criteria to be optimized, optimizing for the price is non-trivial from technical perspective since it is not easily modeled; journeys might get cheaper as they get longer when discounts kick in or different fare zone combinations allow for cheaper tickets. Convenience on the other hand can include a lot of measureable criteria, which results in having to choose which ones of them to use. For our case, we focus on two easily modelable criteria in addition to arrival time: the number of times a passenger has to switch vehicles and the amount of walking that has to be done. Of course the amount of criteria that could be useful is only left to the imagination, but as Delling et al. showed [DDP+12], using four or more criteria is not practical at this time.

Journey

A *path* in a transfer network like a street graph is a list of vertices so that for every consecutive pair of vertices in the path u and v, (u, v) corresponds to an edge. However, a path alone does not suffice to define a journey through a public transit network, as we also need to track which vehicles to ride on and when to enter and disembark. As such, we define a journey through a public transit network as a sequence of journey legs.

A *journey leg* can be either a *ride*, which is a sub-sequence of a trip, or a *transfer* which is a path in the transfer graph. We define the departure time for a ride to be the departure time of the trip at the first stop of the ride's sub-sequence and the arrival time to be the arrival time of the final stop of its sub-sequence. A transfer is not schedule dependent so it only has a duration, which is the sum of the edge-weights of its path. We require a

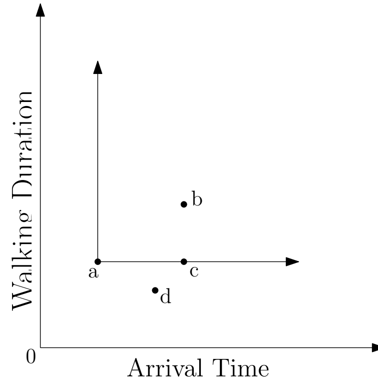

Figure 2.2.: A visualization of different journeys' arrival time and walking duration criteria. Journey b is dominated by a. However, journey c is only weakly dominated by a and d is not dominated at all. The Pareto-optimal set of journeys is a and d. The arrows pointing from a show the area that a dominates.

transfer's departure time to be equal to the arrival time of the previous journey leg if there is one, and its arrival time to be the sum of its departure time and its duration.

For any two consecutive journey legs, the arrival vertex of the first one must equal the departure vertex of the second one, while the arrival-time of the first one must be earlier or equal to the departure time of the second.

The first vertex of the journey is called the *source* vertex, and when we define a *source departure time* for a journey, the departure time at the source vertex must equal or be later than it. The last vertex of the journey is called the *target* vertex and its arrival time represents the arrival time of the journey.

If the first journey leg is a transfer, we call this transfer the *initial transfer*. Likewise, if the last journey leg is a transfer, we call it the *final transfer*. We call the other transfers *intermediate transfers*. See Figure 2.1 for an example of a journey containing every kind of transfer and two consecutive rides. Additionally, if a journey only consists of a single transfer and no other journey legs, this transfer may be referred to as a *direct transfer* since it represents directly transferring from the source to the target vertex using no public transportation.

A journey supports the criteria of the arrival time in seconds, the number of rides and the duration of transfers in seconds, also called *walking duration* throughout this work. Note that not all algorithms presented in Section 1.1 optimize for all of these criteria.

We define a journey to be Pareto-optimal when there exists no other journey which *dominates* it. A journey J dominates another journey L if it is better in every criterion $c_i(J) < c_i(L)$ (*strict domination*) or if it is better or equal in every criterion $c_i(J) \leq c_i(L)$ (*weak domination*) We show a visual intuition of this in Figure 2.2. When not further specified, when we speak about domination, we refer to weak domination, as there is no reason for us to produce journeys which are equal in all criteria, rather it is important to filter out as many journeys as possible as early as possible during the processing to speed it up.

It is also possible to include more criteria than these three. For each journey leg l and an arbitrary criterion c_{new}, define $c_{\text{new}}(l)$ to be the value of the criterion at the journey leg l. Then simply require for two subsequent journey legs l and l_{next}, that $c_{\text{new}}(l) \leq c_{\text{new}}(l_{\text{next}})$ and increase the criterion naturally.

For example, it is possible to optimize the amount of fare zones passed through by all rides by defining $c_{\text{fare}}(l)$ as a set of fare zones and its relation to be the subset relation. Any ride then adds all fare zones it passes through to its set if they aren't present already. This way a journey is not dominated by a journey with an earlier arrival time if it uses fewer fare zones, which would likely result in a cheaper ticket price.

2.3. Algorithms

Many route planning algorithms have Dijkstra's algorithm at their core, however to use it on a public transit network, the network must be expanded is not possible to run it unaltered on a public transit network, as it is only handling time-unrestricted networks like a transfer graph. While it is possible to expand a time restricted network to enable running Dijkstra-based algorithms on it [PSWZ08], it is not ideal for reasons which we will explain in this section.

To compute the time-restricted, multi-modal journeys for public transit networks efficiently, we explain the basic idea of the RAPTOR algorithm. However, as RAPTOR requires a transitively closed transfer graph to model every possible transfer, the ULTRA algorithm is then explained to show how to pre-process a transfer graph to include all important paths in a contracted form, reducing the size of the transfer graph for unlimited transfers considerably.

To understand ULTRA better, however, we will first look into the rRaptor variant of the RAPTOR algorithm which will be used for the ULTRA pre-processing.

Finally, to introduce multiple criteria optimization, we look at the multi-criteria variant of RAPTOR and subsequently show how the MCR algorithm extends it to make it possible to run it on non-transitively closed transfer graphs.

2.3.1. RAPTOR

RAPTOR solves the bicriteria problem of minimizing arrival time and number of rides taken given a source stop $s \in \mathcal{S}$ and a source departure time τ_{dep}. In a round-based manner, for every round k, it computes the minimum arrival time at any target stop using at most k rides.

To do this, for every round k, every stop v is associated with a label $\tau_{\text{arr}}(v, k)$ containing the earliest known arrival time at this round, which is initialized with the value of the previous round, or ∞ if there is none. In the initial round, the arrival time of the source stop, $\tau_{\text{arr}}(s, 0)$, is set to τ_{dep}.

For every following round k, the arrival times $\tau_{\text{arr}}(v, k)$ are now computed for all reachable stops v using at most k trips, using the results of the previous rounds.

This is done by dividing each round in three phases: The first phase sets the labels $\tau_{\text{arr}}(v, k)$ for every vertex v to equal the value of their previous round $\tau_{\text{arr}}(v, k-1)$ to prune all journeys which would not improve the arrival time further.

In the second phase, all relevant routes are collected at first. Each stop v which had been improved in the previous round is processed by first collecting all routes which contain v. For every such route a search is done for the earliest stop v_r which had been both improved in the last round, and whose arrival time $\tau_{\text{arr}}(v_r, k-1)$ allows for boarding a trip of this route. If no such trip can be found, the route is pruned. The result of this collection step is a list of pairs of all routes that need to be scanned in the current round and the first stops for which the scanning can be relevant (r, v_r).

For every pair (r, v_r), the routes are now being scanned. First, the earliest trip $T \in \mathcal{T}$ of route r at stop v_r which departs after the arrival time $\tau_{\text{arr}}(v_r, k-1)$ is selected. Then, all

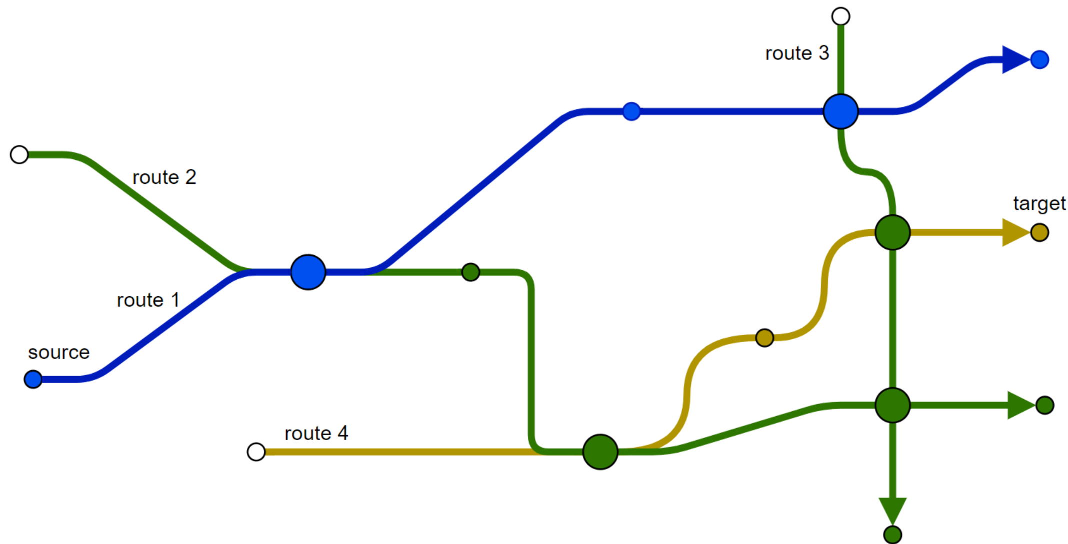

Figure 2.3.: Visualization of the RAPTOR algorithm: starting from the source stop, in the first round, route 1 is scanned. In the second round, both route 2 and 3 are scanned. Finally, in the last round, route 4 is scanned and no further routes can be scanned which would improve the arrival time at the target stop.

stops of the route are iterated in sequence starting at $v_i = v_r$, updating their arrival times $\tau_{\text{arr}}(v_i, k)$ if they are improved by the arrival time of the current trip $T(v_i)_{\text{arr}}$.

At every stop visited like this, it is necessary to also check whether it is possible to embark on an earlier trip for every route we only do a single scan even if it was reached at different arrival times at different stops in the previous round. So if a stop is encountered for which its arrival time in the previous round allows to use an earlier trip, we use that earlier trip for this route scan going forward.

Finally, in the third phase, transfers between routes are processed. For every stop v whose label was improved this round, all of its outgoing edges (v, u) in the transfer graph are collected and if the sum of the arrival time at v, $\tau_{\text{arr}}(v, k)$ and the transfer duration $\tau_\Delta((v, u))$ is smaller than the arrival time at the neighbour u, $\tau_{\text{arr}}(u, k)$ is updated to reflect the arrival time using the transfer. RAPTOR relies in this step on a transitively closed transfer graph, which means that for any two edges (u, v) and (v, w), an edge (u, w) must exist with an edge weight which adheres to the triangle equality: $\tau_\Delta((u, v)) + \tau_\Delta((v, w)) \leq \tau_\Delta((u, w))$. If the graph does not provide this, a Dijkstra-based search must be used to traverse the transfer graph instead.

The algorithm can stop once no more labels have been improved in a round. For a visual example of how RAPTOR will scan routes over time, see Figure 2.3.

Local Pruning

RAPTOR uses two pruning techniques to greatly accelerate the processing. First of all, *local pruning* is used, which accelerates the algorithm by removing the necessity to initialize every round with the labels of the previous round. For every stop a new label is created, which represents the cumulative earliest arrival over all rounds executed so far. Now, whenever a label is going to get updated in a round, it is first checked whether the new label would improve the local pruning label. If it does not, this implies that a journey found in a previous round has a better arrival time already and thus the label is pruned. If it does improve the local pruning label, both it and the label for the current round are

updated. The result is that labels may be initialized in bulk or even before the query starts with the default value ∞ for the arrival time.

Target Pruning

In addition to local pruning, *target pruning* can be used when RAPTOR is used for a one-to-one query and thus a target stop exists. It works essentially the same way as local pruning, by eliminating labels before they update a stop if they are worse than the target pruning label. However in this case, only a single label representing the best arrival time at the target stop over all rounds calculated this far is used for every stop, which incidentally is just the local pruning label of the target stop. This is correct because as soon as the arrival time at the target stop is smaller than ∞ for the first time, any journey that has a later arrival time than that can obviously not improve the earliest arrival time at the target. Note that for every following round, the number of rides taken can also never decrease, so any journey found later in the algorithm can also not improve on this criterion.

2.3.2. MR-∞

In order to make it possible to run RAPTOR on a non-transitively closed transfer graph, like a road graph, it is necessary to use a variant which replaces the transfer phase of the RAPTOR algorithm with a Dijkstra-based search through the transfer graph [DDP+12]. As this MR-∞ was developed as a heuristic for researching the relevance of longer transfers, it is possible to set a maximum transfer duration k, denoted as MR-k, at which point the Dijkstra search will stop for the given round. This algorithm, even when restricted in its transfer duration, finds more journeys compared to a transfer graph containing only hand-picked transfers between stops like it is commonly used for RAPTOR, validating the thesis that longer transfers are still useful. Unfortunately, doing this kind of search also results in more computations during the transfer phase and therefore in a longer query time. To reduce this query time, a contraction technique is employed, but it is still slower than using only RAPTOR.

2.3.3. Contraction

One very effective way to speed up Dijkstra-based searches is to pre-process the transfer graph using a technique called *vertex contraction*. This technique works by contracting vertices from the graph, which means removing the affected vertex and its adjacent edges while inserting *shortcuts*, which are edges that connect neighbours back together so that shortest paths in the remaining graph keep the same length as before the contraction. See figure 2.4 for a visual example of this. Note that some shortcuts may be unnecessary, for example if the neighbour vertices are already connected by an edge with a lesser edge weight than the shortcut. By strategically removing vertices in this way and removing unnecessary shortcuts, Dijkstra-based algorithms can be accelerated significantly. A simple, yet elegant technique that uses this approach is the *Topo-Core* algorithm [DSW15], which uses a contracted *core graph* as an *overlay* and the original graph. A search is done bi-directionally starting from the original graph. Whenever a vertex is visited which is also in the core graph, the search will continue to only scan the core-graph from that vertex on, which results in only scanning a small part of the original graph before, by the bi-directional nature, meeting in the core-graph.

The MR-∞ algorithm uses a similar concept to contract the graph between stops. For all intermediate transfers, it is only interesting to visit other stops so vertices between stops can, but don't need to be contracted. Contracting a graph too much results in a very high vertex-degree of the vertices, which means that the vertices have a lot of edges to other vertices, resulting in many scans to be performed for every vertex and defeating the purpose of the contraction. For public transit networks, contracting up to an vertex-degree of 14 to

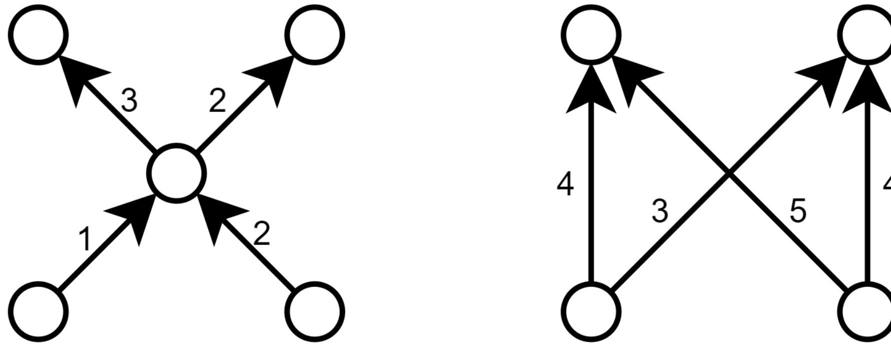

Figure 2.4.: An example of a vertex-contraction operation on the center vertex. Left: before the contraction, Right: after the contraction.

20 is reasonable. The initial and final transfers may need to start and stop from a non-stop vertex, though. Since these will probably be eliminated from the contracted transfer graph, another contraction-based technique is used to connect the source and target vertex to the transfer network using the road network to do one-to-many queries from the source vertex to the stops or in a backwards search, from the target vertex to the stops.

2.3.4. Contraction Hierarchies

The *CH* algorithm is a pre-processing and query technique to vastly accelerate query times for route planning in non-time-restricted transfer networks, making use of the natural hierarchy of e.g.road networks for a contraction strategy. There exist many extensions to this algorithm, some of which are used in this work to calculate the initial and final transfers efficiently, namely *BucketCH* [BBS+19], but need not be explained in detail to understand this work, as they are replaceable by any other one-to-one or one-to-many route planning algorithms, such as Dijkstra's algorithm.

2.3.5. rRAPTOR

Realistically, it is often necessary to not just compute the earliest arrival at a destination given a departure time, but to collect all optimal journeys given during a given time span. It is possible to calculate this by running single RAPTOR queries in sequence for every possible departure time at the source stop during the departure time span. However it is possible to improve on this with the rRAPTOR variant by reusing the labels from previous queries.

Specifically, we order all departure times from every trip originating from the source stop in the given time span from the latest to the earliest, after which we run a query for every one of those departure times. Thanks to this order, we can re-use the rounds and bags from the previous iterations without clearing them. This means that every journey is automatically pruned if its arrival time is later at some stop than from a previous query, or in simpler terms, if an earlier departure leads to the same or worse arrival at a stop than a later departure, it cannot result in a shorter journey and is skipped.

Note that in this way, rRaptor not only produces all optimal journeys starting in the departure time span by arrival time and number of trips used, but it additionally optimizes the duration of the journey. For example, a regular RAPTOR might start on an earlier trip than rRaptor, only to arrive at the same time at the target vertex.

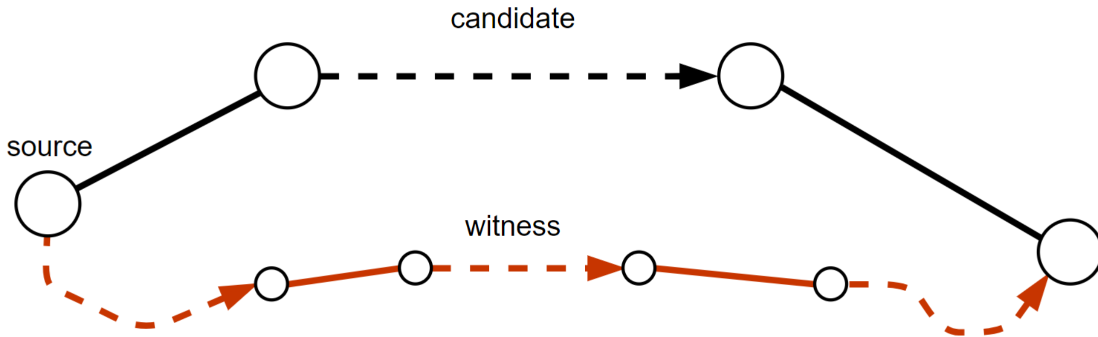

Figure 2.5.: A candidate and a witness journey. If the candidate journey is not dominated by the witness journey, the candidate will be added as a shortcut.

2.3.6. ULTRA

One big limitation of the RAPTOR algorithm, as discussed earlier, is the requirement of a transitive transfer graph, which would result in a huge vertex-degree and quadratic size if the graph is fully connected. Much like some shortcuts in a contraction might be unnecessary, some edges in this transitive transfer graph can be removed as they never appear in any optimal journey. The ULTRA algorithm is a pre-processing technique which, in theory, enumerates every optimal journey possible and only creates shortcuts for transfers which are contained in any of these journeys.

Obviously, collecting every optimal journey is not reasonable, so the algorithm uses properties of optimal journeys to reduce the necessary computation. A key insight being used is the fact that while initial and final transfers vary a lot as they depend on the choice of the respective vertices and times, the amount of useful intermediate transfers is only determined by the time-table of the public transit routes. By computing initial and final transfers efficiently at query-time using a one-to many path-finding algorithm, such as *BucketCH*, which is introduced for this purpose, the pre-processing can fully focus on finding shortcuts only for intermediate transfers.

One useful property of optimal journeys is that the number of useful intermediate transfers is limited; depending on the transfer speed, many transfers are simply not viable as it is always faster to take a trip. As a crude example, it should never be faster to walk hundreds of kilometers from Karlsruhe Main Station to Hamburg Main Station than to take a ride in a train, even if the train ride involves waiting for many hours.

As Delling et al. showed [DDP+12], simply limiting the duration of a transfer by a flat number always results in some journeys which are not found. As it is also impossible to compute every single journey in a reasonable time, another property of optimal journeys is used: a sub-journey, which means any number of consecutive journey legs from the original journey, can be substituted by an optimal journey for its respective source and target vertices as well as the source departure time. In other words, if a better journey exists for any sub-journey of an optimal journey, replacing that sub-journey with the better journey will keep the whole journey optimal. So the idea is to iterate the smallest kinds of journeys which could contain a transfer which could be needed as a shortcut, which would be a journey consisting of a ride, an intermediate transfer and another ride. Every such journey is called a *candidate journey*. As soon as another journey which dominates it is found, called a *witness journey*, it is certain that the shortcut from this candidate is unnecessary.

A witness journey has less restrictions than candidate journeys: it may freely make use of initial and final transfers and is not forced to utilize exactly two rides, but it never

uses more than two rides since it would never dominate the criterion of number of rides taken. See Figure 2.5 for a typical candidate and witness. If a witness journey dominates a candidate journey, the candidate journey is not optimal and can be pruned since the witness can replace it.

These journeys are calculated by using the rRAPTOR variant for every stop and a time span that contains every departure from that stop, but limiting the search to only two rounds and only processing initial and final transfers for witnesses. Note that the nature of the RAPTOR algorithm to be undirected helps by computing all shortcuts whose journeys originate from a single stop, and that the process can easily be parallelized for all stops. After two rounds, all non-dominated candidate journeys are collected, and their transfers added to the shortcut graph.

This graph can then be utilized by a public transit algorithm like RAPTOR or CSA, using a one-to-many Dijkstra-based algorithm for the initial and final transfers, to find all optimal journeys like MR-∞ in a time comparable to an original RAPTOR.

2.3.7. McRAPTOR

While RAPTOR already solves for two criteria, it can be extended to include an arbitrary number of additional criteria with the McRAPTOR variant. A label now not only consists of the arrival time $\tau_{\text{arr}}(v, k)$, but also one or more other criteria c: $c(v, k)$ which now also need to be accounted for when determining dominance of labels over each other, whereas before, it used to suffice to only compare the arrival times. Instead of a single label per round and per stop, now a list of mutually non-dominating labels is kept per round and stop. These collections of labels are generally denoted as *bags*. A label strictly dominates another label if it is better in every criterion, or weakly dominates it if it is better or equal in every criterion. This results in a set of labels for each stop which are mutually non-dominating. Similarly, the result of every round is a set of journeys to the target vertex. Collecting these journeys from every round results in a Pareto set of journeys.

To adapt the RAPTOR algorithm to this, during the route scanning phase, it is necessary to track the trip for every single label in the route bag. The reason for this is that the by the nature of having mutually non-dominating labels at one stop now, these labels can contain different arrival times, so in order to continue to process the route in one loop, it is necessary to update each label with its own trip every time a stop scanned.

The RAPTOR algorithm is modified in the following way: in the first phase, while scanning a route, an empty, temporary route-bag is created to collect all of the labels which are encountered during the route scan. While traversing the stops of the route, the scanning phase is divided in three steps: first, the arrival times of all labels inside of the route bag are updated according to their associated trip and the current stop. Then, the route bag is merged into the bag of the current stop and the current round while. Merging a bag into another bag means that the resulting bag then contains exactly all labels from both input bags which are not dominated by any other label in the bag.

In the final step the route bag is filled with the labels from the current stop bag, but from the previous round. To do this, the labels from the bag of the previous round at the current stop are collected for each such label, a trip is searched which departs after the label's arrival time. If no such trip is found, like with the regular RAPTOR, the label is pruned, otherwise, the trip is associated with the label and the label is merged into the route bag. Note that in a three-criteria algorithm, where only two criteria are tracked directly using the label, at most one label per trip is necessary to be in the route bag at a time as the arrival time can be ignored for dominance in this case. The reason for this is rather simple, as the arrival time of every label will be set to their trip's arrival time for the next stop,

they will be equal and so, when it comes to merging the labels back from the temporary route bags into the stop bags, labels using the same trip will have the same arrival time and hence one will always dominate every other one (unless they are equal and only weak dominance is used).

Lastly, the transfer phase is changed as follows: instead of propagating a single label to all neighbours, all labels from the current bag are merged into the neighbour's bags after updating their distances.

Local and Target Pruning

For local and target pruning, the single labels holding the best arrival times for each stop are replaced with bags holding all non-dominated labels which have been found for their respective stops so far. Every time a label is being updated, the updating label must first successfully merge into the local pruning bag of the corresponding stop and it must not be dominated by any label in the target's local pruning bag.

2.3.8. MCR

MCR is an extension to McRAPTOR like MR-∞ is to RAPTOR; it works on an unlimited, contracted transfer graph to create a set of Pareto-optimal journeys for three or more criteria and multiple modes of individual transportation. McRAPTOR is extended to handle initial and final transfers from non-stop vertices exactly like MR-∞.

Then, the transfer phase is replaced by a multi-criteria Dijkstra search. This means that, like McRAPTOR already does in the scan phase, the Dijkstra algorithm is extended to use labels with arbitrary criteria instead of just the arrival time. Every vertex in the transfer graph is associated with a bag of labels instead of a single label, and whenever a vertex is visited, all of its labels need to be merged into every neighbour after updating them with the transfer duration. Then, every label successfully merged in that way needs to be pushed into the priority queue as well.

This means that the priority queue will hold many labels, and keeping track of which ones are added and which ones are dominated from a bag is a lot of work, which is why MCR implements this by making each bag implement their own priority queue.This lets the priority queue of the Dijkstra search track just bags instead of labels, which reduces its size and volatility considerably as a label can dominate any amount of other labels when merging without making it necessary to update the global priority queue for each one. It is only required to update the global priority queue when the minimal label changes, which provides the key for the bag in the global priority queue.

Unfortunately, using a multi-criteria Dijkstra search also usually means that it is not possible to say for sure when the labels of a vertex cannot be improved further, depending on the choice of criteria. The algorithm becomes *label correcting*, as opposed to *label setting*. The reason for this is that the key for the priority queue is usually a composite key combining multiple criteria, which usually does not result in an order which guarantess that labels pulled from the priority queue are strictly dominated by every one label that was pulled before. In fact, it may happen that labels which have been pulled earlier can be dominated by labels being pulled later. For example, if only the arrival time was to be used as a key for the priority queue, the Dijkstra search might search most of the graph before it visits a vertex where a different criterion might be better, resulting in most of the graph being visited again.

This problem can only be solved by choosing the composite key in a way that no label with a higher key is not dominated by a label with a lower key. Common ways of choosing composite keys are linear combinations or lexicographic sorting of the criteria.

Another problem arises when trying to use more than three criteria with MCR, which is that the result sets become impractically large, depending on the criteria. This is both undesirable since having the choice of hundreds of journeys which are relatively similar is not appealing for a user, and also because this slows down the query. This problem is tackled by using heuristics to prune journeys earlier, even though they may still result in optimal journeys, and by filtering journeys into more distinctive groups, with the intent of only computing journeys which are sufficiently different from other journeys [DDP+12].

2.4. Heuristics

Heuristics can be used to accelerate route planning algorithms, but not all of them still deliver optimal results. Optimizing the RAPTOR algorithm with heuristics can be done by increasing the sub-space that a journey or journey leg dominates in the criteria space. As shown in Figure 2.2, one journey (leg) dominates a certain sub-space. By increasing this space beyond what is shown, the guaranteed optimality is lost, but more journeys are dominated earlier, which improves the run-time of the algorithm. A simple and rather ineffective example can be seen in Figure 5.5. Dibbelt et al. [DDP+12] introduce and examine several heuristics for MCR, of which we pick two for our algorithms because of their effectiveness.

2.4.1. Fuzzy Dominance

The first of the two heuristics is *Fuzzy Dominance*, which makes use of principles borrowed from *Fuzzy Logic* to increase the domination area as shown in Figure 2.6. Normally, the relational operators $<$, $>$ and $=$ are binary operators, which means that they are either true or false, or 1 and 0 respectively. By defining fuzzy relational operators $\mu_<$, $\mu_=$ and $\mu_< \to [0,1]$ instead, it is possible to determine "how much" these statements are true or false. A statement might be considered false using the regular relational operators, but "almost true" using fuzzy operators.

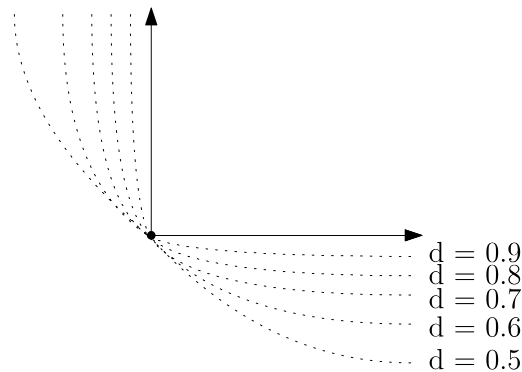

Figure 2.6.: A visualization of the spaces a journey dominates with different degrees of fuzzy-domination, compared to strict dominance (arrows).

For consistency, it is required that $\mu_<(x,y)+\mu_=(x,y)+\mu_>(x,y) = 1$ for any valid values x, y, which is called *Ruspini's condition*. The exact way these functions are defined in this paper and in the MCR paper is by defining a single exponential function $\mu_=(x) := \exp(\frac{\log(\chi)}{\epsilon^2}x^2)$ and using it to define the operators in the following way: $\mu_=(x,y) := \mu_=(x-y)$, $\mu_<(x,y) := 1-\mu_=(x-y)$ if $x-y<0$, otherwise 0, and $\mu_>(x,y) := 1-\mu_=(y-x)$ if $y-x>0$, otherwise 0. The parameters χ and ϵ are used to determine the curve of the fuzzy function and as such, determine the fuzzyness. The fuzzy function can be roughly imagined as a Gaussian normal distribution centered at $x=0$. Then, the result of $\mu_=(\epsilon)$ should be χ.

Using these fuzzy relational operators, the fuzzy domination is defined as a *degree of domination* $d(J_1, J_2) \in [0,1]$ of journey J_1 over J_2. To show this function simply, the following

functions need to be introduced first. Given that M is the number of criteria, the number of criteria in which J_1 is better than J_2 is given as $n_b(J_1, J_2) := \sum_{i=1}^{M} \mu_{i<}(c_i(J_1), c_i(J_2))$ the sum of the fuzzy "less than" values over all criteria. Note that a different fuzzy operator with different χ and ϵ values can be used for each criterion c_i: μ_i.

Likewise, the number of criteria in which J_1 is worse than J_2 is given as $n_w(J_1, J_2) := \sum_{i=1}^{M} \mu_{i>}(c_i(J_1), c_i(J_2))$ and finally, $n_b(J_1, J_2)$ uses the sum of $\mu_{i=}$. Together, from Ruspini's condition, it holds that $n_b + n_w + n_e = M$. The degree of dominance can now be easily defined as $\frac{n_b - n_w}{n_b}$ if $n_b > n_w$, or 0 otherwise. Note that for efficiency, this can be simplified to only evaluate the exponential function $\mu_{i=}$ once for every criterion.

To interpret the values of d, for $d(J_1, J_2) = 0$, we say that J_1 does not dominate J_2, for $d(J_1, J_2) = 1$, it strictly dominates J_2 as defined earlier, and for $d(J_1, J_2) \in (0, 1)$, it *fuzzy-dominates* J_2 with degree $d(J_1, J_2)$. In Figure 2.6 we can visually interpret the effect of different degrees of domination.

2.4.2. Discretized Dominance

A simpler yet still very effective heuristic is the *Discretized Dominance* or *Bucket Dominance*. This heuristic uses strict dominance as is natural for the RAPTOR algorithms, however before determining dominance, one or more criteria are rounded to the nearest value of x_i for the i-th criterion. The effect of this can be seen in Figure 2.7.

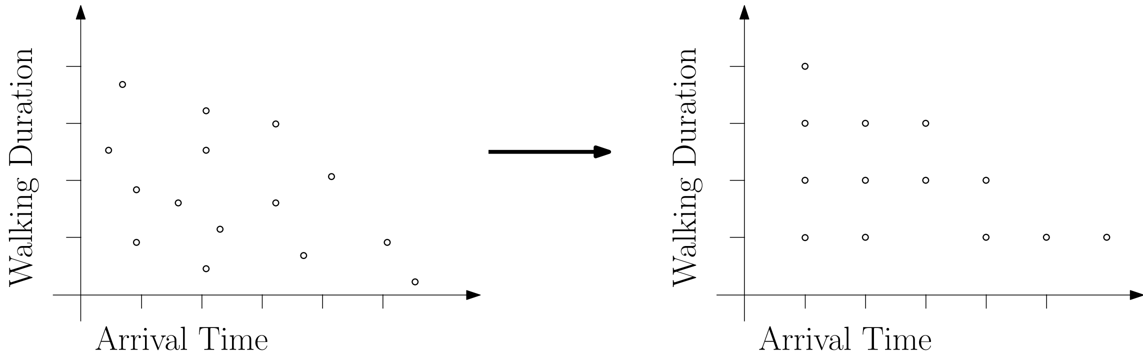

Figure 2.7.: A visualization of how criteria are rounded to a "grid" for two criteria before considering domination, resulting in a single Pareto-optimal journey (leg).

2.4.3. Scoring and Quality

To determine the quality of heuristics, it is necessary to define a metric. This metric should reflect how different a journey is compared to the others, as it is important to include meaningful choices. For example, single minute differences in different criteria between journeys are not big enough to make these journeys meaningfully different, including one one would suffice, while it is important to include journeys which have big differences for example in arrival time and number of rides taken, as either the comfort of having less rides or the importance of an early arrival may be preferable to the user, and thus the choice is meaningful.

To this end, the principles from the Fuzzy Domination heuristic are used to construct this metric in the following way. First, a pair of Norms is defined for the degrees of domination: the *T-Norm* T, or triangular norm, and a complementary *S-Norm* S, or conorm. Both are commutative, associative and monotone functions mapping $[0, 1]^2 \rightarrow [0, 1]$. The T-Norm can be interpreted as a fuzzy conjunction ("and" in binary) while the S-Norm can be interpreted as the according fuzzy disjunction ("or"), and is always defined as $S(x, y) := 1 - T(1 - x, 1 - y)$.

Two well known pairs of T-/S-Norms exist, the minimum and maximum norms $(\min(x, y), \max(x, y))$, and the *product norm/probabilistic sum* $(xy, x + y - xy)$. As the scoring and quality for MCR used only the min/max norms, so will we for better comparability.

Given a Pareto-set of journeys $J_1, \ldots, J_n$ and an S-Norm, a scoring function is defined to be $sc(J) := 1 - S(d(J, J_1), \ldots, d(J, J_n))$. This score is also called the *significance* of a journey.

To determine the quality of a set of journeys, the k most significant journeys are taken from the set and are compared to the k most significant journeys from the baseline set. A $k \times k$ matrix is filled with the similarities between these two sets of journeys where the the location of (i, j) corresponds to the i-th journey from the first set, and the j-th journey from the second set. The similarity for criterion c_i is given as $sim_i(J_1, J_2) := \mu_{i=}(c_i(J_1) - c_i(J_2))$. The overall similarity sim is then defined as $T(sim_1, sim_2, \ldots, sim_M)$.

After populating the matrix, the journeys are greedily matched by the selecting the highest similarity of all unmatched journey pairs in the matrix until all journeys are matched. For each match, the similarity value is then weighted by the fuzzy score of the reference journey and finally both the weighted average similarity and the standard deviation are produced.

2.5. ULTRA

We introduced the basic concept of ULTRA already. To fully understand how we change the ULTRA algorithm in our own work, in this section we explain it in full detail first.

Given a public transit network, ULTRA uses an rRAPTOR range query limited to two rounds on every stop with a range enveloping all departure times from all stops to generate shortcuts. Naturally, this process can be parallelized by running this rRAPTOR query separately for every stop s and combining the resulting shortcuts in the end.

The first route scanning round is split and performed twice to calculate candidates and witnesses separately. Because candidates only ever start from the source stop s, no initial transfers need to be performed for them. Then, a regular rRAPTOR would consider every reachable trip from the source stop for the first scanning round as described in lines 10 through 12. However this can be accelerated by not only collecting departure times of trips starting at s in line 4, but also collecting reachable departures at other trips between these.

This is done by pre-processing every stop v so that the departure time of every trip T departing from stop v is temporarily reduced by the distance $d(s, v)$. Afterward, all departure times for all stops are collected, together with their corresponding trips and stops and sort them by the departure times. For every rRAPTOR iteration, to know which stops to start new witness searches from, it is then only necessary to iterate over the the list of departures which lie between the current departure time of the iteration, and the one from the previous iteration.

This enables us to scan witness journeys efficiently in each iteration. Note that generally it is not important for correctness that all witness journeys are successfully found, as a missing witness can merely lead to unnecessary shortcuts being added, not to missing shortcuts.

Using this process, the witness routes are scanned in line 16 and afterward, intermediate transfers are calculated. This is based on a Dijkstra search in the transfer graph as described earlier, however it is also tracked which transfers are candidates and which are witnesses. During the intermediate transfers round, another optimization step is done which exploits the fact that missing witnesses do not lead to invalid results. By stopping the processing

Algorithm 2.1: ULTRA transfer shortcut computation.

Input: Public transit network $(\mathcal{S}, \mathcal{R}, \mathcal{T}, G)$, with unrestricted transfer graph $G = (\mathcal{V}, \mathcal{E})$
Output: Shortcut graph $G' = (\mathcal{S}, \mathcal{E}')$

```
1  for each s ∈ S do
2  |   Clear all arrival labels and Dijkstra queues
3  |   d(s,∗) ← Compute distances from s to all stops in G
4  |   D ← Collect departure times of trips at s
5  |   for each τ_dep ∈ D in descending order do            // rRAPTOR iteration
6  |   |   Set initial label of s to τ_dep
7  |   |   Collect routes serving updated stops
8  |   |   Scan routes for first RAPTOR round                 // track candidates
9  |   |
10 |   |   From all routes, collect trips reachable from s not scanned previously
11 |   |   for each trip t collected previously do           // initial transfers
12 |   |   |   Find first stop u of trip t reachable from s
13 |   |   └   Set initial label of u to τ_dep + d(s,u)
14 |   |
15 |   |   Collect routes serving updated stops
16 |   |   Scan routes for first RAPTOR round                  // only witnesses
17 |   |   Relax intermediate transfers
18 |   |
19 |   |   Collect routes serving updated stops            // second RAPTOR round
20 |   |   Scan routes for second RAPTOR round
21 |   |   Relax final transfers                               // only witnesses
22 |   |
23 |   |   C ← Collect undominated candidates
24 └   └   E' ← E' ∪ C
```

of transfers after no more candidates are in the queue, the run time of the algorithm can be significantly reduced. This can in return lead to witnesses not to be processed further which may dominate shortcuts in later iterations and as such, the amount of shortcuts which are generated is increased. So a balance parameter is introduced, the *witness limit*. After the last shortcut candidate is settled, witnesses continue to be processed until they exceed the arrival time of the last shortcut candidate plus the walking limit.

Now, the second round starts in line 19 with collecting updated stops from the intermediate transfer round and scanning their routes normally. In this round, like in the first round the algorithm must track which vertices are candidates, and which are witnesses, so that the shortcuts can be retrieved from the candidates later. After this, the final transfers are calculated for witnesses only in line 21. As soon as the last candidate is pulled from the priority queue though, this phase can be stopped, as all candidates are settled and no witnesses remain that could dominate a candidate. In the end, all non-dominated candidates are collected from the second round and merged into the shortcut graph.

No target pruning can be used since there is no target vertex. However, there is the possibility to prune candidates early if the shortcut they would create is already present in the shortcut graph, which in turn also leads to more shortcuts to be added for a faster preprocessing time.

3. Query Algorithm

As our work focuses on the pre-computation of a transfer graph, theoretically any public transit query algorithm can be used on the resulting data as long as the query algorithm only uses criteria which have been considered in the pre-processing.

We will now explain in detail the changes we made to McRAPTOR to accomplish unlimited initial and final transfers from any vertex, which are similar in nature to the changes ULTRA makes to RAPTOR. Instead of being able to answer only stop-to-stop queries, we extend McRAPTOR using BucketCH to allow any vertex for source and target vertices while keeping the multi-criteria optimization. We can use BucketCH for these transfers because our third criterion is the walking distance, which is incidentally exactly what BucketCH produces. If we were to add another criterion that may be affected during transfers, we would need to use a multi-criteria variant for these transfers, too. Afterwards, we explain the details of implementing McRAPTOR for our purpose and the specific ways in which we optimized it.

3.1. McRAPTOR + BucketCH

First of all, to accomplish functional parity with the MCR algorithm, we need to add the possibility to run the McRAPTOR query from and to non-stop vertices.

For this, we use BucketCH, which gives us single-criteria, one-to-many shortest paths on the original road graph. Although technically any other one-to-many algorithm could be used, this technique is a compromise of very good query speed and good pre-processing speed.

BucketCH lets us run a query from a source vertex s to a destination vertex t, after which we can retrieve the distances to different stops without having to re-run the query again. Note that we are only interested in the arrival times at stops because the labels for non-stop vertices are not important for the route-scanning phase. While adding initial distances to non-stop vertices might improve local pruning by a little, the speed up does not compare to the extra work of computing these distances in the first place. BucketCH gives us two ways to retrieve distances: the `forwardDistance(s)` is the distance from the source vertex to the stop s, and the `backwardDistance(s)` is the distance from s to the target vertex. In both cases, the algorithm returns ∞ if there is no path or if the path is longer than the path from the source directly to the target. This is already the first pruning step.

We use this technique to add a label for every reachable stop from the source vertex efficiently during the initialization. Whenever a round finishes, when we collect all vertices which have improved in that round, we then add the backward distance to every label and try to merge it into the target bag, as it is described in the MR-∞ and ULTRA sections.

3.2. Implementation Details

In this section, we go into detail on how we implemented the query algorithm, show what is different compared to the formal definition of the algorithm and explain the choices we made.

Initial Transfers

For the initial transfers, we simply collect all `forwardDistance(s)` for all reached stops s and insert a label with arrival time $\tau_{\text{arr}}(s, 0) = \tau_{\text{dep}} + \texttt{forwardDistance(s)}$ into the bag for s for round 0. Additionally, if the source vertex is itself a stop, we will also add a label with the source departure time at the source vertex. Note that it is not necessary to have a bag for the source vertex if it is not a stop, because it will never be necessary to visit this vertex again since the departure label cannot improve for non-negative weights.

Final Transfers

For final transfers, we do require a bag for the target vertex for every round, as RAPTOR requires a bag for each round to produce journeys depending on the number of rides taken. So if the target vertex is not by itself a stop, we will add one more stop to every round which will represent the target vertex. Obviously this stop does not have any routes or trips associated with it, so after every round we perform one pass over every stop s which has been updated during that round. For each of its new labels, we try to add a final transfer to the target stop by adding the `backwardDistance(s)` to its arrival time and walking distance and then attempt to merge it into the target bag. This way, the regular stopping mechanism of McRAPTOR also works for a destination vertex that is not a stop.

Bags

In McRAPTOR, bags are a simple construct: a dynamic array of labels with a helper function to merge a single label into the bag. This merge function is important though, as it is used very frequently. As such, we try to minimize dynamic data allocations in those bags during the merge operations. In general, when we merge a label into a bag, we have to check every label contained in the bag once for domination. However, if the label we want to merge into the bag is dominated by any label inside the bag, it cannot dominate any other label inside the bag as that would mean that the bag contained labels that dominate each other in the first place.

We use this to perform only a single pass over the labels and just abort if a label dominates the new label. While doing the pass, whenever we encounter a dominated label, we skip it remember the amount of labels skipped like this. For all following labels, if they aren't skipped themselves, they are moved forward in the bag by the number of skipped labels so far. Finally, the bag size is reduced by the amount of skipped labels, including only the non-dominated labels and the new label is appended to the back. An alternative way would be to swap each dominated label with the last label in the bag while decrementing the bag size after every such swap. We implemented the first way to do this because the alternative way does not work with the way we set up our bags for the pre-processing explained in Chapter 4. This allows us to use re-use code between the two algorithms. Additionally, we added one optimization which directly inserts the new label into the first dominated label's space if we encounter one instead of appending it to the end. We can do this because of another result of the fact that bags only contain mutually non-dominating

labels, which is that if a new label dominates a label in a bag, it cannot be dominated by any following label in the bag.

Labels

For our label data structure, we opted to use a single label structure for the journey legs, independent of them being a ride or a transfer, and also used them for the temporary route-bag which requires the trip to be retained for each label. Technically this could be improved by making custom tailored labels for each of those steps which only include the data that they absolutely need for that step. However, the amount of memory that can be saved in this way is miniscule and does not justify the non-trivial complexity increase. So our labels include our criteria: arrival time, walking distance and the trip ID, as is needed by the temporary labels during the route scan. For the purpose of reconstructing journeys after the query is finished, we also need to include information to reconstruct the journey leg. To do this, we need the route ID or the transfer ID from which is responsible for adding this label. These IDs are mutually exclusive and when one is set, the other one is set to an invalid value.

Finally, we also need to to know where this journey leg originated and which label came before this one, so we save the *parent stop*, the *parent label round* and the *parent label index* too. The parent stop is the stop this journey leg departed from, while the parent label round and the parent label index are both used to address the label which should be unpacked next. While it is technically possible to leave out the parent label round from the label structure, it serves a dual use to also determine the number of trips taken thus far from just the label itself, which will be necessary for the heuristics.

Rounds

Because of the fact that a transfer can dominate rides from the current round, having both rides and transfers in the same bags will make it impossible to reliably reference to parent labels during the transfer phase, as the positions of those labels inside of the bags can change. For this reason, we require the bags of the rides to be read-only as soon as we start computing transfers. For this reason, our implementation of McRAPTOR splits up the logical round into two separate rounds in the memory. This means that route scans and transfers now have their own rounds and hence their own bags each. This enables us to track those parent labels reliably. As we mentioned, we track these parent labels by saving the parent-stop, the round of the parent label and the index of the parent label inside of the bag of the parent-stop. The reason for saving the round number is that because of splitting up transfer and scan rounds, when a ride is followed by another ride, these labels are two rounds apart, while a ride followed by a transfer is only one round apart. Alternatively, adding dummy labels into each transfer round could also solve this issue, however it would result in useless labels to be created every round, but we want to keep the amount of labels created to a minimum.

A third way to do this is instead of changing any indexes in the bag when dominating a label, it is possible to deactivate a label while leaving it in the bag. However, our experiments have shown us that this increases the sizes of the bags to a degree that makes the query unreasonably slower.

Having split a round into separate rounds in memory also leads to having to do the third step of the route scan, copying labels from the previous round into the round bag twice: once for the previous route scan round and once for the previous transfer round. This does not significantly slow the algorithm down because the total amount of labels copied during this step does not change.

4. Preprocessing using McULTRA

In this chapter we describe how to adapt the ULTRA pre-processing algorithm for multiple criteria and the problems that need to be tacked while doing so. As ULTRA used MR-∞ as inspiration, we use MCR. We reference the pseudo-code of ULTRA 2.1 to make it clear where these changes should happen.

First, we change the fact that for every round and for every stop there exists exactly one label which describes the earliest arrival time at this stop. Instead, every vertex is now associated with a bag of labels for every round, similarly to MCR, where every label now uses the earliest arrival time and walking distance as criteria. As before, a bag may only ever contain labels which don't dominate each other.

Instead of setting single labels like we do in lines 6 and 13, we create new labels with the arrival times and the walking distance set appropriately, and merge them into the bags of the respective stops.

In the transfer phases in line 17 and 21, to reduce complexity and improve the run-time, we don't track each label individually in the priority queue. Instead, like MCR, we only track bags. Each bag's key is the key of it's smallest label that is still unsettled. To achieve this, a bag's labels are partitioned into a heap area, where every bag manages its own internal priority queue, and a non heap area. Whenever a label is merged into the bag, it will be added to the heap area and the area will be sorted again if necessary because of labels which were dominated and thus were removed. Lastly, if the bag is on the priority queue, we update the bag's position there.

When we need the next label from the priority queue for processing in the transfer phases, we retrieve the bag with the lowest key from the global queue, then retrieve the lowest label from its heap, which moves the label internally to the non-heap partition, and if the bag's heap area still contains labels, push the bag back into the global priority queue again. This dramatically reduces the amount of items in the global queue.

In the scan phases, similarly to McRAPTOR, we use a temporary route bag to store all labels collected during the route and assign trips to them. Since we are using an rRAPTOR variant, like mentioned earlier, we cannot do local pruning, but we can get around having to merge every bag from the previous round into the current round for every vertex. Since we are limited to only two rounds anyways, simply checking all previous bags for domination before attempting to merge a new label into the current round serves the same purpose.

Like this, we have to check at most two other bags before attempting to merge a label into a given bag. The amount of labels to be compared is increasing slightly due to this, but only when labels are being added in the current round which would dominate the labels from the previous rounds.

It might be curious that we only need to check at most two additional bags, since in the McRAPTOR query, we split the transfers and the route scans into two rounds each, which would would result in at most four bags having to be checked before beginning a merge: the initial transfer bags, the first route scan, the intermediate transfer bag and then the second route scan bag, before being able to merge into the final transfer bag.

The reason for this is that we include all information needed for shortcut computation in the labels themselves, so that we do not need to backtrack journeys for shortcut generation. This enables us to lift the restriction we had in the McRAPTOR query on the route scan bags to be read-only. Labels which are created in the transfer phases simply retain the information that they are representing candidates and which specific shortcut they would create if they stay non-dominated.

The result is being able to only have three bags per stop. First, the initial transfer bag, which holds the starting label for every iteration of the rRAPTOR query for s, as well as the initial transfer labels for all relevant witness as described in the ULTRA Section 2.5. Then, we have one bag for the first route scan and the intermediate transfer phase. Note that the candidate labels also recieve their shortcut information in the intermediate transfer phase. Lastly, there is one bag for the second route scan and the final transfers, from which the non-dominated candidates can then be collected.

4.1. Optimizations

ULTRA uses some optimizations to reduce the number of labels which need to be processed. In this section, we examine which ones of these can be adapted and what problems might arise doing that.

The witness limit from ULTRA only uses the arrival time for a criterion. When we allow arbitrary criteria to be added, we need to change the way we determine the pruning limit for the transfer rounds. We therefore created two different ways to do this: a soft limit and a hard limit.

The *soft limit* simply remembers the maximum of all criteria, in this case arrival time and walking distance, of all shortcut candidates from the previous route scan. Then, a walking distance limit and an arrival time limit is added to each maximum respectively. In the transfer phase, we can then simply prune labels which exceed any of those limits (*strict pruning*) or all of those limits (*moderate pruning*).

The *hard limit* introduces a counting mechanism. During the transfer phase, we keep exact count of how many shortcut candidates still are unsettled in all bags. We enhance the bags to provide their own count and update the global number after every operation. Once we know that all shortcuts have been settled, we simply continue to process a certain number of labels until we stop the algorithm.

The soft limit is weaker than the hard limit because it is not known how many labels will still be processed after the last candidate. Even when setting the soft limit to 0, there may still be many witnesses which will be processed if only moderate pruning is employed. We provide benchmarks for different values of the hard limit and the soft limit with moderate pruning in the Experiments chapter.

Both of these limit techniques may result in a problem specific to multi-criteria journeys. By pruning a witness journey before it is dominated naturally, a later rRAPTOR iteration

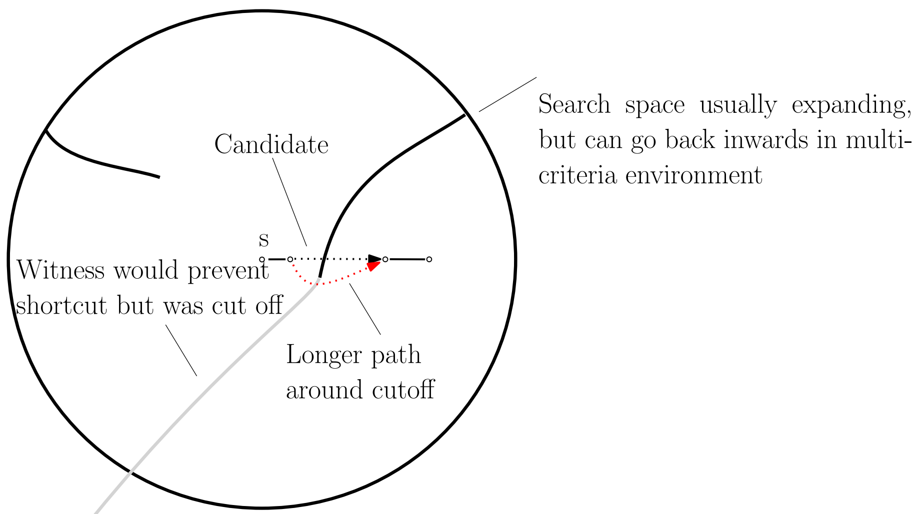

Figure 4.1.: Because the multi-criteria Dijkstra phase is not necessarily just expanding the search space, but could revisit already settled vertices, a journey which would cross over a candidate and would prevent that candidate from becoming a shortcut may be pruned by our optimizations midway, leaving the shortcut to be able to be created with a detour. This is not a problem, however, as one of two cases can happen: Either this shortcut would never have been added anyways, then it is not important for any shortest path and will therefore never lead to a wrong result in the queries, or it is an important shortcut, which means that it is part of a shortest journey at some time, at which point it will not be influenced by such a "cut" and we can simply update the shortcut length with the proper length.

may create a shortcut which would normally not exist because it would be fully dominated by the witness journey that we pruned earlier. This is the case for ULTRA and McULTRA alike, but in the multi-criteria case, this may lead to shortcut candidates which have non-optimal transfer durations. Figure 4.1 illustrates this problem.

In other words, shortcuts may be created which are too long. This only happens when the criteria chosen for the algorithm are not label setting. It is critical when such a shortcut is added to the graph and a later iteration finds a similar shortcut, that the edge weight of the shortcut must now be updated to the minimum of both. In ULTRA on the other hand, it was sufficient to simply trust that the edge weight already contained in the shortcut graph was correct. While this sounds scary, the good part about this is that the shortcuts with wrong distances are always shortcuts which would have been dominated anyways. This means that if a shortcut is important for any iteration, that iteration will still happen and it will update the shortcut graph accordingly. The result is that no important shortcuts remain with incorrect distances in the shortcut graph.

The explanation of this problem is that specifically for multi-criteria journeys, we cannot guarantee that a vertex visited during the transfer phase is never visited again. In ULTRA, such a witness journey would always be pruned at the boundary of the search space. In turn, that results in later iterations to behave normally up until the point where the witness was pruned. Because it is at the edge of the search space, a candidate search will continue to go outward from this point on and therefore not create wrong shortcuts. However since the search can return into a settled part of the graph when using multiple criteria, the end of witness journey may be inside the boundary of the search space and hence could influence searches in later iterations.

One more optimization that ULTRA does is to prune a label as early as possible when it is clear that it can only lead to a shortcut that already exists in the shortcut graph. It is easy to see why this leads to problems with the previous optimization. We adapted this technique and tested it both for the intermediate transfer phase, which is the earliest place to do this, and the second route scan phase, which is the latest place this optimization still makes sense. We also provide numbers for these variants in the Experiments chapter.

4.1.1. Implementation Details

Labels

First we will take a look at the label structure. We can simplifly it quite significantly compared to an implementation of MCR since we dont need to be able to unpack full journeys. At the same time, we need to add more complexity to the label because they now need to identify themselves as candidates or witnesses because a bag can contain both without them dominating each other. As with the McRAPTOR implementation, we keep the trip ID in the label.

Our labels thus contain the following information: arrival time, walking distance, trip id, a shortcut candidate origin stop and a shortcut candidate destination stop, which serve to determine which transfer was responsible for this candidate in case it will result in a shortcut. These fields also serve as a discriminator between shortcut candidates and witnesses, too.

Additionally, labels provide a composite key for the Dijkstra search phases.

Bags

The bags our McRAPTOR implementation are very simple, however as in MCR, they receive more responsibilities in our implementation here. First, as with MCR, a bag now also implements a priority queue to be able to retrieve the label with the smallest key efficiently. Since we cannot avoid settling a stop more than once, instead we do not to settle any individual labels more than once by dividing the bags space in settled and unsettled labels, as described earlier.

Additionally, every bag now also tracks how many candidate labels are contained in the its priority queue as this information is key in understanding how many candidates are still in the global queue. Finally, whenever a bag is changed by merging a label into it, the new label is inserted into the priority queue, which is then rebuilt since a number of labels could have been dominated by the newly added label.

The merge operation itself is handled much like we described in the McRAPTOR variant, however after the operation, a final check is done to see if the bag is currently contained in the global priority queue, and if so, its position in it is updated. As with MCR, the bag's key for the global priority queue is the key of its smallest label.

5. Experiments

In this chapter we compare different values for the optimizations we have introduced and explain our findings. We also show our benchmarks and compare them to other algorithms.

5.1. Data

We used the following transit graphs for our experiments: The public transit network of Switzerland (GTFS data), the public transit network of Germany provided by Deutsche Bahn, and the public transit network of London (GTFS data). See Table 5.1 for their sizes. For the purpose of pre-processing transfers, we used contracted OpenStreetMap[1] data of the transport networks' respective areas. If it is not otherwise stated, these transfer graphs have been previously contracted to a vertex degree of 14 and the movement speed on them has been limited to a casual 4.5 kilometers per hour.

5.1.1. Computer Specs

All of our pre-processing experiments, unless stated otherwise, have been run on a `Gigabyte R282-Z93` Server with 1024 GiB of DDR4 3200MHz ECC memory. It uses two `AMD EPYC 7742` CPUs with 64 cores each, which we utilized fully for the pre-processing, as it is easily parallelized.

The experiments specifically comparing query times of McULTRA-McRAPTOR to MCR were conducted on a `Supermicro SuperServer SYS-6029UZ-TR4+` with 192GiB DDR4 2666 MHz ECC memory. It uses two 8-core `Intel Xeon Skylake SP Gold 6144` CPUs clocked at 3.5 GHz, but only a single core was utilized to run queries. Note that this server and test-setup is comparable to the server used for the original ULTRA experiments.

[1] https://www.openstreetmap.org

Network	Stops	Routes	Trips	Stop events	Vertices	Full edges	Tran. edges
London	20 595	2 107	125 436	4 970 428	183 025	347 737	3 755 200
Switzerland	25 426	13 934	369 534	4 740 929	604 167	1 847 140	4 687 016
Germany	244 055	231 089	2 387 297	48 495 169	6 872 105	21 372 360	22 645 480

Table 5.1.: Sizes of the used public transit networks and their transfer graphs (full and transitive).

We use the `C++` programming language with the gcc compiler version `9.3.1` using `-std=c++17` and optimization flag `-O3`.

5.2. Pre-Processing

In this section we examine our experiments relating to the pre-processing of the shortcut graph.

5.2.1. Key Composition

The key for the priority queue in the transfer phases can be made up of any combination of the label's criteria. We tested the impact of using only the arrival time, only the walking distance, or the sum of both as keys. To our surprise, using only the walking distance as key slightly beats the other combinations of keys when the soft limits increase. However, the differences are still relatively insignificant, as at its best, the walking distance as key is about one percent faster as the rest, while at its worst, it is around three percent worse for other parameters. Using the sum of the arrival time and the walking limit provided preprocessing times which were more stable than using single keys as the preprocessing time remained in between the other key possibilities.

5.2.2. Prune After Last Candidate

When running the transfer phases, it is possible to stop the processing after the last candidate has left the queue, as from that point on, there is no way that a new candidate might appear somewhere. However, doing this will create unnecessary shortcuts in later rRAPTOR iterations, which will not invalidate the query algorithm, but they will slow it down. If the pre-processing time is significantly faster, though, this trade-off might be sensible. For this reason, we examined the effect different methods and parameters have on both run-time and number of shortcuts created on the Switzerland instance. As a baseline, we provide the statistics with none of these optimizations enabled, and then we show the effect of the *soft limit* and the *hard limit*. We show the full results in table 5.3.

Limit Type	Limit Value	Duration [min:s]	Shortcuts [#]
hard limit	0	9:12	806 744
	100	9:11	806 347
	1000	9:31	803 355
soft limit	0	10:03	798 128
	300	10:10	798 012
	600	10:19	797 818
	1200	10:17	797 475
	3600	10:44	796 019
	7200	11:18	794 234
no limit		12:52	784 045

Table 5.3.: Results of limiting the amount of labels computed after the last candidate left the queue in the final transfer.

The *soft limit*, as explained earlier, simply retains the maximum for every criterion from the previous route-scanning phase and for every settled candidate in the transfer phase and adds a criterion-specific limit on top. Any label that exceeds any (strict pruning) or all (moderate pruning) of those limits is pruned. For our experiments shown here, we

chose moderate pruning. This will result in an unknown number of labels to continue to be processed. In our case, the criteria to be considered for parameters would be a walking distance limit and an arrival time limit. However, as both of these parameters grow linearly in the transfer phase, there is not much reason to view them as separate parameters.

When a label's arrival time increases by a certain amount during the transfer phase, so does its walking distance. Therefore we did not expect a significant change when using two parameters compared to when using a single parameter for both, as on average, a label would always be pruned for the lower of the two parameters, making the other one unnecessary. Our experiments showed that there is less than a half percent difference when using wildly different values than simply using the lower value for both. Therefore, we only use the walking distance limit parameter. However the choice of limits has to be considered for every other criterion anew.

The *hard limit* simply counts how many shortcuts are still left in all bags in the queue, and once this count reaches zero, the transfer phase will continue to visit k vertices before halting. This is a simpler mechanism than the soft limit which is faster but also creates some more shortcuts. At the same time, this table gives an intuition of how many shortcut candidates are still dominated by witness journeys in the final transfer phase alone.

To test the impact of this technique when using it on the intermediate transfer, we used a hard limit of 100 for the final transfer for every experiment so that we could observe the direct impact the pruning has for only this part. It is easy to see in table 5.5, that the pruning in the intermediate journeys is very effective in reducing the run-time further. Furthermore, both of these limits are useful, whereas the hard limit allows for even shorter preprocessing times than a soft limit of 0, the hard limit is less effective the higher the limit is because on some iterations, it might be useful to process more labels than on others. At a soft limit of 7200, this technique overtakes the hard limit by producing less shortcuts in less time.

Limit Type	Limit Value	Duration [mm:ss]	Shortcuts [#]
soft limit	0	5:55	883 732
	60	5:56	883 731
	120	5:57	882 649
	300	5:57	879 801
	600	5:58	876 575
	1200	6:03	870 684
	1800	6:04	865 630
	3600	6:14	855 102
	7200	6:34	839 808
hard limit	0	4:56	871 170
	10	6:47	871 317
	100	8:20	824 753
no limit		9:11	806 347

Table 5.5.: Results of limiting the amount of labels computed after the last candidate left the queue in the intermediate transfer, with a hard limit of 100 in the final transfer.

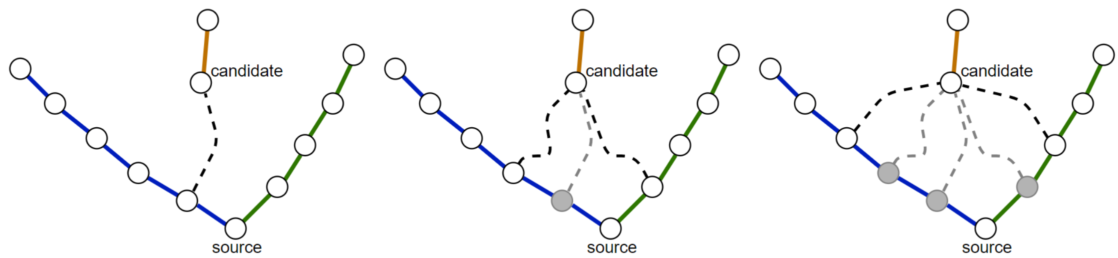

Figure 5.1.: In a multi-criteria environment, pruning the shortcut that was created in the first iteration (left) can lead to multiple shortcuts, which would have been dominated by the first shortcut, being created in the next iteration (center), which in turn can lead to even more shortcuts to be generated when they are pruned in the next iteration.

Pruning Method	Pre-Processing Duration [mm:ss]	Shortcuts [k]
none	9:11	806
early	10:53	2,720
late	8:54	1,016

Table 5.7.: Results of pruning when an existing shortcut has already been found. The graph instance used here is Switzerland with a contraction degree of 14 and a speed limit of 5 km/h.

5.2.3. Prune Existing Shortcuts

The ULTRA pre-processing suggests an optimization which prunes a candidate label as early as possible when a candidate is being processed for which there is an existing shortcut already in the shortcut graph. Since this missing shortcut can then make way for a different shortcut which would otherwise have been dominated, the number of total shortcuts will likely increase. This increase in unwanted shortcuts is tolerable and a trade-off for faster processing speed. However there is a problem with this when we take another criterion into account. While ULTRA can only ever add a single shortcut per vertex and per rRAPTOR iteration, McULTRA can add an arbitrary amount of shortcuts per vertex and iteration because there may be an arbitrary amount of pairwise non-dominating candidates in the vertex' bag. You can see an illustration of how a pruned candidate can lead to several more candidates in each iteration in Figure 5.1.

This means that while the amount of unwanted shortcuts that can be created per vertex and per iteration is limited to a single one for ULTRA, in McULTRA this can actually get out of hand, which is exactly what we observed in table 5.1.

We tried this pruning strategy in the intermediate transfer phase, pruning candidate labels as soon as they were about to be added to a stop. This is the *early* pruning step. Then we also tried this principle in the second round scanning step, to simply skip processing any candidate which would result in a shortcut that already existed. We call this one the *late* pruning step.

This was tested on the contracted Switzerland graph with a limited walking speed. The early pruning step was quite horrible, increasing the pre-processing time by more than 20 percent, the number of shortcuts generated more than doubled. On the other hand, the late pruning step yielded a slight decrease in pre-processing time while only increasing the number of generated shortcuts by 25 percent. So while the early pruning step is worse in every way, the late pruning step might be considered, although the trade-off is still large. Note that these experiments ran using a hard limit of 100 labels for the final transfers and no other optimizations.

5.2.4. Transfer Speed

Originally, ULTRA had the advantage that an increasingly fast transfer speed would result in many shortcuts being pruned simply because the arrival time using the initial transfer becomes increasingly faster than using any public transport vehicle. This is not the case when incorporating the walking distance as a criterion, as a journey that is fast but only consists of a single transfer does not dominate a journey that uses one ride and a shorter transfer, or two rides and less transfer time, etc. Our experiments show that as the speed of transfers increases, the shortcut graphs approach an upper limit in an exponential curve, as we illustrate in Figure 5.2.

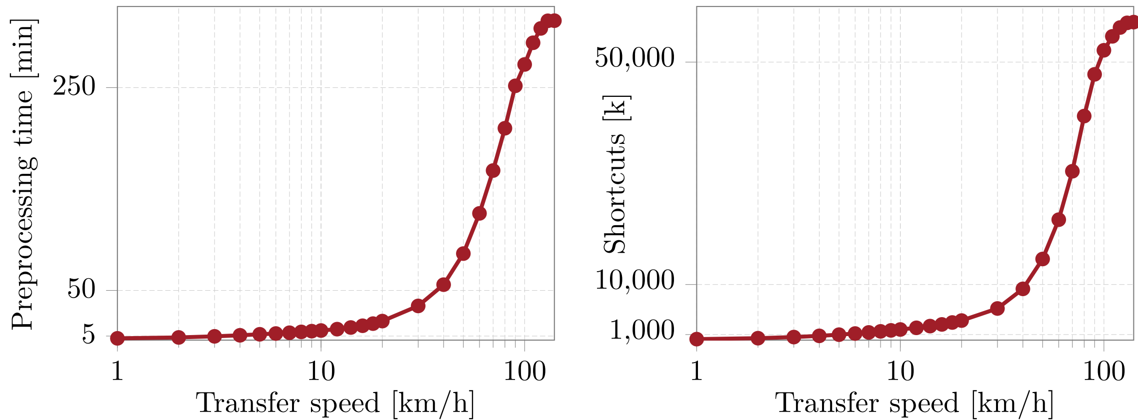

Figure 5.2.: Impact of transfer speed on pre-processing time and number of computed shortcuts, measured on the Switzerland network with a core degree of 14, a hard witness limit of 100 vertices in the final transfer and a soft witness limit of 10 minutes in the intermediate transfer. It is easy to see that with increasing transfer speed, the resulting shortcut graph approaches an upper limit with a steep increase in shortcuts and preprocessing time in starting at 20km/h.

The McULTRA technique remains very effective up to a speed of 30 km/h, but loses its advantage over MCR at speeds of 50km/h and above.

This is less the result of the McULTRA technique than the nature of the walking distance as a criterion. We can see in Figure **??** that the majority of useful shortcuts, even at slow speeds, are very long. So increasing the transfer speed enables even more longer shortcuts to be created, whereas in the bicriteria ULTRA, at faster transfer speeds, transfers become increasingly useless as they are dominated by a direct transfer, which is not the case when using the transfer duration as a criterion, as a journey which may take longer but uses less of a transfer duration will not be dominated. Such journeys are increasingly easy to find, the higher the transfer speed is, which explains the explosion of created shortcuts at higher speeds in Figure 5.2.

Unlike in ULTRA, we did not observe any significant difference in preprocessing speed or number of shortcuts when filtering out isolated stops, which are stops which are isolated from the transfer graph, which means that they can only be reached using at least one ride.

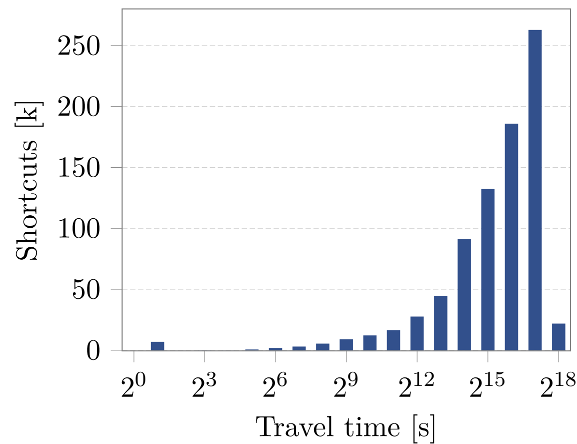

Figure 5.3.: Distribution of the lengths of generated shortcuts on the Switzerland instance restricted to walking speed. The bars with the labels 2^i for $i > 0$ include all shortcuts with travel time τ in the half-closed interval $[2^{i-1}, 2^i)$. The 2^0 bar represents the interval $[0, 2^0)$, i.e., all shortcuts with travel time exactly zero.

5.3. Queries

To evaluate the impact of our pre-processed shortcut graph on the query performance, we tested it with the multicriteria variant McRAPTOR on our preprocessed shortcut graph. The experiments consist of location-to-location queries, which do not need to be stops, with sources, targets, and departure times picked uniformly at random.

We compare our speeds to McRAPTOR without pre-processing, which computes fewer journeys, because it can only answer stop-to-stop queries. Additionally, we compare them to

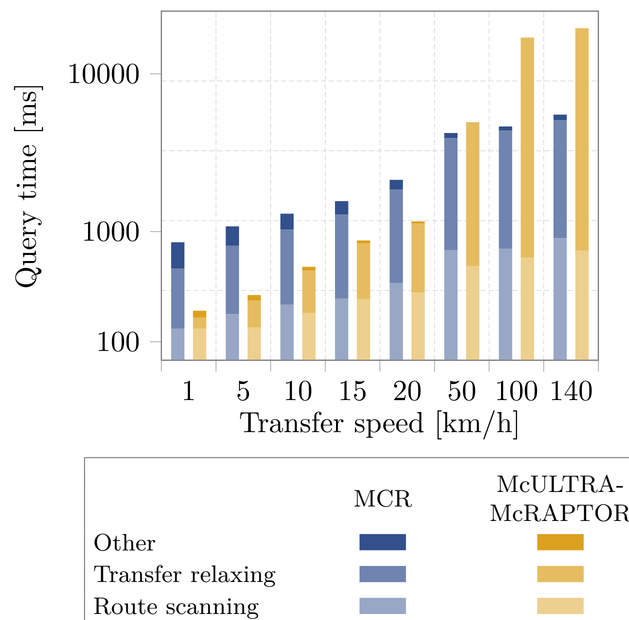

Figure 5.4.: Impact of transfer speed on query times and travel times, measured on the Switzerland network with a core degree of 14, an intermediate witness limit of 10 minutes and a final witness hard limit of 100. All results were averaged over 10 000 random queries. Speed limits were obeyed. Query times are divided into route collecting/scanning, transfer relaxation, and remaining time.

	Algorithm	Full graph	Counts [k]			Time [ms]				
			Routes	Edges	Merges	Init.	Collect	Scan	Transfer	Total
London	McRAPTOR*	○	16	20 011	59 656	2.8	3.2	135	325	476
	MCR	●	17	5 400	8 062	33.3	3.1	133	283	478
	McULTRA + McRAPTOR	●	16	1,383	6,188	2.25	2.96	123	52.9	190
Switzerland	McRAPTOR*	○	134	34 433	68 743	1 901	4.47	10.3	217	621
	MCR	●	139	5 946	11 273	167	12.6	228	309	820
	McULTRA + McRAPTOR	●	140	4 907	13 435	4.46	9.69	217	88.9	336
Germany	McRAPTOR*	○	3 395	308 481	1 132 828	63.5	353	7 525	6 236	14 475
	MCR	●	3 531	175 777	343 773	3 682	387	6 970	30 960	41 998
	McULTRA + McRAPTOR	●	3 483	118 421	461 577	71.4	371	8 027	3 955	12 747

Table 5.8.: Query performance for McRAPTOR, MCR and McULTRA-McRAPTOR. Times are divided into phases: scanning initial transfers, collecting routes, scanning routes, and relaxing transfers. All results are averaged over 10 000 random queries. Note that McRAPTOR (marked with *) only supports stop-to-stop queries with transitive transfers, hence is not capable to produce the same amount of journeys, whereas the other two algorithms support vertex-to-vertex queries on the full graph.

MCR on an unrestricted transfer graph using Core-CH for initial transfers, which produces identical journeys to our solution. The results are shown in Table 5.8.

To better understand how many operations have to be done, we also count how many merge operations were performed. Interestingly, our solution results in more merge operations than MCR, but the merge operation in McULTRA-McRAPTOR is simpler because MCR needs to keep its bags ordered for its Dijkstra-phases, whereas we have simpler transfer phases and perform less work for each merge. The graphs used for the queries for MCR, McRAPTOR and McULTRA-McRAPTOR are identical to the ones used for the preprocessing of McULTRA, which are contracted to degree 14 and restricted to walking speeds as described in Section 5.1. Note that the benchmarks from Dibbelt et al. [DDP+12] include another mode of transport: transfers by cycling, using bicycle lending stations, which are computed by employing an MLC query before every transfer phase. While Dibbelt et al. do not introduce a new criterion for this transfer mode, it does change the resulting experiment numbers slightly compared to our implementation, which uses purely walking for transfers.

Nonetheless, combining McRAPTOR queries with McULTRA preprocessing calculates queries more than twice as fast as MCR for walking speeds.

Finally, we look at the performance of MCR and McULTRA-McRAPTOR when it comes to different allowed transfer speeds in the transfer graph in Figure 5.4. We can see that McULTRA-McRAPTOR is surprisingly efficient on very low speeds, but becomes slower at a fast pace after 20km/h are reached, which coincidentally is the point where walking as a mode of transportation is not feasible anymore. In contrast, MCR starts out with more than double the query time of our technique, but increases slower and can handle transfer

speeds of 140km/h significantly faster. However, at those speeds, it can be argued that the distance travelled is not important anymore as it is usually handled by taxis, etc. So in this case, using cost as a criterion would be better.

5.4. Heuristics

In this section we examine different heuristics to increase the speeds of the algorithm. We score the quality of these heuristics using the system explained in Section 2.4.3 with both a k value of 3 and 6, for a good comparison to MCR [DDP+12].

5.4.1. Overdomination

A very crude way to reduce the number of journeys produced is to make every label dominate more space while keeping strict domination. For every criterion we decrease its value for the purpose of domination before comparing it with another unchanged label. It is possible to simply subtract a constant value per criterion, which is called a *slack*, but in our case we decrease the value by 10% for arrival time and walking distance. The resulting space can be seen in Figure 5.5 and in Table 5.10 we can see that while this approach is very fast and produces the least journeys, the quality of those journeys is rather low. As such, this method should only be considered for demonstration purposes. Note that naturally, by doing this, the order in which labels are added matters for domination purposes.

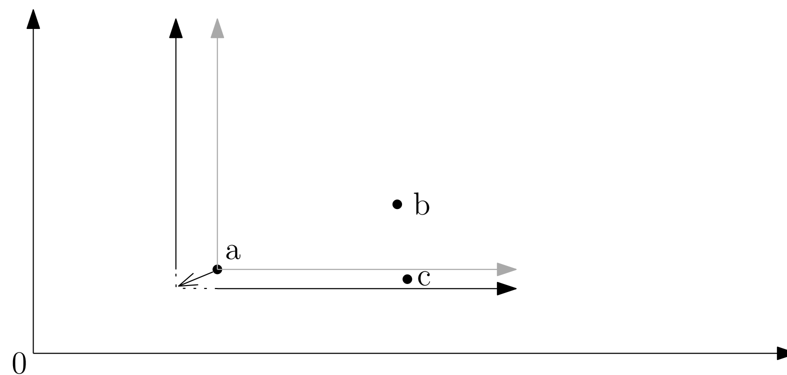

Figure 5.5.: A journey leg *over-dominates* by a percentage of it's criteria values, increasing the area it dominates. The journey leg b is strictly dominated by a while c is only dominated after applying the over-domination to a.

5.4.2. Fuzzy Domination

As described in Section 2.4.1, we implemented Fuzzy Domination for both the query algorithm and the pre-processing technique. For the query algorithms, we chose the following values for (χ, ϵ): For arrival time $(0.8, 60)$, for walking duration $(0.8, 300)$, both values for ϵ in seconds, and for the number of rides we chose $(0.1, 1)$. We chose these values based on the work of Dibbelt et al. [DDP+12] to compare our numbers to theirs. We use the London graph limited to a walking speed of 4.5 km/h. Note that for the pre-processing, we only used arrival time and walking duration for computing the fuzzy degree as the number of rides taken never rises above two. Additionally, we also tried this approach of only using two criteria for the queries, which did not lead to faster queries and is therefore not useful.

The speed-up of this technique for MCR is roughly a factor of 2 while the loss in quality is minimal at just above 3% for our tests. These numbers still hold up when using the pre-processed graph from McULTRA with McRAPTOR, also improving the query time by a factor of 2 while keeping the same quality of journeys of 96.7%. Using the fuzzy dominance technique for the pre-processing yields less impressive results. It barely improves

the run-time of queries. In Table 5.9 we can see that the pre-processing is also barely faster for walking speeds (slower for faster speeds) and that it creates 1/6th less shortcuts for walking speeds to 1/3rd less shortcuts for a speed of 20km/h typical for cycling.

Heuristic	Time [min]	Shortcuts [k]
Switzerland, 20km/h		
Normal	30.5	3.182
Fuzzy	38.8	2.043
Bucket	15.2	2.485
London, 4.5km/h		
Normal	37.5	301
Fuzzy	37	246
Bucket	17	411

Table 5.9.: Pre-processing performance for fuzzy dominance and discretization (bucket) heuristics for London at walking speed and Switzerland at cycling speed.

5.4.3. Discretization

When evaluating the performance for the discretization heuristic, we only used buckets for the walking distance criterion. For efficiency, we always chose a power of two and settled on 256 for the experiments shown in Table 5.10. As can be seen, for the base MCR algorithm, this improves the runtime by more than a factor of 3, however the quality of the produced journeys is reduced compared to the fuzzy domination heuristic.

For queries using the McULTRA pre-processing, a similar picture is shown, improving the query time by a factor of over 3 with slightly worse quality than the fuzzy domination. This yields a total average query time of just 65 milliseconds which can be reduced by slight margin when using the fuzzy-domination technique for the pre-processing to get less shortcuts, together with buckets for the query, resulting in 63 milliseconds.

Using the bucket technique for the walking duration in the pre-processing reduces the time it takes to generate the shortcut graph significantly, as can be seen in Table 5.9, but it comes at the cost of producing more shortcuts, which decreases query times later.

Pre.[1]	Pre. Heuristic	Query Heuristic	Jn. [#]	Time [ms]	Quality-3	SD	Quality-6	SD
○			36	486	100.00%	0.00%	100.00%	0.00%
○		Fuzzy	10.9	226	96.76%	10.98%	97.24%	9.21%
○		Bucket	12.3	141	94.24%	12.33%	95.23%	10.04%
•			36	207	100.00%	0.00%	100.00%	0.00%
•		Fuzzy	10.6	102	96.76%	10.97%	97.23%	9.25%
•		Fuzzy 2*	12.2	109	97.18%	10.58%	97.26%	9.33%
•		Bucket	12.1	65	94.85%	11.99%	95.76%	9.84%
•		Overd.	7.9	48	24.68%	30.37%	39.34%	31.05%
•	Fuzzy		35.9	199	99.26%	8.15%	98.99%	8.31%
•	Fuzzy	Fuzzy	10.6	99	96.75%	10.97%	97.19%	9.26%
•	Fuzzy	Bucket	12.1	63	94.78%	12.03%	95.68%	9.87%
•	Bucket		35.4	219	99.06%	8.43%	98.63%	8.59%
•	Bucket	Fuzzy	10.6	112	96.66%	11.07%	97.02%	9.38%
•	Bucket	Bucket	12.1	72	94.15%	12.39%	95.07%	10.13%

Table 5.10.: Average query performance and number of computed journeys in regards to different heuristics for the London Network using a transfer speed of 4.5km/h. All start and stop nodes, as well as departure times have been chosen uniformly at random for 10 000 queries. Quality of the heuristics for $k = 3$ and $k = 6$ including their respective standard deviations.
[1]: Unticked means that MCR was used, while ticked means that the McULTRA pre-processed graph was used with the McRAPTOR query algorithm.
*: Using only arrival time and walking distance for fuzzyness.

6. Conclusion

We presented an overview of public transit routing techniques and explained the importance of multi-modal, multi-criteria route planning. To solve this problem, we introduced McULTRA, a technique for calculating multi-modal, multi-criteria journeys using unlimited transfers in walking speed. It provides a significant speedup of almost 3 towards MCR, which has been one of the fastest techniques to solve this problem so far. Using benchmarks and experiments, we gave an overview over different speed-up techniques and heuristics to improve our algorithm even further, and examined their strengths and weaknesses.

Using a combination of these techniques, we accomplish high quality journeys with three criteria: arrival time, walking distance and number of rides, in just 63 milliseconds. By using Fuzzy Dominance in the pre-processing, we reduce the size of the generated shortcut graph, while rounding the walking distance in the queries then accelerates the queries dramatically.

Further research can be done to understand other criteria and the impact they have on query and pre-processing times as well as on the sizes of the shortcut graphs. Studies can also look into which criteria have the biggest impact on the consumer satisfaction.

There are a number of topics just finishing or currently being researched to further improve on the ULTRA technique, which includes incorporating a tolerance for delays, adding more complex modes for transfers (rentals, sharing, etc.), improving one-to-many performance and utilizing more diverse public transit routing algorithms efficiently.

The goal for the future is to keep the versatility of the multi-criteria, multi-modal approach while vastly improving its performance to make ubiquitous usage possible. By improving the query times dramatically, we have come one step closer to that goal.

Epilogue

I would like to thank my friends, family and special people in my life for enduring with me and cheering me on. I would also like to thank everyone at the Institute of Theoretical Informatics at KIT for giving me the chance to work on a very exciting and relevant topic, especially my advisors for their endless patience and helpfulness. Finally, I want to thank all the wonderful people around the world doing their best in this current worldwide crisis.

The world is indeed full of peril, and in it there are many dark places; but still there is much that is fair, and though in all lands love is now mingled with grief, it grows perhaps the greater.

J.R.R. Tolkien
The Fellowship of the Ring

Bibliography

[BBS+19] Moritz Baum, Valentin Buchhold, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. UnLimited TRAnsfers for Multi-Modal Route Planning: An Efficient Solution. *CoRR*, abs/1906.04832, 2019.

[BDS+10] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-directed Speed-up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15:2.3:2.1–2.3:2.31, March 2010.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

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

[DDP+12] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing and Evaluating Multimodal Journeys. Technical Report 2012-20, Faculty of Informatics, Karlsruhe Institute of Technology, 2012.

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

[DPSW18] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection Scan Algorithm. *ACM Journal of Experimental Algorithmics*, 23:1.7:1–1.7:56, October 2018.

[DPW09] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering Time-Expanded Graphs for Faster Timetable Information. In *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 182–206. Springer, 2009.

[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.

[DSW15] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Fast exact shortest path and distance queries on road networks with parametrized costs. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '15, New York, NY, USA, 2015. Association for Computing Machinery.

[For56] Lester R. Ford, Jr. Network Flow Theory. Technical Report P-923, Rand Corporation, Santa Monica, California, 1956.

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

[GH04] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. Technical Report MSR-TR-200, Microsoft Research, 2004.

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

[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.

[GW05] Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.

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

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

[NDLS08] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* Search for Time-Dependent Fast Paths. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 334–346. Springer, June 2008.

[OR89] Ariel Orda and Raphael Rom. Traveling without Waiting in Time-Dependent Networks Is NP-hard., March 1989.

[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.

[Sch08a] Dennis Schieferdecker. Systematic Combination of Speed-Up Techniques for exact Shortest-Path Queries. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, January 2008.

[Sch08b] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, February 2008. http://algo2.iti.uka.de/schultes/hwy/schultes_diss.pdf.

[SS05] Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[SS06] Peter Sanders and Dominik Schultes. Robust, Almost Constant Time Shortest-Path Queries in Road Networks. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge* -, November 2006.

[WW03] Dorothea Wagner and Thomas Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03)*, volume 2832 of *Lecture Notes in Computer Science*, pages 776–787. Springer, 2003.

[WZ17] Dorothea Wagner and Tobias Zündorf. Public Transit Routing with Unrestricted Walking. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2017.

Appendix

A. Real World Example

In Figure A.1 we show three example journeys returned from our algorithm on a real-world scenario and real map. The journey departs at 8:00 am from Horwerstrasse and the target is Brambergstrasse in Luzern, Switzerland.

Yellow. The direct transfer from the source to the target arrives at 8:47 am and accordingly, involves 47 minutes of walking.

Red. This journey contains an initial transfer to Schachenstrasse, which requires four minutes to walk, a bus ride to Kasernenplatz and a final transfer requiring 13 minutes, with a final arrival time of 8:33 am and a total walking time of 17 minutes. By using public transport, the arrival time could be reduced considerably. (Multi-Modal)

Blue. This journey has an arrival time of 9:15 am, which means that its arrival time is worse that the previous two mentioned journeys. It also requires to use two bus rides, from Zunacher to Kantonalbank, and then from Kantonalbank to Bramberg Station, instead of just the one ride for the red journey. This means that an algorithm that allows unlimited transfers and only optimizes for arrival time and number of rides, e.g.MR-∞ or ULTRA-RAPTOR, will not show this journey. However as the total walking time for this journey only amounts to three minutes, it may be more desirable for some people, e.g., the elderly, while other people may gladly walk for longer periods for a faster arrival time. This type of journey is found by taken the walking duration as a criterion into consideration, as MCR and McULTRA-McRAPTOR do.

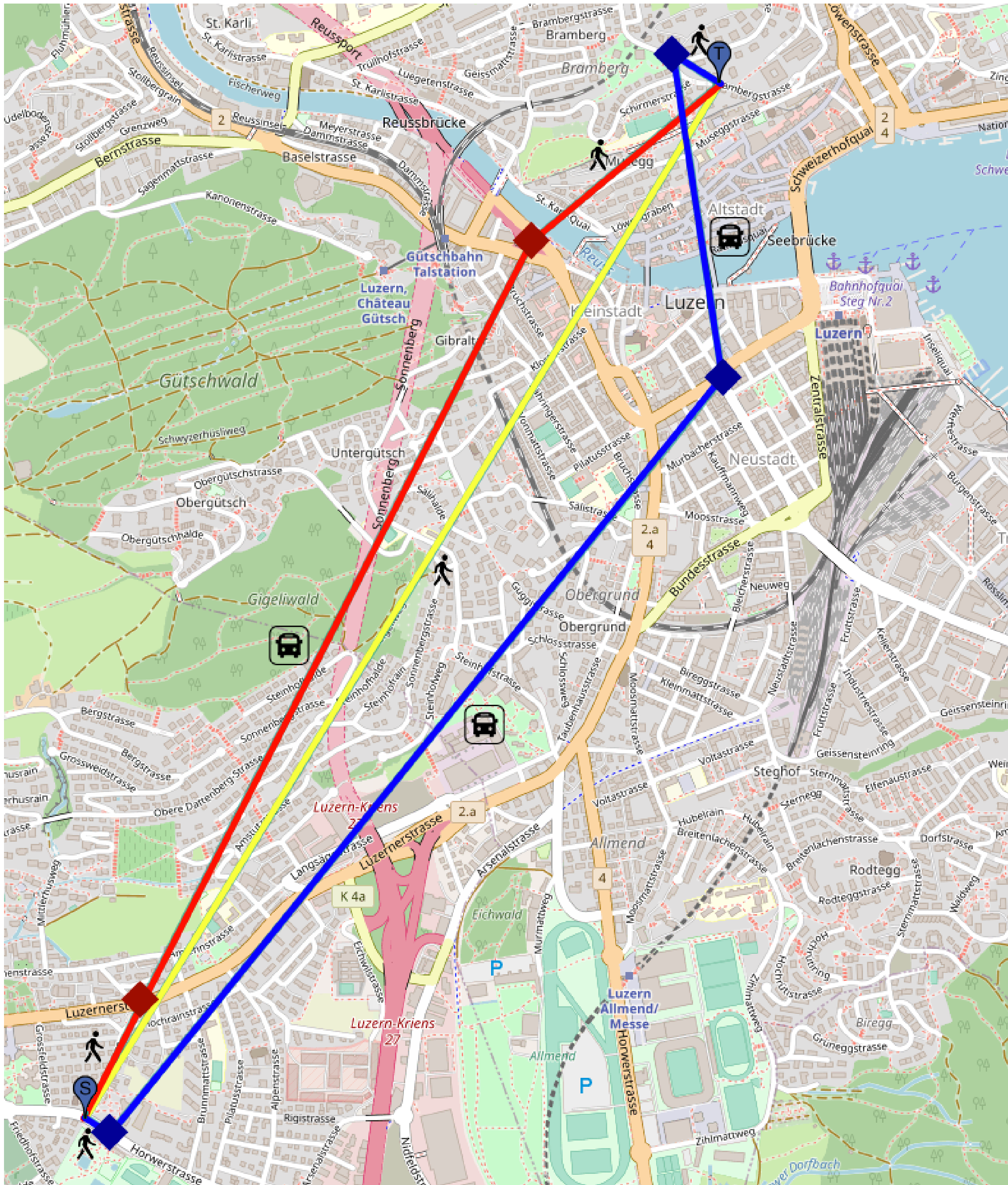

Figure A.1.: A real example of different journeys for a single multi-modal, multi-criteria query from Horwerstrasse (s) to Brambergstrasse (t) in Luzern, Switzerland, with a departure time of 8:00 am.