# Lower bounds using the RAPTOR algorithm

**Untere Schranken mit dem RAPTOR Algorithmus**
Bachelor thesis by David Holland (Student ID: 2437778)
Date of submission: September 4, 2023

1. Review: Prof. Dr. Karsten Weihe
Supervisor: Julian Harbarth, M.Sc.
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Algorithmics group

## Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, David Holland, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbtständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 4. September 2023

D. Holland

# Contents

# 1 Abstract

I present an improvement of the RAPTOR algorithm [2], integrating pre-computed *lower bounds* into the algorithm by improving on the *target pruning*, proposed with the original algorithm.

I further develop a modification of the RAPTOR algorithm, to enable it computing said lower bounds itself, with minimal modification to the original RAPTOR algorithm.

The results will be tested against existing RAPTOR and lower bounds implementations within MOTIS, together with real-world public transit data.

The discrepancies of theoretical improvements vs. real-world results will be discussed and a deeper analysis and mitigation attempt of the hindering factors performed.

# 2 Introduction

Avoiding unnecessary computations, and optimizing the runtime efficiency, is one of many objectives when optimizing and improving algorithms, where reducing the amount of data to process and throwing out non-improving ”branches” as early as possible, is important for a fast runtime.

I therefore discuss the utilization and computation of lower bounds with the RAPTOR algorithm [2], for solving the earliest arrival problem with the least amount of transfers on a public transport network. This main topic can be split into three separate parts.
How can pre-computed lower bounds (minimum required travel time between stops) be used *within* the RAPTOR algorithm, and how can they improve it?
How can the lower bounds be calculated *using* the RAPTOR algorithm?
How do the lower bounds computed by the RAPTOR algorithm compare to lower bounds computed by a Dijkstra algorithm?
How do the changes perform in comparison (tested with real-world data)?

The idea for using lower bounds seems pretty simple. Using lower bounds, which are already pre-calculated, to condense the data to be processed by the algorithm, should result in a reduced runtime of the algorithm. However, as I will show in 6, there are more factors to this theoretical improvement than meets the eye. Furthermore, pre-calculating these bounds with the RAPTOR algorithm itself is not easily adaptable from other approaches or other shortest path algorithms, as RAPTOR uses a completely different data structure and calculation method. A new adaptation of the original RAPTOR is presented, calculating the desired lower bounds. Calculating the lower bounds using the RAPTOR algorithm natively, should allow closer integration and optimization with the algorithm, as well as less overhead. Finally, the difference between the RAPTOR approach for lower bounds calculation and other approaches (e.g. the Dijkstra one), in regard to runtime, overhead and improvement of the actual algorithm, isn’t obvious and needs to be evaluated.

Furthermore, these findings aim to aid the MOTIS project goal, in providing real-time routing for public transport, by reducing calculation time per routing request and providing means of isolating intensive computations to be potentially pre-computed.

Each part of the research question will first be presented and evaluated one after another. The results of the test scenarios with real-world public transit data will be discussed, as well as some of the issues that made them not perform up to expectations and investigations into some of the side effects, hindering optimal performance and limiting their improvement potential.

# 3 Related Work

This thesis is majorly based on the RAPTOR paper [2]. Furthermore, the evaluation is performed as an implementation within *MOTIS* (Multi Objective Travel Information System)[1] [5]. Other related work mainly focuses on improving and speeding up algorithms.

Round-Based Public Transit Routing (RAPTOR)[2]
: This work introduces a novel round-based algorithm for solving the earliest arrival problem for public transit networks. It doesn't utilize the commonly used Dijkstra algorithm, but rather takes a new approach. RAPTOR doesn't need preprocessing steps and can therefore be very flexible and dynamic and enable interactive use cases, as it has shown to be a lot faster than similar preceding approaches.
The algorithm proposes some improvements to the base version, including the so-called *target pruning*, which this thesis aims to extend and improve.

Dijkstra's shortest path algorithm[4]
: This is Dijkstra's original description of his famous shortest-path algorithm. The algorithm operates on a weighted graph consisting of nodes and edges and calculates the shortest path from an origin point to a target point on the graph.
It is used to calculate the lower bounds from a *constant graph* within MOTIS and used as a base implementation of lower bounds calculation to compare against in this thesis.

Fully Realistic Multi-Criteria Timetable Information Systems (inc. MOTIS)[5]
: Among other things, this work introduces MOTIS, a realistic traffic information system, working on timetables under multiple criteria. This work also introduces their multi-criteria Dijkstra algorithm, fittingly called MOTIS-algorithm. It opens up a magnitude of possibilities, including several speed-up techniques, like the utilization of lower bounds.
It is the framework I implemented all my approaches in and used as a platform to conduct my evaluation on. The MOTIS project, implementing [5], already has a RAPTOR implementation, which I used as a baseline for my evaluation.

Exact and Approximate Distances in Graphs[6]
: This work provides an overview of solutions for the shortest path problems and outlines different approaches and open questions.
Among other things, the survey presents the concept of *approximate* distances. This is very similar in concept to using a series of minimum travel times, instead of trips in 5.2 to approximate the minimum time needed to travel from a current to a target stop, willingly accepting an error in the process.

Combining Hierarchical and Goal-Directed Speed-up Techniques for Dijkstra's Algorithm[1]
: This work outlines multiple possible speed-up techniques for a Dijkstra algorithm in context of transportation networks.

[1] Project repository: https://github.com/motis-project/motis

Especially the part about *Goal-Directed Approaches* is interesting, as the general idea of this method is to direct the search in the direction of the target. This is something [2] does with *target pruning* and this thesis tries to improve on.

Faster Transit Routing by Hyper Partitioning[3]

This work's aim is in enabling the utilization of the RAPTOR algorithm with large networks, too big for the standard RAPTOR version.

It could be beneficial to check whether the approaches presented in this thesis can in some form be applied to the concepts introduced in this work.

# 4 Foundations

This thesis' main focus is laid on both improving the RAPTOR algorithm itself, by extending it to enable the usage of lower bounds, and calculating said lower bounds using the RAPTOR algorithm. In the evaluation chapter, the improved *RAPTOR with lower bounds* will be compared to its original counterpart, and the *computation of lower bounds using RAPTOR* to a Dijkstra approach, used in the MOTIS project, which in turn is used as the framework for conducting my evaluation.

To aid in the better understanding of the concepts discussed in this thesis, I will summarize the RAPTOR algorithm, as well as gloss over the method of calculating lower bounds with a Dijkstra algorithm. Note that while some discrepancies between this work and the RAPTOR paper [2] are possible, especially regarding naming conventions, the following parts are a summary and therefore greatly influenced by the work being summarized, especially in the structure of the pseudocode.

## 4.1 RAPTOR

Contrary to the working principles of a Dijkstra algorithm, an algorithm often used as a shortest-path algorithm, the RAPTOR algorithm doesn't work on a graph, but rather uses a completely different approach for modelling the transit network. The RAPTOR algorithm tries to improve on the Dijkstra algorithm for *public transit routing* regarding the *earliest arrival problem*, by constructing a novel algorithm which optimizes after two parameters, namely *the number of transfers* and the *arrival time*. This RAPTOR algorithm "is not based on Dijkstra's algorithm"[2, Section 3] and even drops the usage of a graph-based data structure and instead employs a completely new structure, which models the real-world properties of *public transit networks* much closer and "does not even need a priority queue"[2, Section 3]. The algorithm doesn't traverse a graph until all nodes are visited, like the Dijkstra does, but instead operates in rounds over the underlying data until no improvement is possible, to compute optimal journeys.

### 4.1.1 The data model

The network (e.g. public transit network) is modeled as a timetable $(\Pi, \mathcal{S}, \mathcal{R}, \mathcal{T}, \mathcal{F})$, consisting mainly of an operation time $\Pi \subset \mathbb{N}_0$, a set of stops $\mathcal{P}$, routes $\mathcal{R}$, trips $\mathcal{T}$ and footpaths/transfers $\mathcal{F}$.

**Stops** $p \in \mathcal{P}$

model a topographical point where one can board and alight means of transportation.
A good visualization of stops in the real-world could be bus stops, train stations, etc.
In case of graph-based solutions for calculation of the shortest path, these stops are often represented by *nodes*.

**Routes** $r \in \mathcal{R}$

model the topographical paths, which the means of transportation follow.
A route consists of an ordered series of stops

$$(p_0, p_1, \ldots, p_n) \mid p_i \in \mathcal{P},\ p_i \prec p_{i+1}$$

where $p \prec p' \equiv p' \succ p$ models a stop $p$ being an earlier stop than $p'$.
A good visualization for this concept in the real-world are bus lines, train routes, etc. Note that two different routes $r_1, r_2 \in \mathcal{R} \mid r_1 \neq r_2$ could both contain the same stop $p \in r_1 \wedge p \in r_2$. This is intentional and in other words, both routes serve the same stop $r_1, r_2 \in R_p$, where $R_p$ is the set of routes serving the stop $p$. This models the way you have access to different bus routes at a bus station. Also, these shared stops are the (only) connection points between different routes.
These routes are often represented by the *edges* in a graph.

**Trips** $t \in \mathcal{T}$

are a notable difference between a graph-based structure and the RAPTOR one. The trips model the actual trips a mean of transportation, e.g. a bus or train, takes on the route.

> In the original RAPTOR paper [2], routes were simply defined as a group of trips that follow the same stops. For getting a sequence of trips, ordered from earliest to latest, following a route, the definition $\mathcal{T}(r) = (t_0, t_1, \ldots, t_{|\mathcal{T}(r)-1|})$ was used. In this summary and the following chapters, I also use the convention that trips *follow a route*, but more in the way that trips are concrete instantiations of a route, which in the end boils down to the same concept.

These trips extend the concept of routes by having an associated *arrival* and *departure time* for each stop $p_j$ in a trip $t_i$ $\text{arr}(t_i, p_j), \text{dep}(t_i, p_j) \in \Pi$. If you imagine looking at a bus plan, the listing of the bus lines' stops informs about the *routes*, whereas the table of departure (and arrival) times informs you about the *actual trips* you can take. If you're at the bus stop at a specific time, and you check the plan for the next bus you can catch, you're checking for the next *trip* you can catch, in this model.
This concept doesn't really have a straightforward equivalent when using a graph, but rather needs to be modelled by weighted edges, for example, which commonly only model *travel time* between two nodes (stops), instead of specific trips, or by evaluating each edge for a specific arrival time, which [2] calls *Time-Dijkstra*. In the RAPTOR algorithm, however, they are a central concept.

**Footpaths / Transfers** $\langle p, p' \rangle \in \mathcal{F}$

consist of a pair of stops $\langle p, p' \rangle$, which model a direct connection between $p$ and $p'$, independent of routes and trips. This independence is underlined further, in that footpaths don't have *arrival* and *departure times* associated with them, but rather define a constant *walking time* (or *transfer time*) $l(p, p')$, which models the time it takes to walk / transfer from $p$ to $p'$, e.g. by walking.
Footpaths are also transitive, which means that if $\langle p_i, p_j \rangle \in \mathcal{F}$ is a footpath and $\langle p_j, p_k \rangle \in \mathcal{F}$ is a footpath, $\langle p_i, p_k \rangle \in \mathcal{F}$ is also one.

To summarize this, two stops $p_1, p_2 \in \mathcal{P}$ are either connected

- *directly* by a footpath $\langle p_1, p_2 \rangle \in \mathcal{F} \vee \langle p_2, p_1 \rangle \in \mathcal{F}$
- *directly* by a route which serves both stops $\exists r \in \mathcal{R} : (r \in \mathcal{R}_{p_1} \wedge r \in \mathcal{R}_{p_2})$
- *indirectly* through other stops, which eventually have a direct connection
- *not at all* (although the network would probably be separated then, but in theory it's possible)

You can then run the algorithm on this timetable to construct a journey $\mathcal{J}$, consisting of an orderly series of trips and transfers at specific stops, in order to get the actual travel path from one stop to another.

As already mentioned previously, the algorithm runs in rounds over this timetable, where each round $k$ signifies a trip taken on the journey, so the journey consists of $k$ trips and one less transfer $k-1$ (alighting one trip and boarding another).

The RAPTOR paper [2] mentions multiple possible problems and optimization criterions to solve. The focus in this work lies in the *Earliest Arrival Problem*, with optimizations in regard to the earliest arrival time.

### 4.1.2 The algorithm

#### Preliminaries

The essential part of the algorithm is operating in rounds over the data and storing time values $\tau_k(p) \in \Pi$ for each round $k$ and stop $p$. When no further optimization is possible, the algorithm terminates. Furthermore, a time $\tau_{\text{min}}(p)$ is kept for each stop $p$, which denotes the *earliest known arrival time* at stop $p$ up until the current round.

> Note that, technically, this is already an optimization proposed by the RAPTOR paper [2, *Section 3.1*].

An algorithm run is done for a starting stop $p_s$ and a starting time $\tau_{\text{init}}$, where $\text{dep}(\cdot, p_s) \geq \tau_{\text{init}}$.

**Input:** Source stop $p_s$
**Input:** Travel departure time $\tau_{\text{init}}$

> Technically, an ending stop (or target stop) $p_t$ is also known and used, but this also is an optimization proposed in the RAPTOR paper [2, *Section 3.1*], called *target pruning*, and to differentiate between using and not using it, is of importance later on in this work.

#### Initialization

To start and initialize the algorithm, every $\tau_\cdot(\cdot)$ is set to $\infty$, because no (earliest) arrival time is known yet and calculating it is the aim of the algorithm.

If, after the termination of the algorithm, all labels $\tau_\cdot(p)$ are still $\infty$ (alternatively $\tau_{\text{min}} = \infty$), it means the stop $p$ is unreachable (from the starting stop $p_s$).

**foreach** $p \in \mathcal{P}$ **do**
  **forall** $i$ **do**
    $\tau_i(p) \leftarrow \infty$;
  $\tau_{\text{min}}(p) \leftarrow \infty$;

### Starting conditions

First off, the departure time needs to be set somehow. As the algorithm operates in rounds $k$, starting with *round 1*, it is only logical to set the label $\tau_0(p_s)$ as we haven't made a trip yet ($k = 0$) but already have a time. As the algorithm uses the times of previous rounds, this makes sure, we have exactly that for the first round $k = 1$ and the origin stop $p_s$.

Strictly speaking, the further initialization of $\tau_{\text{min}}(p_s)$ wouldn't be necessary, as calculating the fastest journey from $p_s$ to $p_s$ is trivial and $\tau_{\text{min}}(p_s)$ would never be accessed, but for the sake of consistency with the later update procedure to the labels, we initialize that too.

We also use a new set $\mathcal{M}$ to *mark* specific stops / store *marked* stops.

> Note that this, *again*, is an optimization of the bare-bones algorithm, proposed in the paper [2, *Section 3.1*].

---

$\tau_0(p_s) \leftarrow \tau_{\text{init}}$;
$\tau_{\text{min}} \leftarrow \tau_0(p_s)$;
$\mathcal{M} \leftarrow \{p_s\}$;

---

### Main loop

We then enter the main loop, incrementing $k$ each round. Note that this is an infinite loop that only terminates when the termination condition is fulfilled at the end of an iteration.

---

**foreach** $k \leftarrow 1, 2, \ldots$ **do**
    `// Handle marked stops` $p$
    [...];
    `// Handle each queued route`
    [...];
    `// Footpath or transport faster?`
    [...];
    `// Termination condition`
    **if** $\mathcal{M} = \emptyset$ **then**
        **return**;

---

### Termination condition

To understand the termination condition, it is important to know that in each iteration, a stop $p$ is *marked*

$$\mathcal{M} \leftarrow \mathcal{M} \cup p$$

exactly when an improvement was made, in other words, if a faster routing has been found. The stop is then *unmarked*, upon handling being finished. The marking of a stop therefore operates as a kind of "dirty flag". In conclusion, if no stop has been marked during an iteration, no improvements were made.

If we "consider a route whose last improvement happened at round $k' < k-1$", and "[t]he route was visited again during round $k' + 1 < k$, and no stop along the route improved", then the RAPTOR paper now argues, that "[t]here is no point in traversing it again until at least one of its stops improves"[2, *Section 3.1*]. In other words, if no improvement was made for a route in a round, no improvement will be made for the route in further rounds, if the stop isn't improved by other means e.g. by processing another route. If no improvement was made collectively for *all* routes (no stop was marked in a round), improvement in further rounds is therefore impossible, and the algorithm can terminate.

### Handling marked stops

As a next step, for each *marked* stop $\forall p \in \mathcal{M}$, we add every route $r$ serving the stop $\forall \in \mathcal{R}_p$ to a queue

$$\mathcal{Q} \leftarrow \mathcal{Q} \cup \langle r, p \rangle$$

After we have added all serving routes for a stop, we can *unmark* the stop

$$\mathcal{M} \leftarrow \mathcal{M} \setminus p$$

as we make use of the queue for the rest of the iteration.

In this step, we only want to save the *earliest* stop of a route, though. This is, because we will eventually traverse each stop on a queued route (in order, starting with said "earliest stop"), so queuing duplicates of a route at this point is redundant.

In particular, this means checking if a route-stop pair $\langle r, p' \rangle$, with the current route $r$ and a *different* stop $p'$, is already present in the queue

$$\exists p' \in \mathcal{P} : \langle r, p' \rangle \in \mathcal{Q} \mid p' \neq p$$

If this is indeed the case, and the currently processed stop $p$ precedes the other stop $p'$, the already present pair is substituted

$$\mathcal{Q} \leftarrow (\mathcal{Q} \setminus \langle r, p' \rangle) \cup \langle r, p \rangle$$

---

**foreach** $k \leftarrow 1, 2, \ldots$ **do**
  $\mathcal{Q} \leftarrow \emptyset$;
  **foreach** $p \in \mathcal{M}$ **do**
    **foreach** $r \in \mathcal{R}_p$ **do**
      **if** $\exists p' \in \mathcal{P} : \langle r, p' \rangle \in \mathcal{Q} \mid p' \neq p$ **then**
        **if** $p \prec p'$ **then**
          $\mathcal{Q} \leftarrow (\mathcal{Q} \setminus \langle r, p' \rangle) \cup \langle r, p \rangle$;
      **else**
        $\mathcal{Q} \leftarrow \mathcal{Q} \cup \langle r, p \rangle$;
    $\mathcal{M} \leftarrow \mathcal{M} \setminus p$;
  [...];

---

### Handle the queue

Next, we process every queued pair of routes and stops $\langle r, p_{hop} \rangle \in \mathcal{Q}$. We traverse the route $r$ in order and start with the associated stop $p_{hop}$.

> As the RAPTOR paper explains in the **Improvements** section [2, *Section 3.1*], the marked and therefore queued stops are the (earliest) ones, where a potential transfer and boarding ("hopping on") of a trip is possible. This implies the sufficiency of traversing the route (for each $p_i \in r$) only from the earliest marked stop onwards ($i \in (hop, |r|]$).

To proceed further, we are in need of determining the *earliest trip* $\text{et}(r, p)$ at a stop $p$ for $r$, which is the earliest $t$ where the departure time $\text{dep}(t, p)$ isn't earlier than the arrival in the previous round $\tau_{k-1}(p)$ ($\tau_0(p_s)$ is needed here!)

$$\text{dep}(t, p_i) \geq \tau_{k-1}(p_i)$$

If no such trip exists, the earliest trip is undefined $\text{et}(r, p) = \square$.

The *current trip* $t$ (originally undefined $\square$) is set to this $\text{et}(r, p)$. When an earlier trip can be caught at a stop, $t$ is updated accordingly.

If the arrival time at a stop $p_i$ for the current trip $t$ is now determined to be faster than the *earliest known arrival time* at this stop

$$\text{arr}(t, p_i) < \tau_{\text{min}}(p_i)$$

the current round time $\tau_k(p_i)$, as well as $\tau_{\text{min}}(p_i)$, are updated and the stop is marked

$$\mathcal{M} \leftarrow \mathcal{M} \cup p_i$$

---

**foreach** $k \leftarrow 1, 2, \ldots$ **do**
    [...];
    **foreach** $\langle r, p_{hop} \rangle \in \mathcal{Q}$ **do**
        $t \leftarrow \square$;
        **foreach** $p_i \in r \mid i \in (hop, |r|],\ p_i \prec p_{i+1}$ **do**
            **if** $t \neq \square$ **then**
                **if** $\text{arr}(t, p_i) < \tau_{\text{min}}(p_i)$ **then**
                    $\tau_k(p_i), \tau_{\text{min}}(p_i) \leftarrow \text{arr}(t, p_i)$;
                    $\mathcal{M} \leftarrow \mathcal{M} \cup p_i$;
            **if** $\text{dep}(t, p_i) \geq \tau_{k-1}(p_i)$ **then**
                $t \leftarrow \text{et}(r, p_i)$;
    [...];

---

### Checking footpaths

Finally, as mentioned in 4.1.1, aside from routes, stops can also be connected directly by footpaths. Therefore, for every marked stop $p$, all stops $p'$ are processed, which are accessible from $p$ in the form of a footpath

$$\langle p, p' \rangle \in \mathcal{F} \mid p \in \mathcal{M}$$

The label for $p'$, is updated with the fastest time, either achieved by transportation or by foot, whichever is lowest

$$\tau_k(p') \leftarrow \min\{\tau_k(p'), \tau_k(p) + l(p, p')\}$$

The stop is then marked.

---

**foreach** $k \leftarrow 1, 2, \ldots$ **do**
    [...];
    **foreach** $\langle p, p' \rangle \in F \mid p \in \mathcal{M}$ **do**
        $\tau_k(p') \leftarrow \min\{\tau_k(p'), \tau_k(p) + l(p, p')\}$;
        $\mathcal{M} \leftarrow \mathcal{M} \cup p'$;
    [...];

---

#### Improvement: target pruning

The so-called *target pruning* is one of the improvements proposed in the RAPTOR paper [2]. Some other improvements have already been implemented and mentioned in the summary above, but I chose to single this one out, as it becomes particularly relevant in my approach to utilizing lower bounds in the RAPTOR algorithm.

The paper describes, in the **Improvements** section, that the "RAPTOR does not exploit the fact that we are only interested in journeys to a target stop $p_t$"[2, Section 3.1]. This makes the version of the algorithm described above compute journeys from stop $p_s$ to all other stops without any regard for a desired destination. The paper hereafter describes the *target pruning* improvement to the algorithm.

When handling the queue, as described in 4.1.2, and traversing the stops on a route, we can, in addition to comparing $\text{arr}(t, p_i)$ to the $\tau_{\min}(p_i)$ for that stop, compare it to the $\tau_{\min}(p_t)$ at the target stop! If the arrival time is both lower than $\tau_{\min}(p_i)$ *and* $\tau_{\min}(p_t)$, an improvement was made, and the values can be updated accordingly.

This is sound, as an arrival time $\text{arr}(t, p_i)$ bigger than the fastest known time at the target is *not an improvement* in regard to the *Earliest Arrival Problem*. In other words, a faster journey to the target has already been found, so pursuing this slower one is of no use, therefore updating the values and marking the stop can and should be omitted.

Instead of now checking for both conditions, it is easier to just get the lower value of both of them

$$\min\{\tau_{\min}(p_i), \tau_{\min}(p_t)\}$$

as we check for the arrival time being *lower* anyway.

[...];
**Input:** Target stop $p_t$
[...];
**foreach** $k \leftarrow 1, 2, \ldots$ **do**
  [...];
  **foreach** $\langle r, p_{hop} \rangle \in \mathcal{Q}$ **do**
    [...];
    **foreach** $p_i \in r \mid i \in (hop, |r|],\ p_i \prec p_{i+1}$ **do**
      **if** $t \neq \square$ **then**
        **if** $\text{arr}(t, p_i) < \min\{\tau_{\min}(p_i), \tau_{\min}(p_t)\}$ **then**
          $\tau_k(p_i), \tau_{\min}(p_i) \leftarrow \text{arr}(t, p_i)$;
          $\mathcal{M} \leftarrow \mathcal{M} \cup p_i$;
      **if** $\text{dep}(t, p_i) \geq \tau_{k-1}(p_i)$ **then**
        $t \leftarrow \text{et}(r, p_i)$;
  [...];

## 4.2 Lower bounds with Dijkstra

The Dijkstra algorithm is one of the most popular algorithms when trying to "[F]ind the path of minimum total length between two given nodes"[4, p. 2], also called a *shortest path algorithm*.

A specific implementation is used and described in [5, Section 3.2]. The algorithm is then further adapted to handle multiple criteria [5, *Section 10.2*] and improved on.

One of the speed-up techniques described are *Lower Bounds*[5, *Section 10.2.2*]. This specific speed-up technique is used in the MOTIS project today and will serve as a frame of reference for evaluating my RAPTOR lower bounds algorithm. The usage of lower bounds is also described in [5, *Section 8.5.4*], using the lower bound (minimum of remaining travel time to the target stop) in conjunction with the already traveled time. Construction of the lower bounds is done via the *station graph*[5, Section 8.5.1].

For a quick and concise visualization of Dijkstra's algorithm, refer to [5, Section 3.2 - Algorithm 1].

# 5 Approach

## 5.1 Utilizing lower bounds with RAPTOR

The aim of this section is improving the algorithm further, by possibly not marking certain stops and therefore throwing them out sooner, potentially decreasing the number of rounds necessary to reach the termination condition and therefore reducing calculation time. This improvement is comparable in nature to the aforementioned *target pruning*(4.1.2).

### 5.1.1 Preliminaries

To achieve this, it is first and foremost necessary to provide the algorithm with a target stop $p_t$, the same way it is done in 4.1.2.

Furthermore, a set of pre-calculated lower bounds are necessary. These lower bounds are calculated in regard to a *target stop* $p_t$ and define the *minimally needed travel time* to the target. Of course, in actuality, these times can be more or less accurate, so calling it the *fastest possible travel time* would be an inaccuracy, but they guarantee, as they are *lower bounds*, that no faster travel time is possible.

The quality of the lower bounds can be determined as a combination of two factors, *calculation time* and *optimization time*. *Calculation time* describes the time needed to calculate the lower bounds, whereas *optimization time* describes the amount of time saved by using the calculated lower bounds. These two factors are normally positively correlated, so decreasing the calculation time often times results in a lower efficacy regarding the possible time savings. The exact nature of this correlation is important when trying to optimize the compromise needed to get an optimal calculation time reduction with minimal work to calculate the bounds, and is discussed in 6.

### 5.1.2 Extension to the data model

To now anchor this concept in the realm of the RAPTOR data model, another type of time is needed first. In 4.1.1, I defined $\Pi \subset \mathbb{N}_0$ as the operation time (e.g. seconds of a day). To define a *travel time*, which is a relative time, compared to e.g. an *arrival time*, which is an absolute time, I introduce a new set $\mathcal{I} \subset \mathbb{N}_0$ denoting the *duration times* (e.g. seconds). Note that in a concrete implementation, $\Pi$ and $\mathcal{I}$ could be the same.

The lower bounds $\mathcal{L}_{p_t}$ of target stop $p_t$, are now a set of duration times (travel times) $\iota \in \mathcal{I}$ closely correlated to it. In detail, a lower bound $\mathcal{L}_{p_t}(p) \in \mathcal{L}_{p_t}$ denotes the *minimally needed travel time* from stop $p$ to $p_t$.

### 5.1.3 Extension to the algorithm

As described in 4.1.2, the RAPTOR algorithm normally computes fastest journeys to all other stops, instead of focusing on a single target stop $p_t$. In order to make use of $p_t$ being the single point of interest, an additional check was employed, before any updating and marking happened. In particular, an arrival time for a trip $t$ at a stop $p_i$, $\text{arr}(t, p_i)$, must not only be earlier than the earliest known time *at that stop*, but also than the earliest known time *at the target stop* (*target pruning property*).

I now extend this concept further.

#### Integration of lower bounds

If the arrival time $\text{arr}(t, p_i)$ for a trip $t$ at a particular stop $p_i$, added to the minimally needed travel time $\mathcal{L}_{p_t}(p)$ from stop $p_i$ to target stop $p_t$, *is not* earlier than the earliest known arrival time $\tau_{\text{min}}(p_t)$ at the target stop $p_t$, the trip is no improvement to the journey already calculated (where $\tau_{\text{min}}(p_t)$ originated). In other words,

$$\text{arr}(t, p_i) + \mathcal{L}_{p_t}(p) < \tau_{\text{min}}(p_t)$$

needs to hold, in order for the trip to be considered an improvement (*lower bounds property*).

This is sound, as upon arrival at a stop $p_i$ with the trip $t$, the *minimal* arrival time at $p_t$ is $\text{arr}(t, p_i) + \mathcal{L}_{p_t}(p)$. If this is not faster than the fastest already known and possible time, it is by definition not an improvement and can therefore be omitted from the update and marking process.

Recalling 4.1.2,

$$\text{arr}(t, p_i) < \min\{\tau_{\text{min}}(p_i), \tau_{\text{min}}(p_t)\}$$

was used to determine if $\text{arr}(t, p_i)$ is lower when target pruning is used. If I now need to add the lower bounds to the arrival time, as previously described (modify the left side of the inequality), the check in its current form would be unusable, as a comparison with $\tau_{\text{min}}(p_i)$ would be faulty. In order to keep the check elegant, the previously proposed *lower bounds property* equation needs to be transformed into $\text{arr}(t, p_i) < \tau_{\text{min}}(p_t) - \mathcal{L}_{p_t}(p)$, which is trivially equivalent.

It is also trivial, that

$$\tau_{\text{min}}(p_t) - \mathcal{L}_{p_t}(p) \leq \tau_{\text{min}}(p_t)$$

as $\mathcal{L}_{p_t}(p) \in \mathcal{I} \Rightarrow \mathcal{L}_{p_t}(p) \geq 0$ is always true. Because of the transitive property of

$$\text{arr}(t, p_i) < \tau_{\text{min}}(p_t) - \mathcal{L}_{p_t}(p) \leq \tau_{\text{min}}(p_t)$$

a check for

$$\text{arr}(t, p_i) < \tau_{\text{min}}(p_t) - \mathcal{L}_{p_t}(p)$$

suffices.

In conclusion, a check for the *lower bounds property* supersedes the check for the *target pruning property* and checking for

$$\text{arr}(t, p_i) < \min\{\tau_{\text{min}}(p_i), \tau_{\text{min}}(p_t) - \mathcal{L}_{p_t}(p)\}$$

therefore sufficiently integrates the usage of lower bounds into the algorithm.

```
[...];
Input: Target stop p_t
Input: Lower bounds L_{p_t} for p_t
[...];
foreach k ← 1, 2, ... do
    [...];
    foreach ⟨r, p_hop⟩ ∈ Q do
        [...];
        foreach p_i ∈ r | i ∈ (hop, |r|], p_i ≺ p_{i+1} do
            if t ≠ □ then
                if arr(t, p_i) < min{τ_min(p_i), τ_min(p_t) − L_{p_t}(p)} then
                    τ_k(p_i), τ_min(p_i) ← arr(t, p_i);
                    M ← M ∪ p_i;
            if dep(t, p_i) ≥ τ_{k−1}(p_i) then
                t ← et(r, p_i);
    [...];
```

This minor change also has the benefit of inheriting the runtime properties of the original RAPTOR. As described in the RAPTOR paper[2, End of Section 3], the algorithm takes $\mathcal{O}(K(\sum_{r \in \mathcal{R}} |r| + |\mathcal{T}| + |\mathcal{F}|))$ time for $K$ rounds, where $|r|$ is the number of stops along route $r$. As the changes don't mess with any part related to the runtime calculation, this still holds true.

## 5.2 Computing lower bounds using RAPTOR

To be able to utilize the benefits of the lower bounds optimization for the RAPTOR algorithm, as described in 5.1, I described the need for said lower bounds $\mathcal{L}_{p_t}$ as an input to the algorithm. The question that now naturally arises, is on how to calculate the lower bounds.

Various different approaches are theoretically possible to achieve this goal, e.g. an approach using a Dijkstra-method is used for comparing results in 6, with results varying in accuracy and performance.

My goal for calculating the lower bounds using a (modified) RAPTOR approach, was to have minimal changes to the algorithm produce the lower bounds that can then in turn be used in the actual algorithm. The algorithm would then in some sense be self-sustaining. Furthermore, the RAPTOR paper [2] lists a couple of desirable properties of the RAPTOR algorithm, namely good possible parallelization and good runtime. By modifying the algorithm as little as possible for the lower bounds calculation, the aim is to preserve as much of these desirable properties as possible.

### 5.2.1 Concept

As the lower bounds are to be calculated in regard to a target stop $p_t$, the algorithm needs to be provided with it.

The goal of this lower bounds calculation is to calculate the minimally needed travel time *from every* stop $p$ *to a target* stop $p_t$. If one recalls the aim of the RAPTOR algorithm (4.1), the similarity becomes apparent, as it in turn calculates the fastest arrival time from *a source* stop $p_s$, *to every* stop $p$, which kind of reverses the goal from above.

One could be led to believe that simply inputting $p_t$ as $p_s$ into the algorithm would be sufficient. This is, unfortunately, not necessarily true. As explained in 4.1.1, routes and more importantly trips, consist of an ordered list of stops. Traversing a route in reverse isn't therefore possible without modification, and not trivial. In addition, trips assign an arrival and departure time to each of their stops, which in turn complicates a simple "reversal" further. Also, and most importantly, the aim of the lower bounds is *not* to precompute the whole solution, which is exactly what this naive solution would achieve. This would mean that despite the time saved, during each iteration (*optimization time*) of the algorithm by using these lower bounds, being maximized, the time needed to calculate them (*calculation time*) would also be maximized and therefore tend towards the actual calculation time of the RAPTOR algorithm itself, which would signify poor quality and optimization.

As already mentioned in 4.1.2, the RAPTOR algorithm, without *target pruning* employed, calculates journeys to all other stops, instead of focusing on a single target stop. Normally, this narrowing is desirable. In the case of calculating the lower bounds, however, the absence of *target pruning* and therefore "flooding" behavior of the calculation is actually the most desirable part of it. In other words, by starting from $p_t$, the algorithm will calculate the lower bounds for every other (reachable) stop somewhat natively.

The only real adjustments to the algorithm need to be made at points, where the precedence and order of stops is relevant, as operation needs to happen in a backwards manner, and where an arrival time is needed, as I just introduced a special function for this part. The absence of round-bound labels $\tau_k(p)$ only slims the algorithm down further.

In conclusion, a simpler version of the algorithm is needed, as to only calculate the lower *bounds* instead of the actual values calculated during a regular run of the algorithm.

There are multiple abstractions and simplifications possible.

### 5.2.2 The algorithm

#### Preliminaries

First, a run of the algorithm only requires a singular input: the target stop $p_t$, for which to calculate the lower bounds. One can also argue that since we start the algorithm off with the target stop, it has now become the stop $p_s$ as far as the algorithm is concerned. In order to keep modifications of the original algorithm to a minimum, we define the starting stop $p_s$ to be the target stop $p_t$ for which the *Lower Bounds RAPTOR* calculates the lower bounds.

**Input:** Starting stop $p_s$ (target stop for which to calculate the lower bounds)

### Minimal arrival time

For determining the minimal *travel time* between stops $p_i$ and $p_{i+1}$, it is completely sufficient to determine the lowest *travel time* between $p_i$ and $p_{i+1}$, for all trips $t \in \mathcal{T}_r$ of route $r$. Therefore, a *current trip* with a specific discrete arrival and departure time, satisfying $\text{et}(r, p_i)$, etc., is no longer needed. The travel time for travelling from $p_i$ to $p_{i+1}$ is calculated by subtracting the departure time at stop $p_i$ from the arrival time at stop $p_{i+1}$. I therefore need to determine the lowest value for all $t \in \mathcal{T}_r$

$$\min_{t \in \mathcal{T}_r}(\text{arr}(t, p_{i+1}) - \text{dep}(t, p_i)) \in \mathcal{I}$$

In the normal RAPTOR algorithm (4.1), the time between the arrival at stop $p$ in the previous round $\tau_{k-1}(p)$ and the departure at this stop for the current (earliest) trip $\text{dep}(t, p)$ would be the transfer time. I however will assume all transfers to be instantaneous and an immediate trip to be catchable, in order to further aid the aim of calculating lower *bounds* instead of full-fledged solutions and therefore reducing the *calculation time*. This means the check for an *earliest trip* is completely obsolete, as I assume there is always a trip to catch at the time of arrival, which in turn sets the transfer time to be a constant $0$. As no earliest trip is required anymore, transfers are instantaneous, and a catchable trip is also assumed to be available immediately, we can also drop the need for the $\tau_k(\cdot)$ labels. We will only keep and update the $\tau_{\text{min}}(\cdot)$ labels, which keeps track of the globally minimal arrival time for each stop.

As I don't use a current trip $t$ anymore, the $\text{arr}(t, p)$ function explained in 4.1 can't be used here. That's why I introduce a novel function

$$\text{arr}_{\text{min}}(r, p_{hop}, p_i) \in \Pi$$

In order to calculate the arrival time at stop $p_{i+1}$, one simply needs to add the minimal travel time from $p_i$ to $p_{i+1}$ to the arrival time at $p_i$

$$\text{arr}_{\text{min}}(r, p_{hop}, p_i) + \min_{t \in \mathcal{T}_r}(\text{arr}(t, p_{i+1}) - \text{dep}(t, p_i))$$

An important fact to recollect is that we are still working with "forward oriented" routes, trips, etc., but we need to traverse everything *in reverse* as we need to calculate the lower bounds from a *target stop* and therefore *backwards*. This calls for an adjustment to the previous function, as to calculate the arrival time at stop $p_i$, it needs to be based on the arrival at stop $p_{i+1}$! (traversing happens backwards)

$$\text{arr}_{\text{min}}(r, p_{hop}, p_{i+1}) + \min_{t \in \mathcal{T}_r}(\text{arr}(t, p_{i+1}) - \text{dep}(t, p_i))$$

As this is a recursive function, some kind of termination condition is needed. This is also the reason why $p_{hop}$ is provided as an argument to the function: If the stop to be handled $p_i$ is equal to the stop we "hop on" $p_{hop}$, we don't need to calculate anything, as we already have the arrival time in the form of $\tau_{\text{min}}(p_{hop})$.

In conclusion, I define the *minimal arrival* function as

$$\text{arr}_{\text{min}}(r, p_{hop}, p_i) := \begin{cases} \tau_{\text{min}}(p_{hop}), & \text{if } p_i = p_{hop} \\ \text{arr}_{\text{min}}(r, p_{hop}, p_{i+1}) + \min_{t \in \mathcal{T}_r}(\text{arr}(t, p_{i+1}) - \text{dep}(t, p_i)), & \text{otherwise} \end{cases}$$

or as pseudocode

**Function** $\text{arr}_{\text{min}}(r,\ p_{hop},\ p_i)$**:**
  **if** $p_i = p_{hop}$ **then**
    **return** $\tau_{\text{min}}(p_{hop})$;
  **else**
    **return** $\text{arr}_{\text{min}}(r,\ p_{hop},\ p_{i+1}) + \min_{t \in \mathcal{T}_r}(\text{arr}(t, p_{i+1}) - \text{dep}(t, p_i))$;

#### Initialization

The initialization procedure is almost identical to the native RAPTOR one, described in 4.1.2, except the absence of any $\tau_k(p)$.

**foreach** $p \in \mathcal{P}$ **do**
  $\tau_{\text{min}}(p) \leftarrow \infty$;

#### Starting condition

As I introduced earlier in 5.1.2, technically, the lower bounds calculate a *relative* minimum travel time $\iota \in \mathcal{I}$. This stands in contrast to the ”original” RAPTOR implementation, where the period of operation $\Pi$ is used.

However, by employing a simple trick, one can easily make the lower bounds algorithm work with the period of operation $\Pi$ (absolute time).

If I set the travel departure time to be 0 ($\tau_{\text{init}} \leftarrow 00:00:00$), any calculated arrival time can effectively be interpreted as a duration, in addition to an absolute point in time. An arrival time of, e.g. $01:25:00$, at stop $p$ while calculating the lower bounds for $p_t$ ($\mathcal{L}_{p_t}(p) = 01:25:00$), would signify a minimum travel time / lower bound of *1 hour and 25 minutes*, instead of an actual arrival at 01:25am. The actual, absolute arrival time is no longer needed, as $\text{arr}_{\text{min}}$ operates solely with relative times.

$\tau_{\text{init}} \leftarrow 00:00:00$;
$\tau_{\text{min}}(p_s) \leftarrow \tau_{\text{init}}$;
$\mathcal{M} \leftarrow \{p_s\}$;

#### Main loop

I already explained previously that the algorithm stays mostly true to the original RAPTOR algorithm. One exception that already arose, was the reverse traversal of routes. This is also applicable for the main loop.

The general order of operation stays the same, as well as the termination condition.

**foreach** $k \leftarrow 1, 2, \ldots$ **do**
    `// Handle marked stops` $p$
    [...];
    `// Handle each queued route`
    [...];
    `// Footpath or transport faster?`
    [...];
    `// Termination condition`
    **if** $\mathcal{M} = \emptyset$ **then**
        **return**;

I still operate in rounds over the data, transfer marked stops and their serving routes into a queue, handle the queued items, check for potential footpaths, and finally determine whether to terminate.

The only points where the two algorithms differ, is wherever the direction between stops or an arrival time calculation are of interest.

#### Handling marked stops

Same as with the normal RAPTOR algorithm, I handle each *marked* stop $\forall p \in \mathcal{M}$ and add every route $r$ serving the stop $\forall \in \mathcal{R}_p$ to th queue

$$\mathcal{Q} \leftarrow \mathcal{Q} \cup \langle r, p \rangle$$

and *unmark* the stop

$$\mathcal{M} \leftarrow \mathcal{M} \setminus p$$

Instead of solely saving the *earliest* stop of a route, I now save the *latest* stop. This is needed, as the direction is reversed and therefore the *latest* stop, when handling a route in reverse, becomes the *earliest* one.

If, similar to 4.1.2, a route $r$ is already present in the queue with a different stop, the present entry is replaced, if the current stop $p$ *succeeds* the other stop $p'$.

**foreach** $k \leftarrow 1, 2, \ldots$ **do**
    $\mathcal{Q} \leftarrow \emptyset$;
    **foreach** $p \in \mathcal{M}$ **do**
        **foreach** $r \in \mathcal{R}_p$ **do**
            **if** $\exists p' \in \mathcal{P} : \langle r, p' \rangle \in \mathcal{Q} \mid p' \neq p$ **then**
                **if** $p \succ p'$ **then**
                    $\mathcal{Q} \leftarrow (\mathcal{Q} \setminus \langle r, p' \rangle) \cup \langle r, p \rangle$;
            **else**
                $\mathcal{Q} \leftarrow \mathcal{Q} \cup \langle r, p \rangle$;
        $\mathcal{M} \leftarrow \mathcal{M} \setminus p$;
    [...];

### Handle the queue

When now processing the queue entries in a closely similar manner as 4.1.2, I also traverse the route, but this time from back to front. So instead of looping for each $p_i \in r$ where $i \in (hop, |r|]$, which would signify starting with $p_hop$ and ending at the last stop of the route, the loop now happens for each $p_i \in r$ where $i \in (hop, 0)$, which signifies starting with $p_{hop}$ and ending with the first stop

$$p_i \in r \mid i \in (hop, 0),\ p_i \succ p_{i-1}$$

Note, that as discussed earlier, an *earliest trip* is no longer required. This enables a straight jump to the update procedure, where now instead of using $\text{arr}(t, p_i)$, the new $\text{arr}_{\text{min}}(r, p_{hop}, p_i)$ is used to check for an improvement against the earliest known arrival time, now technically used as the fastest known travel time

$$\text{arr}_{\text{min}}(r, p_{hop}, p_i) < \tau_{\text{min}}(p_i)$$

If an improvement has been made, the $\tau_{\text{min}}$ gets updated with the $\text{arr}_{\text{min}}$ value and the stop is marked.

---

**foreach** $k \leftarrow 1, 2, \ldots$ **do**
    [...];
    **foreach** $\langle r, p_{hop} \rangle \in \mathcal{Q}$ **do**
        **foreach** $p_i \in r \mid i \in (hop, 0),\ p_i \succ p_{i-1}$ **do**
            **if** $\text{arr}_{\text{min}}(r, p_{hop}, p_i) < \tau_{\text{min}}(p_i)$ **then**
                $\tau_{\text{min}}(p_i) \leftarrow \text{arr}_{\text{min}}(r, p_{hop}, p_i)$;
                $\mathcal{M} \leftarrow \mathcal{M} \cup p_i$;
    [...];

---

### Checking footpaths

In case of checking for footpaths, it is almost identical to the part described in 4.1.2, except for the direction, again.

Checking footpaths is important, as despite the simplifications of using the lowest travel time of all trips, as well as assuming an instant transfer time, two stops can very well be connected in a more time efficient manner through footpaths instead of routes. This is very apparent in real world scenarios, as walking from one bus stop to another two streets away can often be much faster than taking potentially multiple trips with other buses to reach it.

As operations are, again, reversed for the purpose of this algorithm, I handle every footpath *reaching* (ending at) each marked stop, instead of *originating* (starting from) it

$$\langle p, p' \rangle \in \mathcal{F} \mid p' \in \mathcal{M}$$

The $\tau_{\text{min}}$ value is updated for the *originating* stop, and it is marked. The stop is then marked.

**foreach** $k \leftarrow 1, 2, \ldots$ **do**
    [...];
    **foreach** $\langle p, p' \rangle \in F \mid p' \in \mathcal{M}$ **do**
        $\tau_k(p) \leftarrow \min\{\tau_k(p), \tau_k(p') + l(p, p')\}$;
        $\mathcal{M} \leftarrow \mathcal{M} \cup p$;
    [...];

# 6 Evaluation

To evaluate my approach and findings, I decided to use an already existing implementation of the RAPTOR algorithm to then apply my approaches on and perform my evaluation on. I decided to use the RAPTOR implementation within *MOTIS* (Multi Objective Travel Information System)[1]. MOTIS enables the use of multiple different routing algorithms and criteria to calculate results (*responses*) of routing requests (*queries*), through the use of modules, e.g. the *routing* module for a Dijkstra implementation. One of those modules is the fittingly named *raptor* module, which, as the name suggests, uses an implementation of the RAPTOR algorithm. Another advantage is an already present *lower bounds algorithm* based on a Dijkstra approach, which I will use to compare my RAPTOR one against. MOTIS additionally uses the formats of *real-world schedule timetables* in a variety of commonly used formats as the input data for the schedule (timetable), which in turn is greatly beneficial for me evaluating the benefit for public transit routing closely.

Although many things are possible with MOTIS, I will focus on the Earliest Arrival Problem, which in the case of the RAPTOR algorithm also optimizes the number of transfers. I will use the MOTIS system, which can also be run as a server, running in batch mode, processing queries in a text file and also outputting the responses in one. This batch mode is therefore extremely convenient for my testing, as I am able to generate a set of random queries from a timetable and test the exact same queries with different implementations and variants to compare them and analyze the performance.

I chose to use the *GTFS* (General Transit Feed Specification Format) format with the Swiss transit network and used the version `2023-05-03_04-15`[2]. This should enable me to generate representative, sufficiently randomized and wide-spread queries compared to a fictive timetable, while also not blowing out of proportion as to the smaller size and volume of the Swiss network compared to e.g. the German one.

**Points Of Interest** Of course, there are a multitude of variables and parameters to analyze. To keep this evaluation concise, a select set of these measurements and results are analyzed as points of interest.

For once, the responses are analyzed regarding their accuracy. In detail, this means comparing the corresponding responses to determine discrepancies in the routing. In theory, the addition of lower bounds shouldn't change the calculated journeys themselves, but rather dismiss non-improving branches earlier in the process, thus speeding up the runtime, similar to the target pruning. To test this, an integrated tool of MOTIS is used, accessed through the `motis compare` sub-command.

Another focus obviously lies in the raw performance gain / loss. Here I'm mainly concerned with two measurements, the `total_calculation_time` ($\Delta_t(\Sigma)$) and the `raptor_time` ($\Delta_t(raptor)$). $\Delta_t(raptor)$ signifies the time the actual RAPTOR algorithm took. It should therefore be a good indicator for improvements or slow-downs directly affecting algorithm runtime. $\Delta_t(\Sigma)$, in contrast, includes pretty much everything

[1] Project repository: https://github.com/motis-project/motis
[2] https://opentransportdata.swiss/en/dataset/timetable-2023-gtfs2020/resource/9cdd66d2-0fa1-46a1-be71-fcfd9a575dcf

needed to calculate a response, including generating queries, collecting statistics, reconstructing the journeys, etc.

In case lower bounds are used, it also includes calculating the lower bounds for the target stop of the query, which in turn gets measured on its own through either the `lower_bounds_time` ($\Delta_t(lb)$), when Dijkstra lower bounds are used, or the `raptor_lower_bounds_time` ($\Delta_t(raptorlb)$), where both *are a component of* $\Delta_t(\Sigma)$). It should therefore be a good indicator for the lower bounds calculation overhead, as well as general slow down of parts not directly benefiting from the lower bounds.

I also gathered some additional *in-depth statistics* covering the number of stops processed and omitted, with each method, to determine the amount of stops being affected by each improvement. It is worth noting that every $\Delta_t$ is in $ms$.

## 6.1 Using Dijkstra-based computed lower bounds with RAPTOR

In this section, I will determine the impact of my approach, integrating the use of lower bounds into the RAPTOR algorithm (5.1). For this to be done correctly, it is important to note that at the time of writing, the existing RAPTOR implementation didn't implement the target pruning improvement, portrayed in 4.1.2, proposed by [2]. This means a sole comparison of the lower bounds implementation, which further *improves on the target pruning*, to a version with target pruning completely absent, wouldn't enable any insight into improvements made exclusively by the lower bounds.

**Target pruning** As explained in 4.1.2, the changes needed to implement target pruning are rather minimal and consist of additionally comparing the arrival time to the earliest known arrival time of the target stop. First, the $\tau_{\text{min}}(p_t)$ needs to be determined and a

$$\min\{\tau_{\text{min}}(p_i), \tau_{\text{min}}(p_t)\}$$

to be calculated for use in the comparison, replacing the simple check against $\tau_{\text{min}}(p_i)$, which boils down to a two line code change.

**Lower bounds** For implementing the other variant, the lower bounds, it is a little more complicated. Before calling the algorithm itself, the lower bounds need to be calculated. In this comparison section, I utilize the already present lower bounds calculation. It is very important to mention that the implementation doesn't make use of the underlying RAPTOR data structure at all, quite the opposite. A so-called *constant graph* is constructed from the data and simplified/optimized. This is done as a pre-computing step once, when initially launching the software in batch mode, not for every query. The calculation done for each query is computing the lower bounds for the query target, which means running a Dijkstra algorithm on the optimized graph. After now providing the algorithm with the lower bounds as an input, they can now be used in place of the target pruning part. So, again, the $\tau_{\text{min}}(p_t)$ needs to be determined and a

$$\min\{\tau_{\text{min}}(p_i), \tau_{\text{min}}(p_t) - \mathcal{L}_{p_t}(p_i)\}$$

to be calculated for use in the comparison, replacing the target pruning check, which boils down to a three line code change.

| optimization type | measurement | average | Q(99) | Q(90) | Q(80) | Q(50) |
|---|---|---|---|---|---|---|
| none | $\Delta_t(raptor)$ | 20.8928 | 44 | 28 | 23 | 20 |
| | $\Delta_t(\Sigma)$ | 21.3527 | 45 | 28 | 24 | 20 |
| target pruning | $\Delta_t(raptor)$ | 20.3853 | 46 | 27 | 23 | 19 |
| | $\Delta_t(\Sigma)$ | 20.8608 | 47 | 28 | 24 | 20 |
| lower bounds | $\Delta_t(lb)$ | 3.4929 | 16 | 5 | 4 | 3 |
| | $\Delta_t(raptor)$ | 20.8282 | 46 | 29 | 24 | 19 |
| | $\Delta_t(\Sigma)$ | 25.3609 | 52 | 34 | 29 | 24 |

Table 6.1: 10,000 queries (Dijkstra lower bounds)

### 6.1.1 The testing

I generated a set of 10000 queries (`motis generate` with `start_type=ontrip_station`, as well as `dataset.begin=20221212`) and ran it through `motis` in batch mode (also using `dataset.begin=20221212`). The run (not the generation) was performed with 32 worker threads, which is the default setting for MOTIS, using each the target pruning and the lower bounds approach once, as well as the *original implementation, without even the target pruning*, to obtain three responses in total. The same run was then repeated with the custom *in-depth statistics*, as runs with them enabled can't be used for runtime evaluation, as gathering of the additional statistics takes up time.

#### Results

Comparing each respective run pair (original vs. target pruning vs. lower bounds) using `motis compare`, yields a result of *0 mismatches*, which indicates an error-free implementation. In other words, every single query, even ones where no connection was found, are identical in their routing response between the three implementations, which should make the performance comparable.

The results of the run are outlaid in table 6.1.

It is worth noting, though, that multiple reruns of the even same input queries, yield fluctuating timing result. This is very likely due to calculation time being impacted by external factors, such as processor speed, available system resources, other applications, and so on.

### 6.1.2 Analyzation

It is noticeable, that the target pruning optimization improves on the not optimized version. Compared to the not optimized run, the lower bounds optimization actually has a slightly lower average runtime. However, when looking at the quantiles, most of them are indeed slower for the lower bounds. Additionally, the total calculation time is noticeably increased, due to the overhead of calculating the lower bounds for the target stop.

However, as one can easily see, the lower bounds implementation clearly isn't improving *on the target pruning* approach *on average*, quite the opposite, it is actually slower. This is consistent with my testing with different query sets and further runs. Upon closer inspection, this is all true, except for the Q(50), however, as this quantile often times performed equivalent or even better than the target pruning one. In conclusion, though, on average the lower bounds implementation seems to perform worse.

As the changes for implementing the usage of lower bounds in the algorithm are rather minimal and the lower bounds calculation itself is already a separate measurement, I highly suspect the reason for this to be the additional array access (fetching the lower bound $\mathcal{L}_{p_t}(p_i)$) and subtraction from $\tau_{\text{min}}(p_i)$ for each stop $p_i$. It could be that its cumulative *additional* calculation time is eliminating any possible gains of visiting less stops and ”branches”. Of course, as explained in 5.1.1, it could be because of bad quality lower bounds, with them not gaining enough timing advantage to offset the additional calculation expense. Running the tests on a non-isolated and not really controllable system, where external factors can easily cause noticeable fluctuations and noise, certainly doesn't help, especially with such a relatively low calculation time per query.

Still, to investigate these results further, I thought of three changes to the code base. One amplifies the positive gain of skipping stops and rounds, therefore reducing the negative impact of the check (*artificial slow down*). The second one moves the check further inside the sub-conditions, to minimize the number of iterations checking the condition (*nested check*). The third one in turn collects statistics about the time taken to calculate the min$\{...\}$ part.

#### Nested check

One of the suspected culprit is the processing time of an additional fetch- and subtract-operation outweighing any potential benefits.

As I argued in 5.1, a check for the *lower bounds property* theoretically supersedes the check for the *target pruning property*. However, this check could also be performed in two separate steps. I would first check for

$$\text{arr}(t, p_i) < \min\{\tau_{\text{min}}(p_i), \tau_{\text{min}}(p_t)\}$$

which is the exact check performed for the target pruning implementation and then, in case this holds true, check for

$$\text{arr}(t, p_i) < \tau_{\text{min}}(p_t) - \mathcal{L}_{p_t}(p_i)$$

in a separate step. In theory, this removes the check for *some* iterations, when the target pruning condition doesn't hold true anyway.

```
[...];
foreach k ← 1, 2, ... do
    [...];
    foreach ⟨r, p_hop⟩ ∈ Q do
        [...];
        foreach p_i ∈ r | i ∈ (hop, |r|], p_i ≺ p_{i+1} do
            if t ≠ □ then
                if arr(t, p_i) < min{τ_min(p_i), τ_min(p_t)} then
                    if arr(t, p_i) < τ_min(p_t) − L_{p_t}(p) then
                        τ_k(p_i), τ_min(p_i) ← arr(t, p_i);
                        M ← M ∪ p_i;
            if dep(t, p_i) ≥ τ_{k−1}(p_i) then
                t ← et(r, p_i);
    [...];
```

| optimization type | measurement | average | Q(99) | Q(90) | Q(80) | Q(50) |
|---|---|---|---|---|---|---|
| target pruning | $\Delta_t(raptor)$ | 19.4310 | 42 | 25 | 22 | 19 |
| | $\Delta_t(\Sigma)$ | 19.8817 | 42 | 26 | 22 | 19 |
| lower bounds | $\Delta_t(lb)$ | 3.3379 | 13 | 5 | 4 | 3 |
| | $\Delta_t(raptor)$ | 19.5306 | 42 | 26 | 22 | 18 |
| | $\Delta_t(\Sigma)$ | 23.8830 | 49 | 32 | 27 | 22 |

Table 6.2: 10,000 queries (nested check)

| optimization type | measurement | average | Q(99) | Q(90) | Q(80) | Q(50) |
|---|---|---|---|---|---|---|
| target pruning | $\Delta_t(raptor)$ | 26.6237 | 45 | 36 | 31 | 25 |
| | $\Delta_t(\Sigma)$ | 27.1398 | 45 | 37 | 32 | 26 |
| lower bounds | $\Delta_t(lb)$ | 3.6237 | 17 | 5 | 4 | 3 |
| | $\Delta_t(raptor)$ | 25.8495 | 44 | 34 | 29 | 25 |
| | $\Delta_t(\Sigma)$ | 30.4409 | 52 | 38 | 34 | 30 |

Table 6.3: 100 queries (20 volatile reads)

Though it is worth mentioning, that especially in early rounds, many updates are performed because $\tau_{\min}(p_t)$ hasn't been narrowed down as much, so the amount of reduction could be minimal. Also, the new additional conditional statement could bear the same problem I had initially, namely slowing the algorithm too much per iteration. The test results can be seen in table 6.2.

This unfortunately doesn't really seem to improve much, except for possibly amplifying the previously observed advantage of the 50th quantile. However, in my testing, the times weren't that consistent in the first place, so this could also simply be fluctuation.

#### Artificial slow down

The second idea is to artificially slow down each iteration of the loop that checks for the update condition. The goal of this idea is to determine, whether the culprit is simply a highly optimized algorithm beating the performance gain of the lower bounds, by simply being extremely fast with fewer instructions.

If I slow down each iteration by a constant amount, each avoided iteration rewards a bigger chunk of saved time, therefore potentially offsetting the cost of additional instructions in every iteration. To try this, I created a `static volatile int slow_access;`. I will then read it in every iteration: `(void)slow_access`. The compiler can't remove this when optimizing for the Release build, as reading a `volatile` variable could potentially cause it to change and can't therefor be omitted.

I increased the number of volatile reads until I saw a definitive improvement across the board at 20 reads (see table 6.3). When increasing the number of reads greatly, this improvement in the difference was even more amplified (see table 6.4).

#### Isolation attempt

To see, whether the $-\mathcal{L}_{p_t}(p_i)$ part really is the slowing factor, I decided to track an additional statistic, wrapped around the min{...} code point (where the lower bounds are used), tracking the *cumulative calculation time*

| optimization type | measurement | average | Q(99) | Q(90) | Q(80) | Q(50) |
|---|---|---|---|---|---|---|
| target pruning | $\Delta_t(raptor)$ | 417.4194 | 695 | 598 | 522 | 415 |
| | $\Delta_t(\Sigma)$ | 417.7204 | 695 | 598 | 522 | 415 |
| lower bounds | $\Delta_t(lb)$ | 3.3978 | 8 | 4 | 4 | 3 |
| | $\Delta_t(raptor)$ | 405.8925 | 635 | 555 | 526 | 417 |
| | $\Delta_t(\Sigma)$ | 410.3118 | 640 | 559 | 530 | 421 |

Table 6.4: 100 queries (1000 volatile reads)

| optimization type | measurement | average | Q(99) | Q(90) | Q(80) | Q(50) |
|---|---|---|---|---|---|---|
| target pruning | $\Delta_t(\min\{...\})$ | 0.4301 | 5 | 2 | 0 | 0 |
| lower bounds | $\Delta_t(\min\{...\})$ | 0.5376 | 6 | 1 | 0 | 0 |

Table 6.5: 100 queries (min measured)

for each query. This will now be able to show what magnitude of significance this part really has (table 6.5)

As we can see, the additional code indeed produces a slowdown, although it doesn't seem quite as significant, as the previous statistics suggested.

## 6.2 Computing RAPTOR-based lower bounds

As explained previously, I aimed to keep the lower bounds algorithm as true to the original as possible, to possibly enable the reusage of a large part of the codebase. There are some parts, however, where some kind of workaround or trick is employed, which makes it a little bit tricky to calculate in reverse.

This doesn't make implementation impossible, though! With an implementation from scratch, the performance wasn't looking too promising (table 6.6).

The raptor lower bounds calculations have skyrocketed. The Dijkstra lower bounds' overhead isn't directly observable, as the optimized graph gets calculated at server launch, before any requests are processed and only the target specific lower bounds are calculated per query. With the RAPTOR lower bounds, calculation of the $\min_{t \in \mathcal{T}_r}(\text{arr}(t, p_{i+1}) - \text{dep}(t, p_i))$ needs to be done every query, just as an example.

I therefore chose to pre-compute the $\min_{t \in \mathcal{T}_r}(\cdots)$ values beforehand. Furthermore, I integrated the algorithm, as true to my approach as possible, with the already existent RAPTOR implementation, instead of a standalone implementation. I utilized a `boolean` field in the query, to tell the algorithm whether I want to calculate the lower bounds, or perform a normal run. For the differences between 5.2 and 5.1, the corresponding code segments are chosen depending on the state of the field. With this approach, the calculation times are much more reasonable, although still not on par with the Dijkstra-based ones (table 6.7).

| optimization type | measurement | average | Q(99) | Q(90) | Q(80) | Q(50) |
|---|---|---|---|---|---|---|
| Dijkstra lower bounds | $\Delta_t(lb)$ | 2.7204 | 6 | 4 | 3 | 2 |
| RAPTOR lower bounds (from scratch) | $\Delta_t(r_l b)$ | 14,944.3370 | 29,271 | 23,433 | 20,420 | 12,558 |

Table 6.6: 100 queries (RAPTOR lower bounds from scratch)

| optimization type | measurement | average | Q(99) | Q(90) | Q(80) | Q(50) |
|---|---|---|---|---|---|---|
| Dijkstra lower bounds | $\Delta_t(lb)$ | 3.4939 | 16 | 5 | 4 | 3 |
| RAPTOR lower bounds | $\Delta_t(r_lb)$ | 12.6313 | 31 | 17 | 13 | 11 |

Table 6.7: 10,000 queries

| optimization type | measurement | average | Q(99) | Q(90) | Q(80) | Q(50) |
|---|---|---|---|---|---|---|
| Dijkstra lower bounds | $\Delta_t(lb)$ | 3.3473 | 15 | 5 | 4 | 3 |
| | $\Delta_t(raptor)$ | 23.5332 | 49 | 32 | 27 | 22 |
| | $\Delta_t(\Sigma)$ | 27.8778 | 54 | 37 | 32 | 26 |
| RAPTOR lower bounds | $\Delta_t(r_lb)$ | 12.6313 | 31 | 17 | 13 | 11 |
| | $\Delta_t(raptor)$ 23.9890 | | 49 | 32 | 28 | 23 |
| | $\Delta_t(\Sigma)$ 37.9966 | | 67 | 49 | 43 | 36 |

Table 6.8: 10,000 queries (lower bounds comparison)

To now determine the quality of the bounds, as it could be the case that the longer calculation time also yields better bounds, I need to use both the Dijkstra lower bounds, and the RAPTOR ones in the main algorithm and compare the results

## 6.3 Using RATPOR-based lower bounds with RAPTOR

Comparison between the lower bounds implementations is easy, as it simply involves running the same algorithm with a different set of pre-computed bounds (table 6.8).

It is obvious that for the current implementation of *RAPTOR lower bounds*, the pre-computation takes a significantly longer time and using them also doesn't yield optimal performance for the algorithm, although this could be a fluctuation. Logically, the same issues as described in section 6.1 arise, when comparing the $\Delta_t(raptor)$ times. The reasons are very likely the same, which shows that it likely isn't the raw quality of the lower bounds that is lacking, but rather some fundamental implementation or theory specific components that cause the usage of lower bounds to slow calculation down.

It is however important to note that during my testing it became apparent that the RAPTOR lower bounds implementation I used, had $393$ non-matching query responses out of the $10000$ generated ones. This definitely hints at some kind of bug. It could be that the existent RAPTOR implementation is only really able to iterate over $\langle p, p' \rangle \in \mathcal{F} \mid p \in \mathcal{M}$, instead of the proposed $\langle p, p' \rangle \in \mathcal{F} \mid p' \in \mathcal{M}$. This would mean that $\neq \forall p, p' \in \mathcal{P} : l(p, p') = l(p', p)$, which could very well be possible.

# 7 Conclusions and future work

My first conclusion, based on the evaluation results, is that the usage of lower bounds in the RAPTOR algorithm, despite saving rounds in theory, has some practical culprits to be overcome. Foremost, the target pruning improvement did noticeably cut down the calculation time of the algorithm, without any apparent drawbacks. The algorithm only needs the target stop as an input parameter, which with MOTIS is already the case, to be able to prune against the target. However, despite my assumptions and reasoning, the *extension* of that criterion, namely the lower bounds, failed to provide a similar improvement. This, without doubt, raises questions about the quality of implementation and potentially about the fundamental construction of the approach. Further testing, with different and differing implementations and data, needs to be conducted to either confirm or dismiss the approach. I still think that the usage of lower bounds as means of speed-up could prove useful for the RAPTOR algorithm, but modifications to the approach or implementation need to be made to increase the positive and reduce the negative impact in real-world scenarios.

In case of the lower bounds calculation using RAPTOR, I think that it definitely has potential to be able to utilize the strengths of the RAPTOR algorithm for calculating lower bounds. Either the concrete implementation or the structure of MOTIS made it difficult to be able to use the approach to its full potential. Because the underlying RAPTOR data structure implementation in MOTIS is, albeit maybe very potent, in my opinion very convoluted and hidden behind a handful of abstraction layers, it prevents the RAPTOR algorithm from being repurposed effectively and hassle-free. In addition to that, some of the implementation specifics, which are very focused on the original RAPTOR algorithm, also make them difficult to use in other scenarios. An improvement of the implementation, or even implementation in another framework/context, and following evaluation, could be very helpful in deciding whether a flaw is situated in the approach or the realization. It would also be interesting to research, whether improvement or minimization of the underlying RAPTOR data model might be possible in a similar manner to MOTIS' constant graph. This could as a result keep the work done in every iteration to a minimum and rather have the underlying data provide the means of determining lower bounds.

Furthermore, it would be interesting to closely examine the quality of the lower bounds, to determine, if the encountered issues could be mitigated by higher quality bounds.

# 8 Appendix

**Algorithm 1:** RAPTOR (4.1)

**Input:** Source stop $p_s$
**Input:** Travel departure time $\tau_{\text{init}}$

**foreach** $p \in \mathcal{P}$ **do** // Init stops to be $\infty \doteq$ unreachable
  **forall** $i$ **do**
    $\tau_i(p) \leftarrow \infty$;
  $\tau_{\min}(p) \leftarrow \infty$;
$\tau_0(p_s) \leftarrow \tau_{\text{init}}$; $\tau_{\min} \leftarrow \tau_0(p_s)$; $\mathcal{M} \leftarrow \{p_s\}$ ; // Starting conditions
**foreach** $k \leftarrow 1, 2, \ldots$ **do** // Round $k$
  $\mathcal{Q} \leftarrow \emptyset$;
  **foreach** $p \in \mathcal{M}$ **do** // Handle marked stops $p$
    **foreach** $r \in \mathcal{R}_p$ **do** // Handle routes $r$ serving stop $p$
      **if** $\exists p' \in \mathcal{P} : \langle r, p' \rangle \in \mathcal{Q} \mid p' \neq p$ **then** // Keep earliest $p$ for $r$
        **if** $p \prec p'$ **then**
          $\mathcal{Q} \leftarrow (\mathcal{Q} \setminus \langle r, p' \rangle) \cup \langle r, p \rangle$;
      **else** // Queue route $r$ with $p$
        $\mathcal{Q} \leftarrow \mathcal{Q} \cup \langle r, p \rangle$;
    $\mathcal{M} \leftarrow \mathcal{M} \setminus p$ ; // Unmark stop $p$
  **foreach** $\langle r, p_{hop} \rangle \in \mathcal{Q}$ **do** // Handle each queued route
    $t \leftarrow \square$;
    **foreach** $p_i \in r \mid i \in (hop, |r|],\ p_i \prec p_{i+1}$ **do** // Orderly traverse stops
      **if** $t \neq \square$ **then**
        **if** $\text{arr}(t, p_i) < \tau_{\min}(p_i)$ **then** // Improvement?
          $\tau_k(p_i), \tau_{\min}(p_i) \leftarrow \text{arr}(t, p_i)$; // Update values
          $\mathcal{M} \leftarrow \mathcal{M} \cup p_i$; // Mark stop $p_i$
        **if** $\text{dep}(t, p_i) \geq \tau_{k-1}(p_i)$ **then** // Earlier trip possible?
          $t \leftarrow \text{et}(r, p_i)$;
  **foreach** $\langle p, p' \rangle \in F \mid p \in \mathcal{M}$ **do** // Footpath or transport faster?
    $\tau_k(p') \leftarrow \min\{\tau_k(p'), \tau_k(p) + l(p, p')\}$;
    $\mathcal{M} \leftarrow \mathcal{M} \cup p'$; // Mark stop $p'$
  **if** $\mathcal{M} = \emptyset$ **then** // Termination condition
    **return**;

**Algorithm 2:** RAPTOR (with target pruning, 4.1.2)

**Input:** Source stop $p_s$
**Input:** Target stop $p_t$
**Input:** Travel departure time $\tau_{\text{init}}$

**foreach** $p \in \mathcal{P}$ **do** // Init stops to be $\infty \doteq$ unreachable
  **forall** $i$ **do**
    $\tau_i(p) \leftarrow \infty$;
  $\tau_{\min}(p) \leftarrow \infty$;
$\tau_0(p_s) \leftarrow \tau_{\text{init}}$; $\tau_{\min} \leftarrow \tau_0(p_s)$; $\mathcal{M} \leftarrow \{p_s\}$ ; // Starting conditions
**foreach** $k \leftarrow 1, 2, \ldots$ **do** // Round $k$
  $\mathcal{Q} \leftarrow \emptyset$;
  **foreach** $p \in \mathcal{M}$ **do** // Handle marked stops $p$
    **foreach** $r \in \mathcal{R}_p$ **do** // Handle routes $r$ serving stop $p$
      **if** $\exists p' \in \mathcal{P} : \langle r, p' \rangle \in \mathcal{Q} \mid p' \neq p$ **then** // Keep earliest $p$ for $r$
        **if** $p \prec p'$ **then**
          $\mathcal{Q} \leftarrow (\mathcal{Q} \setminus \langle r, p' \rangle) \cup \langle r, p \rangle$;
      **else** // Queue route $r$ with $p$
        $\mathcal{Q} \leftarrow \mathcal{Q} \cup \langle r, p \rangle$;
    $\mathcal{M} \leftarrow \mathcal{M} \setminus p$ ; // Unmark stop $p$
  **foreach** $\langle r, p_{hop} \rangle \in \mathcal{Q}$ **do** // Handle each queued route
    $t \leftarrow \square$;
    **foreach** $p_i \in r \mid i \in (hop, |r|],\ p_i \prec p_{i+1}$ **do** // Orderly traverse stops
      **if** $t \neq \square$ **then**
        **if** $\text{arr}(t, p_i) < \min\{\tau_{\min}(p_i), \tau_{\min}(p_t)\}$ **then** // Improvement? (target pruning!)
          $\tau_k(p_i), \tau_{\min}(p_i) \leftarrow \text{arr}(t, p_i)$; // Update values
          $\mathcal{M} \leftarrow \mathcal{M} \cup p_i$; // Mark stop $p_i$
      **if** $\text{dep}(t, p_i) \geq \tau_{k-1}(p_i)$ **then** // Earlier trip possible?
        $t \leftarrow \text{et}(r, p_i)$;
  **foreach** $\langle p, p' \rangle \in F \mid p \in \mathcal{M}$ **do** // Footpath or transport faster?
    $\tau_k(p') \leftarrow \min\{\tau_k(p'), \tau_k(p) + l(p, p')\}$;
    $\mathcal{M} \leftarrow \mathcal{M} \cup p'$; // Mark stop $p'$
  **if** $\mathcal{M} = \emptyset$ **then** // Termination condition
    **return**;

**Algorithm 3:** RAPTOR (with lower bounds, 5.1)

**Input:** Source stop $p_s$
**Input:** Target stop $p_t$
**Input:** Lower bounds $\mathcal{L}_{p_t}$ for $p_t$
**Input:** Travel departure time $\tau_{\text{init}}$

**foreach** $p \in \mathcal{P}$ **do** // Init stops to be $\infty \doteq$ unreachable
  **forall** $i$ **do**
    $\tau_i(p) \leftarrow \infty$;
  $\tau_{\text{min}}(p) \leftarrow \infty$;
$\tau_0(p_s) \leftarrow \tau_{\text{init}}$; $\tau_{\text{min}} \leftarrow \tau_0(p_s)$; $\mathcal{M} \leftarrow \{p_s\}$ ; // Starting conditions
**foreach** $k \leftarrow 1, 2, \ldots$ **do** // Round $k$
  $\mathcal{Q} \leftarrow \emptyset$;
  **foreach** $p \in \mathcal{M}$ **do** // Handle marked stops $p$
    **foreach** $r \in \mathcal{R}_p$ **do** // Handle routes $r$ serving stop $p$
      **if** $\exists p' \in \mathcal{P} : \langle r, p' \rangle \in \mathcal{Q} \mid p' \neq p$ **then** // Keep earliest $p$ for $r$
        **if** $p \prec p'$ **then**
          $\mathcal{Q} \leftarrow (\mathcal{Q} \setminus \langle r, p' \rangle) \cup \langle r, p \rangle$;
      **else** // Queue route $r$ with $p$
        $\mathcal{Q} \leftarrow \mathcal{Q} \cup \langle r, p \rangle$;
    $\mathcal{M} \leftarrow \mathcal{M} \setminus p$ ; // Unmark stop $p$
  **foreach** $\langle r, p_{hop} \rangle \in \mathcal{Q}$ **do** // Handle each queued route
    $t \leftarrow \square$;
    **foreach** $p_i \in r \mid i \in (hop, |r|],\ p_i \prec p_{i+1}$ **do** // Orderly traverse stops
      **if** $t \neq \square$ **then**
        **if** $\text{arr}(t, p_i) < \min\{\tau_{\text{min}}(p_i), \tau_{\text{min}}(p_t) - \mathcal{L}_{p_t}(p)\}$ **then** // Improvement? (lower bounds)
          $\tau_k(p_i), \tau_{\text{min}}(p_i) \leftarrow \text{arr}(t, p_i)$; // Update values
          $\mathcal{M} \leftarrow \mathcal{M} \cup p_i$; // Mark stop $p_i$
      **if** $\text{dep}(t, p_i) \geq \tau_{k-1}(p_i)$ **then** // Earlier trip possible?
        $t \leftarrow \text{et}(r, p_i)$;
  **foreach** $\langle p, p' \rangle \in F \mid p \in \mathcal{M}$ **do** // Footpath or transport faster?
    $\tau_k(p') \leftarrow \min\{\tau_k(p'), \tau_k(p) + l(p, p')\}$;
    $\mathcal{M} \leftarrow \mathcal{M} \cup p'$; // Mark stop $p'$
  **if** $\mathcal{M} = \emptyset$ **then** // Termination condition
    **return**;

**Algorithm 4:** RAPTOR for computing lower bounds (5.2)

**Input:** Starting stop $p_s$ (target stop for which to calculate the lower bounds)

**Function** $\text{arr}_{\text{min}}(r,\ p_{hop},\ p_i)$**:** // Calculate the minimum travel time
  **if** $p_i = p_{hop}$ **then**
    **return** $\tau_{\text{min}}(p_{hop})$;
  **else**
    **return** $\text{arr}_{\text{min}}(r,\ p_{hop},\ p_{i+1}) + \min_{t \in \mathcal{T}_r}(\text{arr}(t, p_{i+1}) - \text{dep}(t, p_i))$;

**foreach** $p \in \mathcal{P}$ **do** // Init stops to be $\infty \doteq$ unreachable
  $\tau_{\text{min}}(p) \leftarrow \infty$;

$\tau_{\text{init}} \leftarrow 00:00:00$; $\tau_{\text{min}}(p_s) \leftarrow \tau_{\text{init}}$; $\mathcal{M} \leftarrow \{p_s\}$; // Starting conditions

**foreach** $k \leftarrow 1, 2, \ldots$ **do** // Round $k$
  $\mathcal{Q} \leftarrow \emptyset$;
  **foreach** $p \in \mathcal{M}$ **do** // Handle marked stops $p$
    **foreach** $r \in \mathcal{R}_p$ **do** // Handle routes $r$ serving stop $p$
      **if** $\exists p' \in \mathcal{P} : \langle r, p' \rangle \in \mathcal{Q} \mid p' \neq p$ **then** // Keep LAST possible $p$ for $r$
        **if** $p \succ p'$ **then**
          $\mathcal{Q} \leftarrow (\mathcal{Q} \setminus \langle r, p' \rangle) \cup \langle r, p \rangle$;
      **else** // Queue route $r$ with $p$
        $\mathcal{Q} \leftarrow \mathcal{Q} \cup \langle r, p \rangle$;
    $\mathcal{M} \leftarrow \mathcal{M} \setminus p$;
  **foreach** $\langle r, p_{hop} \rangle \in \mathcal{Q}$ **do** // Handle each queued route
    **foreach** $p_i \in r \mid i \in (hop, 0),\ p_i \succ p_{i-1}$ **do** // Traverse in reverse
      **if** $\text{arr}_{\text{min}}(r,\ p_{hop},\ p_i) < \tau_{\text{min}}(p_i)$ **then** // Improvement made
        $\tau_{\text{min}}(p_i) \leftarrow \text{arr}_{\text{min}}(r,\ p_{hop},\ p_i)$; // Update value
        $\mathcal{M} \leftarrow \mathcal{M} \cup p_i$; // Mark stop $p_i$
  **foreach** $\langle p, p' \rangle \in F \mid p' \in \mathcal{M}$ **do** // (Reverse) Footpath faster?
    $\tau_{\text{min}}(p) \leftarrow \min\{\tau_{\text{min}}(p), \tau_{\text{min}}(p') + l(p, p')\}$;
    $\mathcal{M} \leftarrow \mathcal{M} \cup p$; // Mark stop $p$
  **if** $\mathcal{M} = \emptyset$ **then** // Termination condition
    **return**

# Bibliography


[1] Reinhard Bauer et al. “Combining Hierarchical and Goal-Directed Speed-up Techniques for Dijkstra’s Algorithm”. In: *ACM J. Exp. Algorithmics* 15 (Mar. 2010). ISSN: 1084-6654. DOI: 10.1145/1671970.1671976. URL: https://doi.org/10.1145/1671970.1671976.

[2] Daniel Delling, Thomas Pajor, and Renato F. Werneck. “Round-Based Public Transit Routing”. In: *Transportation Science* 49.3 (2015), pp. 591–604. DOI: 10.1287/trsc.2014.0534. eprint: https://doi.org/10.1287/trsc.2014.0534. URL: https://doi.org/10.1287/trsc.2014.0534.

[3] Daniel Delling et al. “Faster Transit Routing by Hyper Partitioning”. In: *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*. Ed. by Gianlorenzo D’Angelo and Twan Dollevoet. Vol. 59. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 8:1–8:14. ISBN: 978-3-95977-042-2. DOI: 10.4230/OASIcs.ATMOS.2017.8. URL: http://drops.dagstuhl.de/opus/volltexte/2017/7896.

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

[5] Mathias Schnee. “Fully Realistic Multi-Criteria Timetable Information Systems”. en. PhD thesis. Darmstadt: Technische Universität, Dezember 2009. URL: http://tuprints.ulb.tu-darmstadt.de/1989/.

[6] Uri Zwick. “Exact and Approximate Distances in Graphs — A Survey”. In: *Algorithms — ESA 2001*. Ed. by Friedhelm Meyer auf der Heide. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 33–48. ISBN: 978-3-540-44676-7.