Karlsruhe Institute of Technology

Multicriteria Trip-Based Public Transit Routing

Bachelor Thesis of

Moritz Timo Potthoff

At the Department of Informatics
Institute of Theoretical Informatics

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

Time Period: 24th September 2020 – 23rd December 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, 23. Dezember 2020

Abstract

The Trip-Based Public Transit Routing Algorithm computes Pareto optimal journeys in public transit networks for two criteria: earliest arrival time and minimum number of transfers. This work extends the algorithm to compute Pareto optimal journeys for an additional criterion. Specifically, the Walking Trip-Based algorithm additionally minimizes walking times along journeys and the Fare Zone Trip-Based algorithm optimizes for minimal fare zone subsets as third criterion. Both preprocessing and query stage of the original Trip-Based algorithm are adapted and carefully optimized in the development of the extended algorithms. Both algorithms are extensively evaluated on three real world public transit networks and query running times are compared to those of McRAPTOR. The Walking Trip-Based algorithm is faster than the Walking McRAPTOR algorithm on all three networks and all footpath configurations. The speedup achieved over McRAPTOR ranges from 1.18 to 3.47. Parallelized preprocessing runs in few minutes for small networks and takes only about 36 min for a complex variant of Switzerland. The second extension, the Fare Zone Trip-Based algorithm is significantly slower than the Fare Zone McRAPTOR algorithm. Based on these findings, it is analyzed for which sets of optimization criteria the Multicriteria Trip-Based approach is likely to be fast and possible integrations with other algorithms are discussed.

Deutsche Zusammenfassung

Der Trip-Based Public Transit Routing Algorithmus berechnet Pareto-optimale Routen in öffentlichen Verkehrsnetzwerken für zwei Kriterien: früheste Ankunftszeit und minimale Anzahl Umstiege. Diese Arbeit erweitert den Algorithmus, um Pareto-optimale Routen für ein zusätzliches Kriterium zu berechnen. Der Walking Trip-Based Algorithmus minimiert zusätzlich die Fußwegezeit entlang einer Route. Der Fare Zone Trip-Based Algorithmus optimiert Routen zusätzlich für die minimale Menge an genutzen Tarifzonen. Dazu werden sowohl die Vorberechnungs- als auch die Anfragephase des Algorithmus angepasst und optimiert. Die entstandenen Algorithmen werden ausführlich auf drei realen öffentlichen Verkehrsnetzen evaluiert und Anfragelaufzeiten werden mit denen von McRAPTOR vergleichen. Der Walking Trip-Based Algorithmus berechnet Anfragen auf allen Netzwerken schneller als der McRAPTOR Algorithmus. Die erreichte Beschleuningung gegenüber McRAPTOR liegt zwischen 1.18 und 3.47. Die parallelisierte Vorberechnung läuft für kleine Netzwerke in Minuten und braucht für eine komplexe Variante der Schweiz nur 36 min. Die zweite Erweiterung, der Fare Zone Trip-Based Algorithmus, ist deutlich langsamer als die McRAPTOR-Variante. Basierend auf diesen Erkenntnissen wird analysiert für welche Kombinationen von Optimierungskriterien der multikriterielle Trip-Based-Ansatz vielversprechend ist und mögliche Integrationen mit anderen Algorithmen werden diskutiert.

Contents

1. Introduction

In recent years, public transit has gained significance in everyday life. Along with this, good route planning for public transit networks has become more important. Route planning is a well-studied problem for road networks. Algorithms are tuned to the specific properties and requirements of road networks. Even for very large networks, the resulting algorithms calculate optimal journeys efficiently [BDG+16]. Route planning algorithms for road networks operate directly on a graph modelling the road network. In contrast, public transit routing inherently operates on timetables. Initially, timetables were modelled as graphs to be able to use algorithms designed for road networks on these graphs. The *time-expanded model* rolls out events at a stop (arrival or departure of a trip) into the graph: For each event, a copy of the stop is added as a vertex. Vertices corresponding to subsequent stops of the same trip are connected with directed edges to represent travelling along the trip. Vertices that correspond to the same physical stop are connected to represent the possibility to change trips at the stop. In the *time-dependent model*, every stop is represented by exactly one vertex in the graph. Vertices are added whenever there is any connection between the stops represented by the vertices. Edges in this graph are time-dependent: Each edge has an associated function that models *when* individual trips are possible between the stops [BDG+16, MHSWZ07].

Both approaches have drawbacks. In the time-expanded model, graphs get very large since there are usually many events at every individual stop. For the smaller graphs of the time-dependent model, algorithms must obey the time-dependence of the edges. Moreover, the resulting graphs do not have the same properties as road networks. For instance, road networks usually have a strong hierarchy and small neighborhoods for most stops. In graphs that model timetables, hierarchies are not as strong and neighborhoods can be large. In consequence, existing algorithms that are fast on road networks often do not show the same good performance when used for public transit routing [BDG+16].

This led to the new approach to operate directly on the timetable information. Algorithms like CSA [DPSW13], RAPTOR [DPW15] and Trip-Based Public Transit Routing [Wit15] are specifically designed for public transit routing and exploit characteristics of the problem. For instance, RAPTOR and the Trip-Based Public Transit Routing Algorithm are based on the observation that journeys in public transit networks consist of a seqeunce of trips that are connected by transfers. Additionally, they benefit from simple operations and cache efficient access patterns. Using this new approach, queries can be answered efficiently, even for large networks [BDG+16].

As public transit becomes more widely used, requirements for public transit routing increase. It is no longer sufficient to compute simple earliest arrival queries. For instance, good journeys often contain long walking sections. Without optimizing for walking times, journeys with any amount of walking can be the only solution of a query, regardless of the potential existence of another journey with significantly less walking at the cost of a slightly later arrival time. Therefore, relevant results can often only be achieved in multicriteria settings [BDG+16]. While many algorithms can already efficiently optimize journeys for the two criteria earliest arrival time and minimum number of transfers, there are fewer efficient solutions for more criteria.

Another field of study in routing algorithms is *multimodal* routing. In this setting, journeys are not restricted to one mode of transportation (e.g. public transit, private cars or airplanes) but can combine multiple modes for one journey. This imposes a range of new challenges. Concerning public transit routing, queries are inherently multicriterial. They must optimize criteria such as walking times, number of transfers and journey price [BDG+16]. Therefore, the significance for multicriteria public transit routing has increased.

1.1. Related Work

Multicriteria queries for public transit routing can be answered by existing algorithms. The *Multicriteria Label-Setting* (MLS) algorithm is an extension of Dijkstra's algorithm that optimizes multiple criteria by keeping Pareto sets of labels at each stop [BDG+16]. It can be used to compute Pareto optimal journeys in public transit networks using the time-expanded model [BDG+16, MS07].

Delling, Pajor and Werneck proposed McRAPTOR (*More criteria Round bAsed Public Transit Optimized Router*), an extension of their RAPTOR algorithm [DPW15]. It performs significantly better than the MLS algorithm for queries optimizing three criteria according to [BDG+16]. It is described in detail in Chapter 2. The algorithms developed in this work will be compared to the McRAPTOR algorithm.

1.2. Contribution

This work extends the Trip-Based Public Transit Routing algorithm for use in a multicriteria setting. The original algorithm uses preprocessing on transfers to evaluate queries efficiently. It optimizes arrival times and number of transfers. Two extensions are presented in this work: First, the *Walking Trip-Based Algorithm* optimizes arrival time, number of transfers and total walking time along journeys. Second, the *Fare Zone Trip-Based Algorithm* minimizes arrival time, number of transfers and the subset of fare zones used by a journey to find cheap journeys.

The preprocessing phase of the algorithm is adapted for both variants to keep all necessary transfers while still reducing the number of transfers effectively. Properties of the new criteria concerning the Trip Based algorithm are explored and the correctness of the preprocessing algorithms is proven. The query phase is adjusted to correctly handle the additional criterion. Necessary changes are described in chapter 3 along with possible optimizations. Both extensions are implemented and evaluated on multiple real-world instances in chapter 4. Effectiveness of the resulting preprocessing algorithms is evaluated by comparing it to the standard Trip-Based preprocessing. The query running times of both new algorithms are compared to the relevant McRAPTOR variant and the optimizations suggested in chapter 3 are evaluated. Finally, it is analyzed for which sets of criteria the multicriteria Trip-Based approach is likely to be fast.

2. Preliminaries

This chapter defines basics needed in the rest of this work. First, public transit networks are formally defined. Second, problem statements for public transit routing and different multicriteria optimization problems are introduced. Finally, Trip-Based Public Transit Routing – the algorithm that will be extended in this work – as well as RAPTOR and its variant McRAPTOR that will be used for comparison, are described.

2.1. Public Transit Networks

The following definition of public transit networks combines those from [Wit15] and [SWZ20]. A *public transit network* is a 5-tuple $(\mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{F}, \mathcal{Z})$, consisting of a set of *stops* $\mathcal{S}$, a set of *trips* $\mathcal{T}$, a set of *lines*[1] $\mathcal{L}$, a set of *footpaths* $\mathcal{F} \subseteq \mathcal{S} \times \mathcal{S}$ between stops and a partition $\mathcal{Z} \subseteq \mathcal{P}(\mathcal{S})$ that partitions stops into *fare zones*. A stop $s \in \mathcal{S}$ models a place where passengers can board or disembark vehicles. This can be a train station, an individual platform, a bus stop etc.

Stops in $\mathcal{S}$ are partitioned into disjoint fare zones $\mathcal{Z} \subseteq \mathcal{P}(\mathcal{S})$. For use in the algorithms, each fare zone is associated with an ID. For stop $p \in \mathcal{S}$, $\text{fz}(p)$ is the ID of its fare zone. No further assumptions are made about fare zones.

Each footpath $(p, q) \in \mathcal{F}$ has an associated walking time $\tau_{\text{fp}}(p, q)$. For the algorithms regarded in this work, footpaths are required to be closed under transitivity: $(p, q) \in \mathcal{F}$ and $(q, s) \in \mathcal{F}$ implies $(p, s) \in \mathcal{F}$. Additionally, footpaths must fullfill the triangle inequation as defined by

$$\tau_{\text{fp}}(p, s) \leq \tau_{\text{fp}}(p, q) + \tau_{\text{fp}}(q, s).$$

All public transit networks used in this work use a *walking threshold* θ_{w} that filters all footpaths with walking time greater than θ_{w} before the footpaths are enclosed under transitivity. To simplify algorithms, a footpath $(p, p) \in \mathcal{F}$ exists for any stop $p \in \mathcal{S}$ with $\tau_{\text{fp}}(p, p) \coloneqq 0$.

A trip $t = \langle p_t^0, p_t^1, \dots \rangle \in \mathcal{T}$ models one ride of a vehicle (i.e., a train or bus) between a sequence of stops $p_t^0, p_t^1, \dots$ at specific times. Each trip has an arrival time $\tau_{\text{arr}}(t, p)$ and departure time $\tau_{\text{dep}}(t, p)$ at every stop $p \in \mathcal{S}$ it serves where

$$\tau_{\text{arr}}(t, p) \leq \tau_{\text{dep}}(t, p). \tag{2.1}$$

[1] *Line* is used instead of *route* to avoid confusion with the use of route as a journey in general routing algorithms [Wit15].

Additionally, arrival and departure times are defined for stop indices $i \in \{0, \dots, |t| - 1\}$:

$$\tau_{\text{arr}}(t, i) \coloneqq \tau_{\text{arr}}(t, p_t^i) \quad \text{and} \quad \tau_{\text{dep}}(t, i) \coloneqq \tau_{\text{dep}}(t, p_t^i)$$

Each stop $p \in \mathcal{S}$ has a *departure time buffer* $\tau_{\text{buf}}(p)$. It models the minimum amount of time that must be spent at p after arriving at p (with an initial footpath, a transfer or a trip) before the next trip can be boarded. For simplicity, departure time buffers are not explicitly considered in the algorithms in this work. Instead, departure times can be adapted to implicitly require the departure time buffer whenever necessary, defining

$$\tau_{\text{dep}}(t, p) \coloneqq \tau'_{\text{dep}}(t, p) - \tau_{\text{buf}}(p) \tag{2.2}$$

for original departure times $\tau'_{\text{dep}}(t, p)$ and any trip $t \in \mathcal{T}$ and stop $p \in \mathcal{S}$ therein in a preprocessing step. Modelling departure time buffers like this will be referred to as *implicit* departure time buffers [SWZ20, BBS+19]. Arrival times were previously required to be no later than departure times in inequation 2.1. Using implicit departure time buffers, this inequation does not necessarily hold anymore. For a trip $t \in \mathcal{T}$ and a stop $p \in \mathcal{S}$ therein, $\tau_{\text{buf}}(p) > \tau'_{\text{dep}}(t, p) - \tau_{\text{arr}}(t, p)$ will cause $\tau_{\text{dep}}(t, p) < \tau_{\text{arr}}(t, p)$. This is acceptable since departure times are only relevant when boarding a trip. Using implicit departure time buffers, change times do not have to be considered explicitly in this step. While riding a trip, departure times are irrelevant since the trip was already boarded. Only arrival times at stops are relevant to decide when a stop can be reached.

Trips are partitioned into lines $L \in \mathcal{L}$. Trips $t, u \in \mathcal{T}$ are part of the same line $L \in \mathcal{L}$ if they have the same stop sequence and can be totally ordered using

$$t \preceq u \iff \forall i \in \{0, \dots, |t| - 1\} : \tau_{\text{arr}}(t, i) \leq \tau_{\text{arr}}(u, i).$$

This also ensures that trips in the same line do not overtake each other. Trip t is strictly earlier than u if

$$t \prec u \iff t \preceq u \wedge \exists i \in \{0, \dots, |t| - 1\} : \tau_{\text{arr}}(t, i) < \tau_{\text{arr}}(u, i)$$

For a trip $t \in \mathcal{T}$, $L_t \in \mathcal{L}$ is its line. A line has the same stop sequence as its trips: $L_t = \langle p_t^0, \dots, p_t^{|t|-1} \rangle$. For a stop $p \in \mathcal{S}$, $L(p)$ is a set of tuples (L, i) of lines $L \in \mathcal{L}$ that serve p with stop index i.

$$L(p) = \{(L,\, i) \mid p = p_L^i \text{ for line } L \text{ and } L = \langle p_L^0,\, p_L^1,\, \dots \rangle\}$$

A trip segment $p_t^b \to p_t^e$ is the section of trip t from stop index b to e (inclusively). $p_t^i \mapsto p_u^j$ models a transfer where $(p_t^i, p_u^j) \in \mathcal{F}$ and

$$p_t^i \mapsto p_u^j \implies \tau_{\text{arr}}(t, i) + \tau_{\text{fp}}(p_t^i, p_u^j) \leq \tau_{\text{dep}}(u, j)$$

It is possible to transfer from one stop of a trip to another stop of the same trip ($u = t$).

A *journey* models a way on which a passenger can travel through the network. It consists of alternating trip segments and transfers, each called a *leg* of that journey. Each leg must have the same first stop as the final stop of the previous leg. Additional footpaths at the beginning and end of a journey are possible. In that case, the source stop and target stop of the journey must be defined separately.

2.2. Problem Statement

Public transit routing aims at answering *public transit queries*. To define the problem statement, Pareto dominance and Pareto sets are first introduced. Then, two basic queries, the *earliest arrival query* and the *bicriteria query* will be defined. Finally, *multicriteria* optimization will be described along with the query types considered in this work.

2.2.1. Pareto Dominance and Pareto Sets

For queries that optimize only one criterion, there is a unique optimal solution. In multicriteria optimization, this is no longer the case. The goal of multicriteria optimization is to compute a *Pareto set* of journeys [BDG+16]. A journey J is *dominated* by journey K with regards to a set of criteria if K is at least as good as J in each criterion and strictly better in at least one criterion. Arbitrary tie-breaking is used if journeys are equal in every criterion. A set of journeys $\mathcal{J}$ is a Pareto set if no journey $K \in \mathcal{J}$ dominates any other journey $J \in \mathcal{J}$ and all journeys that are not in $\mathcal{J}$ are dominated by journeys in $\mathcal{J}$. Dominated journeys are not relevant since there is another journey that is better in at least one criterion while it is no worse in any other. A journey J is called *Pareto optimal* if it is not dominated by any other journey.

2.2.2. Public Tranit Queries

A public transit query receives as input a source stop p_{src} and target stop p_{tgt} along with a departure time τ_{dep}. The objective of a query is to compute a Pareto set of valid journeys $\mathcal{J}$ with regard to the optimization criteria. A journey J is *valid* if it starts at p_{src}, ends at p_{tgt} and leaves p_{src} no earlier than τ_{dep} (either with an initial footpath or with the first trip). The *earliest arrival query* optimizes journeys for earliest arrival time as only criterion. Queries in public transit routing often optimize journeys for two criteria, *earliest arrival time* and *minimum number of trips*.[2] If two journeys have the same arrival time, passengers would generally choose the journey with the fewest transfers. Moreover, a journey that has a non-optimal arrival time might be attractive to passengers if it requires fewer transfers than the journey with the optimal arrival time. Technically, this is a multicriteria optimization problem. However, number of trips is a discrete criterion that only has few possible values for typical queries. Algorithms like RAPTOR [DPW15] and Trip-Based Routing [Wit15] take advantage of this. Instead of optimizing the number of trips explicitly, they implicitly consider them during the query: By finding journeys with an ascending number of trips, every previously found journey automatically has a better or equal number of trips than a new one. Therefore, the criterion must not be checked explicitly. The remaining problem of optimizing for the earliest arrival time can then be treated analogously to a unicriteria problem. In the following, the bicriteria *earliest arrival and minimum number of trips query* will simply be referred to as the *bicriteria query*.

In their simplest form, the algorithms described in this work do not compute full journey descriptions but only their properties for the optimization criteria. To correctly define the result set of a query, an *arrival label* for a journey J is a tupel that contains the value of the journey for each optimization criterion. For instance, for the bicriteria query, arrival labels are tuples (τ_{arr}, n) with arrival time τ_{arr} and number of trips n. The query algorithms in this work compute Pareto sets of arrival labels. Journey descriptions can be reconstructed in an additional step.

2.2.3. Multicriteria Optimization

This work focuses on optimizing public transit journeys for additional criteria besides earliest arrival time and number of trips. While the number of trips will still not be handled explicitly, at least two criteria remain. In consequence, the part of an algorithm that previously had to handle only one criterion, earliest arrival time, now has to optimize for at least two. This makes the algorithms more complex.

[2] Using minimum number of trips instead of minimum number of *transfers* allows to differentiate between a direct walking journey using footpath $(p_{\text{src}}, p_{\text{tgt}})$ (with 0 trips) and a journey using exactly one trip of the public transit network – which both have 0 transfers.

Multicriteria optimization will be studied using the additional criteria *minimum walking time* and *minimal fare zone subset*. These will be briefly described in the following. The resulting queries, simply called *minimum walking time query* and *minimal fare zone subset query*, both optimize for the *three* criteria earliest arrival time, minimum number of trips and minimum walking time or minimal fare zone subset, respectively.

Minimum Walking Time

Public transit journeys often include walking sections. Especially, the journey that consists solely of walking (if a footpath $(p_{\text{src}}, p_{\text{tgt}}) \in \mathcal{F}$ exists) is always Pareto optimal since it is the only journey that uses no trips. The walking time of a journey is defined as the sum of all walking times in the journey, i.e., initial and final footpaths as well as transfers. *Minimum walking time* takes walking sections into account and optimizes the walking time along a journey. A journey has minimum walking time if its overall walking time is minimal compared to that of other journeys. Time spent waiting at a stop during a transfer at this stop is not counted as walking time.

Minimal Fare Zone Subset

A common routing objective is to compute cheap journeys. In public transit routing, the price of a journey depends on different fare models that transportation companies use. Considering all possibilities in a general algorithm would make it very complex [BDG+16, MHS06]. Especially, the price can generally not be calculated as the sum of prices of journey segments it uses. To simplify this – and still give a reasonably accurate representation of fare models –, this work uses the *fare zone model* introduced in [DPW15]: Stops are partitioned into fare zones $\mathcal{Z}$. Each stop $p \in \mathcal{S}$ has an associated fare zone $\text{fz}(p)$. In a query, the subset of fare zones that a journey uses is minimized. This model is simple but allows to model many real transportation networks and their fare models with sufficient accuracy. If necessary, the actual price of a journey can be calculated in a (network-dependent) postprocessing-step [DPW15].

A journey uses the fare zone of a stop if it uses a vehicle at that stop. This can be by boarding or disembarking a vehicle at the stop or by passing through the stop in a vehicle. The fare zone of a stop is not used by walking to or from the stop.[3] Fare zone subsets are not totally ordered. In this work, fare zone subset f is defined to be better or strictly better than fare zone subset g by

$$f \leq g \iff f \subseteq g$$

and

$$f < g \iff f \subsetneq g$$

respectively. In all other cases, no comparison can be made. Especially, the cardinalities of fare zone subsets are not sufficient to order them. For instance, some fare zones might be grouped into one price group. Then, a fare zone subset with many fare zones of this group can still be better than another one that contains only two fare zones but from different groups.

2.3. RAPTOR

Delling, Pajor and Werneck proposed RAPTOR (*Round bAsed Public Transit Optimized Router*) in [DPW15]. In contrast to previous solutions, it does not model the timetable

[3] Since footpaths are enclosed under transitivity, this is only relevant for the source and target stop of a journey.

information as a graph. Instead, it operates directly on the timetable information and does not rely on preprocessing.

The basic RAPTOR algorithm answers bicriteria queries: Given a source stop p_{src}, target stop p_{tgt} and departure time τ_{dep}, it calculates a Pareto set of journeys from p_{src} to p_{tgt}, optimizing earliest arrival time and minimum number of trips. RAPTOR operates in *rounds* $k = 1, 2, \ldots$. In round k, it computes optimal arrival times for all stops reached by journeys that use k trips. In each round, RAPTOR explores lines serving stops whose arrival time was improved in the previous round. It scans outgoing footpaths for all stops whose arrival times improved in the current round. The algorithm stops if there is no stop whose arrival time was improved in the current round. This algorithm is described in pseudocode in Algorithm 2.1.

More precisely, RAPTOR holds earliest arrival times $\tau_k(p)$ for journeys using exactly k trips for each stop $p \in \mathcal{S}$ in round k. Additionally, $\tau^*(p)$ is the best arrival time for p seen so far, using journeys with any number of trips. Both are initialized to ∞. Initially, it sets $\tau_0(p_{\text{src}}) \leftarrow \tau_{\text{dep}}$, $\tau^*(p_{\text{src}}) \leftarrow \tau_{\text{dep}}$ and marks p_{src} as improved. The algorithm holds a queue Q of tuples (L, p) of lines $L \in \mathcal{L}$ that need to be scanned in the current round beginning at stop $p \in \mathcal{S}$. In round k, all stops $p \in \mathcal{S}$ that were marked in the previous round (or initially) are considered. For every line $L \in L(p)$, RAPTOR calculates the first marked stop q of L and inserts a tuple (L, q) into Q. All marked stops are then unmarked.

For each entry $(L, p) \in$ Q, RAPTOR traverses line L beginning at p. First, it selects the earliest trip t of L where $\tau_{\text{dep}}(t, p) \geq \tau_{k-1}(p)$ – the earliest trip that is reachable from p using the arrival time at p from the previous round. Then, it scans all stops p_L^i of L starting at p. If $\tau_{\text{arr}}(t, i) < \tau_k(p_L^i)$, the current trip improves the arrival time at p_L^i. Therefore, $\tau_k(p_L^i)$ is updated accordingly and p_L^i is marked. Two pruning techniques are used here to avoid unnecessary scans: $\tau_k(p_L^i)$ is only updated and p_L^i only marked if $\tau_{\text{arr}}(t, i) < \tau^*(p_{\text{tgt}})$ (*target pruning*) and $\tau_{\text{arr}}(t, i) < \tau^*(p_L^i)$ (*local pruning*) hold. Otherwise, either p_L^i has already been reached with a better arrival time in previous rounds or p_{tgt} has already been reached with a better arrival time. Since RAPTOR finds journeys in ascending order of their number of trips, this implies that each journey using p_L^i that could be found would be dominated, since the arrival time only increases in the remaining part of the journey. After this, RAPTOR updates t to the first trip that is reachable at p_L^i, i.e., the earliest trip of L such that $\tau_{\text{dep}}(t, i) \geq \tau_{k-1}(p_L^i)$.

Finally, for every stop p marked in the previous step, the arrival time at every stop q reachable from p using footpaths $(p, q) \in \mathcal{F}$ is updated via

$$\tau_k(q) \leftarrow \min(\tau_k(q), \tau_k(p) + \tau_{\text{fp}}(p, q)).$$

If the arrival time has been improved, q is marked for the next round. If no stops were marked in a round, no arrival times were improved in that round. Therefore, no improvement can be achieved in any further round and the algorithm stops.

The Pareto set of arrival labels $\mathcal{J}$ is calculated by iterating over all rounds $k = 0, 1, \ldots$ used before. The earliest arrival time τ_{min} at p_{tgt} seen so far is initialized to ∞. For each round k, if $\tau_k(p_{\text{tgt}}) < \tau_{\text{min}}$, this round reaches p_{tgt} with a better arrival time than all journeys previously seen and it uses k trips. Therefore, the arrival label $(\tau_k(p_{\text{tgt}}), k)$ is added to $\mathcal{J}$ and τ_{min} is updated. If instead $\tau_k(p_{\text{tgt}}) \geq \tau_{\text{min}}$, a journey with equal or better arrival time has been found in a previous iteration $m < k$. Since that journey uses $m < k$ trips, it dominates the journey that could be found in the current round.

To reconstruct journeys, additional pointers are added for each round and each stop. A pointer from stop p to stop q in round k indicates that stop p was reached from stop q at

the beginning of round k. For each round, the respective journey (if it exists) can be found by traversing pointers backwards from p_{tgt} until p_{src} is reached. After a pointer is used, it is changed to the previous round and the stop that was pointed to.

Algorithm 2.1: RAPTOR BICRITERIA QUERY [DPW15]

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{F}, \mathcal{Z})$, source stop $p_{\text{src}} \in \mathcal{S}$, target stop $p_{\text{tgt}} \in \mathcal{S}$, departure time τ_{dep}.

Data: Queue Q of tuples (L, p) of lines and stops that need to be scanned, earliest arrival times $\tau_i(\cdot)$ per round, best arrival times $\tau^*(\cdot)$ of all rounds.

Output: Pareto set of arrival labels $\mathcal{J}$.

for each i **do**
  $\tau_i(\cdot) \leftarrow \infty$
$\tau^*(\cdot) \leftarrow \infty$
$\tau_0(p_{\text{src}}) \leftarrow \tau_{\text{dep}}$
$\tau^*(p_{\text{src}}) \leftarrow \tau_{\text{dep}}$
if $(p_{\text{src}}, p_{\text{tgt}}) \in \mathcal{F}$ **then**
  $\tau_0(p_{\text{tgt}}) \leftarrow \tau_{\text{dep}} + \tau_{\text{fp}}(p_{\text{src}}, p_{\text{tgt}})$
  $\tau^*(p_{\text{tgt}}) \leftarrow \tau_{\text{dep}} + \tau_{\text{fp}}(p_{\text{src}}, p_{\text{tgt}})$
mark p_{src}
for each $k \leftarrow 1, 2, \ldots$ **do**
  clear Q
  for each *marked stop* p **do**
    for each $(L, i) \in L(p)$ **do**
      if $(L, p') \in \mathsf{Q}$ *for some stop* p' **then**
        substitute (L, p') by (L, p) in Q if p comes before p' in L
      else
        add (L, p) to Q
    unmark p
  for each $(L, p) \in \mathsf{Q}$ **do**
    $t \leftarrow \perp$
    for each *stop* p^i *of* L *beginning with* p **do**
      if $t \neq \perp \wedge \tau_{\text{arr}}(t, i) < \min(\tau^*(p^i), \tau^*(p_{\text{tgt}}))$ **then**
        $\tau_k(p^i) \leftarrow \tau_{\text{arr}}(t, i)$
        $\tau^*(p^i) \leftarrow \tau_{\text{arr}}(t, i)$
        mark p^i
      if $\tau_{k-1}(p^i) < \tau_{\text{dep}}(t, i)$ **then**
        $t \leftarrow$ earliest trip of L such that $\tau_{\text{dep}}(t, i) \geq \tau_{k-1}(p^i)$
  for each *marked stop* p **do**
    for each *stop* $q \in \mathcal{S}$ *such that* $(p, q) \in \mathcal{F}$ **do**
      if $\tau_k(p) + \tau_{\text{fp}}(p, q) < \tau_k(q)$ **then**
        $\tau_k(q) \leftarrow \tau_k(p) + \tau_{\text{fp}}(p, q)$
        mark q
  if *no stops are marked* **then**
    break
$\tau_{\min} \leftarrow \infty$
for each *round* $k = 0, 1, \ldots$ *used above* **do**
  if $\tau_k(p_{\text{tgt}}) < \tau_{\min}$ **then**
    $\tau_{\min} \leftarrow \tau_k(p_{\text{tgt}})$
    $\mathcal{J} \leftarrow \mathcal{J} \cup \{(\tau_{\min}, k)\}$

2.3.1. McRAPTOR

McRAPTOR (*More criteria RAPTOR*) is an extension of RAPTOR that can handle additional optimization criteria [DPW15]. Instead of storing the (unique) optimal arrival time $\tau_k(p)$ for a stop p in round k, it stores a Pareto set of labels. These Pareto sets will be referred to as *bags* $B_k(p)$. A *label* $\ell \in B_k(p)$ is a tuple with one value for each optimization criterion besides number of trips. Changes to the algorithm are needed during the line processing and footpath evaluation steps.

When iterating over the stops of a line L in round k, an initially empty *line bag* B_L is used to keep track of all non-dominated journeys with last trip in L using exactly k trips. For a label $\ell \in B_L$, this trip is stored as its *active trip* $t(\ell)$. At stop p_L^i, the arrival time of each label $\ell \in B_L$ is updated to the arrival time of its trip at p_L^i, $\tau_{\text{arr}}(t(\ell), i)$. Then, B_L is merged into $B_k(p_L^i)$, removing all dominated labels from $B_k(p_L^i)$ – this represents disembarking from the trip at p_L^i in round k. Finally, $B_{k-1}(p_L^i)$ is merged into B_L, removing dominated labels in B_L. Here, the respective active trip $t(\ell)$ is assigned to all new labels ℓ. This step represents the possibility of boarding the trip at p_L^i with all possible labels from round $k-1$.

To evaluate a footpath $(p, q) \in \mathcal{F}$ in round k, a copy of each label in $B_k(p)$ is created and its arrival time is incremented by $\tau_{\text{fp}}(p, q)$. This copy is then inserted into $B_k(q)$, removing dominated labels.

Local and target pruning can be implemented by keeping a *best bag* $B^*(p)$ for each stop p. A label ℓ is only added to $B_k(p)$ if it is dominated neither by $B^*(p)$, nor by $B^*(p_{\text{tgt}})$, with the same argument as for RAPTOR. If a label is added to $B_k(p)$, it is also added to $B^*(p)$ for effective pruning.

The algorithm described above is a generic variant of the algorithm. For a specific query type, the additional values of the labels – which depend on the criteria that must be optimized – need to be calculated during the algorithm. The necessary steps are hard to generalize for all criteria. Broadly, criteria that change while riding a vehicle (e.g. fare zones) need to be considered when iterating over stops of a line. Criteria that change during transfers (e.g. walking time) must be considered when evaluating footpaths. Necessary changes for the criteria that this work focuses on are presented briefly in the following.

Walking Time as Third Criterion

A bag for stop $p \in \mathcal{S}$ holds labels $\ell = (\tau_{\text{arr}}, \tau_{\text{w}})$ with arrival time τ_{arr} at and walking time τ_{w} to stop p. No additional changes are needed for the line evaluation step. When evaluating a footpath $(p, q) \in \mathcal{F}$, its duration $\tau_{\text{fp}}(p, q)$ must also be added to the walking time of the copied label before inserting it into $B_k(q)$.

Fare Zones as Third Criterion

In this case, a bag for stop $p \in \mathcal{S}$ holds labels $\ell = (\tau_{\text{arr}}, f)$ consisting of arrival time τ_{arr} and the subset of fare zones f that p can be reached with. When evaluating stop p_L^i along a line L, labels $\ell = (\tau_{\text{arr}}, f)$ are added into different bags. At each such operation, the fare zone $\text{fz}(p_L^i)$ of the stop must be added to the fare zone subset f of the label before inserting the label into the respective bag. No further changes are needed for footpath evaluation.

2.4. Trip-Based Public Transit Routing

This work extends the *Trip-Based Public Transit Routing Algorithm* first described by Sascha Witt in [Wit15]. It solves the bicriteria query, optimizing arrival time and number

of trips. To avoid confusion with the extensions that will be developed in chapter 3, it will be referred to as the *standard* Trip-Based algorithm. It operates on individual trips rather than lines. To efficiently evaluate possible transfers between trips, transfers are generated and reduced in a preprocessing step. In a query, stops are explored from the source stop in a breadth-first-search (BFS) manner. Here, the preprocessed transfers are used to change between trips. Both steps will be explained in detail in the following sections.

2.4.1. Preprocessing

First, all possible transfers are generated. This step is described in pseudocode in Algorithm 2.2. For each trip t and stop index $i > 0$ on t, all stops q reachable from p_t^i using a footpath $(p_t^i, q) \in \mathcal{F}$ are considered. For each $(L, j) \in L(q)$ – each line L serving q with index j – a transfer $p_t^i \mapsto p_u^j$ is possible to the first trip u of line L with $\tau_{\text{dep}}(u, j) \geq \tau_{\text{arr}}(t, i) + \tau_{\text{fp}}(p_t^i, q)$, if such a trip exists. An example can be found in Figure 2.1. Transfers are only generated if they connect to a different line $L \neq L_t$, or if they reach an earlier trip $u \prec t$ of the same line or an earlier stop $j < i$ along any trip. Transfers to later trips of the same line or later stops of the same trip are not necessary: A transfer to a later stop of the same trip can always be avoided since the stop can be reached by just continuing to ride the current trip. Any journey that could be found using a transfer from t to a later stop of a later trip $u \succeq t$ would be dominated by a journey that can be found by just continuing to ride t to the stop. Since $u \succeq t$, this journey can reach all stops at equal or better arrival times and it uses less transfers. No transfers are created for $i = 0$ since it is not necessary to transfer directly out of the first stop without riding the trip in a network with transitive footpaths.

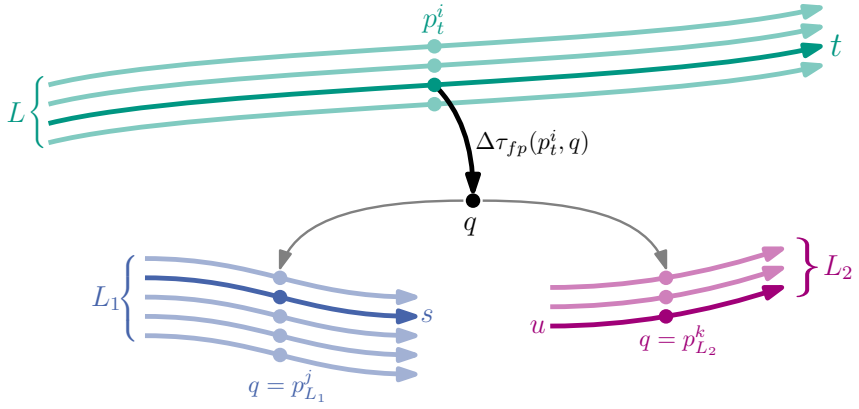


Figure 2.1.: For each trip t, each stop index i, each footpath (p_t^i, q) and each line serving q, a transfer is possible to its first reachable trip.

Two steps of transfer reduction follow. Algorithm 2.3 describes the first step in pseudocode. Here, transfers $p_t^i \mapsto p_u^j$ with $p_u^{j+1} = p_t^{i-1}$, so called *U-turn transfers*, are discarded if they fulfill

$$\tau_{\text{arr}}(t, i-1) \leq \tau_{\text{dep}}(u, j+1). \tag{2.3}$$

In that case, trip u can already be reached at stop p_t^{i-1} and all journeys reaching t at p_t^i can also reach p_t^{i-1}. Note that (2.3) may not always be true, although the existence of $p_t^i \mapsto p_u^j$ implies $\tau_{\text{arr}}(t, i) + \tau_{\text{fp}}(p_t^i, p_u^j) \leq \tau_{\text{dep}}(u, j)$. Stops p_u^j and p_u^{j+1} might have different departure time buffers $\tau_{buf}(\cdot)$ so that $\tau_{\text{dep}}(u, j+1) < \tau_{\text{dep}}(u, j)$ is possible for departure times with implicit departure time buffers that the algorithms use in this work (see equation 2.2 in Section 2.1). An example for U-turn transfers can be found in Figure 2.2.

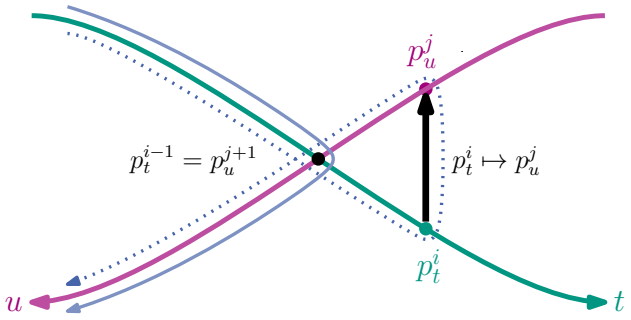


Figure 2.2.: A U-turn transfer $p_t^i \mapsto p_u^j$ with $p_u^{j+1} = p_t^{i-1}$ that fulfills the condition from (2.3) cannot be necessary since any journey using it (dotted blue) can always be reproduced without it (blue).

Algorithm 2.2: Original transfer computation [Wit15]

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{F}, \mathcal{Z})$.
Output: Original transfer set T.

$\mathsf{T} \leftarrow \emptyset$
for each *trip* $t \in \mathcal{T}$ **do**
  for each *stop* p_t^i *on trip* t *with* $i > 0$ **do**
    for each *stop* q *such that* $(p_t^i, q) \in \mathcal{F}$ **do**
      for each $(L, j) \in L(q)$ *with* $j < |L| - 1$ **do**
        $u \leftarrow$ earliest trip of line L with $\tau_{\text{dep}}(u, j) \geq \tau_{\text{arr}}(t, i) + \tau_{\text{fp}}(p_t^i, q)$
        if $L \neq L_t \vee u \prec t \vee j < i$ **then**
          $\mathsf{T} \leftarrow \mathsf{T} \cup \{p_t^i \mapsto p_u^j\}$

Algorithm 2.3: Removal of U-turn transfers [Wit15]

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{F}, \mathcal{Z})$, original transfer set T.
Output: Transfer set T.

for each *transfer* $p_t^i \mapsto p_u^j \in \mathsf{T}$ **do**
  if $p_t^{i-1} = p_u^{j+1} \wedge \tau_{\text{arr}}(t, i-1) \leq \tau_{\text{dep}}(u, j+1)$ **then**
    $\mathsf{T} \leftarrow \mathsf{T} \setminus \{p_t^i \mapsto p_u^j\}$

The remaining transfers are filtered in the second reduction step, see Algorithm 2.4 for a pseudocode description. Here, only transfers that improve arrival times at some stop are kept. The algorithm iterates over all trips $t \in \mathcal{T}$ and scans stops backwards along t. It keeps minimum tentative arrival times $\tau_A(\cdot)$ for each stop, initialized to ∞ for each new trip. For stop index $i > 0$, τ_A is first updated for all stops q reachable via footpaths $(p_t^i, q) \in \mathcal{F}$ (which includes p_t^i) using

$$\tau_A(q) = \min\left(\tau_A(q),\, \tau_{\text{arr}}(t, i) + \tau_{\text{fp}}(p_t^i, q)\right).$$

For each transfer $p_t^i \mapsto p_u^j$ from p_t^i, it is checked if the transfer leads to better arrival times along trip u. For that, it is checked if arrival times can be improved at any stop q reachable from p_u^k for $k > j$ via footpaths $(p_u^k, q) \in \mathcal{F}$ (again, including p_u^k) using

$$\tau_A(q) = \min\left(\tau_A(q),\, \tau_{\text{arr}}(u, k) + \tau_{\text{fp}}(p_u^k, q)\right).$$

If the transfer does not improve arrival times at any of these stops, it is discarded. Scanning $k = j$ cannot improve the arrival time since the arrival time has already been minimized by exploring footpaths out of p_t^i. The first stop p_t^0 along t does not need to be scanned since there are no outgoing transfers from there and no further transfers for any other stops will be checked afterwards. All preprocessing steps can be trivially parallelized on the trip-level.

Algorithm 2.4: Standard Trip-Based transfer reduction [Wit15]

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{F}, \mathcal{Z})$, original transfer set T without unnecessary U-turn transfers.
Data: Earliest arrival times $\tau_A(\cdot)$.
Output: Reduced transfer set T.

for each *trip* $t \in \mathcal{T}$ **do**
  $\tau_A(\cdot) \leftarrow \infty$
  for $i \leftarrow |t| - 1, \ldots, 1$ **do**
    for each *stop* q *with* $(p_t^i, q) \in \mathcal{F}$ **do**
      $\tau_A(q) \leftarrow \mathsf{min}\left(\tau_A(q),\, \tau_{\text{arr}}(t, i) + \tau_{\text{fp}}(p_t^i, q)\right)$
    for each *transfer* $p_t^i \mapsto p_u^j \in \mathsf{T}$ **do**
      $\mathsf{keep} \leftarrow$ false
      for each *stop* p_u^k *on trip* u *with* $k > j$ **do**
        for each *stop* q *with* $(p_u^k, q) \in \mathcal{F}$ **do**
          $\mathsf{keep} \leftarrow \mathsf{keep} \vee \tau_{\text{arr}}(u, k) + \tau_{\text{fp}}(p_u^k, q) < \tau_A(q)$
          $\tau_A(q) \leftarrow \mathsf{min}\left(\tau_A(q),\, \tau_{\text{arr}}(u, k) + \tau_{\text{fp}}(p_u^k, q)\right)$
      if $\neg\mathsf{keep}$ **then**
        $\mathsf{T} \leftarrow \mathsf{T} \setminus \{p_t^i \mapsto p_u^j\}$

The transfer reduction does not claim to produce a minimal set of transfers: A transfer can be kept although it is not necessary for any Pareto optimal journey. A transfer is unnecessary if for any journey section using it, there is another, equal or better, journey section between the same stops without the transfer. The preprocessing algorithm explores *some* of these alternative sections. If the relevant alternative section is not among those explored by the preprocessing algorithm, an unnecessary transfer can be kept. A transfer is *necessary* for a journey J if neither J nor any alternative journey that dominates J can be found without the transfer. It can now be proven that no transfer that is necessary for any Pareto optimal journey is removed by the transfer reduction.

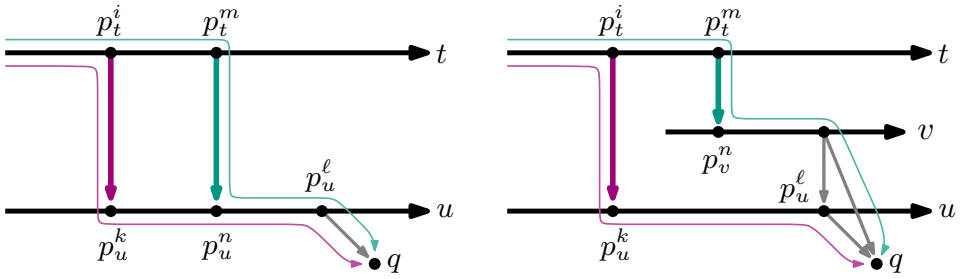
Theorem 2.1. Standard Trip-Based transfer reduction *(Algorithm 2.4) does not remove any transfer that is necessary for a Pareto optimal journey.*

Proof. By contradiction. An illustration can be found in Figure 2.3. Let $p_t^i \mapsto p_u^k$ be a transfer necessary for a Pareto optimal journey J – i.e., J cannot be found without the transfer. Assume that $p_t^i \mapsto p_u^k$ is removed by Algorithm 2.4.

Then, at each stop p_u^ℓ for $\ell > k$, the tentative arrival time $\tau_A(p_u^\ell)$ is already at least equally good as the one that can be achieved using the transfer, $\tau_{\text{arr}}(u, \ell)$. Similarly, at all stops q reachable from p_u^ℓ using a footpath $(p_u^\ell, q) \in \mathcal{F}$, the tentative arrival time $\tau_A(q)$ is at least equally good as $\tau_{\text{arr}}(u, \ell) + \tau_{\text{fp}}(p_u^\ell, q)$. Hence, by design of the algorithm, p_u^ℓ or q has been reached with a non-later arrival time $\tau_A(p_u^k)$ using another transfer $p_t^m \mapsto p_v^n$ from p_t^m, $m \geq i$ that was considered before and a possible footpath from stops after p_v^n. Stop

p_t^m can be reached from p_t^i by riding t. Trip v can be u or any other trip of any line. In each case, the partial journey J' from p_t^i to p_u^ℓ or q that uses $p_t^m \mapsto p_v^n$ is no worse than the one using $p_t^i \mapsto p_u^k$.

Because the arrival time at p_u^ℓ or q is no worse, all following trips in J can be reached using the alternative journey segment J' instead of $p_t^i \mapsto p_u^k$. Since using $p_t^i \mapsto p_u^k$ implies that trip t and trip u would be used, replacing the section with J' (using t and v) does not increase the number of used trips. Therefore, J can also be found without $p_t^i \mapsto p_u^k$, which contradicts the assumption. □



(a) Alternative transfer to $v = u$

(b) Alternative transfer using another trip v

Figure 2.3.: If transfer $p_t^i \mapsto p_u^j$ is dominated at a stop p_u^ℓ or q (lilac), another, non-worse way of reaching p_u^ℓ or q from p_t^i can be found (green). The alternative can use a transfer to the same trip (2.3a) or to another trip (2.3b). In each case, the arrival time at p_u^ℓ or q and the number of used trips obtained with the alternative is no worse than using $p_t^i \mapsto p_u^j$.

2.4.2. Standard Trip-Based Bicriteria Query

Given a source stop p_{src}, target stop p_{tgt} and a departure time τ_{dep}, the bicriteria query computes the Pareto set of arrival labels (τ_{arr}, n) for journeys leaving p_{src} no earlier than τ_{dep} and connecting to p_{tgt} with arrival time τ_{arr} and n used trips. The standard Trip-Based bicriteria query works analogously to a BFS starting from p_{src} on a graph with trips as vertices and transfers as edges. The query is described in pseudocode in Algorithm 2.5.

It stores the first stop index $\mathsf{R}(t)$ that has already been reached for each trip t, initialized to ∞. Trip segments that can be reached with a journey with n trips are stored in queues Q_n, $n = 0, 1, \ldots$. To efficiently evaluate if p_{tgt} can be reached from a trip segment, a set $\mathcal{L}_{\text{tgt}}$ of tuples $(L, i, \Delta\tau)$ is computed. This indicates that p_{tgt} can be reached from line L at stop index i using a footpath with duration $\Delta\tau$:

$$\mathcal{L}_{\text{tgt}} = \{(L, i, \tau_{\text{fp}}(q, p_{\text{tgt}})) \mid (q, p_{\text{tgt}}) \in \mathcal{F} \wedge (L, i) \in L(q)\}$$

The algorithm starts by collecting the trips that are reachable from stops q reachable from p_{src} using footpaths $(p_{\text{src}}, q) \in \mathcal{F}$ starting at τ_{dep}. For each such stop q, it considers the earliest trip t of each line L, where $(L, i) \in L(q)$, that fulfills

$$\tau_{\text{dep}}(t, i) \geq \tau_{\text{dep}} + \tau_{\text{fp}}(p_{\text{src}}, q).$$

If $i < \mathsf{R}(t)$, the trip has been reached at a new earliest stop and the trip segment $p_t^i \mapsto p_t^{\mathsf{R}(t)}$ is added to queue Q_0. The trip segment ends at stop index $\mathsf{R}(t)$ since all stops after that are

covered by trip segments that have been enqueued before. The reached index is updated by $\mathsf{R}(u) \leftarrow \min(\mathsf{R}(u), i)$ for all trips u with $u \succeq t \wedge L_u = L_t$, including t. This marks later trips of the same line as reached at that stop to avoid redundant scanning. Due to $u \succeq t$, trip u reaches all its stops no earlier than t and scanning it cannot lead to any improvement.

Starting with the trip segments in Q_0, trip segments in Q_n for $n = 0, 1, \ldots$ are explored until the next queue is empty. The query maintains the minimum arrival time $\tau_{\min}$ that p_{tgt} could be reached with so far. For a trip segment $p_t^b \to p_t^e$, it is first checked if p_{tgt} is reachable via a footpath. A tuple $(L_t, i, \Delta\tau) \in \mathcal{L}_{\text{tgt}}$ with $b < i \leq e$ indicates that p_{tgt} is reachable from p_t^i with walking time $\Delta\tau$. If $\tau_{\text{arr}}(t, i) + \Delta\tau < \tau_{\min}$, this yields a new non-dominated arrival label $(\tau_{\text{arr}}(t, i) + \Delta\tau, n + 1)$. The label is added to the result set, dominated labels are removed and $\tau_{\min}$ is updated. Again, stop index $i = b$ does not need to be scanned since a potential footpath to p_{tgt} would have already been found when scanning the trip segment that $p_t^b \to p_t^e$ was found from. If $\tau_{\text{arr}}(t, i) + \Delta\tau \geq \tau_{\min}$, the arrival time at p_{tgt} is no better than the best one found so far. Since all journeys that have previously been found use no more trips than the journey in question, the journey cannot be Pareto optimal and must not be added.

In the second step, outgoing transfers from the trip segment $p_t^b \to p_t^e$ are scanned. If $\tau_{\text{arr}}(t, b + 1) \geq \tau_{\min}$, the trip segment is pruned. Since the query considers journeys in ascending order of their number of trips, all previous journeys have no more than the current number of trips. Therefore, if the arrival time at index $b + 1$ is already dominated by the optimal arrival time that was previously found, journeys that can be found using the trip segment would always be dominated. If the trip segment was not pruned, each transfer $p_t^i \mapsto p_u^j$ from the precomputed set of transfers T where $b < i \leq e$ is scanned. If $j < \mathsf{R}(u)$, the transfer reaches u at a new earliest stop and a trip segment $p_u^j \to p_u^{\mathsf{R}(u)}$ is enqueued into Q_{n+1}. The trip segment ends at $\mathsf{R}(u)$ since all stops after that have already been scanned or are part of other enqueued trip segments. The reached indices $\mathsf{R}(v)$ for later trips v in the same line as u are updated using $\mathsf{R}(v) \leftarrow \min(\mathsf{R}(v), j)$. If $j \geq \mathsf{R}(u)$, trip u or an earlier trip of the same line has already been reached at stop index j and the trip segment of u does not need to be scanned. Transfers from p_t^b do not need to be considered because transfers – as footpaths – are enclosed under transitivity and any optimal journey reachable with such transfers would already be found from the stop that the trip segment $p_t^b \to p_t^e$ was enqueued from.

Journeys can be reconstructed by traversing the used trip segments backwards. For that, each trip segment points to the trip segment it was enqueued from. The queues are replaced by an array and all entries are kept until the end of the query. Then, trip segments can be traversed backwards from the last trip segment of each journey. To construct the necessary transfers, each trip segment keeps the transfer it was reached with (collectively, these also indicates the actual stops of each trip segment that were used for the journey). Final footpaths must by stored separatly since p_{tgt} might be reached from different stops of one trip segment.

Algorithm 2.5: STANDARD TRIP-BASED BICRITERIA QUERY [Wit15]

Input: Public transit network ($\mathcal{S}$, $\mathcal{T}$, $\mathcal{L}$, $\mathcal{F}$, $\mathcal{Z}$), reduced transfer set T, source stop p_{src}, target stop p_{tgt}, departure time τ_{dep}.
Data: Reached index $\mathsf{R}(\cdot)$, lines reaching p_{tgt} $\mathcal{L}_{\text{tgt}}$, trip segment queues Q_n.
Output: Pareto set of arrival labels $\mathcal{J}$.

$\mathcal{J} \leftarrow \emptyset$
$\mathcal{L}_{\text{tgt}} \leftarrow \emptyset$
$\mathsf{Q}_n \leftarrow \emptyset$ for $n = 0, 1, \ldots$
$\mathsf{R}(t) \leftarrow \infty$ for all trips t
for each *stop* q *with* $(q, p_{\text{tgt}}) \in \mathcal{F}$ **do**
  for each $(L,\, i) \in L(q)$ **do**
    $\mathcal{L}_{\text{tgt}} \leftarrow \mathcal{L}_{\text{tgt}} \cup \{(L,\, i,\, \tau_{\text{fp}}(q, p_{\text{tgt}}))\}$
for each *stop* q *with* $(p_{\text{src}}, q) \in \mathcal{F}$ **do**
  for each $(L,\, i) \in L(q)$ **do**
    $t \leftarrow$ earliest trip of L such that $\tau_{\text{dep}}(t, i) \geq \tau_{\text{dep}} + \tau_{\text{fp}}(p_{\text{src}}, q)$
    ENQUEUE($t,\, i,\, 0$)
  if $q = p_{\text{tgt}}$ **then**
    $\mathcal{J} \leftarrow \mathcal{J} \cup \{(\tau_{\text{dep}} + \tau_{\text{fp}}(p_{\text{src}}, p_{\text{tgt}}),\, 0)\}$
$\tau_{\text{min}} \leftarrow \infty$
$n \leftarrow 0$
while $\mathsf{Q}_n \neq \emptyset$ **do**
  for each *trip segment* $p_t^b \to p_t^e \in \mathsf{Q}_n$ **do**
    for each $(L_t,\, i,\, \Delta\tau) \in \mathcal{L}_{\text{tgt}}$ *with* $b < i \leq e$ *and* $\tau_{\text{arr}}(t, i) + \Delta\tau < \tau_{\text{min}}$ **do**
      $\tau_{\text{min}} \leftarrow \tau_{\text{arr}}(t, i) + \Delta\tau$
      $\mathcal{J} \leftarrow \mathcal{J} \cup \{(\tau_{\text{min}},\, n+1)\}$, removing dominated entries
    if $\tau_{\text{arr}}(t, b+1) < \tau_{\text{min}}$ **then**
      for each *transfer* $p_t^i \mapsto p_u^j \in \mathsf{T}$ *with* $b < i \leq e$ **do**
        ENQUEUE($u,\, j,\, n+1$)
  $n \leftarrow n + 1$

Procedure ENQUEUE(*trip* t, *index* i, *number of trips* n)
  if $i < \mathsf{R}(t)$ **then**
    $\mathsf{Q}_n \leftarrow \mathsf{Q}_n \cup \{p_t^i \to p_t^{\mathsf{R}(t)}\}$
    for each *trip* u *with* $u \succeq t \wedge L_t = L_u$ **do**
      $\mathsf{R}(u) \leftarrow \mathsf{min}(\mathsf{R}(u),\ i)$

3. Multicriteria Trip-Based Public Transit Routing Algorithm

The original Trip-Based Public Transit Routing Algorithm optimizes two criteria: earliest arrival time and number of transfers [Wit15]. This work aims at extending the algorithm to optimize more criteria. Specifically, two extended versions will be described. One will additionally minimize walking times, the second will additionally minimize the subset of used fare zones.

Trip-Based Routing is a two-stage algorithm. First, transfers are computed and reduced in a preprocessing step. Then – using the reduced set of transfers – queries can be evaluated efficiently. The performance of queries benefits from the effective transfer reduction described in Algorithm 2.4. With an additional criterion, more journeys can become Pareto optimal. For instance, a journey that was previously dominated by another one that had an earlier arrival time can now become relevant if it beats the other journey in walking time or fare zones. Therefore, more transfers are relevant for the query phase. To ensure correctness, none of these transfers may be discarded in the preprocessing-phase. Hence, the extended algorithms must use adjusted versions of the preprocessing algorithms that allow to *prove* that no necessary transfer is discarded. Besides that, preprocessing should still be reasonably fast. On the other hand, it is crucial for the query performance to discard as many transfers as possible. The extensions developed in this work will be described in the following sections along with proofs of correctness and discussions of possible improvements.

3.1. Trip-Based Routing for Minimum Walking Times

Both the preprocessing and the query phase of the Trip-Based algorithm have to be adapted for the minimum walking time query. The resulting algorithms are described in the following sections. Possible optimizations for the query algorithm are presented.

3.1.1. Preprocessing

Standard Trip-Based preprocessing (Section 2.4.1) is divided into three steps: original transfer computation, removal of U-turn transfers and general transfer reduction. The original transfer computation needs no changes: Even considering walking times as a criterion, transferring to another trip of the same line can still only be needed if the transfer reaches an earlier trip or an earlier stop. Likewise, U-turn transfers can be handled as before. Specifically, avoiding U-turn transfers (and instead transferring at a previous joint

stop of two trips) even reduces walking times. Therefore, no U-turn transfer that was not needed for the bicriteria query can be necessary for an optimal journey.

Considering only minimum arrival times at stops is not sufficient to ensure that the transfer reduction step does not discard any necessary transfer. A transfer might lead to a worse arrival time but a better walking time at a stop. This would be disregarded if only arrival times are used, causing the potentially necessary transfer to be discarded. Therefore, a full Pareto set $\mathsf{S}(p)$ of labels $\ell = (\tau_{\text{arr}}, \tau_{\text{walk}})$, tuples with arrival time τ_{arr} and walking time τ_{walk}, must be stored for each stop p. The original reduction algorithm (Algorithm 2.4) checks if new arrival times are earlier than previously seen ones. In this case, it overwrites the old arrival time by the new, earlier one. For the extended version, there is no total order on labels $(\tau_{\text{arr}}, \tau_{\text{walk}})$. Therefore, the "is smaller" check must be replaced by a "is not dominated" check. Instead of overwriting values, new labels must be added to $\mathsf{S}(p)$ and dominated labels therein must be removed.

Storing a full Pareto set $\mathsf{S}(p)$ is less efficient than storing a single arrival time $\tau_A(p)$. There is no longer one optimal value and dominance checks and insertions take considerably more time than simply comparing and overwriting values. However, this loss in efficiency cannot be avoided. First, walking times must be considered as described above. Total orderings could be restored by considering arrival times and walking times separately from each other, using arrival time $\tau_A(p)$ and walking time $\tau_W(p)$ for each stop $p \in \mathcal{S}$. Yet, this is not sufficient to identify all necessary transfers. Figure 3.1 illustrates the example presented here. Consider a transfer $p_t^i \mapsto p_u^j$ that is necessary because it has a Pareto optimal arrival time $\tau_{\text{arr}}(u, j+1) = 10{:}25$ and walking time $\tau_w^j = \tau_{\text{fp}}(p_t^i, p_u^j) = 5\,\text{min}$ at stop p_u^{j+1}. Assume now that a transfer $p_t^{i+1} \mapsto p_v^k$ as well as a footpath $(p_v^{k+1}, p_u^{j+1}) \in \mathcal{F}$ exists. Let both walking times be $\tau_{\text{fp}}(p_t^{i+1}, p_v^k) = \tau_{\text{fp}}(p_v^{k+1}, p_u^{j+1}) = 1\,\text{min}$. Assume that $\tau_{\text{arr}}(v, k+1) = 10{:}30$. Then, the alternative journey J_1 using this transfer reaches p_u^{j+1} with arrival time 10:31 and walking time 2 min. Assume that another transfer $p_t^{i+1} \mapsto p_w^l$ exists, again with a footpath $(p_w^{l+1}, p_u^{j+1}) \in \mathcal{F}$. Let both walking times now be $\tau_{\text{fp}}(p_t^{i+1}, p_w^l) = \tau_{\text{fp}}(p_w^{l+1}, p_u^{j+1}) = 5\,\text{min}$ and let $\tau_{\text{arr}}(w, l+1) = 10{:}16$. Then, the second alternative journey J_2 using this transfer reaches p_u^{j+1} with arrival time 10:21 and walking time 10 min. Both transfers originate at a later stop along t. Therefore they are processed before $p_t^i \mapsto p_u^j$. While scanning them, journeys J_1 and J_2 are explored. Since labels for arrival time and walking time are kept separately, J_1 minimizes $\tau_W(p_u^{j+1})$ to 2 min and J_2 minimizes $\tau_A(p_u^{j+1})$ to 10:21. When it is checked if $p_t^i \mapsto p_u^j$ is necessary at p_u^{j+1}, both its arrival time $\tau_{\text{arr}}(u, j+1) = 10{:}25$ and walking time $\tau_w^j = 5\,\text{min}$ are individually worse than the values in $\tau_A(p_u^{j+1})$ and $\tau_W(p_u^{j+1})$, respectively. In that case, $p_t^i \mapsto p_u^j$ will be discarded although it is necessary for a Pareto optimal journey.

Using the Pareto sets $\mathsf{S}(\cdot)$, the extended reduction algorithm can follow the structure of the original version. The reduction algorithm is described in pseudocode in Algorithm 3.1. Stops are explored backwards along each trip t. For each stop q reachable from stop p_t^i along the trip using a footpath $(p_t^i, q) \in \mathcal{F}$, the label

$$(\tau_{\text{arr}}(t, i) + \tau_{\text{fp}}(p_t^i, q),\ \tau_{\text{fp}}(p_t^i, q))$$

is added to $\mathsf{S}(q)$, removing dominated entries. This indicates that it is possible to reach q at $\tau_{\text{arr}}(t, i) + \tau_{\text{fp}}(p_t^i, q)$ and with walking time $\tau_{\text{fp}}(p_t^i, q)$. Then, for each transfer $p_t^i \mapsto p_u^j$ it is checked if it produces any non-dominated label at stops along or reachable from trip u after stop index j. Each stop q that is reachable from p_u^k, $k > j$ using a footpath $(p_u^k, q) \in \mathcal{F}$, can be reached from p_t^i with arrival time $\tau_{\text{arr}}(u, k) + \tau_{\text{fp}}(p_u^k, q)$ and walking time $\tau_{\text{fp}}(p_t^i, p_u^j) + \tau_{\text{fp}}(p_u^k, q)$. Hence, if the label

$$(\tau_{\text{arr}}(u, k) + \tau_{\text{fp}}(p_u^k, q),\ \tau_{\text{fp}}(p_t^i, p_u^j) + \tau_{\text{fp}}(p_u^k, q)) \tag{3.1}$$

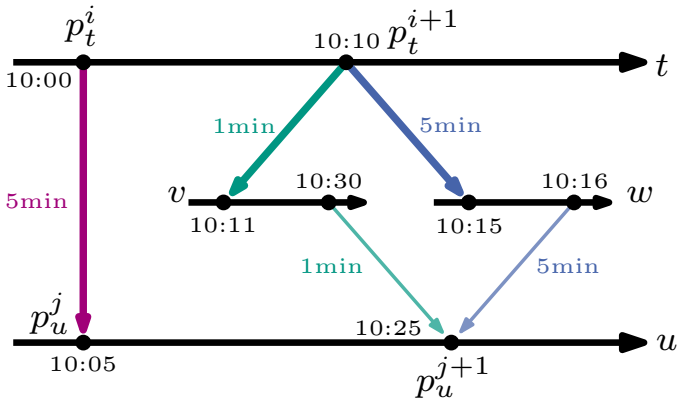


Figure 3.1.: Keeping arrival and walking times separately leads to incorrect transfer removals. Walking times are displayed next to transfers, departure times below stops along trips and arrival times above stops along trips. Transfer $p_t^i \mapsto p_u^j$ (lilac) reaches p_u^{j+1} with Pareto optimal arrival time 10:25 and walking time 5 min. Previously, $p_t^{i+1} \mapsto p_v^k$ (green) and $p_t^{i+1} \mapsto p_w^l$ (blue) were scanned. $p_t^{i+1} \mapsto p_v^k$ minimized $\tau_W(p_u^{j+1})$ to 2 min while $p_t^{i+1} \mapsto p_w^l$ minimized $\tau_A(p_u^{j+1})$ to 10:21. Hence, $p_t^i \mapsto p_u^j$ is discarded, although no other transfer dominates it at p_u^{j+1} in both arrival and walking time.

is not dominated by $\mathsf{S}(q)$, it is added to $\mathsf{S}(q)$ and the transfer will be kept. Dominated labels are removed. It can now be proven that the preprocessing is correct.

Algorithm 3.1: Walking Trip-Based transfer reduction

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{F}, \mathcal{Z})$, original transfer set T.
Data: Pareto sets $\mathsf{S}(\cdot)$ of labels $(\tau_{\text{arr}}, \tau_{\text{walk}})$ per stop.
Output: Reduced transfer set T.

```
1  for each trip t ∈ 𝒯 do
2      S(·) ← ∅
3      for i ← |t| − 1, ..., 1 do
4          for each stop q such that (p_t^i, q) ∈ ℱ do
5              S(q) ← S(q) ∪ {(τ_arr(t, i) + τ_fp(p_t^i, q), τ_fp(p_t^i, q))}, removing dominated
                 entries
6          for each transfer p_t^i ↦ p_u^j ∈ T do
7              keep ← false
8              for each stop p_u^k on trip u with k > j do
9                  for each stop q such that (p_u^k, q) ∈ ℱ do
10                     (τ_a, τ_w) ← (τ_arr(u, k) + τ_fp(p_u^k, q), τ_fp(p_t^i, p_u^j) + τ_fp(p_u^k, q))
11                     if (τ_a, τ_w) is not dominated by S(q) then
12                         S(q) ← S(q) ∪ {(τ_a, τ_w)}, removing dominated entries
13                         keep ← true
14             if ¬keep then
15                 T ← T \ {p_t^i ↦ p_u^j}
```

Theorem 3.1. WALKING TRIP-BASED TRANSFER REDUCTION *(Algorithm 3.1) does not remove any transfer that is necessary for a Pareto optimal journey.*

Proof. By contradiction. Let $p_t^i \mapsto p_u^j$ be a transfer necessary for a Pareto optimal journey J_{complete}, i.e., neither J_{complete} nor any other, equally good, journey can be found without the transfer. Assume that $p_t^i \mapsto p_u^j$ is removed by Algorithm 3.1. To check if $p_t^i \mapsto p_u^j$ is necessary, the algorithm scans journeys J from p_t^i to each stop q reachable using a footpath $(p_u^k, q) \in \mathcal{F}$ from a stop p_u^k for $k > j$ along the reached trip. At each such stop, it is checked if the label $(\tau_{\text{arr}}, \tau_{\text{walk}})$ as defined in (3.1) is not dominated by $\mathsf{S}(q)$. The label $(\tau_{\text{arr}}, \tau_{\text{walk}})$ considers the arrival times along trip u as well as walking times for the transfer and a potential following footpath. If the transfer is removed, it has not produced a non-dominated label at any of these stops.

Thus, at each stop q as defined above, there is a label $(\tau_{\text{arr}}^1, \tau_{\text{walk}}^1)$ that dominates $(\tau_{\text{arr}}, \tau_{\text{walk}})$. Since Pareto sets are reset for trip t and stops are scanned backwards along t, this label must have been produced by a journey J^1 from p_t^ℓ for $\ell \geq i$ using an outgoing transfer $p_t^\ell \mapsto p_v^k$ and a potential footpath from a stop following p_v^k. The alternative journey J^1 starts at p_t^ℓ and ends at q. Arrival times only increase along trips and walking times do not change along trips. Therefore J^1 can be extended to a journey J^2 from p_t^i to q by travelling from p_t^i to p_t^ℓ along t. The label $(\tau_{\text{arr}}^1, \tau_{\text{walk}}^1)$ remains valid for J^2 at q. Therefore, the dominating label $(\tau_{\text{arr}}^1, \tau_{\text{walk}}^1)$ at q implies that there is an alternative journey J^2 from p_t^i to q which dominates journey J in arrival and walking time.

The original trip section J uses trips t and u. Similarly, J^2 uses at most trips t and v. Hence, the number of trips does not change when replacing J by J^2. Since J^2 corresponds to a dominating label at q, J can be replaced by J^2 in J_{complete}. The journey section after J in J_{complete} can still be reached using J^2. Since J^2 does not need $p_t^i \mapsto p_u^j$, an alternative journey that reaches an equally good arrival label as J_{complete} can also be found without $p_t^i \mapsto p_u^j$ which contradicts the assumption. □

3.1.2. Query

The query receives as input a source stop p_{src} and a target stop p_{tgt} along with a departure time τ_{dep}. It computes a Pareto set $\mathcal{J}$ of arrival labels $(\tau_{\text{arr}}, n, \tau_{\text{walk}})$ which indicate that p_{tgt} can be reached at arrival time τ_{arr} with walking time τ_{walk} using a journey with n trips that leaves p_{src} no earlier than τ_{dep}. Similar to the standard Trip-Based earliest arrival query, the Walking Trip-Based query explores trips starting from the source stop p_{src}, scanning stops along reached trips and using the precomputed transfers to change between trips. As opposed to the original variant, it is no longer sufficient to just store one index $\mathsf{R}(t)$ per trip t containing the first reached stop along the trip. Different stops along a trip might be reachable with different minimum walking times. The original variant would prune parts of the search that reach a stop that was already reached before. However, the same stop can be reached again with a better walking time compared to the walking time of previous scans in the same query. Therefore, for each stop in each trip, the query must know the minimum tentative walking time needed to reach the stop with that trip. Minimum tentative walking times will be stored in a new data structure W. Possible implementations for W will be discussed at the end of this section. Generally, it supports two operations: The current minimum walking time for stop index i of trip t can be retrieved using $\mathsf{W}.\text{GET}(t, i)$. For a trip t, first stop index i and walking time τ, $\mathsf{W}.\text{UPDATE}(t, i, \tau)$ updates the walking time for all stop indices $j \geq i$ along t where $\tau < \mathsf{W}.\text{GET}(t, j)$ to τ. Walking times $\mathsf{W}.\text{GET}(t, \cdot)$ along a trip t are monotonically decreasing: Since travelling along a trip does not change the walking time, a walking time τ that can be achieved at stop index i can also be achieved at all stop indices $j > i$. The data structure stores optimal walking times. Therefore, walking

times can only change between subsequent stops if the later stop is reachable with a better walking time than its predecessor. An example can be found in Figure 3.2.

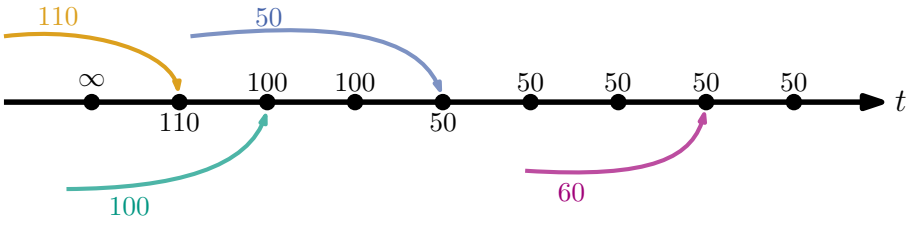


Figure 3.2.: A trip t and transfers reaching stops along t with different walking times. Minimum walking times are noted along the trip. When no transfer reaches a stop, the walking time is ∞ (left of the yellow transfer). Every new minimum walking time is stored for each following stop (yellow, green and blue transfers). Non-optimal walking times do not change the walking times stored for following stops (lilac transfer).

Algorithm 3.2 describes the query algorithm in pseudocode. Like the reached index R in the original query, W is initialized to ∞ for all entries. The first two steps of the query are similar to the original: Every line L that can reach p_{tgt} from stop index i with walking time $\Delta\tau$ is stored as a tuple $(L, i, \Delta\tau)$ in $\mathcal{L}_{\text{tgt}}$. For all stops q reachable from p_{src} using a footpath with walking time $\tau_{\text{fp}}(p_{\text{src}}, q)$, the earliest trip t of every line L with $(L, i) \in L(q)$ that is reachable as defined by

$$\tau_{\text{dep}}(t, i) \geq \tau_{\text{dep}} + \tau_{\text{fp}}(p_{\text{src}}, q)$$

is selected. For each such trip, it is checked if a new trip segment must be enqueued. If $\tau_{\text{fp}}(p_{\text{src}}, q) < \mathsf{W}.\textsc{get}(t, i)$, p_t^i is reached with a new optimal walking time. The trip segment $p_t^i \to p_t^j$ is added to Q_0, where the end index j is the last index where $\tau_{\text{fp}}(p_{\text{src}}, q) < \mathsf{W}.\textsc{get}(t, j)$ holds. All later stops have already been reached with an equal or better walking time. They have been enqueued in other trip segments and therefore must not be enqueued again. For each trip $u \succeq t$ (including t) of the same line after t, walking times are updated using $\mathsf{W}.\textsc{update}(u, i, \tau_{\text{fp}}(p_{\text{src}}, q))$. This indicates that those stops must not be scanned with walking times worse than or equal to $\tau_{\text{fp}}(p_{\text{src}}, q)$. This is correct also for trips $u \succ t$ since its stops have worse arrival times along u than along t. Therefore, they cannot produce non-dominated total journeys unless they have better walking times.

After these steps, trip segments $p_t^b \to p_t^e$ from queues Q_n, $n = 0, 1, \ldots$ are scanned until Q_n is empty. First, it is checked if p_{tgt} is reachable from $p_t^b \to p_t^e$ using the tuples in $\mathcal{L}_{\text{tgt}}$. For each tuple $(L_t, i, \Delta\tau) \in \mathcal{L}_{\text{tgt}}$ for $b < i \leq e$, arrival time $\tau_{\text{arr}} \leftarrow \tau_{\text{arr}}(t, i) + \Delta\tau$ and total walking time $\tau_{\text{walk}} \leftarrow \mathsf{W}.\textsc{get}(t, i) + \Delta\tau$ for the journey at p_{tgt} are computed. If the arrival label $(\tau_{\text{arr}}, n + 1, \tau_{\text{walk}})$ is not dominated by $\mathcal{J}$, it is added to $\mathcal{J}$, removing dominated labels. The journey uses $n + 1$ trips since trip segments in queue Q_n have been reached using n trips and the trip segment itself is part of trip $n + 1$. Stop $i = b$ must not be scanned: If a footpath $(p_t^b, p_{\text{tgt}}) \in \mathcal{F}$ exists and q is the stop that p_t^i was reached from, a footpath $(q, p_{\text{tgt}}) \in \mathcal{F}$ must exist as well because the footpaths are enclosed under transitivity. Hence, a potential journey would have already been found from q.

Next, possible transfers to new trip segments are checked. Target pruning is used to check if scanning the current trip segment $p_t^b \to p_t^e$ can yield any non-dominated journeys. In the original Trip-Based query, this only considers arrival times. Here, both arrival times and walking times must be taken into account. It can be observed that arrival times get worse along a trip while the walking times improve along a trip (see Figure 3.2). Therefore, the

best possible journey including $p_t^b \to p_t^e$ can have an arrival time no better than that of the first relevant stop, $\tau_{\text{arr}}(t, b+1)$ and a walking time no better than that of the last stop, $\mathsf{W}.\textsc{get}(t,\, e)$. If a journey with these values and $n+2$ used trips is already dominated by $\mathcal{J}$, the transfers are pruned. It is correct to use $n+2$ trips here because a transfer from the current trip segment leads to another trip from which p_{tgt} could be found. Therefore, all journeys that could be found by scanning the trip segment use at least $n+2$ trips. If the trip segment is not pruned, all outgoing transfers from its stops are explored. For a transfer $p_t^i \mapsto p_u^j \in \mathsf{T}$ with $b < i \leq e$, it is first checked if it improves walking times at p_u^j. The current walking time consists of the walking time that p_t^i was reached with and the walking time needed for the transfer:

$$\tau_{\text{walk}} = \mathsf{W}.\textsc{get}(t,\, i) + \tau_{\text{fp}}(p_t^i, p_u^j)$$

If $\tau_{\text{walk}} \geq \mathsf{W}.\textsc{get}(u,\, j)$, p_u^j has already been scanned or enqueued with a better arrival time and no new trip segment must be added. Otherwise, the end of the trip segment is calculated as the last stop index $k > j$ such that $\tau_{\text{walk}} < \mathsf{W}.\textsc{get}(u, k)$. The trip segment $p_u^j \to p_u^k$ is added to the next queue Q_{n+1}. Stops after k must not be enqueued since they have already been scanned or enqueued with an equal or better walking time than τ_{walk}. Additionally, W is updated with the new walking time τ_{walk} for all trips $v \succeq u$ (including u) of the same line. Scanning them with the same or worse walking times cannot lead to better journeys.

Optimized Target Pruning

The Walking Trip-Based query uses target pruning to avoid scanning transfers from a trip segment $p_t^b \to p_t^e$ if any journey that could be found using those transfers would be dominated. For that, the best possible journey that could be found using transfers from the current trip segment is created. If this journey is already dominated by $\mathcal{J}$, transfers do not have to be scanned. Since walking times decrease along trip t, the walking time at the last stop of the trip segment, $\mathsf{W}.\textsc{get}(t,\, e)$, is used for the optimal journey.

In fact, using other walking times can lead to more effective target pruning. It can be observed that if a higher walking time is used for the optimal journey, pruning becomes more effective. The probability that the optimal journey is dominated by $\mathcal{J}$ increases so that more trip segments can be pruned. Of course, this is only possible if it is ensured that all transfers from which non-dominated journeys can be found are indeed explored. Two optimizations are possible: First, the query can use the walking time of the first relevant stop of the trip segment, $\mathsf{W}.\textsc{get}(t,\, b+1)$. This walking time is no better than $\mathsf{W}.\textsc{get}(t,\, e)$ since walking times decrease along t. To see why this is correct, it must be considered that $p_t^b \to p_t^e$ has been enqueued by an Enqueue operation (see Algorithm 3.2). During that, all walking times for stop indices in $\{b, \ldots, e\}$ have been updated to the same value τ_{original}. Following Enqueue operations can have improved walking times along t to τ_{new}, starting from a stop index $j \leq e$ (changes for stop indices $j > e$ are not relevant for $p_t^b \to p_t^e$). Correctness must be ensured if $p_t^b \to p_t^e$ is pruned for walking time $\mathsf{W}.\textsc{get}(t,\, b+1)$ and it would not be pruned for $\mathsf{W}.\textsc{get}(t,\, e)$ (which is possible if $j > b+1$). In this case, the relevant transfers from the trip segment $p_t^j \to p_t^e$ that are missed in this iteration will be scanned when the trip segments for which walking times along t were improved are scanned. Transfers from all stops until p_t^e will be scanned since they are either covered by the first trip segment that improved walking times after p_t^b or by other trip segments that further improved them. In any case, the last trip segment which led to an improvement of walking times before p_t^e must at least reach p_t^e since it has a better walking time. Transfers from stops in $p_t^{b+1} \to p_t^{j-1}$ do not have to be scanned since the optimal journey which uses the walking time that $p_t^b \to p_t^e$ was enqueued with is already dominated.

Algorithm 3.2: WALKING TRIP-BASED QUERY

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{F}, \mathcal{Z})$, reduced transfer set T, source stop p_{src}, target stop p_{tgt}, departure time τ_{dep}.

Data: Minimum tentative walking times $\mathsf{W}(\cdot, \cdot)$ for each trip and each stop, lines reaching p_{tgt} $\mathcal{L}_{\text{tgt}}$, trip segment queues Q_n.

Output: Pareto set of arrival labels $\mathcal{J}$.

$\mathcal{J} \leftarrow \emptyset$
$\mathcal{L}_{\text{tgt}} \leftarrow \emptyset$
$\mathsf{Q}_n \leftarrow \emptyset$ for $n = 0, 1, \ldots$
W.CLEAR()
for each *stop* q *such that* $(q, p_{\text{tgt}}) \in \mathcal{F}$ **do**
  for each $(L, i) \in L(q)$ **do**
    $\mathcal{L}_{\text{tgt}} \leftarrow \mathcal{L}_{\text{tgt}} \cup \{(L, i, \tau_{\text{fp}}(q, p_{\text{tgt}}))\}$
for each *stop* q *such that* $(p_{\text{src}}, q) \in \mathcal{F}$ **do**
  for each $(L, i) \in L(q)$ **do**
    $t \leftarrow$ earliest trip of L such that $\tau_{\text{dep}}(t, i) \geq \tau_{\text{dep}} + \tau_{\text{fp}}(p_{\text{src}}, q)$
    ENQUEUE$(t, i, 0, \tau_{\text{fp}}(p_{\text{src}}, q))$
  if $q = p_{\text{tgt}}$ **then**
    $\mathcal{J} \leftarrow \mathcal{J} \cup \{(\tau_{\text{dep}} + \tau_{\text{fp}}(p_{\text{src}}, p_{\text{tgt}}), 0, \tau_{\text{fp}}(p_{\text{src}}, p_{\text{tgt}}))\}$
$n \leftarrow 0$
while $\mathsf{Q}_n \neq \emptyset$ **do**
  for each *trip segment* $p_t^b \rightarrow p_t^e \in \mathsf{Q}_n$ **do**
    for each $(L_t, i, \Delta\tau) \in \mathcal{L}_{\text{tgt}}$ *with* $b < i \leq e$ **do**
      $\tau_{\text{arr}} \leftarrow \tau_{arr}(t, i) + \Delta\tau$
      $\tau_{\text{walk}} \leftarrow \mathsf{W}$.GET$(t, i) + \Delta\tau$
      if $(\tau_{\text{arr}}, n+1, \tau_{\text{walk}})$ *is not dominated by* $\mathcal{J}$ **then**
        $\mathcal{J} \leftarrow \mathcal{J} \cup \{(\tau_{\text{arr}}, n+1, \tau_{\text{walk}})\}$, removing dominated entries
    if $(\tau_{\text{arr}}(t, b+1), n+2, \mathsf{W}$.GET$(t, e))$ *is not dominated by* $\mathcal{J}$ **then**
      for each *transfer* $p_t^i \mapsto p_u^j \in \mathsf{T}$ *with* $b < i \leq e$ **do**
        ENQUEUE$(u, j, n+1, \mathsf{W}$.GET$(t, i) + \tau_{\text{fp}}(p_t^i, p_u^j))$
  $n \leftarrow n + 1$

Procedure ENQUEUE(*trip* t, *index* i, *number of trips* n, *total walking time* τ)
  if $\tau < \mathsf{W}(t,i)$ **then**
    $j \leftarrow$ first stop after i such that $\mathsf{W}(t, j) \leq \tau$
    $\mathsf{Q}_n \leftarrow \mathsf{Q}_n \cup \{p_t^i \rightarrow p_t^j\}$
    for each *trip* u *with* $u \succeq t \wedge L_t = L_u$ **do**
      W.UPDATE(u, i, τ)

Second, it can be observed that even the walking time at p_t^{b+1} can be improved after enqueuing $p_t^b \to p_t^e$ by an Enqueue operation for another trip segment $p_t^j \to p_t^k$ for $j \leq b+1$. In that case, $p_t^b \to p_t^e$ might not be pruned, although the optimal possible journey that uses the walking time it has been enqueued with would be dominated by $\mathcal{J}$. Since all relevant transfers from stop in $p_t^b \to p_t^e$ are scanned again for $p_t^j \to p_t^k$, this is less efficient. The worst walking time that can be used for pruning while still preserving correctness is the walking time that $p_t^b \to p_t^e$ was enqueued with. Instead of using (potentially improved) values from $\mathsf{W}.\textsc{get}(t, \cdot)$, that walking time can be stored along with the trip segment in Q_n. This allows to use it for more effective target pruning. The data structure needed to store trip segments becomes larger and therefore less cache efficient but this is compensated by omitting the need to access the walking time data structure.

Walking Time Data Structure

Minimum tentative walking times must be stored in W for all trips and all stops therein. Different data structures to achieve this are studied in this work. The most intuitive data structure stores walking times in a two-dimensional array that has one line for each trip and one entry for each stop therein. Get operations are simple array accesses. The Update operation must iterate over all entries starting at the given index i and update the individual entries. Since values only decrease along the trip, this can be stopped when the first entry with equal or better walking time has been reached. Clearing the data structure for the next query requires resetting every entry to ∞. Pseudocode for this variant can be found in Data Structure 3.3. Since every stop has a separate entry – although multiple subsequent stops might have the same walking time –, many individual operations have to be executed. However, each individual operation is fast. The Walking Trip-Based query often scans larger sections of a trip. Here, the array based data structure with no additional information reaches high cache efficiency. However, resetting the data structure for every query has to reset each individual entry.

To avoid the constant overhead needed to reset all entries for every query, the *timestamp*-approach used for Dijkstra queries can be used [Zei20]. Here, a two-dimensional array is used to store labels $(\tau_{\text{walk}}, \theta)$ with walking time τ_{walk} and timestamp θ. The timestamp identifies the query that the label originates from. To achieve this, the query uses a global timestamp. At each access to an entry, the timestamp θ of the label is first compared with the global timestamp. If they are not equal, the label originates from a previous query. The walking time must be reset to ∞ and θ is set to the global timestamp. Besides that, operations work as in the previous version. Clearing the data structure only requires an incrementation of the global timestamp. This variant is described in pseudocode in Data Structure 3.4. Compared to the simple array based variant described above, this variant saves the time needed to reset values for each query. On the other hand, each individual operation becomes more complex. The necessary timestamp comparison triggers expensive branch instructions. Furthermore, cache efficiency deteriorates since an additional timestamp has to be stored for each entry.

Finally, the observation that walking times only decrease along a trip allows for a third variant. Instead of storing one value for each stop index, it is sufficient to store stop indices at which a new minimum walking time (compared to the previous stop index) can be achieved along with the respective walking time. For this, a list of tuples (i, τ_w) is stored for each trip t. A tuple (i, τ_w) signals that all stops with stop index $j \geq i$ along the trip can be reached with walking time τ_w. By selecting the last tuple that is relevant for an index, the smallest walking time can be found. To update a trip at index i with walking time τ, the relevant tuple must be found. If there is already a tuple for index i with walking time τ_{prev}, its walking time must be updated to $\min(\tau_{\text{prev}}, \tau)$. Otherwise, a new tuple (i, τ) must be inserted after the found tuple and all following tuples with walking time $\tau_{other} \geq \tau$

must be deleted. Clearing the data structure requires clearing the lists. This variant is described in pseudocode in Data Structure 3.5. In contrast to the two previous variants, values are only stored for new minima. If walking times change infrequently along a trip, this means that less operations have to be executed. Specifically, the update operation only needs to change few tuples instead of all relevant entries. On the other hand, maintaining the dynamic data structure is expensive. Efficient insert and delete operations are possible using a linked list. However, using linked lists comes at the cost of worse cache efficiency. A cache efficient implementation using an array has inefficient insert and delete operations since these require tuples to be repositioned.

Data Structure 3.3: ARRA-BASED WALKING TIME DATA STRUCTURE

Data: Array of arrays of walking times $W[t][p]$ for each trip and each stop.

Procedure CLEAR()
  $W[t][i] \leftarrow \infty$ for all trips t and stop indices $i = 0, \ldots, |t| - 1$ therein

Function GET(*trip t, stop index i*)
  return $W[t][i]$

Procedure UPDATE(*trip t, stop index i, walking time τ*)
  $j \leftarrow i$
  while $j < |t| \wedge \tau < W[t][j]$ **do**
    $W[t][j] \leftarrow \tau$
    $j \leftarrow j + 1$

Data Structure 3.4: ARRAY-BASED WALKING TIME DATA STRUCTURE WITH TIMESTAMPS

Data: $W[t][p]$, array of arrays of tuples (τ_w, θ) with walking time τ_w and timestamp θ, for each trip and each stop, global timestamp $\mathsf{timestamp} = 0$.

Procedure CLEAR()
  $\mathsf{timestamp} \leftarrow \mathsf{timestamp} + 1$

Function GET(*trip t, stop index i*)
  $(\tau_w, \theta) \leftarrow W[t][i]$
  if $\theta \neq \mathsf{timestamp}$ **then**
    $W[t][i] \leftarrow (\infty, \mathsf{timestamp})$
    return ∞
  return τ_w

Procedure UPDATE(*trip t, stop index i, walking time τ*)
  $j \leftarrow i$
  while $j < |t| \wedge \tau <$ GET(t,j) **do**
    $W[t][j] \leftarrow (\tau, \mathsf{timestamp})$
    $j \leftarrow j + 1$

Data Structure 3.5: MINIMA-BASED WALKING TIME DATA STRUCTURE

Data: $W[t]$, array of lists of tuples (i, τ_w) with stop index i and walking time τ_w for each trip t.

```
1  Procedure CLEAR()
2      W[t] ← ⟨⟩ for all trips t

1  Function GET(trip t, stop index i)
2      (j, τ) ← find last tuple in W[t] with j ≤ i by linear search
3      return τ

1  Procedure UPDATE(trip t, stop index i, walking time τ)
2      (j, τ) ← find last tuple in W[t] with j ≤ i by linear search
3      if j = i then
4          replace (j, τ_w) with (j, τ)
5      else
6          insert new tuple (i, τ) after the found tuple in W[t]
7      (k, τ') ← first tuple after the changed one
8      while τ' ≥ τ do
9          remove (k, τ') from W[t]
10         (k, τ') ← successor of the removed tuple
```

3.2. Trip-Based Routing for Minimal Fare Zone Subsets

Similar to the Walking Trip-Based algorithm, both the preprocessing and the query phase have to be adapted for the minimal fare zone subset query. The resulting algorithms are described in the following sections.

3.2.1. Preprocessing

Fare Zone Trip-Based preprocessing requires more changes than preprocessing for the Walking Trip-Based algorithm. One key observation is that – unlike walking times which only change during transfers – fare zones change while riding a trip. In the original Trip-Based preprocessing, transfers are only initially created if they link to a trip of another line, an earlier trip of the same line or an earlier stop of the same trip. This is too strict for fare zones. Consider the example in Figure 3.3: one trip t along five stops a, b, c, d, e where stops a, b, d, e are in fare zone 0 and only stop c is in fare zone 1. Assume there is only one footpath $(b, d) \in \mathcal{F}$ and the transfer $b = p_t^1 \mapsto p_t^3 = d$ exists (i.e., p_t^3 can be reached). Then, for a query with $p_{\text{src}} = a$ and $p_{\text{tgt}} = e$, any journey must use t to get away from a and reach e. Therefore, each journey has the same arrival time at e. Any journey J using transfer $p_t^1 \mapsto p_t^3$ only uses fare zone 0. It is not dominated since any other journey must additionally use fare zone 1. Therefore, the transfer $p_t^1 \mapsto p_t^3$, *along* trip t to a later stop of t is necessary. The same is true for transfers to *later* trips of the same line (if the transfer target stop cannot be reached along the original trip).

This must be reflected in the original transfer computation step. Now, transfers are only discarded if they connect to the same stop and the same trip – i.e., if they form a self-loop. All other transfers are potentially necessary. See Algorithm 3.6 for a pseudocode description of the updated computation.

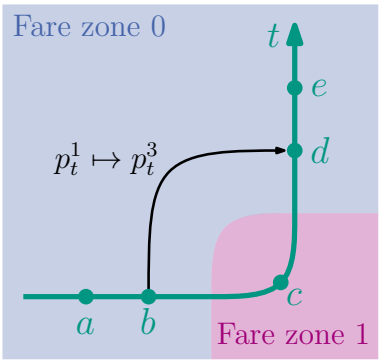


Figure 3.3.: When optimizing fare zone subsets, transfers *along* a trip or to later trips of the same line can be necessary.

U-turn transfers can be removed just as in the original preprocessing algorithm. A U-turn transfer that is removed there does not improve arrival times. Moreover, it also cannot improve fare zones since using a U-turn transfer requires visiting two additional stops that might belong to otherwise unused fare zones.

Algorithm 3.6: Original transfer computation for fare zones

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{F}, \mathcal{Z})$.
Output: Original transfer set T.

```
1 T ← ∅
2 for each trip t ∈ 𝒯 do
3     for each stop p_t^i on trip t with i > 0 do
4         for each stop q such that (p_t^i, q) ∈ ℱ do
5             for each (L, j) ∈ L(q) with j < |L| − 1 do
6                 u ← earliest trip of line L with τ_dep(u, j) ≥ τ_arr(t, i) + τ_fp(p_t^i, q)
7                 if u ≠ t ∨ j ≠ i then
8                     T ← T ∪ {p_t^i ↦ p_u^j}
```

Finally, transfer reduction must be updated for fare zones. This algorithm is described in pseudocode in Algorithm 3.7. Generally, the approach from transfer reduction for walking times can be used again: For each stop $p \in \mathcal{S}$, $\mathsf{S}(p)$ stores a Pareto set of labels $\ell = (\tau_{\text{arr}}, f)$, tuples with arrival time τ_{arr} and fare zone subset f.

For each trip t, stops p_t^i are scanned backwards along t. Because fare zones can change while riding along t, existing labels that were created when scanning a later stop p_t^j, $j > i$ might disregard the fare zone $\text{fz}(p_t^i)$ of the current stop. Therefore, $\text{fz}(p_t^i)$ must be added into the fare zone subset of all existing labels in the Pareto sets $\mathsf{S}(p)$ of all stops $p \in \mathcal{S}$ before exploring transfers out of p_t^i. Then, previously created labels are valid for the new iteration and can be used for more effective dominance checks.[1] Since the previous iteration has finished, changing the labels afterwards does not affect it. By adding the new fare zone $\text{fz}(p_t^i)$ in every step, labels from all previous iterations can be reused. See Figure 3.4 for an example.

[1] Without adding the new fare zones, pareto sets $\mathsf{S}(q)$ would have to be cleared in every stop-iteration to ensure correctness. This would lead to less effective transfer reduction since less alternative paths could dominate a transfer.

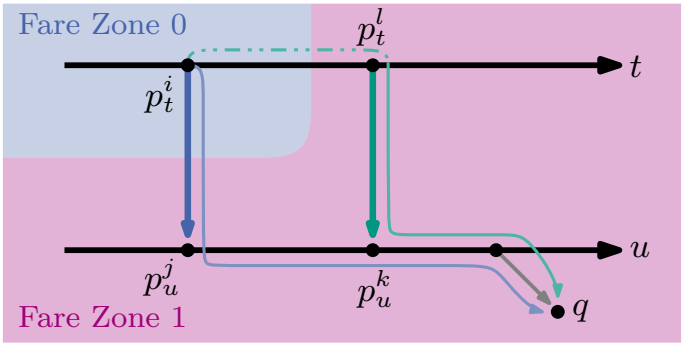


Figure 3.4.: Without update, the blue path might be dominated by the green path. However, the green path disregards the section $p_t^i \to p_t^l$ (dotted green) and does not include fare zone 0.

During the reduction, it is important to always add the necessary fare zones to the respective labels. All paths that the transfer reduction algorithm explores must be seen as subpaths inside a longer trip. Especially, initial and final footpaths of a total journey are not considered as transfers. Therefore, a transfer $p_t^i \mapsto p_u^j$ is only required if trip t has already been used and trip u will be used afterwards. This means that the fare zone of the first and last stop of every explored subpath must be added, even if the stop is reached with a footpath. After updating labels from previous rounds for the fare zone of p_t^i, the new label

$$(\tau_{\text{arr}}(t,i) + \tau_{\text{fp}}(p_t^i, q),\ \{\text{fz}(p_t^i), \text{fz}(q)\})$$

is added to $\mathsf{S}(q)$ for all stops q reachable from p_t^i using a footpath $(p_t^i, q) \in \mathcal{F}$. Dominated labels are removed.

Then, transfers $p_t^i \mapsto p_u^j$ can be explored. At this point, there could be multiple labels $(\tau_a,\ f)$ in $\mathsf{S}(p_t^i)$. However, only the optimal label $(\tau_{\text{arr}}(t,i),\ \{\text{fz}(p_t^i)\})$ must be used for further evaluation. For any other label, f must at least include $\text{fz}(p_t^i)$ and can therefore not be better. All other arrival times cannot be earlier than $\tau_{\text{arr}}(t,i)$ since they originate from p_t^i or later stops along t. Hence, all other labels would be dominated. The current fare zones along trip u are initialized to $f \leftarrow \{\text{fz}(p_t^i),\ \text{fz}(p_u^j)\}$, considering the fare zone of p_t^i and the fare zone of the reached stop p_u^j. Now, stops p_u^k for $k > j$ of the reached trip can be explored. It is important to explore stops in ascending order so that the correct fare zones for stops of u can be calculated efficiently. For each stop p_u^k, its fare zone $\text{fz}(p_u^k)$ is added to f and checked if the resulting label

$$(\tau_{\text{arr}}(u,k),\ f) \tag{3.2}$$

is not dominated by $\mathsf{S}(p_u^k)$. In that case, the transfer will be kept, and the label is inserted into $\mathsf{S}(p_u^k)$ and dominated labels are removed. Similarly, at each stop p_u^k, stops $q \neq p_u^k$ reachable using footpaths $(p_u^k, q) \in \mathcal{F}$ are explored. Again, the reached fare zone $\text{fz}(q)$ must be added to the current fare zones along u since reaching q in the preprocessing scenario is only relevant if another trip out of q would be used later. If the resulting label

$$(\tau_{\text{arr}}(u,k) + \tau_{\text{fp}}(p_u^k, q),\ f \cup \{\text{fz}(q)\}) \tag{3.3}$$

is not dominated by $\mathsf{S}(q)$, the transfer is kept. The label is added to $\mathsf{S}(q)$, again removing dominated entries. The fare zone of a stop $q \neq p_u^k$ reached by a footpath must not be added to the fare zone subset used to continue the scan along u. Transfers that have not been marked to be kept are removed.

It can now be proven that no necessary transfers are discarded. A transfer is necessary for a Pareto optimal journey J if neither J nor any equally good journey can be found without the transfer.

Algorithm 3.7: Fare zone Trip-Based transfer reduction

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{F}, \mathcal{Z})$, original transfer set T without U-turn transfers.
Data: Pareto sets $\mathsf{S}(\cdot)$ of labels (τ_{arr}, f) with arrival time τ_{arr} and fare zone subset f per stop.
Output: Reduced transfer set T.

for each *trip* $t \in \mathcal{T}$ **do**
  $\mathsf{S}(\cdot) \leftarrow \emptyset$
  for $i \leftarrow |t| - 1, \ldots, 1$ **do**
    for each *stop* q **do**
      for each $(\tau_a, f) \in \mathsf{S}(q)$ **do**
        update (τ_a, f) to $(\tau_a, f \cup \{\text{fz}(p_t^i)\})$ in $\mathsf{S}(q)$
    for each *stop* q *such that* $(p_t^i, q) \in \mathcal{F}$ **do**
      $(\tau_a, f) \leftarrow (\tau_{\text{arr}}(t, i) + \tau_{\text{fp}}(p_t^i, q), \{\text{fz}(p_t^i), \text{fz}(q)\})$
      if (τ_a, f) *is not dominated by* $\mathsf{S}(q)$ **then**
        $\mathsf{S}(q) \leftarrow \mathsf{S}(q) \cup \{(\tau_a, f)\}$, removing dominated entries
    for each *transfer* $p_t^i \mapsto p_u^j \in \mathsf{T}$ **do**
      $\mathsf{keep} \leftarrow$ false
      $f \leftarrow \{\text{fz}(p_t^i), \text{fz}(p_u^j)\}$
      for each *stop* p_u^k *on trip* u *with* $k > j$ *in ascending order* **do**
        $f \leftarrow f \cup \{\text{fz}(p_u^k)\}$
        if $(\tau_{\text{arr}}(u, k), f)$ *is not dominated by* $\mathsf{S}(p_u^k)$ **then**
          $\mathsf{S}(p_u^k) \leftarrow \mathsf{S}(p_u^k) \cup \{(\tau_{\text{arr}}(u, k), f)\}$, removing dominated entries
          $\mathsf{keep} \leftarrow$ true
        for each *stop* $q \neq p_u^k$ *such that* $(p_u^k, q) \in \mathcal{F}$ **do**
          $f' \leftarrow f \cup \{\text{fz}(q)\}$
          $\tau_a \leftarrow \tau_{\text{arr}}(u, k) + \tau_{\text{fp}}(p_u^k, q)$
          if (τ_a, f') *is not dominated by* $\mathsf{S}(q)$ **then**
            $\mathsf{S}(q) \leftarrow \mathsf{S}(q) \cup \{(\tau_a, f')\}$, removing dominated entries
            $\mathsf{keep} \leftarrow$ true
      if $\neg\mathsf{keep}$ **then**
        $\mathsf{T} \leftarrow \mathsf{T} \setminus \{p_t^i \mapsto p_u^j\}$

Theorem 3.2. Fare zone Trip-Based transfer reduction *(Algorithm 3.7) does not remove any transfer that is necessary for a Pareto optimal journey.*

Proof. By contradiction. Let $p_t^i \mapsto p_u^j$ be a transfer necessary for a Pareto optimal journey J_{complete}. Assume that $p_t^i \mapsto p_u^j$ is removed by Algorithm 3.7. To check if $p_t^i \mapsto p_u^j$ is necessary, the algorithm scans journeys J from p_t^i to each stop q reachable using a footpath $(p_u^k, q) \in \mathcal{F}$ from a stop p_u^k for $k > j$ along the reached trip. For each such stop q, it is checked if the label (τ_{arr}, f) as defined in (3.2) or (3.3) is not dominated by $\mathsf{S}(q)$. In

the label (τ_{arr}, f), τ_{arr} considers the arrival time along trip u as well as the walking time for a potential following footpath. Its fare zones f consider $\text{fz}(p_t^i)$, all fare zones used by travelling along u up to p_u^k and that of q, in the case of (3.3). If the transfer is removed, it has not produced a non-dominated label at any of these stops.

Thus, at each stop q reachable from p_u^k for $k > j$ using a footpath $(p_u^k, q) \in \mathcal{F}$, there is a label $(\tau_{\text{arr}}^1, f^1)$ that dominates (τ_{arr}, f). Since Pareto sets are reset for trip t and stops are scanned backwards along t, this label must have been originally produced by a journey J^1 from p_t^ℓ for $\ell \geq i$ using an outgoing transfer $p_t^\ell \mapsto p_v^k$ and a potential footpath from a stop following p_v^k. The alternative journey J^1 starts at p_t^ℓ and ends at q. Arrival times only increase along trips. Since existing labels are updated for each new stop along t, the label also considers all fare zones for stops p_t^m for $i \leq m < \ell$. Therefore J^1 can be extended to a journey J^2 from p_t^i to q by travelling from p_t^i to p_t^ℓ along t. The label $(\tau_{\text{arr}}^1, \tau_{\text{walk}}^1)$ remains valid for J^2 at q. Therefore, the dominating label $(\tau_{\text{arr}}^1, \tau_{\text{walk}}^1)$ at q implies that there is an alternative journey J^2 from p_t^i to q that produces a dominating label.

The original trip section J uses trips t and u. Similarly, J^2 uses at most trips t and v. Hence, the number of trips does not change when replacing J by J^2. Since J^2 corresponds to a dominating label at p_u^k or q, J can be replaced by J^2 in J_{complete}. The journey section after J in J_{complete} can still be reached using J^2. The resulting journey is equally good as J_{complete}. Since J^2 does not need $p_t^i \mapsto p_u^j$, it can be found without $p_t^i \mapsto p_u^j$. This contradicts the assumption. □

3.2.2. Query

For a source stop p_{src}, target stop p_{tgt} and departure time τ_{dep}, the Fare Zone Trip-Based query computes a Pareto set $\mathcal{J}$ of arrival labels $(\tau_{\text{arr}}, n, f)$ that indicate that there is a journey from p_{src} to p_{tgt} using n trips that uses exactly the fare zones in f and leaves p_{src} no earlier than τ_{dep}.

Since walking times are totally ordered, the Walking Trip-Based query only had to store one label $\mathsf{W}(\cdot, \cdot)$ per trip and stop. Fare Zone subsets are not totally ordered. There is not always a unique optimal fare zone subset for a stop at a trip. Therefore, the Fare Zone Trip-Based query must hold a Pareto set $\mathsf{F}(t, p)$ of subsets of fare zones for each trip t and each stop p therein. Since there is no unique optimal value in $\mathsf{F}(t, p)$ (which was the case for walking times), Queues Q_0, Q_1, ... store trip segments $p_t^b \to p_t^e$ along with the fare zone subset f they were reached with from p_{src}. The fare zone subset f is used instead of subsets from $\mathsf{F}(t, b)$ whenever scanning stops of the trip segment. This disregards other subsets in $\mathsf{F}(t, b)$ which is justified at the end of this section. Storing the fare zone subset f along with the trip segment allows to construct the correct fare zone subsets while scanning through t.

The query algorithm is described in pseudocode in Algorithm 3.8. The Enqueue operation for the fare zone query is described in pseudocode in Algorithm 3.9. The first two steps are similar to the walking query: First, for each stop q that has a footpath with walking time $\Delta\tau$ to p_{tgt} and each line L serving q with stop index i, a tuple $(L, i, \Delta\tau)$ is added to $\mathcal{L}_{\text{tgt}}$. The walking time is needed to compute the arrival time at p_{tgt}. Since this part of the journey is spent walking, no changes must be considered for fare zones. Second, trip segments reachable from p_{src} at τ_{dep} are enqueued. For each stop q reachable from p_{src} and each line serving q with index i, the earliest reachable trip t is selected. Starting with fare zones $\{\text{fz}(p_t^i)\}$, stops $j > i$ along t are scanned and the fare zone of each stop is added to the current fare zone subset. The scan ends at the last stop j where the current fare zone subset is not dominated by $\mathsf{F}(t, j)$. Then, the resulting trip segment $p_t^i \to p_t^j$ is added to Q_0 along with the fare zone subset $\{\text{fz}(p_t^i)\}$ that it was reached with. If p_t^i was

already reached with equal fare zones (the fare zone subset $\emptyset$ is not possible, therefore it cannot have been reached with better fare zones), no trip segment is enqueued. This is similar to the walking query except that fare zones change along a trip and therefore need to be updated during the scan. The same scan is repeated for t and all following trips u along the line and the constructed fare zone subsets are added to the Pareto set at $\mathsf{F}(\cdot,\cdot)$ to update the data structure for effective pruning.

Algorithm 3.8: Fare zone Trip-Based query

Input: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{F}, \mathcal{Z})$, reduced transfer set T, source stop p_{src}, target stop p_{tgt}, departure time τ_{dep}.

Data: Pareto sets of subsets of fare zones $\mathsf{F}(t,i)$ for each trip t and each stop index i therein, lines reaching p_{tgt} $\mathcal{L}_{\text{tgt}}$, queues Q_n of tuples $(p_t^b \to p_t^e,\ f)$ with a trip segment $p_t^b \to p_t^e$ and a fare zone subset f.

Output: Pareto set of arrival labels $\mathcal{J}$.

$\mathcal{J} \leftarrow \emptyset$
$\mathcal{L}_{\text{tgt}} \leftarrow \emptyset$
$\mathsf{Q}_n \leftarrow \emptyset$ for $n = 0, 1, \ldots$
$\mathsf{F}(t,\ p) \leftarrow \emptyset$ for each trip t and all stop indices p therein
for each *stop* q *such that* $(q, p_{\text{tgt}}) \in \mathcal{F}$ **do**
    for each $(L,\ i) \in L(q)$ **do**
        $\mathcal{L}_{\text{tgt}} \leftarrow \mathcal{L}_{\text{tgt}} \cup \{(L,\ i,\ \tau_{\text{fp}}(q, p_{\text{tgt}}))\}$
for each *stop* q *such that* $(p_{\text{src}}, q) \in \mathcal{F}$ **do**
    for each $(L,\ i) \in L(q)$ **do**
        $t \leftarrow$ earliest trip of L such that $\tau_{\text{dep}}(t,i) \geq \tau_{\text{dep}} + \tau_{\text{fp}}(p_{\text{src}}, q)$
        ENQUEUE$(t,\ i,\ 0,\ \{\text{fz}(p_t^i)\})$
    if $q = p_{\text{tgt}}$ **then**
        $\mathcal{J} \leftarrow \mathcal{J} \cup \{(\tau_{\text{dep}} + \tau_{\text{fp}}(p_{\text{src}}, p_{\text{tgt}}),\ 0,\ \emptyset)\}$
$n \leftarrow 0$
while $\mathsf{Q}_n \neq \emptyset$ **do**
    for each $(p_t^b \to p_t^e,\ f) \in \mathsf{Q}_n$ **do**
        $f' \leftarrow f$
        for each *stop index* $b < i \leq e$ **do**
            $f' \leftarrow f' \cup \{\text{fz}(p_t^i)\}$
            if $(L_t,\ i,\ \Delta\tau) \in \mathcal{L}_{\text{tgt}}$ **then**
                $\tau_{\text{arr}} \leftarrow \tau_{\text{arr}}(t,i) + \Delta\tau$
                if $(\tau_{\text{arr}},\ n+1,\ f')$ *is not dominated by* $\mathcal{J}$ **then**
                    $\mathcal{J} \leftarrow \mathcal{J} \cup \{(\tau_{\text{arr}},\ n+1,\ f')\}$, removing dominated entries
        $f' \leftarrow f$
        if $(\tau_{\text{arr}}(t, b+1),\ n+2,\ f)$ *is not dominated by* $\mathcal{J}$ **then**
            for each *stop index* $b < i \leq e$ **do**
                $f' \leftarrow f' \cup \{\text{fz}(p_t^i)\}$
                for each *transfer* $p_t^i \mapsto p_u^j \in \mathsf{T}$ **do**
                    $f'' \leftarrow f' \cup \{\text{fz}(p_u^j)\}$
                    ENQUEUE$(u,\ j,\ n+1,\ f'')$
    $n \leftarrow n + 1$

Algorithm 3.9: Fare zone Trip-Based query enqueue operation

Data: Public transit network $(\mathcal{S}, \mathcal{T}, \mathcal{L}, \mathcal{F}, \mathcal{Z})$, Pareto sets of subsets of fare zones $\mathsf{F}(t,i)$ for each trip t and each stop index i therein, queues Q_n of tuples $(p_t^b \to p_t^e,\ f)$ with a trip segment $p_t^b \to p_t^e$ and a fare zone subset f.

```
1  Procedure ENQUEUE(trip t, index i, number of trips n, fare zone subset f)
2      if ∄f' ∈ F(t, i) : f' ⊆ f then
3          for each trip u with u ⪰ t ∧ L_t = L_u do
4              k ← i + 1
5              f_u ← f
6              while k < |u| and ∄f' ∈ F(u, k) : f' ⊆ f_u do
7                  f_u ← f_u ∪ {fz(p_u^k)}
8                  F(u,k) ← F(u,k) ∪ {f_u}, removing dominated entries
9                  k ← k + 1
10             if u = t then
11                 Q_n ← Q_n ∪ {(p_t^i → p_t^{k-1}, f)}
```

After this, trip segments from queues Q_n, $n = 0, 1, \ldots$ are scanned until the next queue is empty. In contrast to the walking query, a trip segment $p_t^b \to p_t^e$ with fare zones f must be scanned stop by stop to add the relevant fare zones to f. If at stop index i, there is a tuple $(L_t,\ i,\ \Delta\tau) \in \mathcal{L}_{\text{tgt}}$, p_{tgt} can be reached with a footpath from p_t^i. In that case, the relevant journey with arrival time $\tau_{\text{arr}}(t,i) + \Delta\tau$, the updated fare zones in f (including $\text{fz}(p_t^i)$) and $n+1$ used trips is constructed. If it is not dominated, it is added to $\mathcal{J}$, removing dominated entries. Then, transfers from $p_t^b \to p_t^e$ are scanned. First, target pruning is used to see if any non-dominated journey can be created by scanning the transfers. From trip segment $p_t^b \to p_t^e$, any such journey must have at least arrival time $\tau_{\text{arr}}(t, b+1)$ (stop index b is not scanned since transfers are transitive), use at least $n+2$ trips and use all fare zones f that the trip segment has been reached with. Hence, the optimal possible journey has arrival label $(\tau_{\text{arr}}(t, b+1),\ n+2,\ f)$. If this label is already dominated by $\mathcal{J}$, transfers do not have to be evaluated. Otherwise, stops are scanned along the trip segment, keeping track of the relevant fare zones. For each transfer $p_t^i \mapsto p_u^j \in \mathsf{T}$, it is checked if a new trip segment starting at p_u^j must be enqueued. The subset of used fare zones includes the fare zones f that were used to reach the trip segment $p_t^b \to p_t^e$, the fare zones along t up to index i and $\text{fz}(p_u^j)$. If this subset is dominated by $\mathsf{F}(u,j)$, p_u^j has been reached before with a dominating fare zone subset and no trip segment must be added. Otherwise, it is scanned along the trip to find the last stop where the new fare zone subset (including fare zones for stops along the trip) is not dominated. This is the end of the new trip segment – every later stop has been reached with dominating fare zones before at this trip or an earlier trip of the same line. Finally, Pareto sets $\mathsf{F}(u,j)$ for $j \geq i$ are updated for all following trips u, adding the relevant fare zone subsets and removing dominated entries.

Scanning stops one by one along each trip segment[2], allows to construct the correct fare zone subset at each stop using only the fare zone subset f that the trip segment was enqueued with. It is not necessary to scan a trip segment $p_t^b \to p_t^e$ for every fare zone subset in $\mathsf{F}(t,b)$. This is illustrated in Figure 3.5. When the trip segment $p_t^b \to p_t^e$ was

[2]This causes no overhead when checking if p_{tgt} can be reached since it must be checked if p_{tgt} is reachable for each stop. For transfer evaluation, the overhead is minimal: Stops that do not have any outgoing transfers need to be scanned now, whereas this is otherwise not needed in a practical implementation. There, transfers are sorted by their start stop so that scanning all transfers from stops of a trip segment corresponds to scanning a continuous part of the transfers. This overhead is outweighed since scanning trip segments stop by stop allows to efficiently compute the relevant fare zone subset.

enqueued along with its fare zone subset f, f was not dominated by $\mathsf{F}(t, b)$. If enqueuing another trip segment $p_t^{b'} \to p_t^{e'}$ inserts another fare zone subset f' into $\mathsf{F}(t, b)$ after that, t or an earlier (better) trip of its line will be scanned with fare zones f' when processing $p_t^{b'} \to p_t^{e'}$. Additionally, since a fare zone subset has been added to $\mathsf{F}(t, b)$, $p_t^{b'} \to p_t^{e'}$ cannot start later than at stop index b. If f' is still not dominated at $\mathsf{F}(t, e)$ – which was the last stop where f was not dominated by $\mathsf{F}(t, e)$ at the time of enqueuing $p_t^b \to p_t^e$ – the new trip segment $p_t^{b'} \to p_t^{e'}$ reaches over p_t^e. Otherwise, an even better trip segment starts between stop indices b and e that reaches p_t^e. In every case, $p_t^b \to p_t^e$ is fully enclosed by one or multiple trip segments along t or earlier trips of L_t. Since these will be scanned with the respective fare zone subsets, a scan with the alternative fare zone subsets from $\mathsf{F}(t, b)$ is not needed for $p_t^b \to p_t^e$.[3]

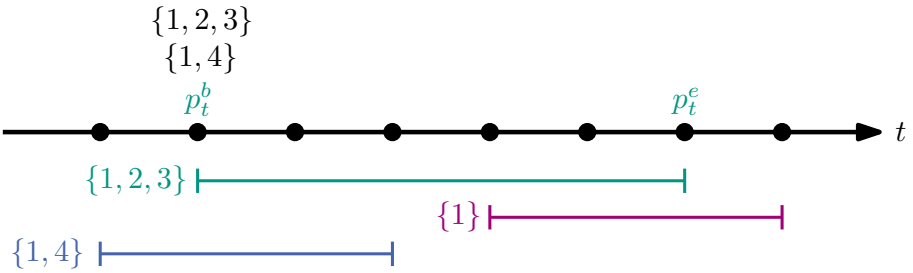


Figure 3.5.: A trip t and three trip segments along it with the fare zones they were reached with and in the order they were enqueued in. For a trip segment $p_t^b \to p_t^e$ (green), other trip segments might have added additional fare zone subsets at p_t^b (blue). However, $p_t^b \to p_t^e$ must only be scanned for its original fare zone subset. The stops in $p_t^b \to p_t^e$ will be scanned again for the other fare zone subsets when scanning the other trip segment (blue). If this does not cover all stops of $p_t^b \to p_t^e$, additional trip segments (lilac) must have been added which cover the remaining stops.

[3] This also shows that scanning $p_t^b \to p_t^e$ can turn out not to be needed. Instead of scanning it anyway, the trip segment could be removed from the queue once another trip segment that encloses it (or multiple other trip segments that collectively enclose it) and has (or have) dominating fare zone subsets is (are) enqueued. However, keeping the necessary additional information and checking the entire queue on every enqueue operation appears to be too expensive for the benefit that could be achieved.

4. Evaluation

The extended Trip-Based algorithms are evaluated in this chapter. First, the experimental setup and the used networks are described and results for the standard Trip-Based algorithm are compared to (standard) RAPTOR. Then, both the Walking and the Fare Zone Trip-Based algorithm will be evaluated in detail and the results will be compared to those of the relevant McRAPTOR variants.

4.1. Overview

All algorithms have been implemented in C++17 and compiled using GCC 9.3.1 (64-Bit) with optimization flags `-O3` and `-march=native`. Implementation details can be found in Appendix A. All query algorithms as well as preprocessing for Bern were evaluated on a machine with two 8-core Intel Xeon Skylake SP Gold 6144 CPUs clocked at 3.5 GHz with 24.75 MiB of L3 Cache and 192 GiB of DDR4-2666 RAM. Because of high memory consumption, preprocessing for Switzerland and London was evaluated using a machine with two 64-core AMD EPYC Zen2 7742 CPUs clocked at 2.25 GHz with 16 MiB of L3 cache and 1 024 GiB of DDR4 3200 RAM.

Query running time measurements are averages over a constant set of 10 000 (Bern and Switzerland for walking queries) or 1 000 (all others) queries that were randomly generated once for each network. Unless otherwise noted, journey reconstruction was always run when running queries for both the McRAPTOR and extended Trip-Based algorithms. No journey reconstruction was run for the Standard Trip-Based and RAPTOR algorithms. These algorithms also do not maintain the data structures solely needed for journey reconstruction. Preprocessing running times are measured for single runs. All running times are measured without the time necessary to load or store the relevant network data.

Data from three real networks was used. Data for Switzerland was sourced from a publicly available GTFS feed[1]. It was previously used in [SWZ20]. The Bern network was extracted from this data. The London network uses data from Transport for London[2]. The public transit data was joined with footpath networks extracted from OpenStreetMap[3] [SWZ20]. An overview can be found in Table 4.1. A *stop event* is the departure or arrival of a trip at a stop. A *connection* is the elementary connection of two subsequent stops in any trip.

[1] `https://gtfs.geops.ch/`
[2] `https://data.london.gov.uk/`
[3] `https://download.geofabrik.de/`

	Bern	Switzerland	London
Stops	535	25 125	20 595
Lines	242	13 786	2 107
Trips	17 447	350 006	125 436
Stop events	218 492	4 686 865	4 970 428
Connections	201 045	4 336 859	4 844 992

Table 4.1.: Overview of the networks used for evaluation.

Since both Trip-Based and RAPTOR use footpath graphs enclosed under transitivity, the walking threshold θ_w used to filter footpaths before calculating transitive footpaths has a large influence on the complexity of the networks and thereby the performance of the algorithms. An overview of the influence of the threshold can be found in Figure 4.1. It is also shown visually in Figures 4.2, 4.3 and 4.4. Bern and Switzerland are evaluated with walking thresholds from 100 s to 900 s whereas London is evaluated with thresholds from 50 s to 250 s. For higher walking thresholds, the footpaths become too complex.

It must be noted that walking thresholds are hardly comparable between different networks: For footpaths enclosed under transitivity, the number of footpaths rises the most when shorter footpaths have a joint end-stop and can therefore form a new footpath, which can, in turn, join with others. In this way, footpaths far longer than the threshold can be created. In public transit networks, footpaths connect stops. Therefore, footpaths can only join if the walking threshold allows footpaths from one stop to be long enough so that other stops in the surroundings are reachable. This shows that the influence of walking thresholds depends on the distance of stops in the networks (in this case measured in walking time). It also explains the different walking threshold influence seen in Figure 4.1. Switzerland consists of sparse areas as well as dense cities. These correspond to areas with high stop-to-stop distances and such with low distances. Combined, they show a continuous rise in the number of footpaths for all walking thresholds. Bern is one of the cities used in Switzerland. Here, average distances between stops are lower. Therefore, the number of footpaths rises stronger for low thresholds. For higher thresholds, the influence drops since other stops are alredy reachable from most stops. In the dense network of London, this effect is even more noticable. It has roughly 80 % of the stops of Switzerland but covers a far smaller area. Hence, the distance between stops is significantly smaller than for Bern and Switzerland. This explains the strong rise in the number of footpaths for walking thresholds above 250 s.

In the evaluation, *original transfers* refers to the number of transfers after original transfer creation and U-turn reduction. *Reduced transfers* is the number of transfers left after the transfer reduction step was run.

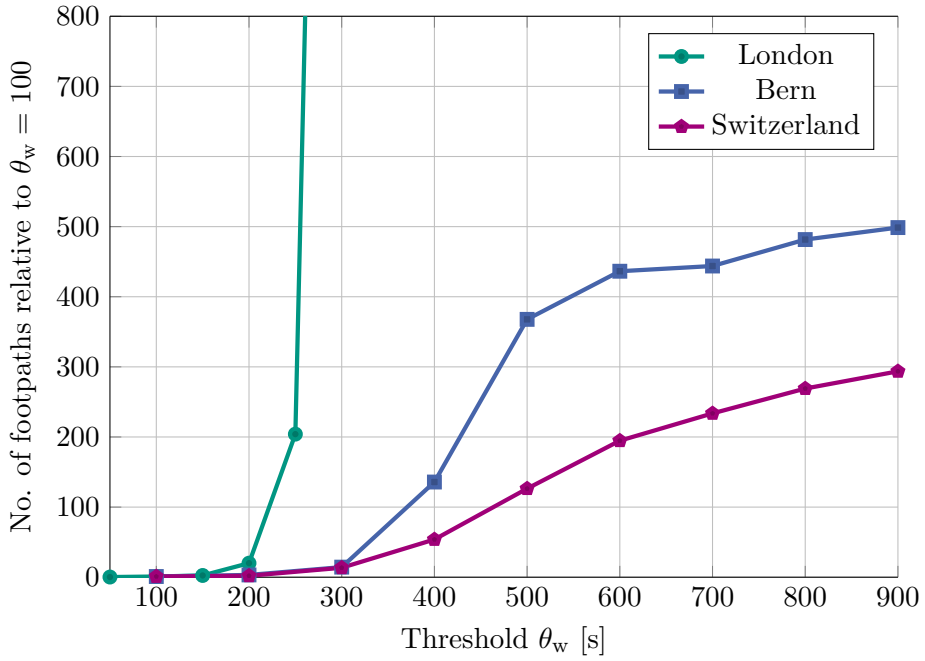
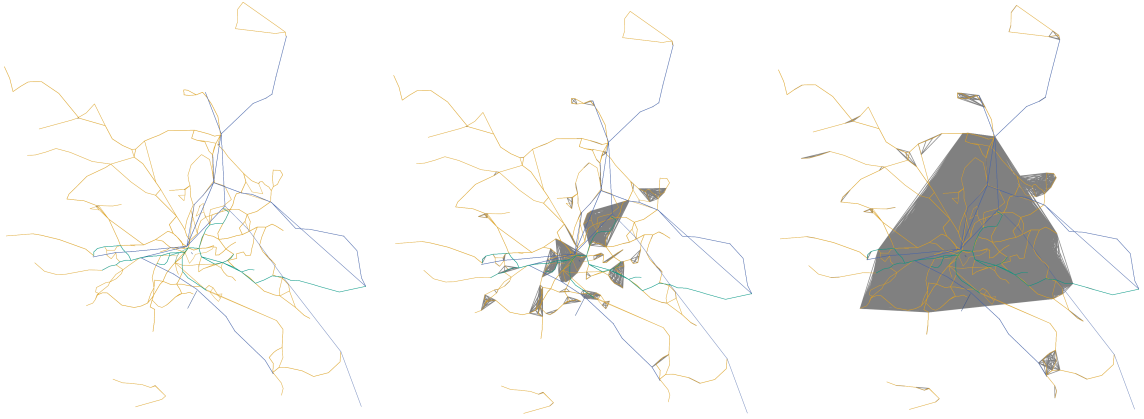
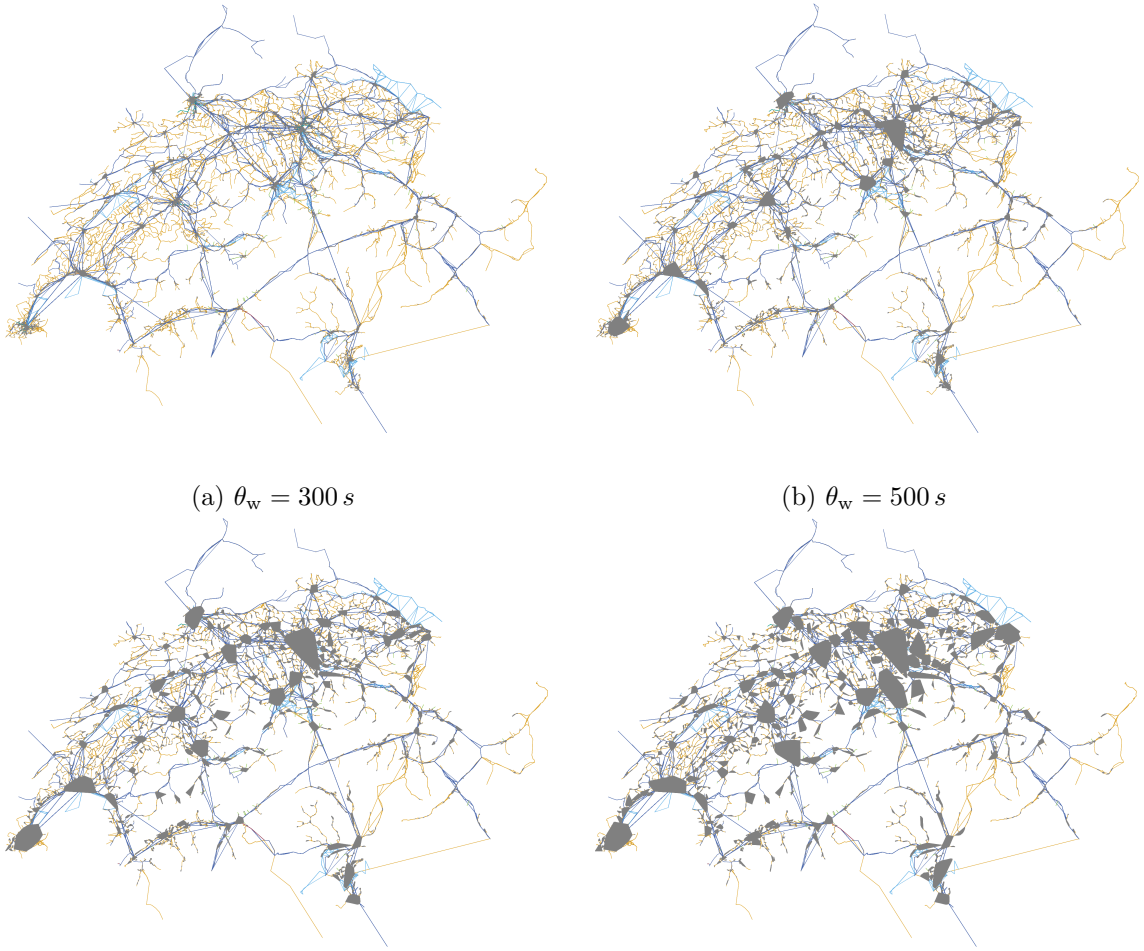


Figure 4.1.: Ratio of the number of footpaths for a given threshold θ_w compared to the number of footpaths for $\theta_w = 100\,s$.



(a) $\theta_w = 100\,s$ (b) $\theta_w = 300\,s$ (c) $\theta_w = 500\,s$

Figure 4.2.: Illustration of the Bern network for various walking thresholds. Lines are shown in colors, footpaths in gray. For low thresholds, only few footpaths exist. For $\theta_w = 300\,s$, some areas form cliques. At $\theta_w = 500\,s$, the Bern network is almost fully connected.



(a) $\theta_w = 300\,s$ (b) $\theta_w = 500\,s$

(c) $\theta_w = 700\,s$ (d) $\theta_w = 900\,s$

Figure 4.3.: The Switzerland network contains dense city networks and large sparse areas. With rising threshold, larger areas become connected.

(a) $\theta_w = 150\,s$ (b) $\theta_w = 200\,s$ (c) $\theta_w = 250\,s$

Figure 4.4.: The London network has almost as many stops as Switzerland, but on a much smaller area. With comparably low thresholds, many of these become connected.

4.1.1. Standard Trip-Based

To establish a baseline for the performance of the Trip-Based algorithm, the standard Trip-Based preprocessing and query algorithms are evaluated using the three networks introduced before. Standard Trip-Based preprocessing reduces transfers significantly while requiring little preprocessing time. Preprocessing times grow with larger thresholds θ_w. However, preprocessing effectiveness – measured in the share of original transfers that are discarded by the reduction step (Algorithm 2.4) – grows as well.

The development of the preprocessing effectiveness for all three networks is depicted in Figure 4.5, exact numbers are available in Table 4.2. Exemplarily, the preprocessing effectiveness is analyzed in detail for the Bern network: For $\theta_w = 100\,s$, 73 % of transfers are discarded. This value grows to 95 % for $\theta_w = 300\,s$ and above 99 % for all higher thresholds. Hence, preprocessing becomes more effective for higher thresholds. The absolute number of reduced transfers grows from 325 056 for $\theta_w = 100\,s$ to a maximum of 1 740 140 for $\theta_w = 500\,s$. Still, the growth factor of 5.35 is very small compared to the simultaneous growth in the number of footpaths by a factor of around 367. For thresholds above 500 s, the number of reduced transfers even drops slightly: For $\theta_w = 900\,s$, there are 4.7 % fewer transfers than at $\theta_w = 500\,s$. For the very high thresholds, the network contains more and longer footpaths. This means that more alternative trips and footpaths can be explored when checking if a transfer is necessary. Moreover, shorter transfers can be dominated by longer transfers. This can explain the drop in the number of reduced transfers: The basic set of transfers is larger, but for each transfer, there are more alternatives that can dominate it.

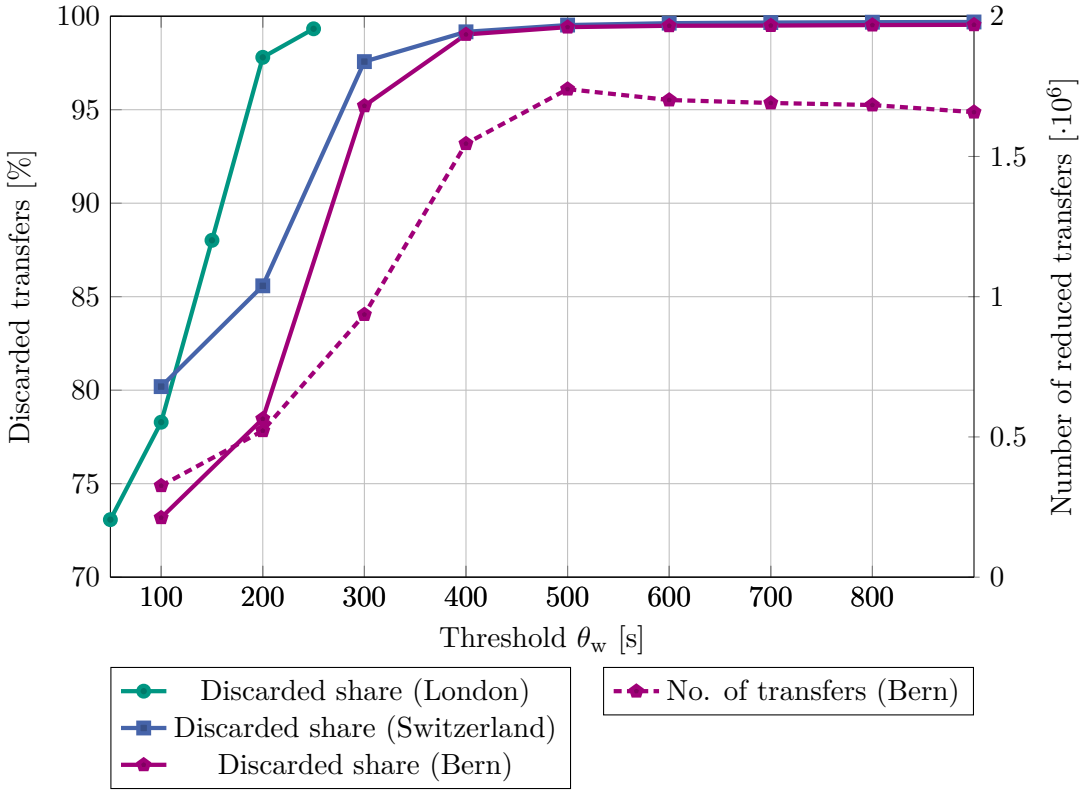


Figure 4.5.: Share of original transfers reduced by standard Trip-Based preprocessing for all networks. Exemplarily, the total number of reduced transfers is shown for Bern (dashed).

Preprocessing running times can be found in Figure 4.6 and Table 4.2. Preprocessing is generally fast for low thresholds – below one second for thresholds up to 300 s using 16 threads for Bern. The small number of footpaths generates comparably few transfers, and scanning transfers and footpaths to determine if a transfer is needed is fast. Running times grow significantly for higher thresholds. The highest running time for Bern is reached at 72.8 s for $\theta_w = 900$ s. Here, far more original transfers must be checked, and more work needs to be done to evaluate if a single transfer is needed. By itself, preprocessing is still relatively fast for higher thresholds, ranging around one minute for most thresholds on the small network of Bern. For larger networks like Switzerland and London, preprocessing is fast for low thresholds. However, it becomes slower for higher thresholds – even with heavy parallelization. For $\theta_w = 250$ s, preprocessing takes more than one hour for London using 128 threads.

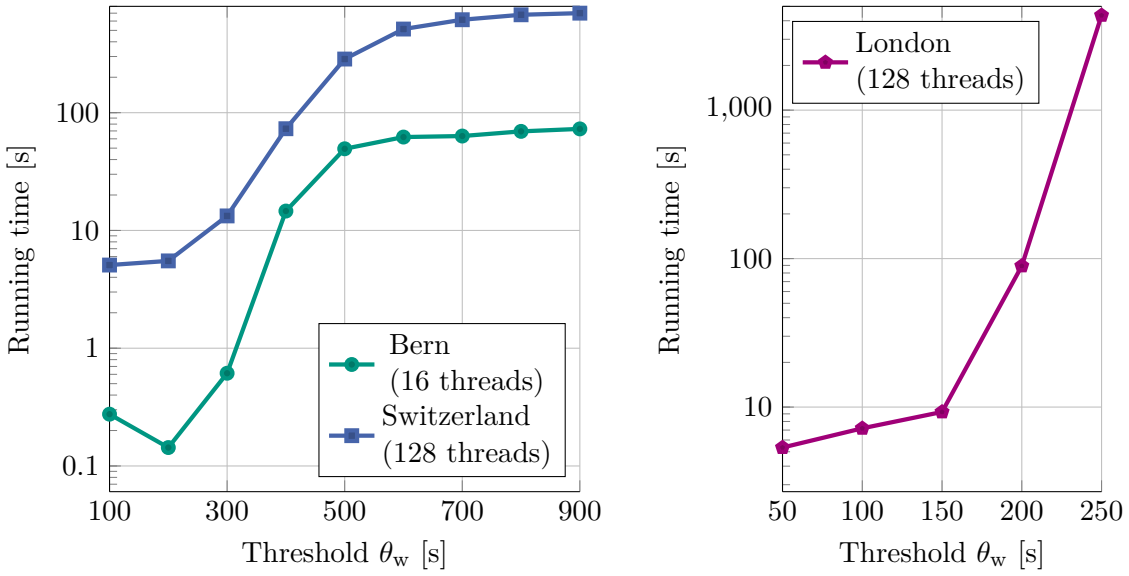


Figure 4.6.: Running time for standard Trip-Based preprocessing on all three networks.

Standard Trip-Based queries are faster than RAPTOR queries for all thresholds on all networks. Full details are available in Figures 4.7 and 4.8 as well as in Table 4.2. In the following, they will be exemplarily analyzed for Bern.

Absolute running times grow in parallel to the number of footpaths for both Trip-Based and RAPTOR. Trip-Based running times are up to 2.46 times faster for low thresholds. With more complex footpath networks, the speedup converges to 1.75 and appears to stay on that level. This gives Trip-Based queries a considerable advantage over RAPTOR queries. The highest advantage of Trip-Based over RAPTOR is reached at thresholds up to 300 s. For these thresholds, preprocessing is fastest (below one second), thus combining very fast preprocessing with high speedups compared to RAPTOR. The high speedups are achieved even though preprocessing for these thresholds is less effective compared to higher thresholds (discarding between 73 % and 95 % of transfers). This shows that even the query itself is highly efficient. For higher thresholds, transfer reduction becomes more important. Here, standard Trip-Based queries still hold an advantage of around 1.75 against RAPTOR. However, this is only reached by discarding around 99 % of all original transfers.

For the Switzerland network, the speedup is around 1.8 for low thresholds but rises to 3 for $\theta_w = 900$ s. London first shows a speedup of around 1.8 for $\theta_w = 50$ s which declines to 1.37 for $\theta_w = 150$ s and then grows again to 2.73 at $\theta_w = 250$ s. The difference to Bern can be explained by the different network characteristics which in turn influence how footpaths

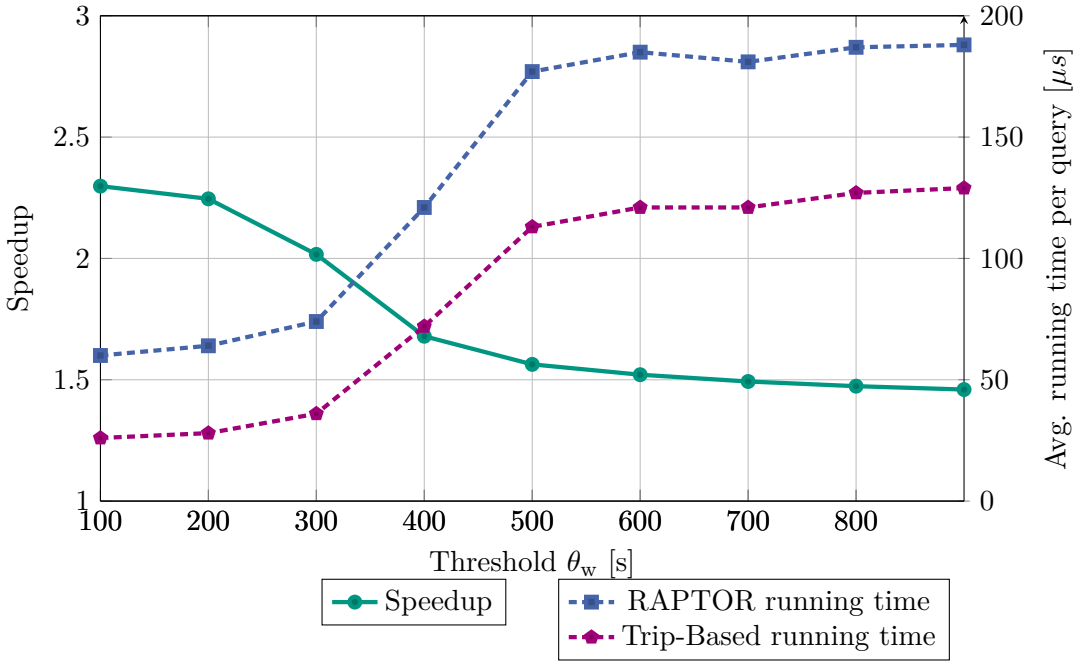


Figure 4.7.: Speedups of Trip-Based queries compared to RAPTOR queries for Bern. Additionally, actual running times of both algorithms are shown (dashed).

develop depending on the walking threshold. In conclusion, actual speedups achieved by the standard Trip-Based algorithm depend on the network and the specific threshold. Generally, speedups range from 1.4 to 3 for the combinations evaluated in this work.

		Transfers			Running times		
					Prepr.	Query	
Network	θ_w	Original	Reduced	Discarded		Trip-B.	RAPTOR
Unit	[s]	[$\cdot 10^6$]	[$\cdot 10^6$]	[%]	[mm:ss]	[μs]	[μs]
Bern	100	1.2	0.3	73.1	00:00.27	26	60
	300	19.5	0.9	95.2	00:00.61	36	74
	500	298.1	1.7	99.4	00:49	113	177
	900	361.2	1.6	99.5	01:12	129	188
SW	100	40.8	8.0	80.1	00:05	3574	6357
	300	709.5	17.2	97.5	00:13	4 027	7 248
	500	6 400.7	30.2	99.5	04:35	4 673	10 459
	900	10 412.8	32.0	99.6	11:40	4 707	13 948
London	50	31.8	8.5	73.0	00:05	1 795	3 210
	150	171.8	20.5	88.0	00:09	2 311	3 185
	250	10 679.3	71.8	99.3	72:28	4 797	13 141

Table 4.2.: Preprocessing and query results for the standard Trip-Based algorithm for all networks.

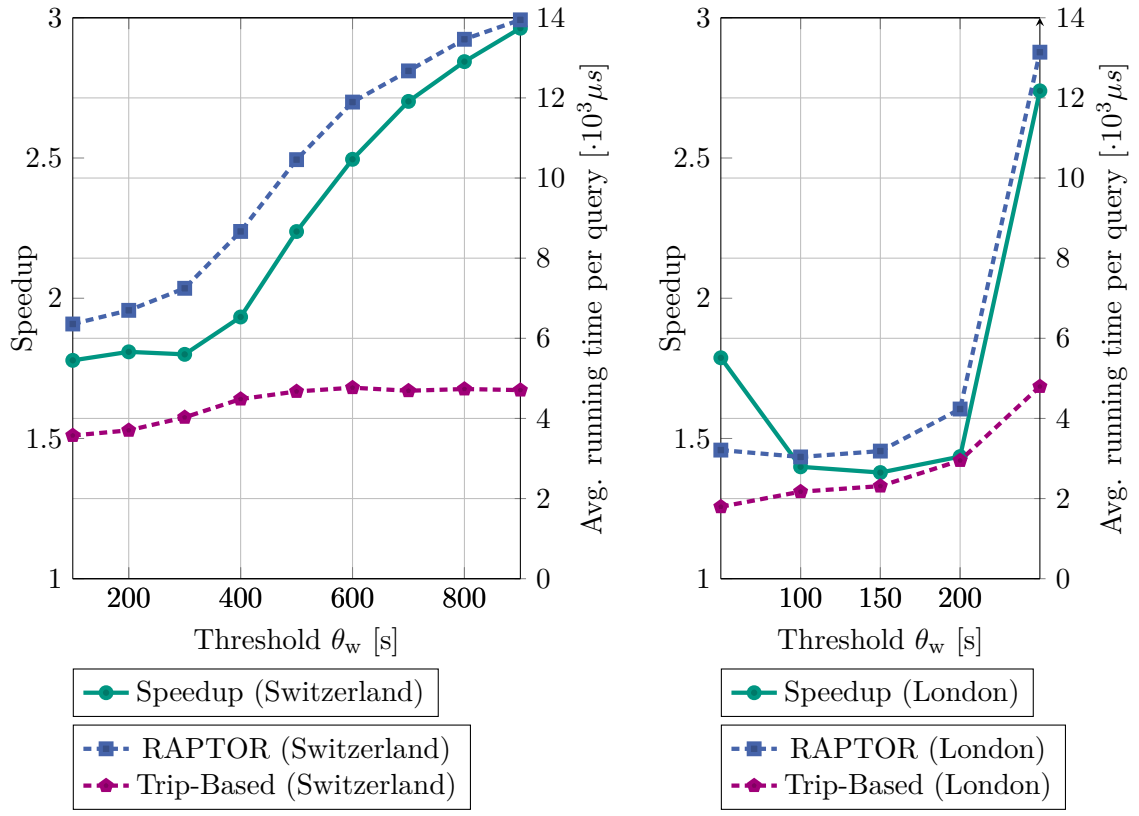


Figure 4.8.: Speedups of Trip-Based queries compared to RAPTOR queries for Switzerland and London. Additionally, actual running times are shown (dashed).

4.2. Walking Trip-Based Algorithm

Next, the Walking Trip-Based algorithm, the first extension of the standard Trip-Based algorithm, is evaluated. Specifically, effectiveness and running times of preprocessing as well as speedups from parallel preprocessing are evaluated. Query running times are analyzed and compared to the running times of Walking McRAPTOR. Furthermore, the different data structure versions as presented in Section 3.1.2 are compared and the effect of optimized target pruning is evaluated.

4.2.1. Walking Trip-Based Preprocessing

The first two preprocessing steps (original transfer generation and U-turn reduction) do not change for Walking Trip-Based compared to the standard Trip-Based algorithm. Therefore, the evaluation will focus on the transfer reduction step (see Algorithm 3.1). Measurements of preprocessing times still include all three steps. On all three networks, Walking Trip-Based preprocessing takes more time than the standard variant. For Bern, preprocessing is fast for thresholds up to $\theta_w = 300\,s$, with running times below or around one second. Beginning with $\theta_w = 400\,s$, running times deteriorate. This development is similar to the standard Trip-Based algorithm. Besides $\theta_w = 100\,s$, Walking Trip-Based preprocessing always takes at least twice as long as standard preprocessing. For higher thresholds, the factor rises to 2.75. On Switzerland, the running time is 2.1 times higher for small thresholds, rising to about 3.1 times for $\theta_w = 900\,s$. For London, a much denser graph, the factor peaks at 4.27 for $\theta_w = 150\,s$ and declines for higher thresholds. Running time comparisons can be found in Figure 4.9. Detailed running times for the networks are available in Figures 4.10 and 4.11. Exact numbers can be found in Table 4.5 at the end of this section.

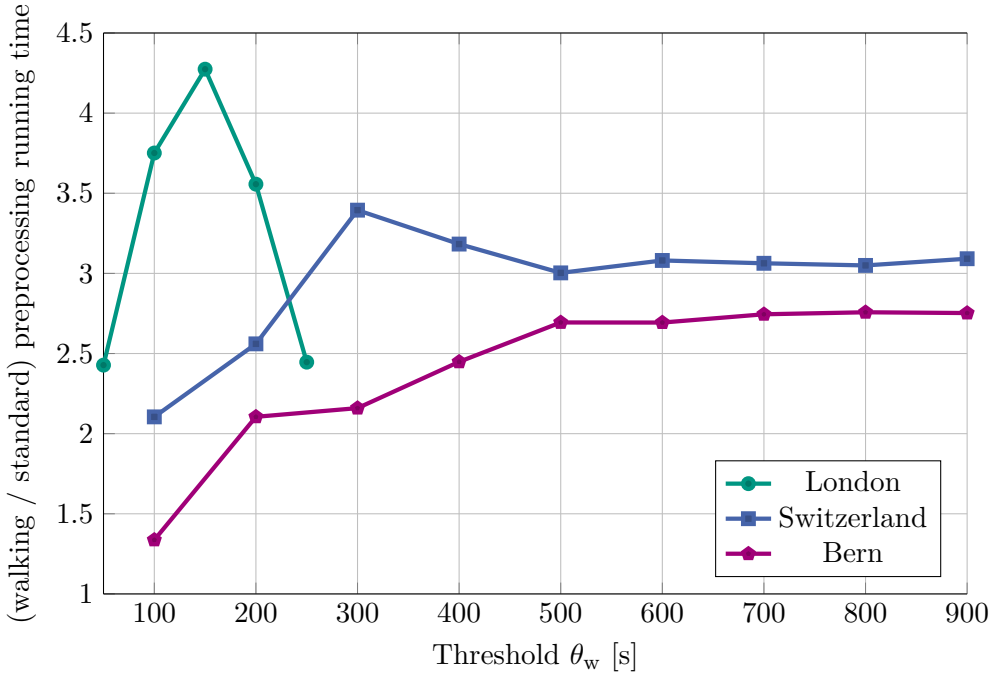


Figure 4.9.: Ratio of Walking Trip-Based preprocessing running times compared to standard Trip-Based preprocessing running times for all networks.

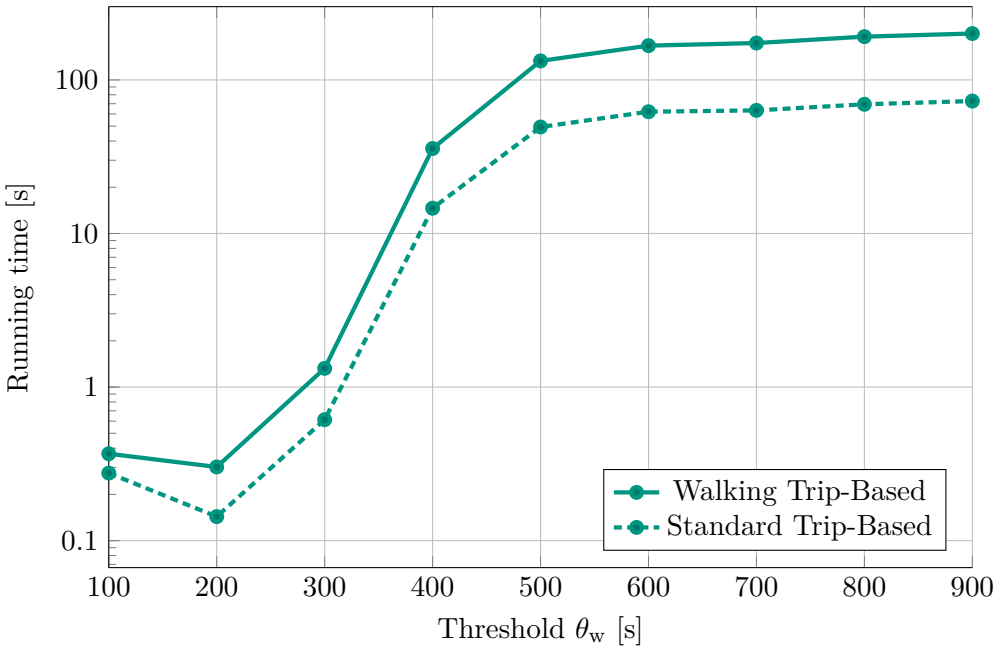


Figure 4.10.: Walking Trip-Based preprocessing running times for Bern using 16 threads.

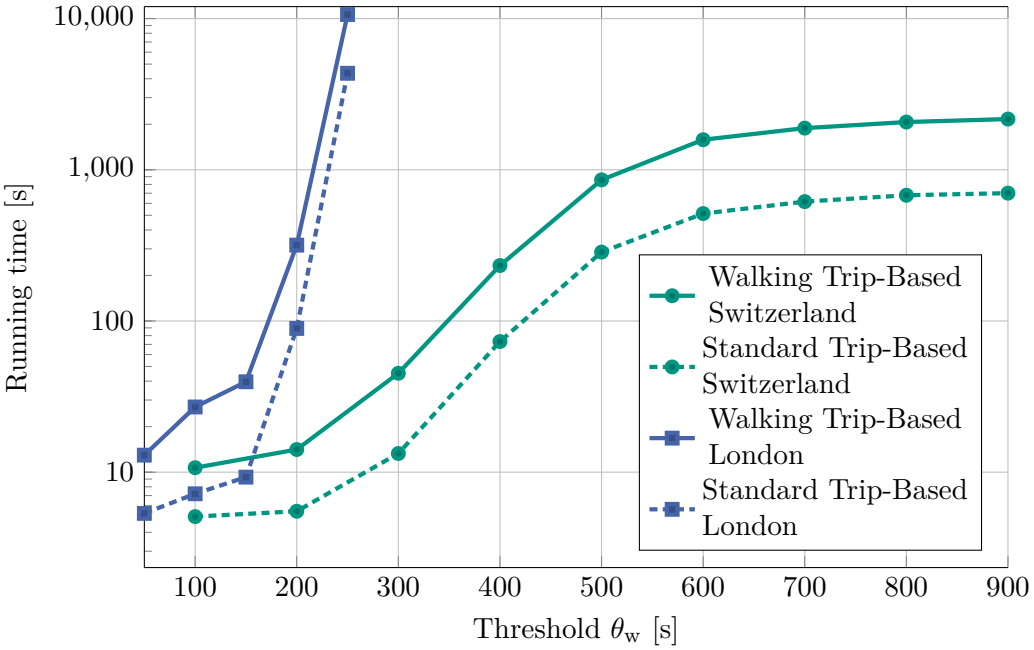


Figure 4.11.: Walking Trip-Based preprocessing running times for Switzerland and London using 128 threads.

For the same number of original transfers, the Walking Trip-Based preprocessing takes longer than the standard variant. This is caused by the more complex operations that have to be used in the transfer reduction step. The standard variant works on totally ordered optimal arrival times. For the walking variant, these are replaced by Pareto sets of tuples with arrival and walking times. Dominance checks and insertions for Pareto sets are more complex than "is smaller" checks and assignments for simple arrival times. This explains the time difference for the preprocessing algorithms. As shown in Section 3.1.1 and Figure 3.1, using Pareto sets cannot easily be avoided in the Trip-Based preprocessing setting.

Total running times are still low for most thresholds on all networks. Low thresholds (below 400 s for Bern and Switzerland, below 200 s) allow parallelized preprocessing in under one minute. For higher thresholds, this rises significantly. The highly parallelized variant needs around 36 min for the Switzerland network with $\theta_w = 900\,s$. Preprocessing for London with $\theta_w = 250\,s$ takes almost three hours. In both cases, more than 10 billion original transfers have to be processed. The approach to use footpaths enclosed under transitivity reaches its limits for such high thresholds. However, for thresholds that are typically used for transitive footpaths, preprocessing is still fast.

For fast queries, effective transfer reduction is crucial. While more transfers are relevant for walking queries, the reduction should still reduce as many transfers as possible in reasonable preprocessing time. Figure 4.12 shows that the Walking Trip-Based preprocessing algorithm still discards a high share of the original transfers. In fact, the maximum discrepancy between walking and standard preprocessing reaches only 3.2 (Bern), 2.8 (Switzerland) and 8.8 (London) percentage points for the worst threshold while it is significantly lower for most thresholds. Yet, because of the high percentage that is generally discarded, small differences in percentages can still cause significantly more transfers to be kept. To assess the effectiveness of Walking Trip-Based preprocessing more accurately, the ratio of the number of reduced transfers for Walking Trip-Based preprocessing compared to standard Trip-Based preprocessing is evaluated. Figure 4.13 shows this ratio in dependence on the walking threshold. For all three networks, the Walking Trip-Based transfer reduction comes

closest to the standard reduction for low thresholds. For Bern and Switzerland, the walking reduction keeps less than 20 % more transfers for thresholds below 300 s. Above that, the ratio rises significantly for both networks. For Bern, it reaches a plateau at around 2.5 for thresholds above 400 s. This is also the area where less additional footpaths are created by rising thresholds (see Figure 4.1). For Switzerland, the ratio continues to rise even for high thresholds, but the growth declines. It reaches the highest ratio of all three networks with around 3.5. For London, the ratio rises almost linearly with the thresholds. Since the network is denser than the others, the ratio is higher for equal thresholds.

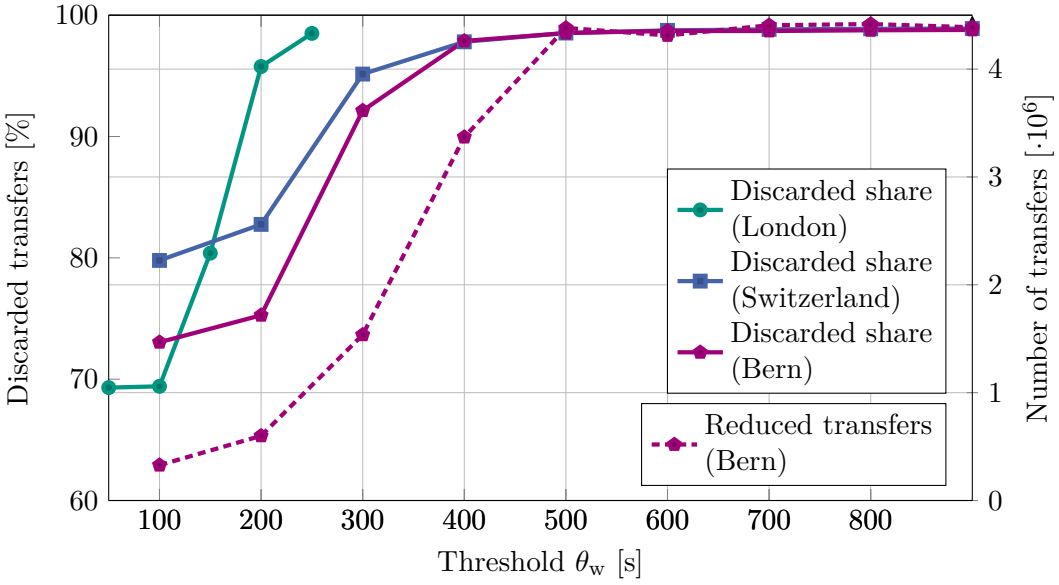


Figure 4.12.: Share of original transfers that was discarded by Walking Trip-Based preprocessing. Additionally, the number of reduced transfers is shown for Bern.

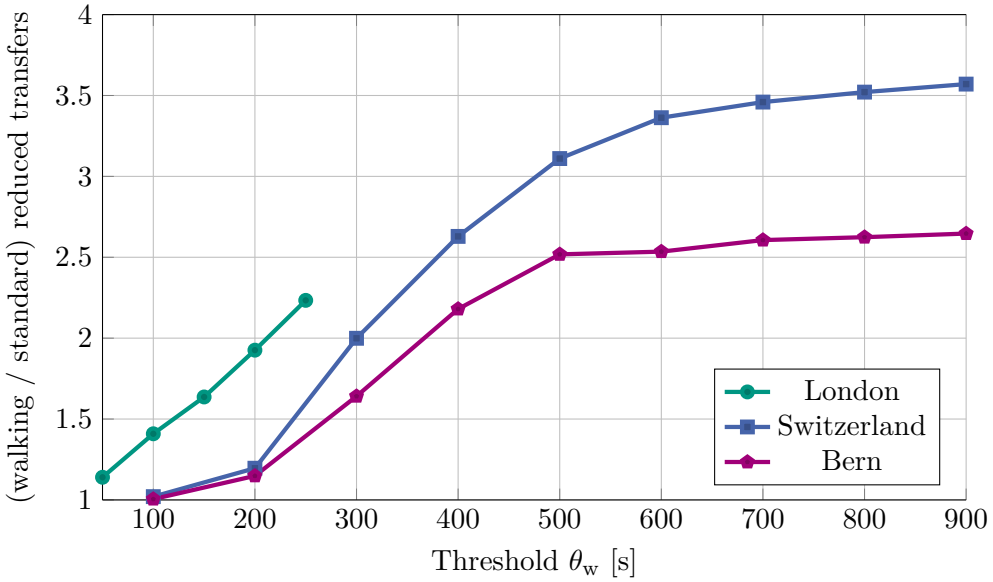


Figure 4.13.: Ratio of reduced transfers for the Walking Trip-Based preprocessing compared to the standard Trip-Based preprocessing.

Multiple conclusions can be drawn: First, Walking Trip-Based preprocessing is generally less effective than preprocessing in the standard variant. This is as anticipated because more transfers are necessary for optimal walking query journeys. Second, Walking Trip-Based preprocessing becomes monotonically more effective with rising thresholds when judging by the share of original transfers that is discarded. This is analogous to the standard variant and primarily based on the higher number of original transfers. Actually, a more detailed analysis shows that while the number of original transfers remains the same between the standard and walking variant, the ratio of reduced transfers between the two variants actually rises with rising thresholds. Since the absolute number of reduced transfers has most influence on the query speed, this shows that the Walking Trip-Based preprocessing performs worse for higher thresholds. Yet, this can in part be explained by the additional criterion: With higher walking thresholds, there are more longer footpaths. These are part of optimal journeys more frequently and can therefore not be reduced as often as shorter footpaths.

All preprocessing steps described in Section 3.1.1 can be trivially parallelized on the trip level. The speedup achieved by this parallelization will be evaluated using the Bern network for different thresholds. The speedup for a number of threads n is calculated as the quotient of the running time for the parallel version with n threads and the running time of the sequential version. Since no superlinear speedup can be expected fot this type of parallelization, the optimal result would be a speedup of n. Results are shown in Figure 4.14. For threshold $\theta_w = 100\,s$, the speedup is low, only reaching around 2 even for 16 threads. In this case, parallelization overheads and time spent to sequentially join the results from different threads come into play. Bern with $\theta_w = 100\,s$ is a small network with only 1.2 million original transfers. The parallel variant on 16 threads only takes 0.36 seconds for preprocessing. Overheads and a larger share of sequential processing time justify the low speedup. With rising thresholds, speedups improve significantly, as the impact of overhead and sequential work declines. For large thresholds, speedups with n threads reach almost n. For instance, a speedup of 15 is achieved using 16 threads for $\theta_w = 700\,s$. Since parallel preprocessing running times are low for the thresholds with low speedups (only 1.3 s for $\theta_w = 300\,s$), it can be concluded that the parallel preprocessing reaches high speedups for the relevant thresholds.

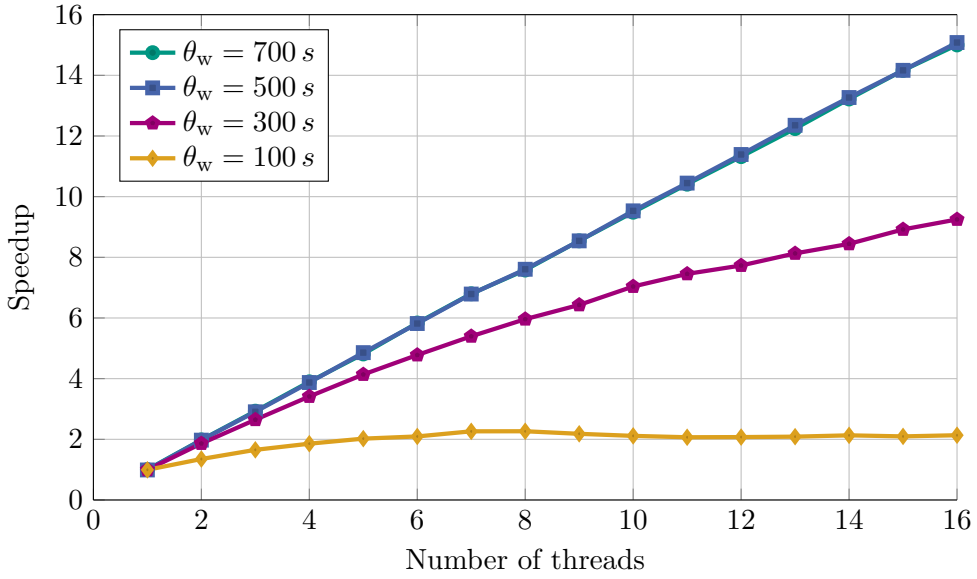


Figure 4.14.: Parallelization speedups for Walking Trip-Based preprocessing using the Bern network with different thresholds.

4.2.2. Walking Trip-Based Query

By presenting the Walking Trip-Based algorithm, it has been shown that the walking query can be solved using the Trip-Based approach. However, the relevance of this result largely depends on the performance of the queries. To evaluate this, query running times are compared to those of Walking McRAPTOR (Section 2.3.1, [DPW15]). Unless otherwise noted, the implementation used for evaluation uses the simple array based walking time data structure (Data Structure 3.3) as well as the second variant of optimized target pruning described in Section 3.1.2.

Figure 4.15 shows results for Bern. The Walking Trip-Based query is consistently faster than the Walking McRAPTOR query. The speedup of Walking Trip-Based compared to Walking McRAPTOR ranges from 1.18 for $\theta_w = 300\,s$ to 2.14 for $\theta_w = 800\,s$. Generally, higher thresholds result in higher speedups. This is analogous to the Walking Trip-Based preprocessing effectiveness: Starting with $\theta_w = 500\,s$, the preprocessing discarded upwards of 98.5 % of original transfers. This corresponds to a speedup between 1.85 and 2.14. For $\theta_w = 400\,s$ transfer reduction effectiveness dropped to 97.8 %. This small drop in effectiveness coincides with a significant drop in speedup to 1.37 compared to 1.85 for $\theta_w = 500\,s$. The speedup seems to be prone to less effective preprocessing. Even slightly less effective preprocessing seems to reduce the speedup significantly. Therefore, the Walking Trip-Based query is dependent on highly effective preprocessing to reach high speedups compared to Walking McRAPTOR. It can also be noted that the point with the lowest speedup, $\theta_w = 300\,s$, corresponds to the point of the highest discrepancy between standard and Walking Trip-Based preprocessing effectiveness, as described in the previous section.

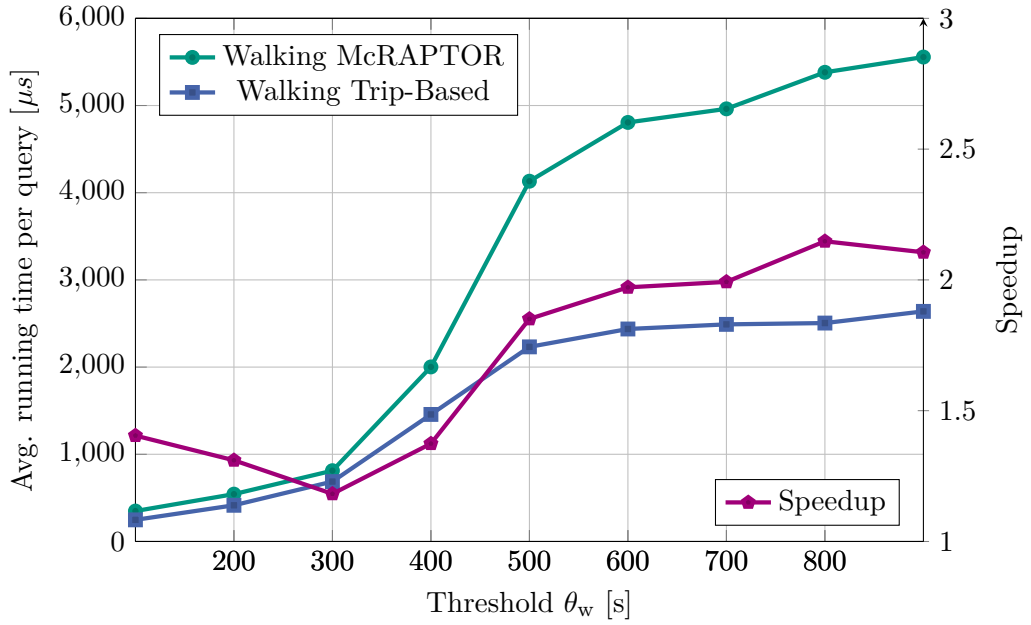


Figure 4.15.: Walking Trip-Based query running times and speedup compared to Walking McRAPTOR for Bern.

However, the preprocessing effectivness is not the only influence that causes varying speedups. For all networks, the highest speedups are obtained where the Walking Trip-Based reduction has the highest factor of reduced transfers more than the standard variant. Additionally to the preprocessing effectiveness, local and target pruning have high influence on the search space that the query has to explore and therefore its speed. This will be analyzed in detail along with the profiling numbers in Table 4.3.

The absolute query running times for the two algorithms show that the Walking Trip-Based algorithm is less prone to becoming slower because of more complex footpaths. It achieves most of its advantage over McRAPTOR between thresholds 300 s and 500 s for both Bern and Switzerland. Here, Walking McRAPTOR running times increase significantly more than Walking Trip-Based running times. For higher thresholds, Walking Trip-Based query running times remain relatively stable. This can be caused by transfer reduction which mitigates the influence of rising thresholds in comparison to Walking McRAPTOR.

Evaluation for Switzerland indicates similar results (Figure 4.16). With speedups between 1.2 and 1.92, Walking Trip-Based again performs better than Walking McRAPTOR. The speedup develops similarly to the speedup achieved for Bern. For $\theta_w = 100$ s, it starts at 1.59 and then declines to its minimum of 1.2 at $\theta_w = 300$ s. With higher thresholds, it rises monotonically. The rising speedup for higher thresholds overlaps with the more effective transfer reduction. The comparably worse speedups around 300s and 400s again coincide with the points of highest discrepancy between the effectiveness of the preprocessing variants.

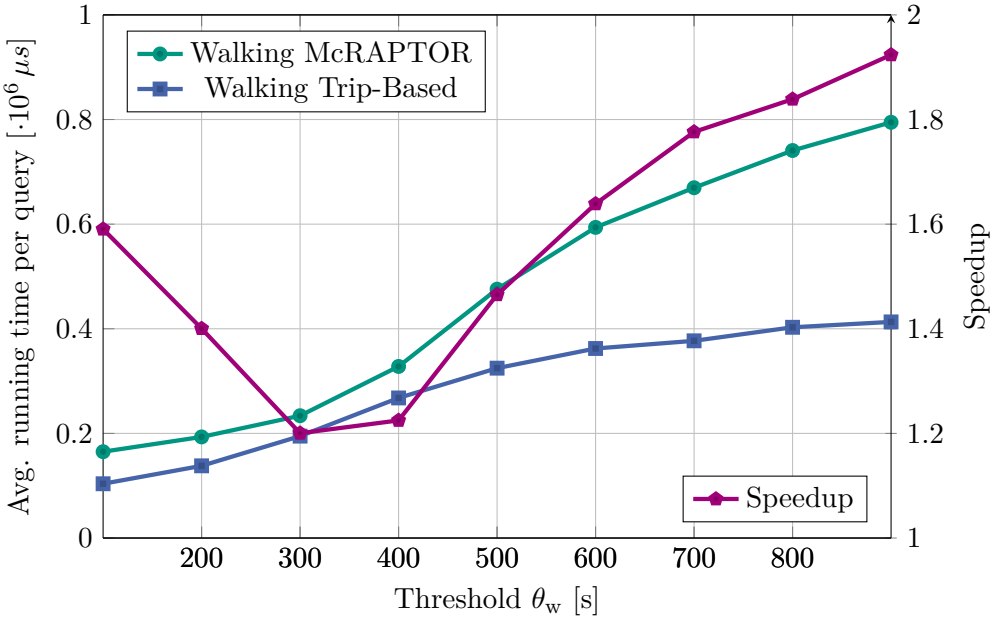


Figure 4.16.: Walking Trip-Based query running times and speedup compared to Walking McRAPTOR for Switzerland.

On the London network, the Walking Trip-Based query generates better speedups than the standard Trip-Based query, ranging between 1.69 for $\theta_w = 150$ s and 3.47 for $\theta_w = 250$ s. Figure 4.17 shows the results. While less data points are evaluated for the London network, the speedup shows a similar trend as for Bern and Switzerland: It first declines for rising thresholds and after its minimum at $\theta_w = 150$ s, it rises again. It should be noted that the achieved speedup of 3.47 requires almost three hours of preprocessing using 128 parallel threads. Compared to that, the speedups between 1.7 and 2.3, achieved for lower thresholds, need below one minute ($\theta_w \leq 150$ s) and around 5 minutes ($\theta_w = 200$ s) of preprocessing time using 128 threads.

It can be concluded that the Walking Trip-Based algorithm is consistently faster than the Walking McRAPTOR algorithm for all networks and all thresholds. The speedup it achieves depends strongly on the network. Generally, higher thresholds improve the speedup. The high speedups for very high thresholds can only be reached with extensive preprocessing.

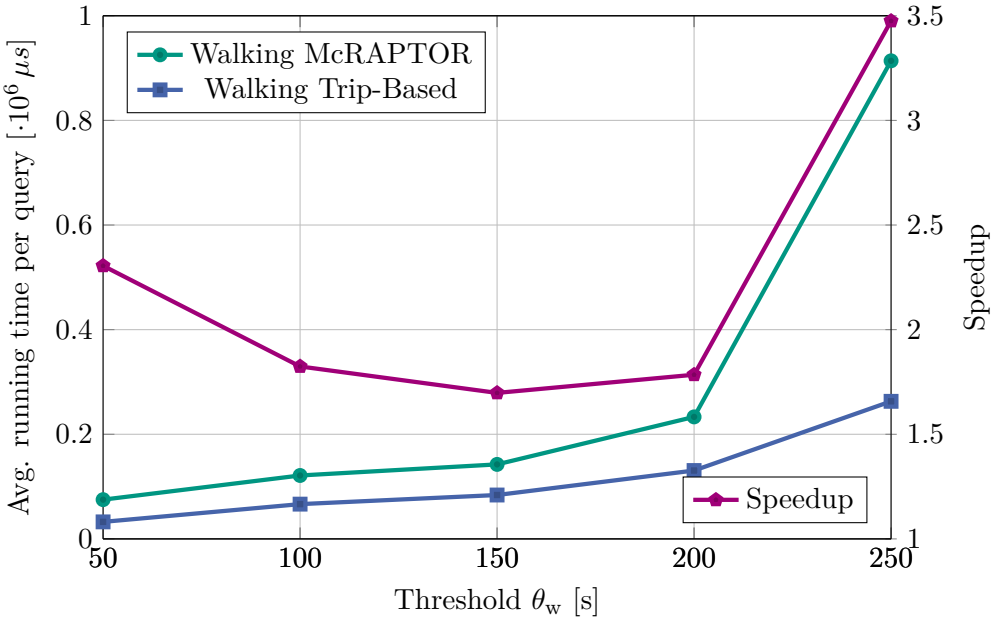


Figure 4.17.: Walking Trip-Based query running times and speedup compared to Walking McRAPTOR for London.

Walking Query Profiling

To better understand which parts of the Walking Trip-Based query are important for fast running times, queries are profiled on the Bern network. Results can be found in Table 4.3. This section uses the terminology introduced in Algorithm 3.2. *Trip segments tested* refers to trip segments for which the ENQUEUE-Operation was called, regardless of whether they were actually enqueued. In contrast, *Trip segments enqueued* refers to trip segments that were enqueued, i.e., trip segments $p_t^b \to p_t^e$ for which their initial stop p_t^b was reached with better walking time than its tentative minimum walking time. All other trip segments are filtered out by local pruning. *Trip segments pruned* is the number of trip segments for which transfers were not scanned because they were pruned by target pruning. *Trip segments explored* is the number of trip segments that are not pruned and from which outgoing transfers are explored.

	Bern			
Walking threshold θ_w [s]	100	300	500	900
Trip segments tested	2 618	28 328	127 235	198 752
Trip segments enqueued	474	1 618	4 475	5 254
Local pruning percentage [%]	81.8	94.2	96.4	97.3
Stops scanned	2 787	8 195	21 777	25 368
Trip segments pruned	190	869	2 967	3 569
Trip segments explored	284	749	1 507	1 684
Target pruning percentage [%]	40.0	53.7	66.3	67.9
Number of trips	5	6	6	6
Number of journeys	3	9	32	37

Table 4.3.: Profiling for Walking Trip-Based queries. Values displayed are averages per query over 10 000 random queries.

Profiling shows that both local pruning and target pruning are important. Local pruning filters between 81 % and 97 % of trip segments for which enqueue was called. The data shows that (1) it is important to keep the minimum walking times in the data structure updated for effective local pruning and (2) the number of accesses to the walking time data structure is high compared to the number of trip segments actually enqueued. Therefore, each individual access must be fast. Target pruning filters between 40 % and 67 % of trip segments. Scanning these trip segments (and other trip segments that might subsequently be enqueued due to this) would cause significantly more work. Therefore, efficient target pruning is important to keep the search space small.

Both pruning techniques are generally more effective for higher thresholds and therefore more complex footpaths. For the Bern network, the lowest speedup was obtained for $\theta_{\mathrm{w}} = 300\,s$. For thresholds above 500 s, the speedup increases significantly. Therefore, comparing the pruning effectiveness for these thresholds can help explain the comparably low speedup at medium thresholds. At $\theta_{\mathrm{w}} = 300\,s$, local pruning already discards 94.2 % of all trip segments that are tested. This is almost as high as for higher thresholds. On the contrary, target pruning only discards 53.7 % of all trip segments before scanning its transfers. This is significantly lower than the 66.3 % and higher achieved for thresholds $\theta_{\mathrm{w}} \geq 500\,s$. Pruning less trip segments enlarges the search space. Therefore, less effective target pruning can be seen as the main reason for comparably low speedups at thresholds around 300 s. This underlines the significance of optimized target pruning, which is already used for this evaluation. Its influence will be evaluated at the end of this section.

Walking Time Data Structure Comparison

The Walking Trip-Based query uses a walking time data structure to keep track of minimum walking times that a stop can be reached with at a trip. Section 3.1.2 described different ways of implementing this data structure. The different versions are experimentally evaluated on the Bern network. Figure 4.18 shows a comparison of total query running times as averages over 10 000 queries without journey reconstruction using the different versions. The most basic version (TB *normal*, Data structure 3.3) that only uses a simple array is the most efficient version for all thresholds. The variant that uses additional timestamps (TB *timestamp*, Data structure 3.4) to avoid resetting all values for each query is slightly slower for all thresholds. The minima-based data structure (TB *minima*, Data structure 3.5) is significantly slower than the normal version. Queries using it perform worse than Walking McRAPTOR queries. This shows that no improvements could be achieved by using more elaborate data structures. The highly cache-efficient array supports the easy operations and linear access patterns best. The benefit of saving time for resetting labels does not compensate for the additional cost per operation induced by timestamps. The operations on the data structure are executed frequently so that each single operation must be fast. The expected overheads described in Section 3.1.2 are not compensated by the reduced time needed to clear the data structure or fewer operations that must be executed. To preserve cache efficiency in the minima-based version, its lists have been implemented using an array. Here, the linear time insert and delete operations far outweigh the advantage obtained from executing fewer operations.[4]

Optimized Target Pruning

In Section 3.1.2, two approaches for more effective target pruning were described. The implementation used for evaluation so far implements the second variant. It stores trip

[4] An alternative implementation using a linked list showed even worse performance. While insert and delete operations only need constant time, chache efficiency is worse compared to the implementation using an array.

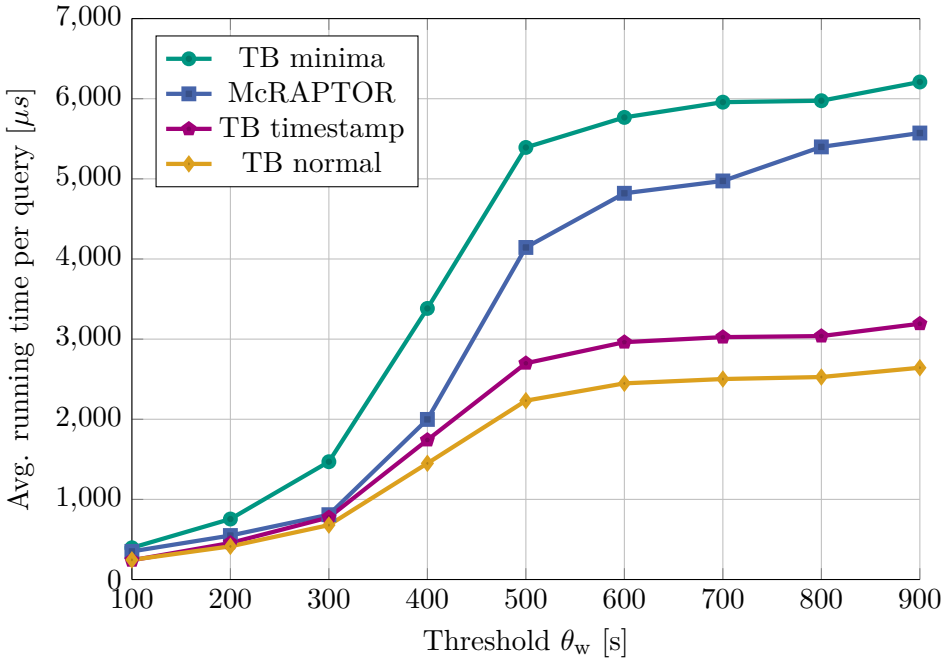


Figure 4.18.: Walking Trip-Based query running times for different implementations of the walking times data structure. Running times are averages per query over 10 000 queries without journey reconstruction.

segments along with the walking times they were enqueued with for the most effective target pruning. The influence of this optimization is evaluated by comparing it with the basic variant described in Algorithm 3.2 using the Bern and Switzerland networks.

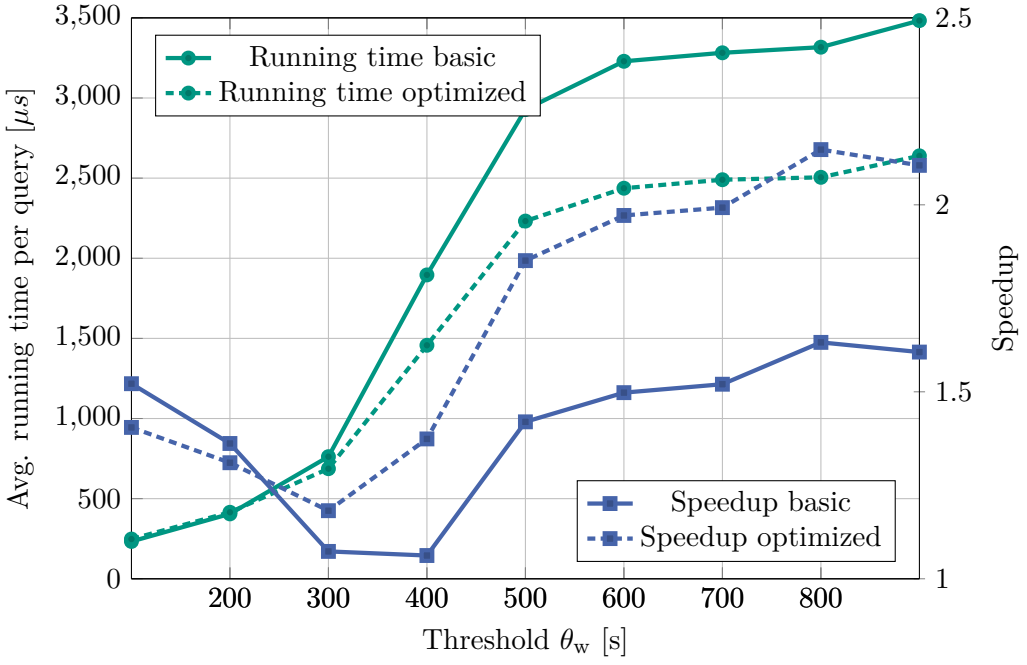


Figure 4.19.: Comparison of Walking Trip-Based query speedups (compared to Walking McRAPTOR) and running times for the optimized variant (dashed) and the basic variant of target pruning (solid) using the Bern network.

Results for Bern (Figure 4.19) show that the query using the less effective basic target pruning is slower than the optimized variant for thresholds above 200 s. It is up to 32 % slower at $\theta_w = 600$ s and stays on that level for all thresholds above 400 s. In comparison to Walking McRAPTOR, the speedup drops significantly compared to the optimized variant. Only for thresholds below 300 s, the basic target pruning variant is slightly faster than the optimized variant. For $\theta_w = 100$ s, the basic variant is 6 % faster, for $\theta_w = 200$ s, it is 2.4 % faster than the optimized variant. To explain the faster running times for low thresholds, trip segments that are not pruned although they could be pruned using the optimized variant must be examined. Pareto optimal journeys can still be found when scanning these trip segments: If p_{tgt} can be reached from a trip segment, the values from the walking time data structure are used to construct the arrival label. If a journey is found for such a trip segment, it leads to better target pruning for other trip segments that are scanned afterwards. Likewise, updating the walking times of more trips earlier in the query improves local pruning. This effect is significant for low thresholds since the search space is small so that small changes lead to improved query running times. Profiling data for queries using basic pruning on Bern is available in Table 4.4. It shows that for $\theta_w = 100$ s, the number of explored trip segments is slightly higher than with the basic variant (Table 4.3). However, the number of scanned stops is lower which explains the better running times, as described above. For higher thresholds, non-optimized target pruning prunes 53.5 % ($\theta_w = 500$ s) and 54.2 % ($\theta_w = 900$ s) of all trip segments. This is significantly less than the 66 % and 67.9 % achieved by optimized target pruning. In consequence, the query with non-optimized target pruning explores 32.6 % and 33.4 % more trip segments in total. Queries become slower even though more efficient local pruning partially compensates for the less efficient target pruning. Between 0.4 and 1.6 percentage points more trip segments are pruned by local pruning. The (unnecessarily) larger search space causes that more trip segments are enqueued with non-optimal walking times and then pruned by local pruning.

	Bern		
Walking threshold θ_w [s]	100	500	900
Trip segments tested	2 726	307 947	373 546
Trip segments enqueued	485	5 936	7 010
Percentage pruned [%]	82.2	98.0	98.1
Stops scanned	2 840	27 524	32 027
Trip segments pruned	190	3 177	3 803
Trip segments explored	294	2 759	3 207
Percentage pruned [%]	39.2	53.5	54.2
Number of trips	5	6	6
Number of journeys	3	32	37

Table 4.4.: Profiling for Walking Trip-Based queries using non-optimized target pruning. Values displayed are averages per query over 10 000 random queries.

For Switzerland (Figure 4.20), the basic variant is between 13.2 % and 42.8 % slower than the optimized variant and the percentage rises monotonically with the walking threshold. The speedup compared to McRAPTOR drops significantly. For $\theta_w = 400$ s, it is smaller than one, i.e., Walking McRAPTOR queries are faster than Walking Trip-Based queries with non-optimized target pruning in that case. The behavior described for low thresholds for Bern does not appear for the Switzerland network.

It can be concluded that the optimized target pruning described in Section 3.1.2 has a large influence on query running times. It increases query speeds for all networks except the small

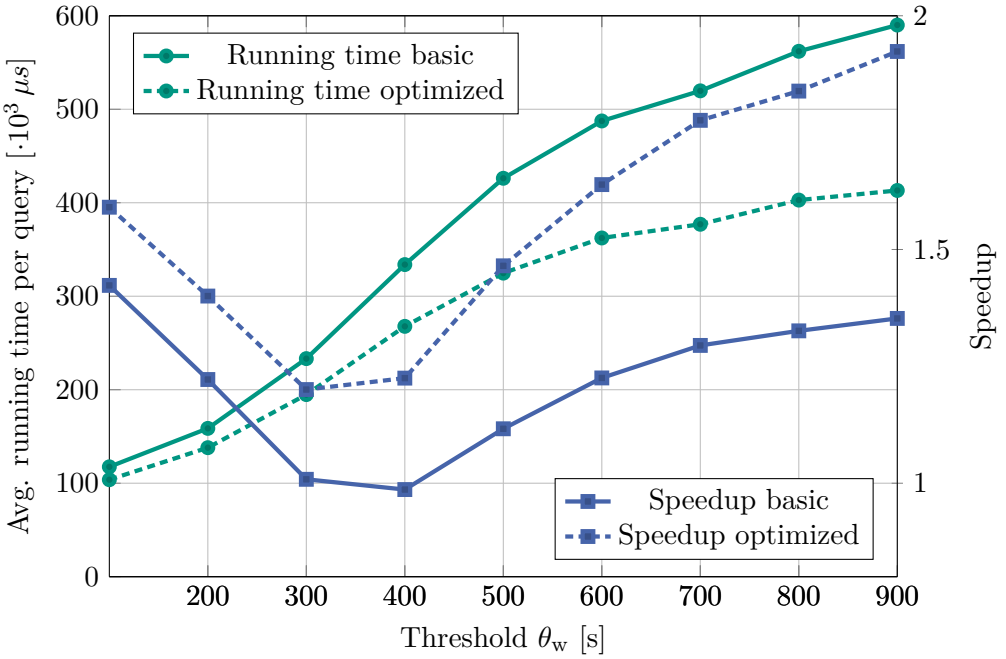


Figure 4.20.: Comparison of Walking Trip-Based query speedups (compared to Walking McRAPTOR) and running times for the optimized variant (dashed) and the basic variant of target pruning (solid) using the Switzerland network.

Bern network with low thresholds. Here, it causes slightly higher running times for low thresholds whereas it significantly reduces running times for higher thresholds. Optimized target pruning significantly improves the speedup compared to Walking McRAPTOR for most cases while only impairing them slightly for some cases. Profiling data shows that the optimization raises the share of trip segments that is pruned by target pruning significantly.

Besides optimizing target pruning, the optimization also allows to use the walking time stored along with a trip segment at any point where $\mathsf{W}(t, \cdot)$ would be needed to scan the trip segment. Besides improving cache locality, the walking time data structure is then solely used for local pruning. This raises the question if the query can become more effective by spending less time on maintaining the walking time data structure. In the version evaluated so far, walking times are updated for all following trips. To save time for this, less trips could be updated. Of course, this reduces local pruning effectiveness in the query. Experimental evaluation has shown that query running times increase when updating (1) no following trip or (2) only a fixed number of following trips. The time needed to update the data structure that can be (partially) saved does not compensate for the time needed to process additional trip segments that are unnecessarily enqueued and scanned. Profiling has shown that a large number of potential trip segments are filtered by local pruning before being enqueued (Table 4.3). Combined with the comparably higher cost for scanning an unnecessary trip segment than updating walking times for a trip, the importance of local pruning explains why this technique does not improve running times.

		Transfers			Running times		
					Prepr.	Query	
Network	θ_w	Original	Reduced	Increase		Trip-B.	RAPTOR
Unit	[s]	[$\cdot 10^6$]	[$\cdot 10^6$]		[h:mm:ss]	[$\cdot 10^3 \mu$s]	[$\cdot 10^3 \mu$s]
Bern	100	1.2	0.3	1.005	0:00:00.36	0.248	0.349
	300	19.5	1.5	1.640	0:00:01	0.687	0.812
	500	298.1	4.3	2.517	0:02:12	2.232	4.132
	900	361.2	4.3	2.646	0:03:20	2.639	5.556
SW	100	40.8	8.2	1.020	0:00:10	103	164
	300	709.5	34.3	1.999	0:00:45	194	233
	500	6 400.7	94.0	3.110	0:14:17	324	475
	900	10 412.8	114.3	3.570	0:36:05	413	794
London	50	31.8	9.7	1.140	0:00:12	32	75
	150	171.8	33.7	1.636	0:00:39	83	142
	250	10 679.3	160.3	2.233	2:57:16	262	913

Table 4.5.: Preprocessing and query results for the Walking Trip-Based algorithm for all networks. *Increase* is the increase in the number of reduced transfers compared to the standard Trip-Based algorithm.

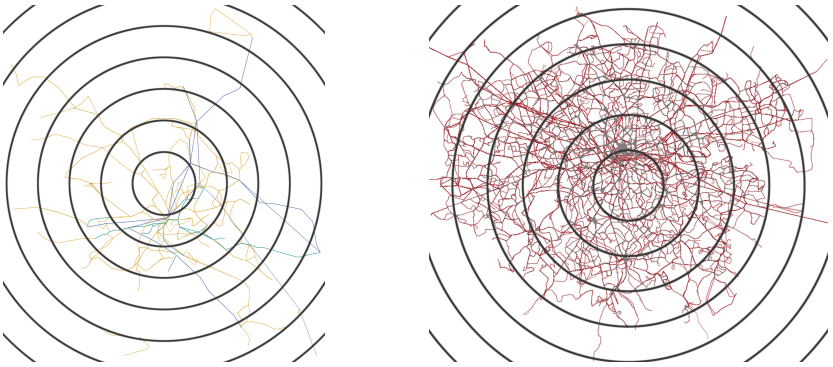
4.3. Fare Zone Trip-Based Algorithm

In this section, the second extension, the Fare Zone Trip-Based algorithm (Section 3.2) will be evaluated. Again, both the preprocessing and the query stage will be considered. Preprocessing is evaluated for transfer reduction effectiveness and running times. Queries are evaluated for their running times in comparison to the Fare Zone McRAPTOR variant (Section 2.3.1, [DPW15]). Since fare zone queries are more relevant to urban transportation networks that use fare zone-based pricing models, the Fare Zone Trip-Based algorithm will only be evaluated using the networks for Bern and London. For comparability, fare zones for both networks are defined as eight circular, ring-shaped fare zones of the same width centered at the center of the network. A visualization of both networks with their fare zone borders can be found in Figure 4.21.

4.3.1. Fare Zone Trip-Based Preprocessing

For the Fare Zone Trip-Based algorithm, more transfers have to be generated in the original transfer creation phase. Table 4.6 shows that for most networks, this increases the number of original transfers by less than one percent. The only cases where significantly more transfers must be generated are Bern networks for thresholds below 300s. Here, 14.92 % ($\theta_w = 100$ s), 8.8 % ($\theta_w = 200$ s) and 1.99 % ($\theta_w = 300$ s) more transfers are created. It should be noted that the first two cases are also those with a low overall number of transfers.

The total preprocessing running time generally increases compared to the standard Trip-Based algorithm. On the other hand, Fare Zone Trip-Based preprocessing is faster than preprocessing for the Walking variant for London with low thresholds. The running times of preprocessing for the three variants can be seen in comparison in Figure 4.22. For Bern and thresholds below 400 s, Fare Zone Trip-Based preprocessing takes between 1.49 and



(a) Bern (b) London

Figure 4.21.: Visualization of fare zones for both networks used for evaluation. Black circles indicate borders between individual fare zones.

2.86 times longer than the standard variant. This is only 1.04 to 1.32 times longer than the Walking Trip-Based preprocessing. For higher thresholds, the difference grows to up to 7.41 times at 900s compared to the standard variant and 2.69 times for the walking variant. On the London network, Fare Zone Trip-Based preprocessing is 2.33 times to 6.37 times slower than the standard variant. On the other hand, it is up to 25 % faster than the Walking variant for thresholds up to 150s. For thresholds above that, it needs up to 2.26 times as much time.

		Original Transfers		
Network	θ_w	Standard	Fare Zone	Increase
Unit	[s]	[$\cdot 10^6$]	[$\cdot 10^6$]	[%]
Bern	100	1.21	1.39	14.92
	300	19.53	19.92	1.99
	900	361.24	362.77	0.42
London	50	31.81	31.82	0.03
	150	171.83	173.47	0.95
	250	10 679.31	10 704.52	0.23

Table 4.6.: Comparison of the number of original transfers for the standard Trip-Based algorithm and the Fare Zone Trip-Based algorithm.

To understand why preprocessing times grow less for smaller thresholds, it is important to consider how transfers behave when optimizing for fare zones. A transfer $p_t^i \mapsto p_u^j$ remains in the same fare zone if $\text{fz}(p_t^i) = \text{fz}(p_u^j)$. Otherwise, it changes fare zones. If a transfer remains in one fare zone, the subset of fare zones does not change when using it (it might still change along the reached trip). Therefore, if more transfers remain in one fare zone, less different fare zone subsets have to be considered during the reduction. In consequence, the Pareto sets of fare zone subsets remain smaller than if more transfers change fare zones, which reduces complexiy. This is obvious in the extreme case: If there is only one fare zone, every transfer remains in this fare zone. This means that the fare zone subset is always the

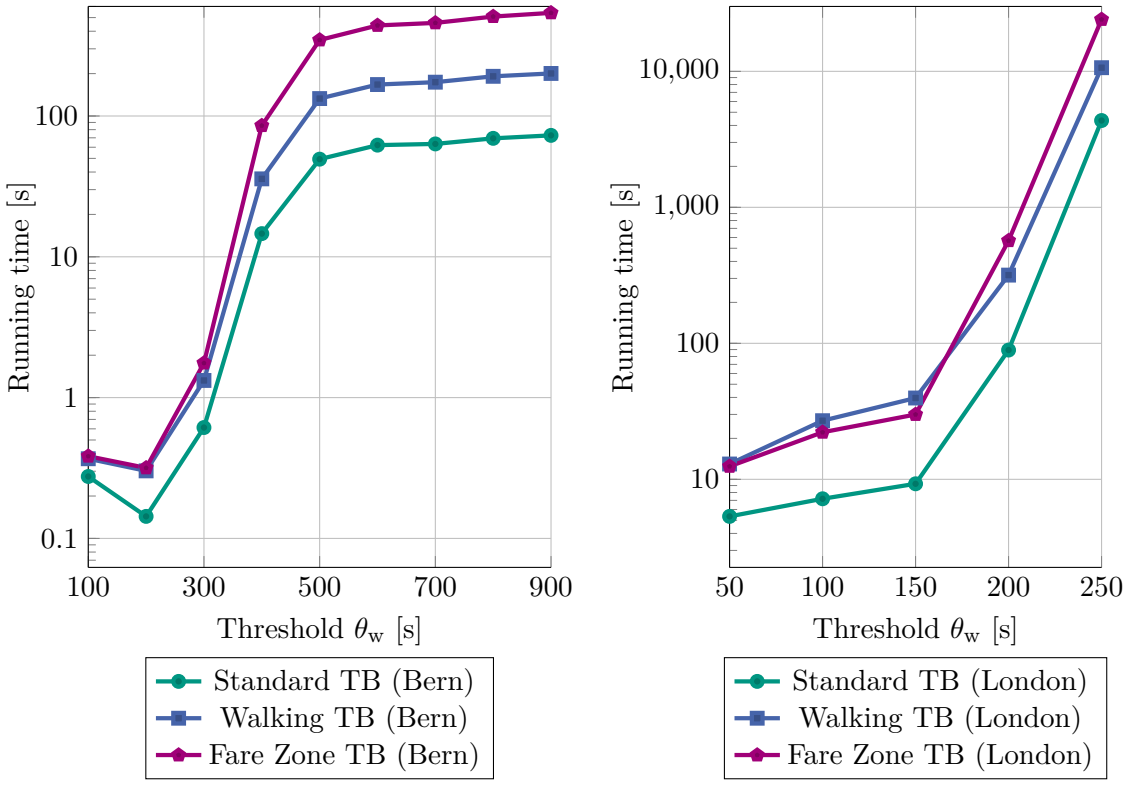


Figure 4.22.: Preprocessing running times compared for the three variants of Trip-Based algorithms for Bern (16 Threads) and London (128 Threads).

same. The Pareto sets have only one element and preprocessing gets nearly as fast as in the standard case.

This effect becomes visible for lower thresholds. Low thresholds mean that most footpaths are short. Since every transfer needs a footpath, most transfers are short, too. Therefore, the probability that a transfer changes fare zones is small. For the Bern network, this effect is small since the network covers a rather small area. The eight fare zones therefore have comparably small width and footpaths changing between fare zones become more likely (see Figure 4.21a). On the other hand, the London network covers a larger area (so that fare zone borders are further apart relativ to the actual size of the network, see Figure 4.21b) and is evaluated for even lower thresholds. For these low thresholds, the effect becomes so strong that preprocessing is faster than in the Walking Trip-Based case. For higher thresholds, more longer footpaths are generated (since more stops are transitively connected) and more transfers can change fare zones. Then, Pareto sets become larger and the running time increases.

Generally, it can be noted that Fare Zone Trip-Based preprocessing is still fast for small thresholds. For Bern, running times are below two seconds for thresholds up to 300s. For London, running times up to $\theta_w = 150\,s$ are below 30s for the highly parallelized variant. For higher thresholds, running times increase significantly. Exact numbers for preprocessing running times and effectiveness can be found in Table 4.7 at the end of this section.

Preprocessing is only useful if it can significantly reduce the number of transfers that the query has to consider. This is first evaluated using the share of transfers that are discarded by the Fare Zone Trip-Based reduction. For Bern, between 75.9 % and 99.1 % of transfers are discarded. As seen for the other versions, the reduction is more effective for higher thresholds. The reduction is even slightly more effective (compared to the higher number of

transfers that it considers) than the Walking variant. This is likely influenced by transfers that do not change fare zones. For London, between 72.7 % and 99.2 % of transfers are discarded. Especially for low thresholds, this is significantly more (up to 8.5 percentage points) than the Walking variant discards. Here, the effect described for preprocessing running times above plays out again.

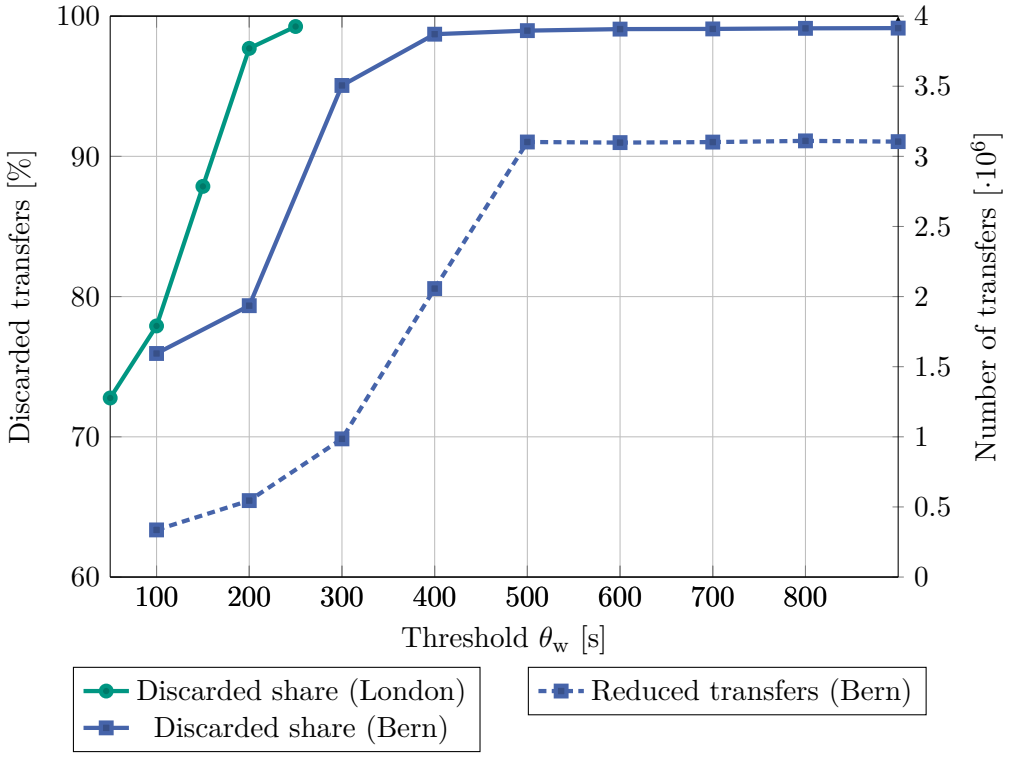


Figure 4.23.: Preprocessing effectiveness for Fare Zone Trip-Based using 8 circular fare zones on London and Bern.

Another sign for the less complex behavior of transfers for low thresholds can be found in Figure 4.24. It shows that for low thresholds, there are only very little additional reduced transfers compared to the standard variant. This is especially significant as the Fare Zone Trip-Based reduction operates on a larger basic set of original transfers. The ratio rises for higher thresholds, but clearly less than for walking times. The Trip-Based preprocessing only considers a well-defined search space when looking for stops for which a transfer is necessary. Because the ring-shaped fare zones used for evaluation here create contiguous fare zones, it is likely that only few different fare zone subsets are ever created during preprocessing (compared to the $2^8 = 256$ that are theoretically possible). In turn, there are less different values for this criterion than for arbitrary walking times. This causes less transfers to be necessary during preprocessing.

It can be concluded that Fare Zone Trip-Based Prepocessing is highly effective. Specifically, spacial locality of fare zones helps limit the number of possible fare zone subsets and small walking thresholds reduce the number of transfers that cross fare zones. In almost all cases, the number of reduced transfers is lower than for the Walking Trip-Based preprocessing.

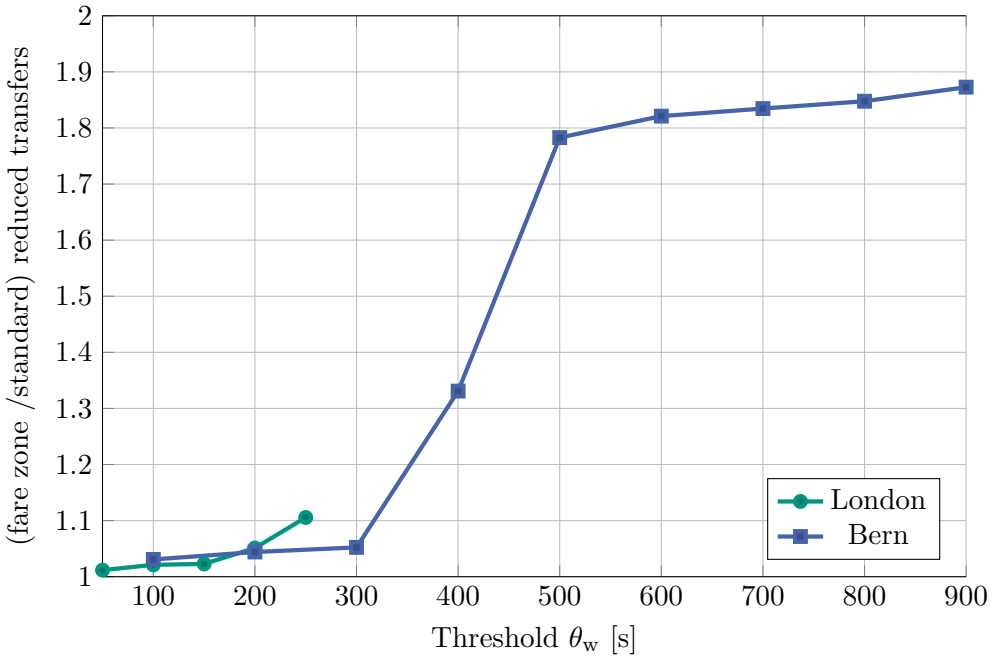


Figure 4.24.: Ratio of reduced transfers for standard Trip-Based and for Fare Zone Trip-Based.

4.3.2. Fare Zone Trip-Based Query

Query running times for the Fare Zone Trip-Based algorithm are considerably slower than those achieved by the Fare Zone McRAPTOR algorithm. Absolute running times as well as the "speedup" obtained by the Fare Zone Trip-Based query (which is consistently below 1) are shown in Figures 4.25 and 4.26. Exact running times are available in Table 4.7. Absolute running times show that Fare Zone Trip-Based queries are slower than the McRAPTOR queries. This is largely independent of the threshold. The speedup improves slightly for higher thresholds. Here, more effective preprocessing might compensate for the slower queries. However, even with the high degrees of transfer reduction that are achieved in these cases, the Fare Zone Trip-Based queries take more than (Bern) or almost (London) twice as long as the McRAPTOR queries.

Even with highly effective preprocessing (often more effective than that of the Walking Trip-Based algorithm), the queries cannot reach the performance of Fare Zone McRAPTOR. This suggests that the query itself is inherently slow, even with effective preprocessing. This shows a discrepancy to both other Trip-Based algorithms considered before. In these cases, the queries were inherently fast enough so that the combination with effective preprocessing resulted in a better performance than RAPTOR or McRAPTOR queries. The results from Section 4.3.1 show that preprocessing itself is highly effective. Therefore, the low speedup cannot be caused by ineffective preprocessing, but must be caused by inherently slow queries.

The evaluation of different data structure variants in Section 4.2.2 and query profiling (Table 4.3) have shown that the Trip-Based Walking query largely depends on fast operations on the individual trips and efficient pruning. All Trip-Based queries must update values along trips. For the Walking Trip-Based queries, the minimum walking time that the trip segment was reached with, is propagated along the trip. This means that the same value is used at each stop. When optimizing for fare zones, the subset of fare zones that must be used at a stop can change at each stop along the trip. This causes an overhead at each

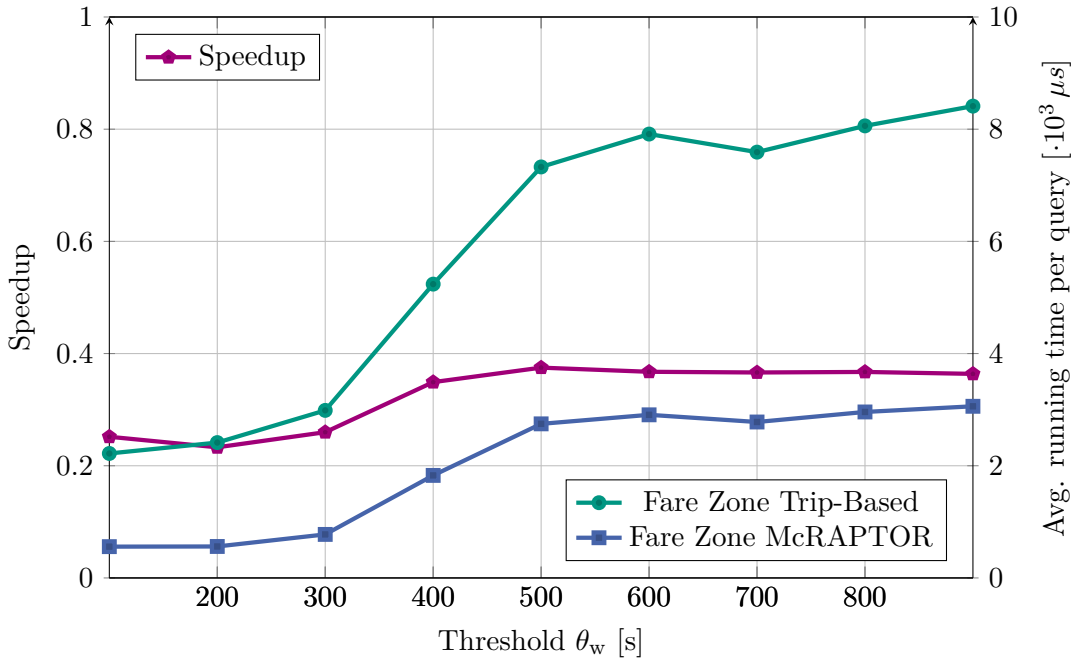


Figure 4.25.: Running times for Fare Zone Trip-Based queries and comparison to Fare Zone McRAPTOR queries for the Bern network.

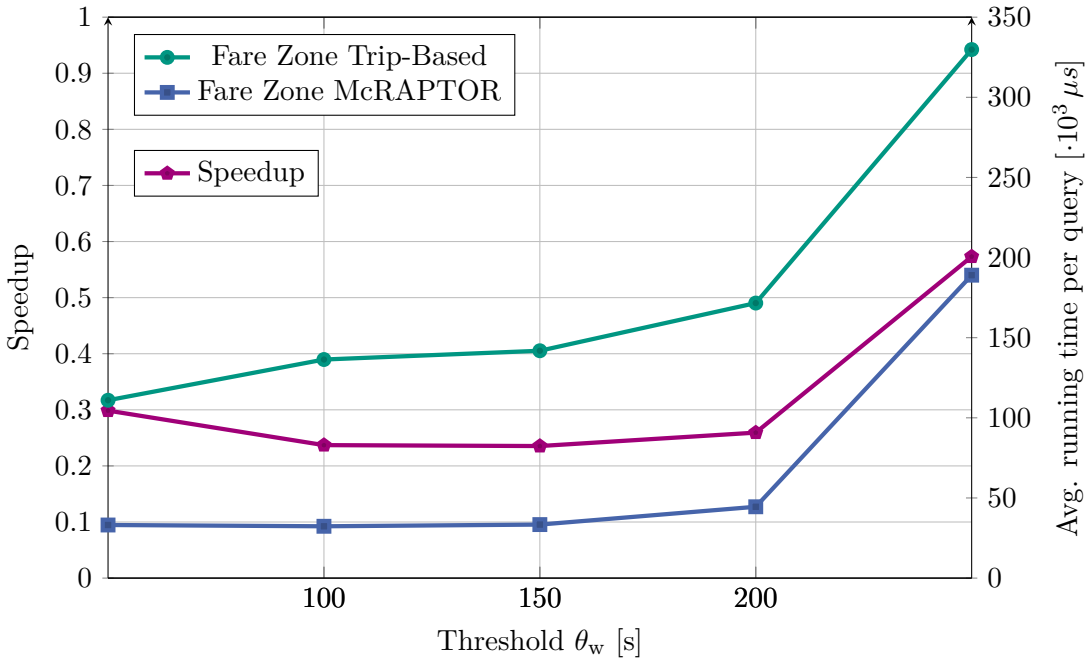


Figure 4.26.: Running times for Fare Zone Trip-Based queries and comparison to Fare Zone McRAPTOR queries for the London network.

stop to obtain and add the relevant fare zones. Cache locality is worse because additional data structures must be accessed. This problem arises whenever scanning stops along trips. Even more significant is that fare zone subsets are not totally ordered. Unlike the Walking Trip-Based query, the Fare Zone Trip-Based query has to keep Pareto set of fare zone subsets at each stop. This makes the frequent enqueue operations significantly more expensive.

Of course, stops also need to be scanned by the McRAPTOR queries. However, they are not scanned on trip level but for every line, and every line is scanned at most once per round (with potentially multiple labels, see Section 2.3.1, [DPW15]). Each line consists of 72 (Bern) or 59 (London) trips on average (see Table 4.1). On the other hand, the number of rounds a Fare Zone McRAPTOR query executes is typically significantly lower. For Bern, there are 5 to 9 rounds on average; for London the maximum number of rounds is 10. Therefore, scanning each line at most once per round instead of scanning each trip potentially multiple times per query is more efficient. The standard and Walking Trip-Based queries take advantage of the more detailed information that can be accessed when scanning stops on the trip level. This allows for efficient pruning and fast scans since arrival times are rolled out into trips and only the first reached index (bicriteria query) or the minimum walking time (walking query) must be considered – so that no Pareto sets must be used. This advantage is lost for fare zone subsets. The more expensive trip segment scans, caused by changing fare zones along a trip and operations on Pareto sets, outweigh the advantage obtained from working on the trip level. Fare Zone McRAPTOR has an advantage because it executes fewer expensive scans.

		Transfers			Running Times		
					Prepr.	Query	
Network	θ_w	Original	Reduced	Increase		Trip-B.	RAPTOR
Unit	[s]	[$\cdot 10^6$]	[$\cdot 10^6$]		[h:mm:ss]	[$\cdot 10^3 \mu$s]	[$\cdot 10^3 \mu$s]
Bern	100	1.3	0.3	1.030	0:00:00.3	2.219	0.558
	300	19.9	0.98	1.052	0:00:01	2.987	0.776
	500	299.5	3.1	1.782	0:05:46	7.327	2.747
	900	362.7	3.1	1.872	0:08:59	8.411	3.060
London	50	31.8	8.6	1.011	0:00:12	110	33
	150	173.4	21.0	1.022	00:29	141	33
	250	10704.5	79.3	1.105	6:41:01	329	189

Table 4.7.: Preprocessing and query results for the Fare Zone Trip-Based algorithm for the Bern and London networks. *Increase* is the increase in the number of reduced transfers compared to the standard Trip-Based algorithm.

5. Conclusion

The conclusion of this work first summarizes the results found in Chapters 3 and 4 and then gives an outlook on further optimizations and future work.

5.1. Summary

The standard Trip-Based Public Transit Routing Algorithm computes Pareto optimal journeys for the two criteria earliest arrival time and number of trips. This work shows that the Trip-Based Public Transit Routing Algorithm can be extended to compute Pareto optimal journeys for more than two criteria. Specifically, two extensions of the algorithm were developed. The Walking Trip-Based algorithm computes Pareto optimal journeys for earliest arrival time, number of trips and minimum walking time. The Fare Zone Trip-Based algorithm optimizes journeys for earliest arrival time, number of trips and minimal subset of used fare zones. Both algorithms were implemented and tested using real world public transit networks.

The preprocessing stage of the Trip-Based algorithm was adapted for both extensions to ensure correctness. For both variants, Pareto sets had to be used to keep track of optimal ways of reaching stops. For the Fare Zone Trip-Based preprocessing, values had to be updated in between stops to ensure their validity – this reflects that fare zones change while using a trip whereas walking times remain constant. Additionally, more original transfers had to be generated for the Fare Zone Trip-Based algorithm.

While both extensions optimize journeys for three criteria, the different additional criterion, minimum walking time versus minimal fare zone subset, required different approaches for the query algorithms. Walking times are totally ordered. Therefore, the query algorithm only needs to store optimum tentative walking times. In contrast, there is no total order on fare zone subsets. This requires the query to maintain Pareto sets of fare zone subsets. Furthermore, walking times only change when using initial or final footpaths or transfers. For one ride of a vehicle, the total walking time remains constant. This makes it easy to scan trip segments in the query. Evaluating walking times can easily be integrated into the query since transfers are processed individually. On the contrary, fare zones do not change during transfers, but while riding a vehicle. This causes significant overhead when scanning trip segments. Fare zones of stops that the trip segment travels through must be obtained and added to the current subset of fare zones. Combined with the Pareto sets of fare zones, this makes trip segment scans for the Fare Zone Trip-Based query significantly

more complex than those for the Walking Trip-Based query. Since the preprocessing and – in case of the Fare Zone variant – the query stage of the Trip-Based algorithm now use full Pareto sets, this work sets the foundation to use the Trip-Based algorithm to optimize more than three criteria.

Evaluation using real world public transit networks showed that the transfer reduction becomes significantly less effective for the Walking Trip-Based algorithm compared to the standard variant. For higher walking thresholds – when more footpaths exist in the network – the number of reduced transfers rises by a factor of up to 2.6 (Bern), 3.5 (Switzerland) and 2.2 (London). Preprocessing for walking takes longer than in the standard variant. For less dense networks, it takes around 3 min (Bern, 16 threads) and 36 min (Switzerland, 128 threads) for high thresholds. With very high thresholds compared to the stop distance in a network, preprocessing becomes slow. While preprocessing for fare zones is faster for very small thresholds, it is slower for most thresholds. However, it is significantly more effective than the walking reduction. For fare zones, only up to 87 % more transfers remain than for the standard Trip-Based algorithm.

The query algorithms of both extensions were compared to implementations of the respective variant of McRAPTOR. An overview of the results for all Trip-Based algorithms studied in this work can be found in Figure 5.1. The Walking Trip-Based algorithm is faster than the Walking McRAPTOR algorithm for all networks and all thresholds. Speedups for the networks used for evaluation range from 1.18 to 3.47. Generally, better speedups are achieved for higher thresholds. Profiling showed that pruning techniques are important for fast queries. Different potential optimizations explored in this work show that Walking Trip-Based queries must be carefully tuned. Even slight improvements in pruning techniques have significantly improved running times. On the other hand, overheads from potential optimizations can seriously deteriorate the performance and a standard technique for route planning algorithms failed to improve running times.

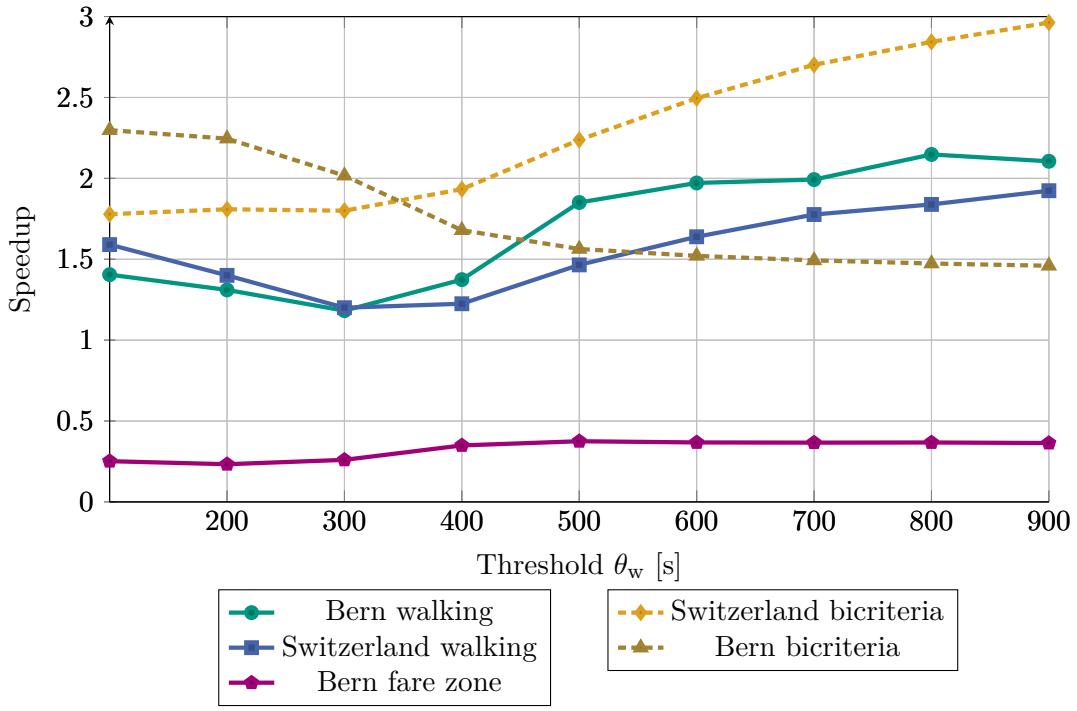


Figure 5.1.: Speedup of all variants of Trip-Based algorithms compared to the relevant variant of RAPTOR or McRAPTOR for Bern and Switzerland. Speedups above 1 indicate that the Trip-Based variant is faster.

Despite its comparably more effective preprocessing, Fare Zone Trip-Based queries cannot achieve similar performance. They take consistently around twice as long as the Fare Zone McRAPTOR queries. It was demonstrated that this is not caused by ineffective preprocessing, but by the more complex operations that the Trip-Based queries have to execute for fare zones. Specifically, Fare Zone McRAPTOR queries have to execute significantly fewer expensive trip scans and are therefore faster in total.

These results show that it is possible to replicate the advantage of the standard Trip-Based algorithm for queries optimizing three criteria. For queries that additionally optimize walking times, the algorithm developed in this work is faster for all thresholds and significantly faster for most thresholds than the Walking McRAPTOR algorithm. However, the performance strongly depends on the optimization criteria. The worse results for the Fare Zone Trip-Based query show that fast trip scans are key to fast queries. This implies that multicriteria Trip-Based queries are most likely to be faster than McRAPTOR queries if (1) they can avoid Pareto sets during queries and (2) if the criterion does not change along trips. This makes multicriteria Trip-Based algorithms attractive for queries optimizing (only) three criteria that are totally ordered and that do not change along trips. Only for these queries, both conditions are met. For other sets of criteria, the more efficient approach of McRAPTOR that scans trips less frequently seems to lead to better running times.

5.2. Outlook

While the Walking Trip-Based query is faster than Walking McRAPTOR for all thresholds, the speedup is low for some thresholds. It was demonstrated that this is partially caused by less effective target pruning for these thresholds. The optimizations for target pruning studied in this work have already significantly improved target pruning effectiveness. However, the share of trip segments pruned by target pruning is still significantly lower than that pruned by local pruning. It could be studied if more granular target pruning can improve running times. Currently, only the arrival time of the first stop of a trip segment is used for target pruning. For trip segments with large differences between arrival times, it could be worth to use target pruning for individual parts of the trip segment.

It was shown that the Trip-Based approach can be used for queries optimizing more than two criteria by adapting the preprocessing- as well as the query-stage of the original algorithm. Both parts of the algorithm raise the possibility to integrate other techniques. The following ideas were developed in collaboration with the first advisor of this work.

The preprocessing part of the algorithms developed in this work checks if a transfer is necessary by exploring a limited range of alternatives. If alternatives yield better results, the transfer is discarded. Since only some alternatives are explored, this does not produce a *minimum* set of reduced transfers. For multi-modal route planning, ULTRA [BBS+19] generates a near minimal set of shortcuts that model transfers for any non-schedule based mode of transportation (e.g., walking) in a public transit network. Sauer et al. have shown that ULTRA preprocessing can be integrated into the standard Trip-Based algorithm [SWZ20]. The resulting algorithm, ULTRA-Trip-Based, uses ULTRA preprocessing and the Trip-Based query. This not only reduces the number of transfers the query has to consider, but also allows the algorithm to work with unrestricted footpaths as opposed to footpaths enclosed under transitivity. For the bicriteria query, ULTRA-Trip-Based achieves a speedup of up to 4 compared to ULTRA-RAPTOR, which uses ULTRA preprocessing for RAPTOR queries [SWZ20]. To further improve the speedup for minimum walking time queries and benefit from unrestricted walking, future works could integrate ULTRA into the Walking Trip-Based Algorithm. Since the Walking Trip-Based transfer reduction is comparably ineffective, this could improve query running times. Optimizing walking times in public

transit networks with footpaths enclosed under transitivity is primarily relevant for networks with very high thresholds. However, the threshold cannot be raised arbitrarily high. The algorithms become too slow for the complex networks. Lifting the artificial restriction from transitive footpaths also allows to find *better* journeys [WZ17, Sau18, BBS$^+$19]. Therefore, allowing unrestricted walking is attractive even beyond reaching better running times.

The results of this work also show that Multicriteria Trip-Based algorithms likely only perform better for a limited set of criteria. More complex criteria, such as minimal fare zone subset, lead to expensive operations, which minimizes the advantage the standard Trip-Based algorithm gains from working on the trip level. McRAPTOR does not suffer from the more expensive operations to the same degree since it executes them less frequently. By integrating ULTRA, McRAPTOR and Trip-Based Routing, the strengths of each algorithm could be combined. A resulting algorithm could use the fast and effective preprocessing from ULTRA for transfers on the trip level. The McRAPTOR query setup could be used to minimize the number of scans along trips. If the Trip-Based setup – scanning transfers from the trip level – could be integrated into the McRAPTOR query, the algorithm could (1) benefit from more effective preprocessing and (2) avoid having to scan through trips of a line to find the first reachable trip. The resulting algorithm could use unrestricted walking and potentially effectively optimize more complex query types, including complex individual criteria or more than three criteria.

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.

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

[DPSW13] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. In *International Symposium on Experimental Algorithms*, pages 43–54. Springer, 2013.

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

[MHS06] Matthias Müller-Hannemann and Mathias Schnee. Paying less for train connections with MOTIS. In *5th Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'05)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[MHSWZ07] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable information: Models and algorithms. In *Algorithmic Methods for Railway Optimization*, pages 67–90. Springer, 2007.

[MS07] Matthias Müller–Hannemann and Mathias Schnee. Finding All Attractive Train Connections by Multi-Criteria Pareto Search. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*, pages 246–263. Springer, 2007.

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

[SWZ20] Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. Integrating ULTRA and Trip-Based Routing. In *20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[Wit15] Sascha Witt. Trip-Based Public Transit Routing. In *Algorithms-ESA 2015*, pages 1025–1036. Springer, 2015.

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

[Zei20] Tim Zeitz. Algorithmen für Routenplanung, 2. Vorlesung. University Lecture, 2020.

Appendix

A. Implementation Details

The implementations of McRAPTOR as well as those of the preprocessing algorithms for Trip-Based are based on existing implementations. Trip Based Query implementations are based on an implementation using unrestricted walking and non-transitive footpath graphs.

Data Structures

Pareto sets have been implemented using a dynamic vector. The insert function scans the existing labels once and removes dominated labels by replacing them with other labels. If no label was found that dominates the new label, it is added. The insert function returns whether the label was inserted (i.e., whether it was not dominated) so that an additional dominance check can be saved.

Similar to [DPW15], *fare zone subsets* were implemented using a 16-bit bitvector. A fare zone can be added using a logical *or*, dominance checks are possible using logical *and*.

Fare zones are stored in a global array which contains an entry with the relevant fare zone ID for each stop ID.

In the *minima-based walking time data structure* (Data Structure 3.5), the relevant tuple for a stop index must be found by linear search over all tuples for the trip. To avoid redundant linear searches when accessing subsequent indices of one trip, the data structure holds an index to the last used tuple. During subsequent accesses, linear searches restart from that index. To achieve this, the query resets the pointer whenever a trip is accessed at an arbitrary index.

Journey Reconstruction

For *McRAPTOR* journey reconstruction, labels of each round must hold a pointer to the label they were reached from. Since labels are kept in Pareto sets where they can change positions, the route scanning step and the footpath evaluation step must now use separate rounds. That way, labels from the route scanning step do not change positions anymore when they are pointed to in labels of stops reached with a footpath. In the route scanning steps, labels from the previous two rounds (that model one round in the original algorithm) must be used.

In both multicriteria *Trip-Based* variants, the queues are replaced by a vector in which all labels are kept until the query is finished. That way, each trip segment label can store a pointer to the trip segment it was reached from. Additionally, they store the transfer they were reached with to correctly reconstruct the footpaths and the first and last stops that were used in each trip segment. Final footpaths (i.e., footpaths that connect to p_{tgt}) must be stored separately per journey since p_{tgt} can be reached multiple times from one trip segment.

Transitive Footpaths

For evaluation, the original footpaths of each network were enclosed under transitivity using Dijkstra's algorithm. Since this can produce up to $\Theta(|\mathcal{S}|^2)$ footpaths, the original footpaths are first filtered by a walking threshold θ_{w} so that only footpaths (p, q) with $\tau_{\text{fp}}(p, q) \leq \theta_{\text{w}}$ remain in $\mathcal{F}'$. A one-to-all Dijkstra query on a graph $(\mathcal{S}, \mathcal{F}')$ from each stop $p \in \mathcal{S}$ yields distances $d_p(q)$ to all stops $q \in \mathcal{S}$. A footpath (p, q) is added to $\mathcal{F}$ for each stop q that has finite distance $d_p(q)$ from p and its walking time is defined as $\tau_{\text{fp}}(p, q) \coloneqq d_p(q)$.