Arc-Flags for Public Transit Routing

Patrick Steil

patricksteil@freenet.de

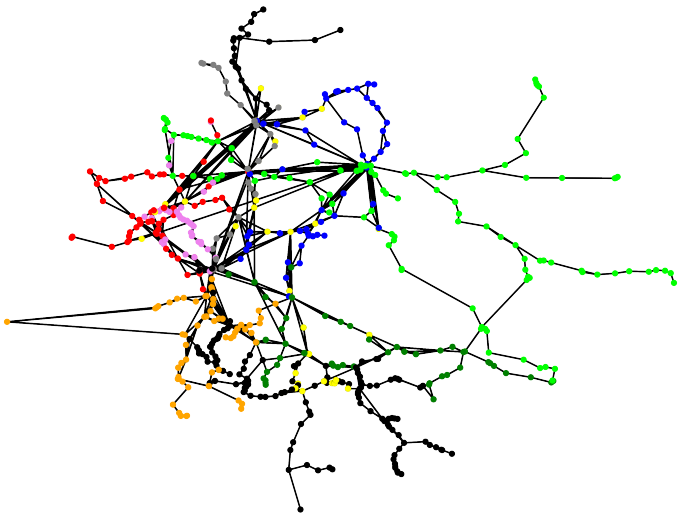
October 26, 2022

3653914

Bachelor Thesis

at

Algorithm Engineering Group Heidelberg
Heidelberg University



Supervisor:
Univ.-Prof. PD. Dr. rer. nat. Christian Schulz

Co-Supervisors:
Ernestine Großmann
Jonas Sauer (*KIT*)

Acknowledgments

I would like to express my gratitude to my supervisor, Prof. Christian Schulz, who guided me throughout this project and my semesters at the university.
Many thanks to Ernestine Großmann and Jonas Sauer, who helped me with every open question and with whom I had many interesting discussions in weekly meetings.
I would also like to thank Prof. Yordan Todorov on this occasion. Last but not least, I thank my family and friends for their continuous support throughout my life.
Likewise, I thank Patrick Brosi for providing us with the S/RE, IC/ICE and Germany datasets from his website `https://gtfs.de/`.

I hereby declare that I have written this thesis independently and have not used any sources or aids other than those indicated.

Heidelberg, October 26, 2022

Patrick Steil

Abstract

This work is about route planning on public transit networks, more specifically it is about improving an existing algorithm with respect to query time. On country-sized networks, other state-of-the-art algorithms are clearly too slow with more than 10 ms per query. Our algorithm outperforms them by a speed-up of $10-18$ resulting in query times less than a millisecond on such networks. The new Arc-TB algorithm consists of the Trip-Based Public Transit Routing (TB) and an adapted version of the Arc-Flags speed-up technique. It is described how the two underlying algorithms work and how Arc-Flags can be applied to the TB algorithm.

Contents

CHAPTER 1

Introduction

Route planning is an important part of our everyday lives, whether we are looking for the fastest route between two places using a car, or we want to get to work using public transport. In the past, for longer car rides, you had to look up on maps yourself which route you wanted to take. You were at the mercy of problems such as traffic jams or road work. Even journeys by public transport had to be planned either by looking at timetables or by asking staff at local stations. Nowadays, there are online services, such as Google Maps [1] or Apple Maps [2], which answer queries automatically. In addition, live data such as traffic jams or train delays are also taken into account when calculating the route. For road networks, algorithms have been optimized to such an extent that queries on large networks, such as Europe, are answered in the millisecond range [1]. Especially speed-up techniques play an important role in archieving such query times. But these algorithms and techniques are not as well suited for public transit networks [4]. This may be because the topology of the underlying graph is different and the optimization criteria are completely different. Car drivers are more interested in the shortest routes between origin and destination, whereas users of public transport often want to optimize not only the arrival time but also the number of transfers or even the length of the walking distance. Information from timetables can be modelled as a graph, and thus solved using Dijkstras Algorithm variants. Two different approaches to graph modelling are known: the *time-expanded* and *time-dependent* model [18]. One disadvantage of Dijkstras Algorithm is the costly use of a priority queue, so current algorithms such as RAPTOR [8], CSA [9] or Trip-Based Public Transit Routing (TB) [23] avoid using one. More information on this in Chapter 3.

Our contribution. The current problem with the above algorithms is that they are all significantly too slow for interactive use on country-sized networks. There are

[1] `https://www.google.de/maps`
[2] `https://www.apple.com/de/maps/`

approaches, such as Public Transit Labelling (PTL), which achieve very fast query times at the expense of higher memory consumption. However, the memory consumption for large networks is enormous and unusable in practice. Therefore, in this work, we present the Arc-TB algorithm, which is based on Trip-Based Public Transit Routing and the speed-up technique Arc-Flags et al. [17]. This technique was originally developed for road networks. Especially on large-scale networks, Arc-TB beats other state-of-the-art algorithms without consuming too much additional memory. The new algorithm depends on a freely selectable parameter k, which can either be increased (resulting in more memory consumption and precomputation time, but faster queries) or decreased depending on the application.

Structure. This work is structured as follows. First, the necessary notation is introduced (see Chapter 2), and then the state-of-the-art algorithms are explained in Chapter 3, as well as the TB algorithm and the concept of Arc-Flags in more detail (see 3.1, 3.2). Our new algorithm Arc-TB is first introduced in Chapter 4 and then in Chapter 5 compared with other algorithms like RAPTOR and CSA on different datasets. At the very end, an outlook on future work is given.

CHAPTER 2

Preliminaries

Public transit networks are described using aperiodic timetables, consisting of a set of stops, routes, trips and transfers. A stop is a physical location where users can get on or off trains, buses or similar. A transfer allows users to move between two stops, for examle on foot or with a rental bike. Routes are sequences of stops that are served by trips at specific times. Trips are vehicles that travel along a route. An important aspect of this model is that trips on the same route cannot *overtake* each other. Two vehicles on the same route *overtake* each other if the latter of the two arrives at a stop earlier than the other.
As mentioned before, several problems exist in the context of time-dependent networks like public transit. Solutions to multi-criteria problems are *Pareto sets* representing journeys. A journey dominates another one if it is not worse in any criterion. It is known that general multi-criteria Pareto optimization is $\mathcal{NP}$-hard [19], but natural criteria are efficiently managed. Among known problems, one is the *earliest arrival problem*, which given an origin stop, destination stop, and departure time computes a journey with the earliest arrival time possible. Combined with minimizing the number of transfers, solutions in the Pareto-set are tupel (π_n, n), where π_n represents the arrival time using n transfers.

2.1 Trip-Based Public Transit Routing

We follow the notation introduced by Witt [23] in his work and explain it in this section.

Trip. A trip t travels along a sequence of stops $\vec{p}(t) = \left\langle p_t^0, p_t^1, \dots \right\rangle$. The set of all stops is denoted by P.

Times. For a stop p_t^i of trip t, we define the arrival time $\tau_{\text{arr}}(t, i)$, the departure time $\tau_{\text{dep}}(t, i)$ and the minimum change time inside a stop by $\Delta\tau_{\text{ch}}(p_t^i)$.

Line. Lines are routes, meaning trips with the same stop sequence are grouped into lines. We denote the set of all lines by $\mathcal{R}$. Since trips of the same line cannot overtake each other, one can order them by

$$t \preceq u \iff \forall i \in [0, |\vec{p}(t)|) : \tau_{\text{arr}}(t, p_t^i) \leq \tau_{\text{arr}}(u, p_u^i)$$

and thus define

$$t \prec u \iff t \preceq u \wedge \exists i \in [0, |\vec{p}(t)|) : \tau_{\text{arr}}(t, p_t^i) < \tau_{\text{arr}}(u, p_u^i)$$

One defines $\vec{p}(L_t) = \vec{p}(t)$ as the stop sequence of the line of trip t and

$$\mathbf{L}(p) = \left\{ (L, i) \mid p = p_L^i,\, L \in \mathcal{R},\, \vec{p}(L) = \left\langle p_L^0, p_L^1, \dots \right\rangle \right\}$$

as the set of lines at a stop p.

Trip segment. The trip segment $p_t^b \to p_t^e$ is a section of the trip t between stop p_t^b and p_t^e where $b < e$.

Transfer. A transfer between trip t and u (with $t \neq u$) is defined by $p_t^b \to p_u^e$, i. e. a passenger can transfer between stop p_t^b and p_u^e in time $\Delta\tau_{\text{fp}}(p_t^b, p_u^e)$ via e.g. a footpath. Only stops that are geographically close to each other are connected by transfers. An example of this would be the different tracks of a stop.
A transfer is called *valid* if the following holds:

$$p_t^b \to p_u^e \implies \tau_{\text{arr}}(t, b) + \Delta\tau_{\text{fp}}(p_t^b, p_u^e) \leq \tau_{\text{dep}}(u, e)$$

See Figure 2.1 as an example.

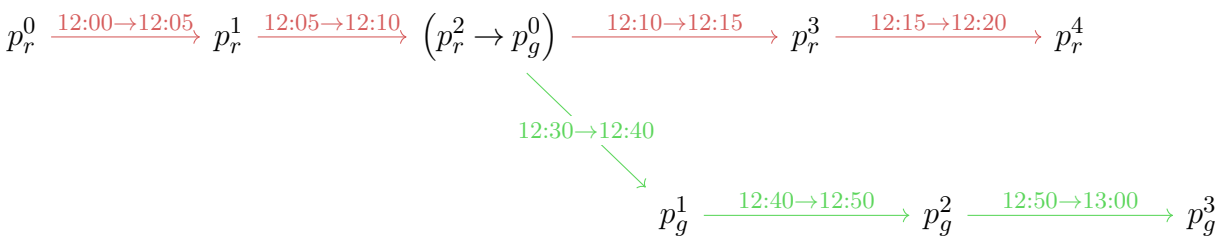


Figure 2.1: Example to illustrate the notation: Let r be the red trip and g be the green one. There is a *valid* transfer $p_r^2 \to p_g^0$ (with $\Delta\tau_{\text{fp}}(p_r^2, p_g^0) < 20\,\text{min}$), and $\mathbf{L}(p_r^2) = \{(g, 0), (r, 2)\}$. And for example $\tau_{\text{arr}}(r, 1) = 12{:}05$ or $\tau_{\text{dep}}(g, 0) = 12{:}30$.

2.2 Arc-Flags

Important concepts for the ARC-FLAGS technique are now explained. The idea behind ARC-FLAGS is to store additional information about shortest paths on edges ("*arcs*"). These can then be used during the query to limit the search space and thus speed it up.

Graph. A graph $G = (V, E)$ is a tuple of a set of nodes V and a set of edges $E \subseteq V \times V$. Edges connect two nodes, can be *directed* or *undirected* and they can be assigned weights using a function $l : E \to \mathbb{R}$. For example, in road networks, such a weight can represent the length of a road or the duration of its travel. Node weights can also be introduced using a weight function $c : V \to \mathbb{R}$. We denote by $n = |V|$ the number of nodes and $m = |E|$ the number of edges. A *path* $p = \left\langle v_1, v_2, \ldots, v_{|p|} \right\rangle$ is a sequence of nodes between v_1 and $v_{|p|}$ such that two consecutive nodes are connected by an edge. The weight of a path is the sum of all edge weights. If $v_1 = v_{|p|}$, p is called a *cycle*.

Shortest Path. If no cylce with negative weight (so-called *negative cycles*) exists, then given a source node s and a target node t, the *shortest path problem* is well-defined. The path $p = \langle s, \ldots, t \rangle$ is called the *shortest path* if there is no other path p' between s and t with a smaller weight. In the following, we denote the *shortest path* between nodes s, t by $p_{\min}(s, t)$. The shortest distance between s and t is denoted by $\pi_{s,t}$, and if no path between them exists, $\pi_{s,t}$ is ∞. Otherwise, the shortest distance $\pi_{s,t}$ is equal to the weight of $p_{\min}(s, t)$. If edge weights are non-negative, then DIJKSTRAS ALGORITHM finds the shortest paths between two nodes in $\mathcal{O}(m + n \log n)$ [10]. For arbitrary edge weights, the BELLMAN-FORD ALGORITHM yields correct shortest paths in $\mathcal{O}(nm)$ [3].

Partitioning Problem. The partitioning problem asks for a partition of a graph $G = (V, E)$ into $k \in \mathbb{N}$ pairwise disjoint blocks V_i, $i \in [0, k)$ such that the number of edges passing between two distinct blocks is minimized. In addition, one wants to limit the size of the blocks so that each block has approximately the same size. For a subset $W \subseteq V$, we denote by $c(W) = \sum_{v \in W} c(v)$ the total weight of the nodes W if node weights are present. Otherwise, the weight of a node is 1, thus $c(W) = |W|$. To limit the size of one block, one uses an *imbalance* parameter $\varepsilon > 0$ and introduces the so-called *balancing constraint* (2.1).

$$\forall i \in [0, k) : |V_i| \leq (1 + \varepsilon) \left\lceil \frac{c(V)}{k} \right\rceil \tag{2.1}$$

For more details on recent advances in (hyper–) graph partitioning, see e. g. [5] or [6].

CHAPTER 3

Related Work

Many different approaches exist to solve the various routing problems such as the *earliest arrival problem* or more complex *multi-criterion problems*. The latter consist of finding Pareto-optimal journeys that meet certain criteria for a given source and target stop and departure time.

CSA. One approach is the Connection Scan Algorithm (CSA) [9] which in its basic variant only minimizes the arrival time. The algorithm is based on an array of connections sorted in ascending order by departure time. A connection c is a trip segment of length 1 and describes the travel of a vehicle between two stops $c_{\text{dep}}, c_{\text{arr}}$ with departure and arrival time $\pi^c_{\text{dep}}, \pi^c_{\text{arr}}$. Given source stop p_{src}, target stop p_{tgt} and departure time π_{dep}, the algorithm works in a dynamic programming fashion. For each stop p, the current best arrival time π_p is stored based on the connections already scanned. The algorithm iterates over all connections c that depart later than the given departure time of the query ($\pi^c_{\text{dep}} \geq \pi_{\text{dep}}$) and updates the arrival time of the current stop $\pi_{c_{\text{arr}}}$ if necessary. As soon as a connection is scanned that departs later than the current best arrival time at the goal destination, the algorithm terminates. It is very cache-efficient, but on large networks, many "unnecessary" connections are scanned. However, this algorithm is very effective for smaller networks, such as those of a transport association like Rhein-Neckar-Verbund[1]. Such networks have about 5 000 stops and nearly 50 000 trips. Country-sized networks consist of over half a million stops and several million trips. To obtain faster query times on country-sized networks, Strasser and Wagner [21] introduced the Accelerated Connection Scan Algorithm (ACSA) extension. The basic idea here is a combination of multilevel overlay graphs and CSA. For more information on overlay graphs, see [14]. This reduces the number of "unnecessary" connections that are scanned during a query.

[1] `https://www.rnv-online.de/fahrtinfo/`

RAPTOR. Another approach is Round-bAsed Public Transit Optimized Router (RAPTOR) [8], which works on lines and calculates Pareto-optimal solutions for arrival time and number of transfers. The algorithm works in rounds, where in each round $k \in \mathbb{N}$ a best arrival time is calculated. During a query, RAPTOR traverses the routes in a breadth-first search manner, i. e. it collects routes in a queue, iterates over them, and checks if transfers to new routes are possible. In each round, the current best arrival time at every stop is stored until it finds no improvements. Advantages of RAPTOR are the absence of pre-computation and the possibility of parallelization, e. g. when iterating over the stops of the routes. See Figure 3.1 as an illustration of the query.

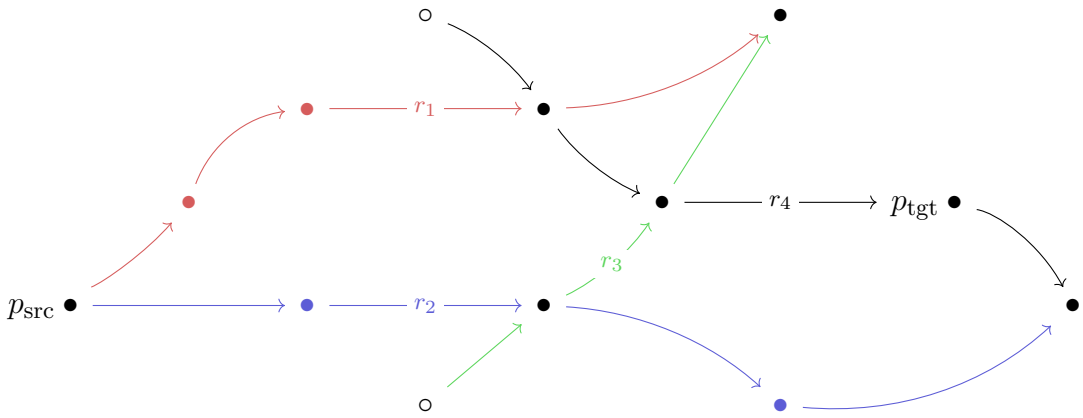


Figure 3.1: Illustration of the RAPTOR query from p_{src} to p_{tgt}. The two routes r_1, r_2 are scanned in the first round and after that, the remaining routes r_3, r_4. Stops denoted by $\circ$ are not scanned by the algorithm.

PTL. A very efficient speed-up technique is Public Transit Labelling (PTL) [7], which is based on the idea of Hub Labelling [11]. For each node $v \in V$ one stores a set of nodes, so-called *hubs*, together with the shortest distances to v. *Shortest paths queries* between two nodes can be answered very fast by just computing the intersection of the two *hubs*. However, this method only returns the shortest distance between u and v, not the path. To obtain the shortest path, each *hub* must also store the path that leads to it, which results in extremely high memory consumption. Delling et al. [7] achieve fast results, which are orders of magnitude faster than state-of-the-art, but the memory consumption is already very large and they do not perform path unpacking, meaning their algorithm only reports arrival times.

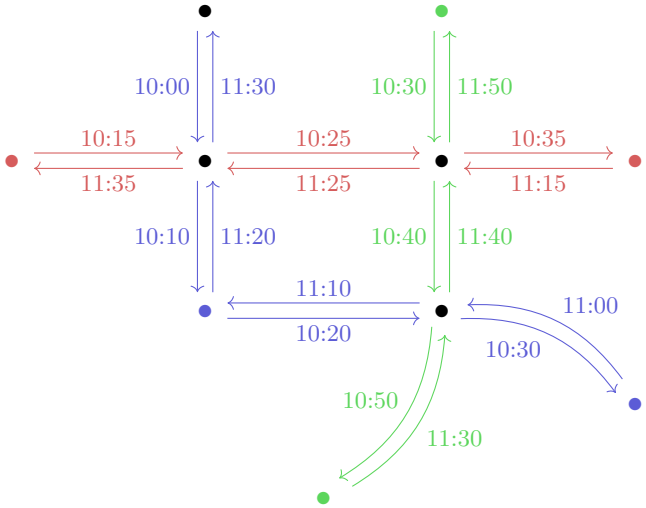
Graph based. In addition, two major graph-based models exist, a *time-expanded* and *time-dependent* model. The idea of the *time-expanded* model is to introduce nodes for each event (e. g. boarding or changing). Edges connect stops either because they belong to the same trip or because transfers are possible at these events. A big advantage of this way of modelling is the fact that all edges have equal weight. Therefore, speed-up techniques are easily adaptable. Moreover, the time component of the graph makes it a directed acyclic graph (DAG). To compute shortest paths, one does not need a priority queue, because breadth-first searches provide correct shortest paths (see [15], chapter "Shortest Paths"). Despite the fact that this model has a simple structure, speed-up techniques, which are known from road networks, are not effective [4]. Moreover, graphs of this type become very large. On the other hand, the *time-dependent* approach models stops as nodes and connections between them as edges. Weights of edges are piecewise linear functions mapping departure to arrival times. For this type of graph, Dijkstras Algorithm variants are able to solve multi-criteria problems, see e. g. [12] or [22]. As an illustration of the two models described, see Figure 3.2. For an overview of recent advances in algorithms for route planning, refer to [2].

3.1 Trip-Based Public Transit Routing

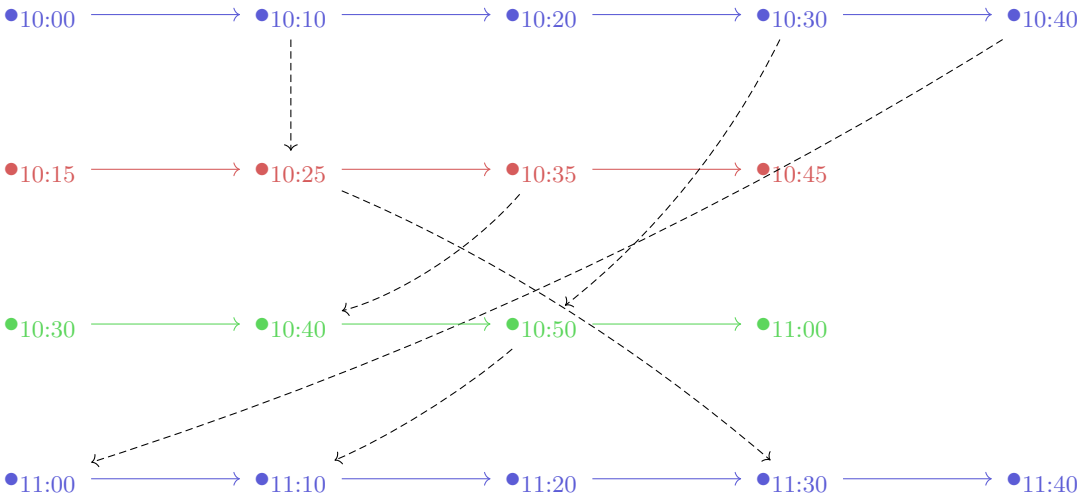
The Trip-Based Public Transit Routing (TB) algorithm was published by Sascha Witt in 2015 [23]. The basic idea of the TB algorithm is based on a breadth-first search through a *time-expanded* graph, where the levels of the breadth first search (BFS) indicate the number of transfers. Thus, the algorithm provides Pareto-optimal results in terms of arrival time and the number of transfers. The DAG property of the underlying graph allows correct computation of shortest paths without a costly priority queue. The algorithm consists of two phases: the pre-computation and the query. Both are explained in the next sections.

3.1.1 Preprocessing

In the preprocessing, all possible transfers between trips are calculated, although some of them are not necessary to compute Pareto-optimal solutions. Therefore, the pre-computation is also divided into two steps: initial computation and reduction.
The initial computation consists of iterating over all trips t and their stops $p_t^i, \forall i \in [0, |\vec{p}(t)|)$, collecting every (transfer-) reachable stop q. Next, one finds the earliest trip u of a line L by iterating over $(L, j) \in \mathbf{L}(q)$ in order to create a *valid* transfer $p_t^i \to p_u^j$. Since trips of the same route do not overtake each other, one does not need to compute transfers to later trips of line L. The reason is that a passenger cannot improve his arrival time by taking a later trip. Another observation is that there is no need to calculate transfers between the first and the last stop of a trip, as passengers



(a) *time-dependent* model.



(b) *time-expanded* model.

Figure 3.2: Example graphs to show the differences between between *time-dependent* and *time-expanded*. **Note**: Both graphs model the same timetable. Due to space reasons, not the entire *time-expanded* graph is shown.

do not board a train to get off at the same stop and start another trip.
After the initial computation, some transfers can be removed since they do not contribute to any Pareto-optimal solution. For example let $p_t^i \to p_u^j$ be a transfer where $p_u^{j+1} = p_t^{i-1}$ and

$$\tau_{\text{arr}}(t, i-1) + \Delta\tau_{\text{ch}}(p_t^{i-1}) \leq \tau_{\text{dep}}(u, j+1)$$

holds. This unnecessary transfer is called a *U-turn transfer* since one can access u from t at the preceding stop.
Another approach to reduce the number of transfers is to keep only those transfers that lead to improved arrival times. For each stop p_t^i along a trip t, one remembers both the earliest arrival time $\tau_{\text{A}}\left(p_t^i\right)$ and the earliest change time $\tau_{\text{C}}\left(p_t^i\right)$. If there is no minimum change time, only τ_{A} is used. Subsequently, one runs "backwards" over the stops of a trip t, meaning $i \in \{|\vec{p}(t)|-1, |\vec{p}(t)|-2, \ldots, 0\}$. For each stop p_t^i, one updates both times τ_{A}, τ_{C} according to (3.1) and (3.2).

$$\tau_{\text{A}}\left(p_t^i\right) = \min\left\{\tau_{\text{A}}\left(p_t^i\right), \tau_{\text{arr}}(t,i)\right\} \tag{3.1}$$

$$\tau_{\text{C}}\left(p_t^i\right) = \min\left\{\tau_{\text{C}}\left(p_t^i\right), \tau_{\text{arr}}(t,i) + \Delta\tau_{\text{ch}}(p_t^i)\right\} \tag{3.2}$$

Similarly, the times for all stops accessible by foot are updated. Now one checks for each transfer $p_t^i \to p_u^j$ whether it is necessary to improve at least one of the two times τ_{A}, τ_{C}. To do this, the algorithm iterates over all stops p_u^k, $\forall k \in (j, |\vec{p}(u)|)$ along the trip u and also over all stops that can be reached by foot to see if it can improve one of the times. If yes, it keeps the transfer, otherwise it discards it.
The entire pre-computation can be parallelized, as each trip can be processed independently. This allows a pre-computation of a few minutes on country-sized networks (if the given transfer set is not too big). The author mentions that the number of transfers can be reduced even further by computing all possible earliest arrival queries between all stops and keeping only the used transfers. However, Witt [23] says that this is too costly and would make the pre-computation much slower compared to the fast reduction steps. But exactly such "unnecessary" transfers are found during the Arc-Flags pre-calculation and subsequently removed, see Section 4.1.

3.1.2 Query

Given a source stop p_{src}, target stop p_{tgt} and departure time τ_{dep}, a query runs as follows: in a queue, the algorithm collects trips that are reachable from the source stop and then iterates in a breadth-first search over all trip segments that are reachable via transfers. More precisely: the algorithm has n distinct queues Q_n of trip segments, where n corresponds to the maximum number of transfers. In his work, Witt bounded the number of distinct queues by $n \leq 15$. Additionally, one maintains an index $R(t)$ for every trip t indicating which parts of t the algorithm already scanned. In the

following set $\mathcal{L}$ (see 3.3) one stores all lines that lead to the destination, either because the destination is served by the line or because a passenger arrives at a station from which he can reach the destination by walking:

$$\begin{aligned}\mathcal{L} = &\{(L,i,0) \mid (L,i) \in \mathbf{L}(p_{\text{tgt}})\} \\ &\cup \left\{\left(L,i,\Delta\tau_{\text{fp}}(q,p_{\text{tgt}})\right) \mid (L,i) \in \mathbf{L}(q) \wedge \exists \text{transfer from } q \text{ to } p_{\text{tgt}}\right\}\end{aligned} \tag{3.3}$$

To store the Pareto-optimal results, one maintains a result set J with tupels of arrival time and number of transfers. The algorithm starts collecting all reachable lines

$$(L,i) \in \mathbf{L}(q) : \forall q \in \{p_{\text{src}}\} \cup \{b \in P \mid \exists \text{ transfer from stop } p_{\text{src}} \text{ to } b\}, \tag{3.4}$$

and the first trip t (reachable at time τ) from each line and inserts the trip segment $p_t^i \to p_t^{R(t)}$ into Q_0. Then, the algorithm updates $R(u) \leftarrow \min\{R(u), i\}$ for every trip u where $t \preceq u \wedge L_t = L_u$ to avoid scanning the same stop sequence of this route using later trips again. Now the algorithm starts the actual BFS work on the queues $Q_0, Q_1, \ldots$. For every trip segment $p_t^b \to p_t^e \in Q_k, k \in [0,n]$, the algorithm takes the following three steps:

1. **Reached:** It is checked whether we reach our target p_{tgt}. If yes, we iterate over all tripels $(L_t, i, \Delta\tau) \in \mathcal{L}$ with $i > b$ and *insert* $(\tau_{\text{arr}}(t,i) + \Delta\tau, k)$ into the result set J. The tuple is only *inserted* if the current arrival time $\tau_{\text{arr}}(t,i) + \Delta\tau$ is better than the previous arrival time τ_{prev} from $(\tau_{\text{prev}}, k) \in J$. If the tupel is inserted, the previous arrival time is replaced.

2. **Pruning:** We test if this trip segment can be omitted by checking if $\tau_{\text{arr}}(t,i)$ is worse than the previously found arrival time at the target.

3. **Expanding:** If the trip segment was not discarded in the previous step, the algorithm looks at all transfers originating from that trip segment. We only add transfer $p_t^i \to p_u^j$ (with $b < i \leq e$) to the queue Q_{k+1} if $j < R(u)$. If we add the transfer, we update $R(u)$ analogously as before.

A high-level pseudocode can be found in Algorithm 1; the full algorithm can be found in Appendix 6.

3.2 Arc-Flags

Arc-Flags is a speed-up technique for traditional shortest path algorithms like Dijkstras Algorithm. Möhring et al. [17] report a speed-up factor of 500 when using bidirectional Dijkstras Algorithm. The original Dijkstras Algorithm algorithm maintains the current best distance for each node and orders all unfinished

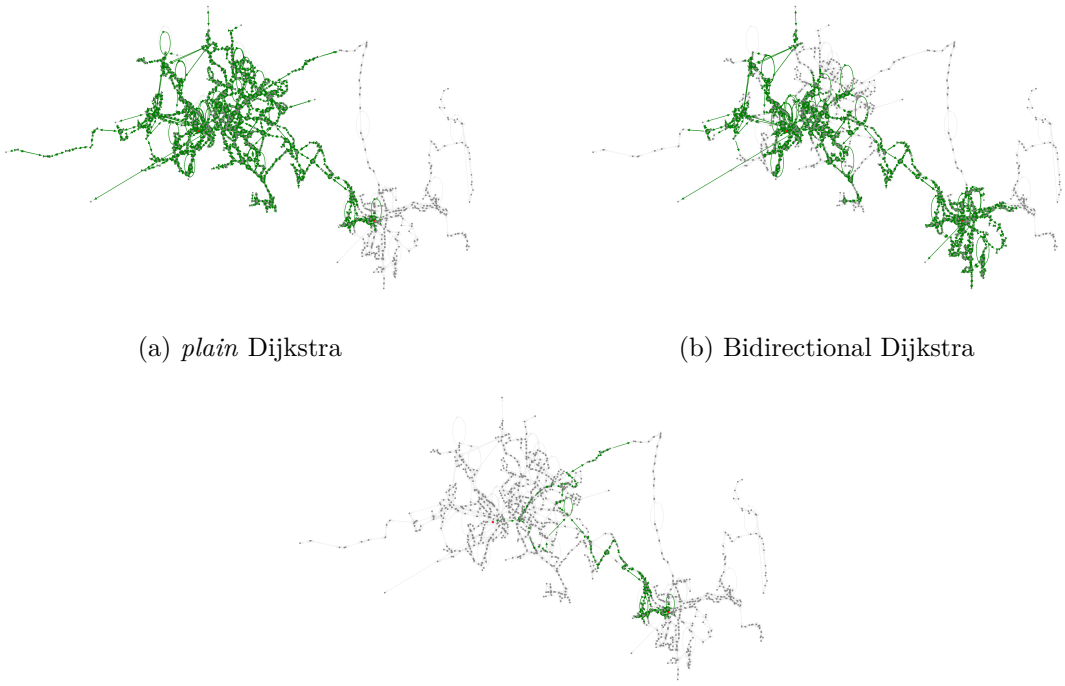
Algorithm 1: Earliest arrival query - this pseudocode is from Witt's paper [23]

Input : Timetable, transfer set T, source stop p_{src}, target stop p_{tgt}, departure time τ
Output: Result set J

```
/* initialise every data structure needed                    */
/* see the other pseudocode 6 for more detail                */
/* now we start the BFS phase                                */
```

$\tau_{\min} \leftarrow \infty$
$n \leftarrow 0$
while $Q_n \neq \emptyset$ **do**
  foreach *trip segment* $p_t^b \rightarrow p_t^e \in Q_n$ **do**
    foreach $(L_t, i, \Delta\tau) \in \mathcal{L}$ *with* $b < i \wedge \tau_{\text{arr}}(t,i) + \Delta\tau < \tau_{\min}$ **do**
      $\tau_{\min} \leftarrow \tau_{\text{arr}}(t,i) + \Delta\tau$
      update result set and $J \leftarrow J \cup \{(\tau_{\min}, n)\}$
    end
    if $\tau_{\text{arr}}(t, b+1) < \tau_{\min}$ **then**
      foreach *transfer* $p_t^i \rightarrow p_u^j \in T$ *with* $b < i \leq e$ **do** `ENQUEUE`$(u,j,n+1)$
    end
  end
  $n \leftarrow n+1$
end
procedure `ENQUEUE`(*trip* t, *index* i, *number of transfers* k)**:**
  if $i < R(t)$ **then**
    $Q_n \leftarrow Q_n \cup \left\{ p_t^i \leftarrow p_t^{R(t)} \right\}$
    foreach *trip* u *with* $t \preceq u \wedge L_t = L_u$ **do** $R(u) \leftarrow \min\{R(u), i\}$
  end

nodes according to their current distance. In each step, the algorithm considers all outgoing edges of the node with the minimum distance and thus tries to improve the distances of the neighbours. In this way, the algorithm explores the graph and eventually finds the correct distances and paths to all nodes in the graph. The basic idea of bidirectional Dijkstras Algorithm is to search for the shortest path between two nodes simultaneously from the source, but also backwards from the target (on the reverse graph). If parallelized, bidirectional Dijkstras Algorithm is on average a factor of 4 faster than the original Dijkstras Algorithm algorithm. See Algorithm 5 for the pseudocode of bidirectional Dijkstras Algorithm and Figure 3.3 to compare the search spaces of the different algorithms.



(a) *plain* Dijkstra

(b) Bidirectional Dijkstra

(c) Arc-Flag & Dijkstra

Figure 3.3: Plotted in green are edges scanned by the mentioned algorithms during the same shortest path query.

3.2.1 Overview

The basic idea of Arc-Flags is to perform most of the work during the preprocessing phase and to save time during a query.
For each vertex $v \in V$, let $S(v)$ be the set of edges that lie on a shortest path from v. During a shortest path query starting from v, Dijkstra only needs to check the edges $e \in S(v) \subseteq E$, i. e. only consider edges from any shortest path from v. Clearly, it is pointless memory-wise to store these sets for all nodes. Therefore, one partitions the set of nodes into k blocks of approximately the same size. More precisely, one defines a function $r : V \to \{1, \ldots, k\}$ which assigns a number between 1 and k to each node. For each edge $e \in E$ one defines a bit vector $b_e : \{1, \ldots, k\} \to \{1, 0\}$ of length k, where bit i represents block i. These bits are also called flags, hence the name Arc-Flags. The bits for an edge $e \in E$ are defined using Equation (3.5). The flag i is set to 1 as soon as the edge is part of a shortest path that leads to a node of block i.

$$b_e(i) = 1 \iff \exists p_{\min}(s,t),\ s,t \in V :\ r(t) = i \wedge e = (u,v) \in p_{\min}(s,t) \tag{3.5}$$

Thus one can restrict Dijkstras Algorithm in finding the shortest path between s,t, namely one now only looks at edges e for which $b_e(r(t)) = 1$ holds. See Algo-

rithm 14 for pseudocode for the original Dijkstras Algorithm algorithm combined with Arc-Flags. The additional memory consumption is $\Theta(km)$ since exactly k bits must be stored for each of the m edges and is manageable for $k \ll n$. The larger k, the smaller the partitions and the smaller the search space during the query. Figure 3.4 illustrates an example.

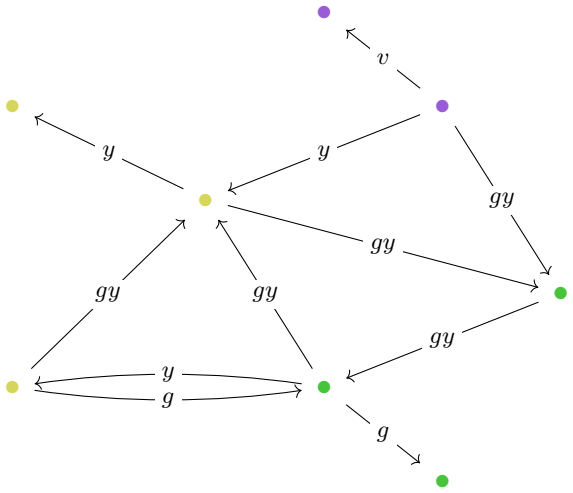


Figure 3.4: Example graph with Arc-Flags. Nodes with same color denote same partition. The flags set to 1 are denoted by the starting character of the colour, **y**ellow, **v**iolet and **g**reen.

3.2.2 Partitioning

The question with Arc-Flags is: how do you partition a given graph to get "good" flags? To answer this question, we define $N_G(\mathcal{P})$ to be the number of bits set to 1 on all edges of G given the partition $\mathcal{P}$. The fewer bits set to 1, the smaller the search space of the Dijkstras Algorithm search, since more edges can be excluded. Therefore, the query will be better on average, the smaller $N_G(\mathcal{P})$ is. Thus, we can describe which partitions are "better" than others (given graph G). Let $\mathcal{P}, \mathcal{P}'$ be two partitions:

$$\mathcal{P} \preceq \mathcal{P}' \iff N_G(\mathcal{P}) \leq N_G(\mathcal{P}')$$

Möhring et al. [17] describe 4 different ways to compute partitions. The simplest method is the so-called *grid method*. The basic idea here is to plot the graph on a two-dimensional plane and then place a grid on top of it. Each cell in the grid corresponds to a partition block. But since this grid approach does not transfer any

Algorithm 2: Dijkstras Algorithm with Arc-Flags.

Input : graph $G = (V, E)$, edge weight function $l : E \to \mathbb{R}_0^+$, source node s, target node t, block function $r : V \to \{1, \ldots, k\}$, bit vectors $b_e : \{1, \ldots, k\} \to \{1, 0\}\ \forall e \in E$

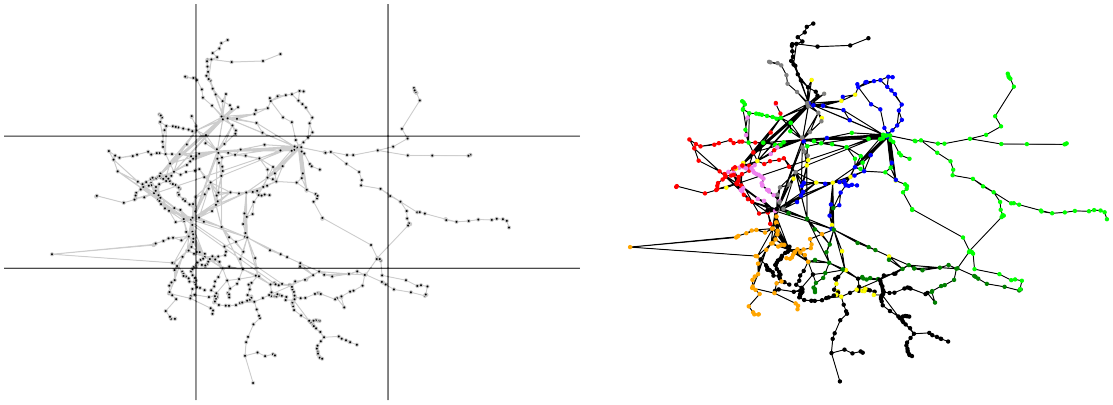
/* $d[v]$ keeps track of the distance from s to v */
/* $s[v]$ marks if node v has already been scanned */

```
1  foreach v ∈ V do d[v] ← ∞, s[v] ← 0
2  d[s] ← 0
3  Q ← {(s, 0)}
4  while Q ≠ ∅ do
5      u ← Q.deleteMin(), s[u] ← 1
6      if u = t then return
7      foreach e = (u, v) ∈ E do
8          if b_e(r(t)) = 1 ∧ s[v] = 0 ∧ d[u] + l(e) < d[v] then
9              d[v] ← d[u] + l(e)
10             if v ∈ Q then Q.decreaseKey(v, d[v])
11             else Q.insert(v, d[v])
12         end
13     end
14 end
```

topological properties of the graph into the partition, this method performed the worst. The next two approaches are very similar in that partitions are computed using data structures that store points in space. The first of the two methods uses a *quad-tree*, a tree structure which divides the underlying space into four quadrants in each iteration. This recursion can be stopped if there are fewer than $b \in \mathbb{N}$ nodes in the current region. "Node-dense" parts of the graph are partitioned into multiple regions, thus to some extent, the topology of the graph is carried over into the partition. The other approach is based on the use of *k-d-trees*, which also recursively divides the space, not in equal-sized quarters, but depending on the distribution of the nodes. In each step, the median of a region is calculated w.r.t. an alternating component (such as latitude or longitude) and based on the median, the region is thus divided into two. Thus, the partition of the *k-d-trees* becomes even "finer" than that of the *quad-trees*. The last, and best approach compared to the other methods, is to use a black–box partitioner, such as METIS [2] or KaHIP [3]. A major advantage of this approach is that the partitioner does not operate on an embedding of the graph, but makes decisions based only on the topology. Moreover, since the partitions have significantly fewer *cross-edges*, i. e. edges which run between different blocks, the pre-computation time

[2] `https://github.com/KarypisLab/METIS`
[3] `https://github.com/KaHIP/KaHIP`

is significantly lower than for all three previous methods. This is because an efficient calculation of the flags calculates the shortest paths starting from all *cross-edges*, and therefore, the number of *cross-edges* is crucial for the pre-computation time. We elaborate on pre-computation in Section 3.2.3. Figure 3.5 shows the difference between the *grid method* and the use of a black–box partitioner.



(a) 3×3 *grid-method* (b) KaHIP with $k = 9$, imbalance 20%

Figure 3.5: Comparison between partitioning with *grid-method* and using KaHIP on a german long distance train network.

3.2.3 Computation

A direct but very inefficient way to compute all flags is to compute all shortest paths from all nodes to all other nodes (so called *all-pair-shortest-path*). Thus, the effort is $\mathcal{O}\left(nm + n^2 \log n\right)$ if Dijkstras Algorithm can be used to find shortest paths. Otherwise, it is $\mathcal{O}\left(n^2 m\right)$. But Möhring et al. show in [13] that one can compute correct flags without *all-pair-shortest-path*. The basic idea is based on the observation that all shortest paths to a node $v \in V$ in a block j pass through so-called *boundary arcs*. An edge $e = (u, v)$ is called *boundary arc* if $r(u) \neq r(v)$ holds; u, v are called *boundary nodes*. Therefore, it is sufficient to compute shortest paths to all *boundary nodes*, or equivalently: backward shortest paths from all *boundary nodes*. For all edges e within a block j, $b_e(j) = 1$ is set, and the bits of the other blocks depend on the backward shortest paths tree from the *boundary nodes*.

CHAPTER 4

Arc-Flags Meet Trip-Based Public Transit Routing

In the following, we will explain how the Arc-Flags technique can be adapted to the TB algorithm. We call the resulting algorithm Arc-TB. As with Arc-Flags, there are two phases, the pre-computation and the query.
Unlike Arc-Flags, in the TB algorithm one does not look for the shortest path, but for Pareto-optimal paths in terms of arrival time and the number of transfers. As explained in Section 3.1, the query is based on a breadth-first search. Edges in this *time-expanded* graph correspond to transfers, i. e. in the Arc-TB algorithm one flags transfers between trips. This makes sense because a user only transfers if this transfer takes him to the destination. More formally: one defines a bit vector $b_{p_t^b \to p_u^e} : \{1, \dots, k\} \to \{1, 0\}$ for every transfer $p_t^b \to p_u^e$ and a function $r : P \to \{1, \dots, k\}$, which maps stops to blocks. Let $p_J(p_{\text{src}}, p_{\text{tgt}}, \tau)$ be a Pareto-optimal path between stops $p_{\text{src}}, p_{\text{tgt}} \in P$ departing at time τ, meaning this path corresponds to one entry in the result set J of the TB Algorithm. Then, the flags for any transfer $p_t^b \to p_u^e$ are defined using Equation (4.1). This means that the ith bit of a transfer is set if that transfer is part of some Pareto-optimal path leading to a stop in the ith partition.

$$\begin{aligned} b_{p_t^b \to p_u^e}(i) = 1 \iff \exists p_J(p_{\text{src}}, p_{\text{tgt}}, \tau),\ p_{\text{src}}, p_{\text{tgt}} \in P,\ \text{time}\,\tau : \\ r(p_{\text{tgt}}) = i \wedge p_t^b \to p_u^e \in p_J(p_{\text{src}}, p_{\text{tgt}}, \tau) \end{aligned} \tag{4.1}$$

4.1 Preprocessing

The pre-computation includes the partitioning as well as the flag calculation; both are explained in the following.

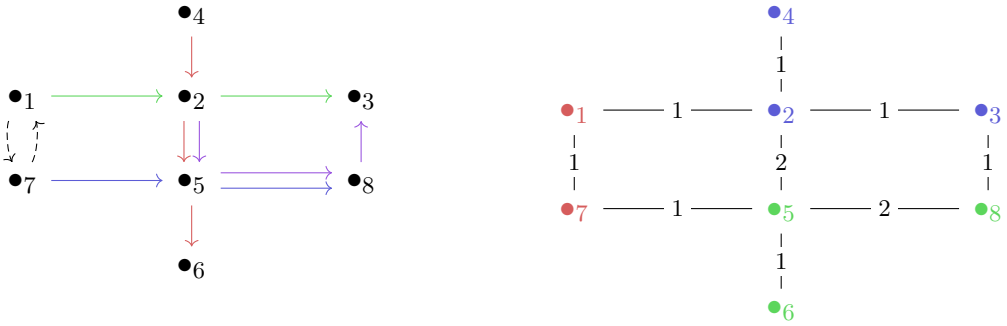


Figure 4.1: On the left is an example network with 4 trips (red, green, blue, purple) and a transfer (dashed arrows). On the right you can see the (partitioned) *layout* graph, where only edge weights were calculated using the trips and transfers. Note that the colours represent the partitions.

4.1.1 Partitioning

Given a data set, the so-called *layout* graph is generated first (see Figure 4.1). The node set $V = P$ corresponds to all stops and edges correspond to connections between two nodes, i. e. for each trip segment of length 1 and each transfer one edge is generated. The weight c_e of an edge $e = (u, v)$ between stops $u, v \in V$ corresponds to the number of trips which operate between u and v.
Based on this graph, one creates the partition $\mathcal{P}$, which is needed for the flag pre-computation using the partitioning program KaHIP[1]. KaHIP is a set of open-source graph partitioning algorithms developed by Sanders and Schulz [20, 16]. We used the program `kaffpaE` with the configuration `-preconfiguration=ssocial` and an imbalance of 20% in all of our experiments. KaHIP is based on a multilevel approach, i.e. the input graph is coarsened, initially partitioned and locally improved during uncoarsening. It has been shown that overall better results are obtained when coarsening is computed using clustering, rather than edge matching as usual (hence we use `preconfiguration=ssocial`). The number of partitions $k \in \mathbb{N}$ can be chosen depending on the application. The larger k, the fewer stops are in a partition, therefore the search space is more restricted during a query and the algorithm is faster. However, this comes at the cost of pre-computation time and memory consumption.

4.1.2 Flag-computation

For each trip t and all stops p_t^i on the trip, the algorithm performs calcArcFlags and computes the Pareto sets with respect to arrival time and the number of transfers to all stops in P (*all-to-all* pre-computation). See Algorithm 3 for the pseudocode

[1] `https://github.com/KaHIP/KaHIP`

of calcArcFlags. After each BFS round, the current best path is unpacked from all stops using stored parent pointers, correctly flagging all transfers. For each stop, one remembers which trip t and stop index i led to the current best arrival time, and for each tuple (t, i) of trip and stop index, one stores the transfer that led to that trip. This step is called unwindParentPointer.

Improvements. Some parts of the algorithm can be optimized. The simplest improvement is in unwindParentPointer: One does not need to check all stops in each iteration, but only the stops where the arrival time has improved.
Another aspect lies in the breadth-first search of calcArcFlags because every outgoing transfer of a trip segment is considered. This means that the breadth-first search traverses the whole graph. Clearly, this is inefficient. In the normal query (see Algorithm 1), transfers of trips are not considered if they result in "too late" trips (compared to the previous best arrival time $\tau_{\min}$). Similar considerations now apply here. For every stop $p \in P$ one keeps track of the current best arrival time $\tau_{\min}(p)$. Then, transfers of trip segments $p_t^b \to p_t^e$ are not considered if the following equation holds:

$$\tau_{\text{arr}}(t, b) > \max\left\{\tau_{\min}\left(p_t^i\right); \forall i \in [b, e]\right\}$$

Another speed-up aspect is based on so-called *transferless stops*. A stop p_t^i of a trip t is called a *transferless stop* if there is no transfer of the form $p_t^i \to p_u^j$. The naive approach executes calcArcFlags from all stops p_t^i, i.e., from each stop a kind of *one-to-all* query. But for correct flag computation, it is not necessary to perform calcArcFlags from a *transferless stop*, thus one can skip these nodes. See Figure 4.2 for an example.

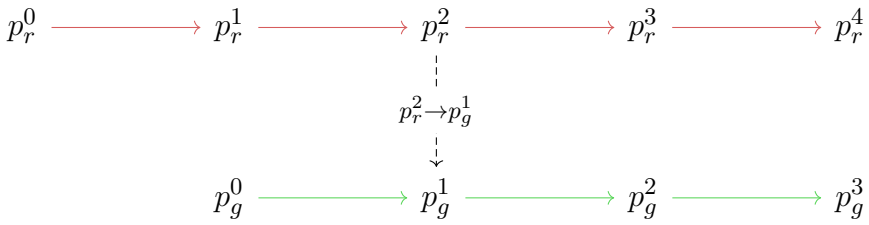


Figure 4.2: Example to demonstrate *transferless stops*. It is not necessary to run the calcArcFlags method from, p_r^0 or p_r^1, since in both cases the first transfer is found only from p_r^2. I.e. for the red trip the method calcArcFlags of p_r^2 has to be executed only once to calculate flags of transfers correctly.

Taking advantage of *transferless stops*, we can disregard between $25 - 50\%$ of all stops p_t^i in the data sets we use. However, this does not correlate with a speed-up factor, as *transferless stops* which are very late in the evening do not save the

same amount of time as a *transferless stop* early in the morning. This is because a passenger can reach more trips during the day if he starts travelling early in the morning rather than in the evening. In the experiments, we achieve a speed-up of about $2-4$ by skipping *transferless stops*. The computation calcArcFlags can be parallelized, since each execution of calcArcFlags can be performed independently. This can be achieved by each process executing calcArcFlags for each stop along an assigned trip. Finally, the flags of all processes are merged using logical OR operations. As already mentioned in Section 3.1.1, all transfers which have no flags set to 1 can be removed, as they do not belong to any Pareto-optimal solution. In the experiments, we show that the proportion of these edges is $2-5\%$.

4.2 Query

The Arc-TB query is only a slight modification of the original query. One is only allowed to add transfers that have the correct flag set to the next queue. See Algorithm 4 for pseudocode of this modified query.

Algorithm 3: calcArcFlags - one to all preprocessing

Input : Timetable, transfer set T, trip u, stop index i

$J \leftarrow \emptyset$
foreach $k = 0, 1, \ldots$ **do** $Q_k \leftarrow \emptyset$
foreach *trip* t **do**
  $R(t) \leftarrow \infty$
  foreach $i \in [0, |\vec{p}(t)|)$ **do** $\text{par}(t, i) = \text{NULL}$
end
foreach $q \in P$ **do** $\tau_{\min,q} \leftarrow \infty$, $\text{stopEventPar}(q) = (\text{NULL}, \text{NULL})$
`ENQUEUE(`u, i, 0`)`
$n \leftarrow 0$
while $Q_n \neq \emptyset$ **do**
  foreach *trip segment* $p_t^b \rightarrow p_t^e \in Q_n$ **do**
    foreach $i \in [b, e]$ **do**
      if $\tau_{\text{arr}}(t, i) \geq \tau_{\min,p_t^i}$ **then** continue
      $\tau_{\min,p_t^i} \leftarrow \tau_{\text{arr}}(t, i)$
      $\text{stopEventPar}\left(p_t^i\right) \leftarrow (t, i)$
      foreach *transfer* $p_t^i \rightarrow p_u^j \in T$ **do**
        $\text{par}(u, j) \leftarrow \left(p_t^i \rightarrow p_u^j\right)$
        `ENQUEUE(`u,j,$n+1$`)`
      end
    end
  end
  $n \leftarrow n + 1$
  `unwindParentPointer()`
end
procedure `unwindParentPointer():`
  foreach $p \in P$ **do**
    $(t, i) \leftarrow \text{stopEventPar}(p)$
    transfer $tr \leftarrow \text{par}(t, i)$
    flags $f \leftarrow \{0, \ldots, 0\}$
    $f(r(p)) = 1$
    while $tr \neq \text{NULL}$ **do**
      `/* `$\oplus$ `means bitwise or */`
      $b_{tr} \leftarrow b_{tr} \oplus f$
      `/* transfer` tr `has the form` $p_k^o \rightarrow p_u^j$ `*/`
      $tr \leftarrow \text{par}(k, o)$
    end
  end

Algorithm 4: ARC-TB - Earliest arrival query

Input : Timetable, transfer set T, source stop p_{src}, target stop p_{tgt}, departure time τ, partition function $r : P \to \{1,\dots,k\}$, for all transfer $p_t^b \to p_u^e$: $b_{p_t^b \to p_u^e} : \{1,\dots,k\} \to \{1,0\}$

Output: Result set J

```
/* ...                                                              */
/* main query is same as before, only difference is in the BFS      */
```

$\tau_{\min} \leftarrow \infty$
$n \leftarrow 0$
while $Q_n \neq \emptyset$ **do**
  foreach *trip segment* $p_t^b \to p_t^e \in Q_n$ **do**
    foreach $(L_t, i, \Delta\tau) \in \mathcal{L}$ *with* $b < i \wedge \tau_{\text{arr}}(t,i) + \Delta\tau < \tau_{\min}$ **do**
      $\tau_{\min} \leftarrow \tau_{\text{arr}}(t,i) + \Delta\tau$
      update result set and $J \leftarrow J \cup \{(\tau_{\min}, n)\}$
    end
    if $\tau_{\text{arr}}(t, b+1) < \tau_{\min}$ **then**
      foreach *transfer* $p_t^i \to p_u^j \in T$ *with* $b < i \leq e$ **do**
        if $b_{p_t^i \to p_u^j}(r(p_{\text{tgt}})) = 1$ **then**
          ENQUEUE$(u, j, n+1)$
        end
      end
    end
  end
  $n \leftarrow n + 1$
end

CHAPTER 5

Experiments

We perform our experiments on two machines A and B. Machine B is used to compute the experiments for the large country-sized instances and A for the rest of the experiments. Machine A is an `Intel Xeon Silver` 4216, 16-`Core CPU` with 96`GB DDR4-2933 DIMM` clocked at 2.10 GHz (max. 3.20 GHz), machine B is an `AMD EPYC` 7702P, 64-`Core CPU` with `1 024 GB DDR4-3200 DIM` clocked at 2.00 GHz (max. 3.35 GHz). All algorithms are implemented in `C++` and compiled with `g++` and the following parameters: `-fopenmp -O3 -march=native -std=c++17 -pipe`.
CSA computes solutions that minimize arrival time, whereas Arc-TB, TB, and RAPTOR compute Pareto-optimal solutions, i.e., minimize the number of transfers in addition to arrival time.
We do not expect a large speed-up on small datasets, but rather on country-sized networks such as Germany or Switzerland. This is due to the nature of the Arc-Flags technique. The results confirm our assumption. All results are based on 10 000 random queries, meaning source, destination and departure time are chosen uniformly at random.

5.1 Datasets

All datasets (Rhein-Neckar-Verbund[1], IC/ICE[2], Karlsruhe[3], S/RE[2], Paris[4], Sweden[5], Switzerland[6] and Germany[2]) used in our experiments are in GTFS format (*General Transit Feed Specification*[7]) and are listed in Table 5.1.

[1] `https://opendata.rnv-online.de/dataset/gtfs-general-transit-feed-specification`
[2] `https://gtfs.de/`
[3] `https://www.kvv.de/fahrplan/fahrplaene/open-data.html`
[4] `https://navitia.opendatasoft.com/explore/`
[5] `https://trafiklab.se/`
[6] `https://gtfs.geops.ch/`

dataset	stops	routes	trips	local	long-distance
IC/ICE	1 599	2 231	2 877		•
Rhein-Neckar-Verbund	2 080	1 078	15 843	•	
Karlsruhe	4 014	2 929	50 970	•	
S/RE	14 178	18 796	67 408	○	○
Paris	41 957	12 877	320 407	•	
Switzerland	37 049	22 193	253 602	•	•
Sweden	49 177	65 947	551 355	•	•
Germany	663 875	573 851	3 338 502	•	•

Table 5.1: An overview of the data sets with which the experiments are performed. A distinction is also made between local and long-distance traffic. These are indicated by • in the corresponding columns. Note: S/RE trains (*S-Bahn* and *Regional Express*) are cross-regional, which stop not only at large stations, but also at smaller regional ones. Accordingly, we classify the dataset as a mixture of both local and long-distance trains (denoted by ○).

5.2 Results

In Figure 5.1 and Table 5.2, it can be seen that on smaller networks like RNV the speed-up technique does not have a great effect, but only beats the other algorithms (at the expense of memory) using more than 32 partitions. The algorithm behaves similarly for Karlsruhe, Paris and the IC/ICE network. Such "dense" networks do not partition well. By "dense" we mean that there are many trips in a short time and a transfer is possible at almost every stop. This makes partitioning difficult.
On the S/RE dataset, Arc-TB with $k = 512$ creates a query time that is about a factor of 4 faster than the original TB (see Figure 5.2 and Table 5.2). With this number of partitions, the pre-computation takes just under 6 hours and the flags have a memory consumption of 229 MB.
On the two countries, Switzerland and Sweden, Arc-TB shows its strength. On Switzerland, it achieves an average runtime of 695 μs (compared to TB: 12 902 μs, CSA: 15 604 μs, and RAPTOR: 23 643 μs). This is a speed-up factor of more than 18,5. For more detail, see Figure 5.3 and Table 5.3. On the Sweden instance, Arc-TB answers queries in 331 μs, compared to TB 4 521 μs, CSA 16 563 μs and RAPTOR 10 660 μs, using an additional storage of around 3 GB. For a detailed overview, see

[7] `https://developers.google.com/transit/gtfs/reference/#general_transit_feed_specification_reference`

k	**prepro. time**	**query time** [μs]	**memory** [MB]	% **set flags**
		RNV		
1	-	115	0	100%
8	00:00:14	140	41	34%
16	00:00:20	126	42	27%
32	00:00:33	102	43	23%
64	00:00:46	82	47	19%
128	00:01:24	71	53	16%
256	00:02:47	66	66	14%
512	00:05:37	66	92	12%
		S/RE		
1	-	1 127	0	100 %
8	00:11:31	1 372	99	35 %
16	00:16:01	1 072	101	27 %
32	00:27:16	784	105	21 %
64	00:43:50	543	113	17 %
128	01:24:42	390	129	14 %
256	02:01:51	347	160	13 %
512	05:50:28	266	229	11 %

Table 5.2: The tables show an overview of the results for the RNV & S/RE dataset depending on k, the number of partitions. The pre-computations were performed with 32 threads on the A machine. $k = 1$ shows the original TB Algorithm.

Figure 5.4 and Table 5.3. The largest dataset, the Germany network, could not be processed using my algorithm, as this would take over 150 days.

k	**prepro. time**	**query time** [μs]	**memory** [MB]	% **set flags**
		Swiss		
1	-	12 902	0	100%
8	07:09:55	6 830	1 331	25%
16	10:07:31	4 471	1 444	18%
32	17:38:33	2 672	1 536	14%
64	23:09:53	1 875	1 638	11%
128	12:19:37	1 306	1 843	9%
256	23:27:44	900	2 150	8%
512	46:07:37	695	3 072	7%
		Sweden		
1	-	4 521	0	100 %
8	01:44:04	2 806	847	24 %
16	02:29:54	2 115	884	17 %
32	04:01:00	1 458	919	13 %
64	06:23:03	1 043	968	10 %
128	12:09:12	731	1 229	8 %
256	19:32:12	555	1 434	7 %
512	10:30:57	385	1 946	6 %
1 024	20:39:26	331	3 072	5 %

Table 5.3: The tables show an overview of the results for the Swiss & Sweden dataset depending on k, the number of partitions. All pre-computations (where k is not **bold**) were performed with 32 threads on the A machine. **Bold** k denoted pre-computation on the B machine with 128 threads. $k = 1$ shows the original TB Algorithm.

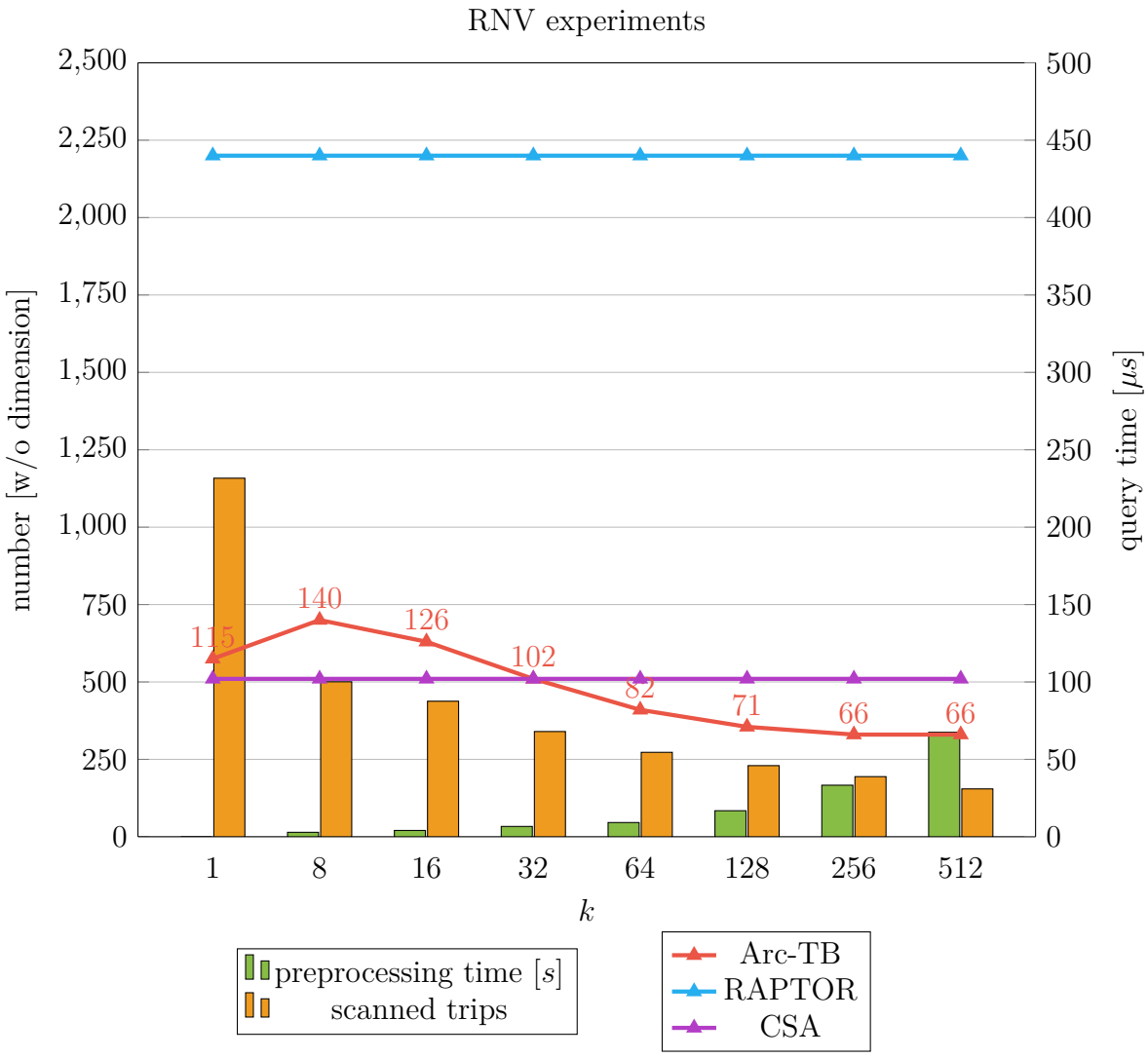


Figure 5.1: Plot of the experiments of the RNV data set. On the abscissa the number of $k \in \mathbb{N}$ is plotted, query times (denoted by ▲) are plotted against the right axis and pre-computation as well as the number of scanned trips per query (of the Arc-TB) against the left.

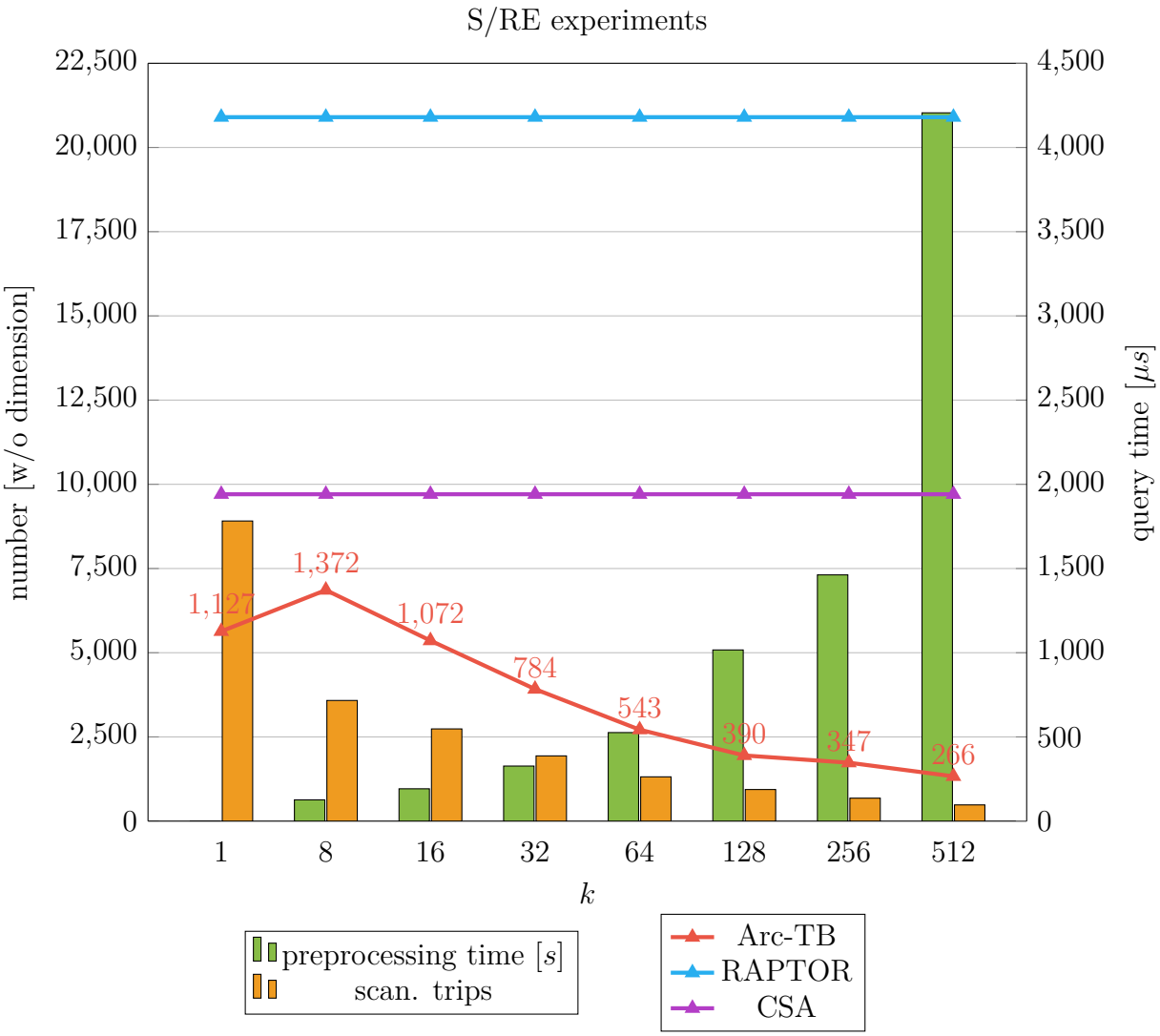


Figure 5.2: Plot of the experiments of the S/RE data set. On the abscissa the number of $k \in \mathbb{N}$ is plotted, query times (denoted by ▲) are plotted against the right axis and pre-computation as well as the number of scanned trips per query (of the Arc-TB) against the left.

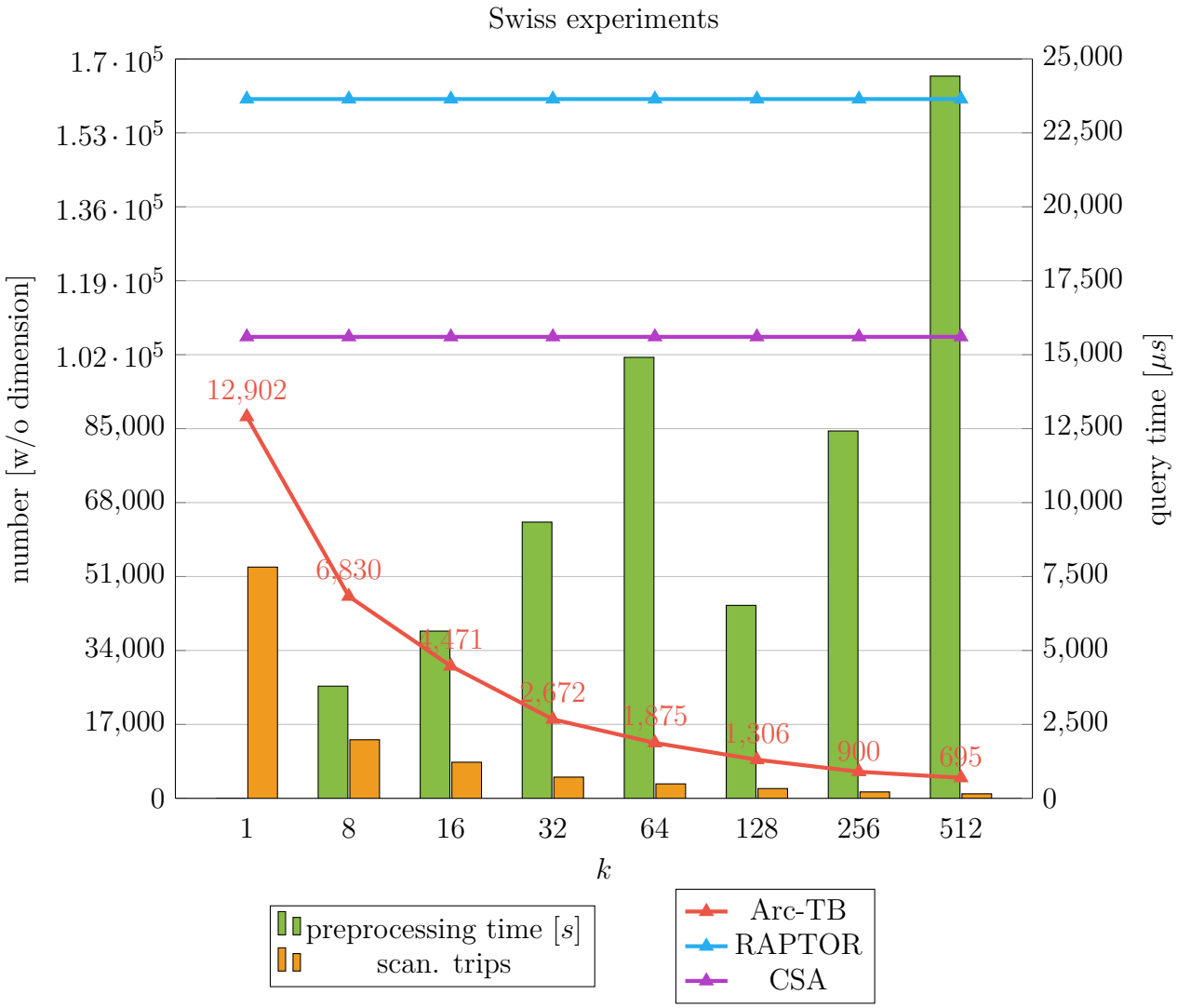


Figure 5.3: Plot of the experiments of the Swiss data set. On the abscissa the number of $k \in \mathbb{N}$ is plotted, query times (denoted by ▲) are plotted against the right axis and pre-computation as well as the number of scanned trips per query (of the Arc-TB) against the left. **Note**: For $k < 128$, the pre-computation has been done on the A machine with 32 threads, for $k \geq 128$, the B machine with 128 threads has been used.

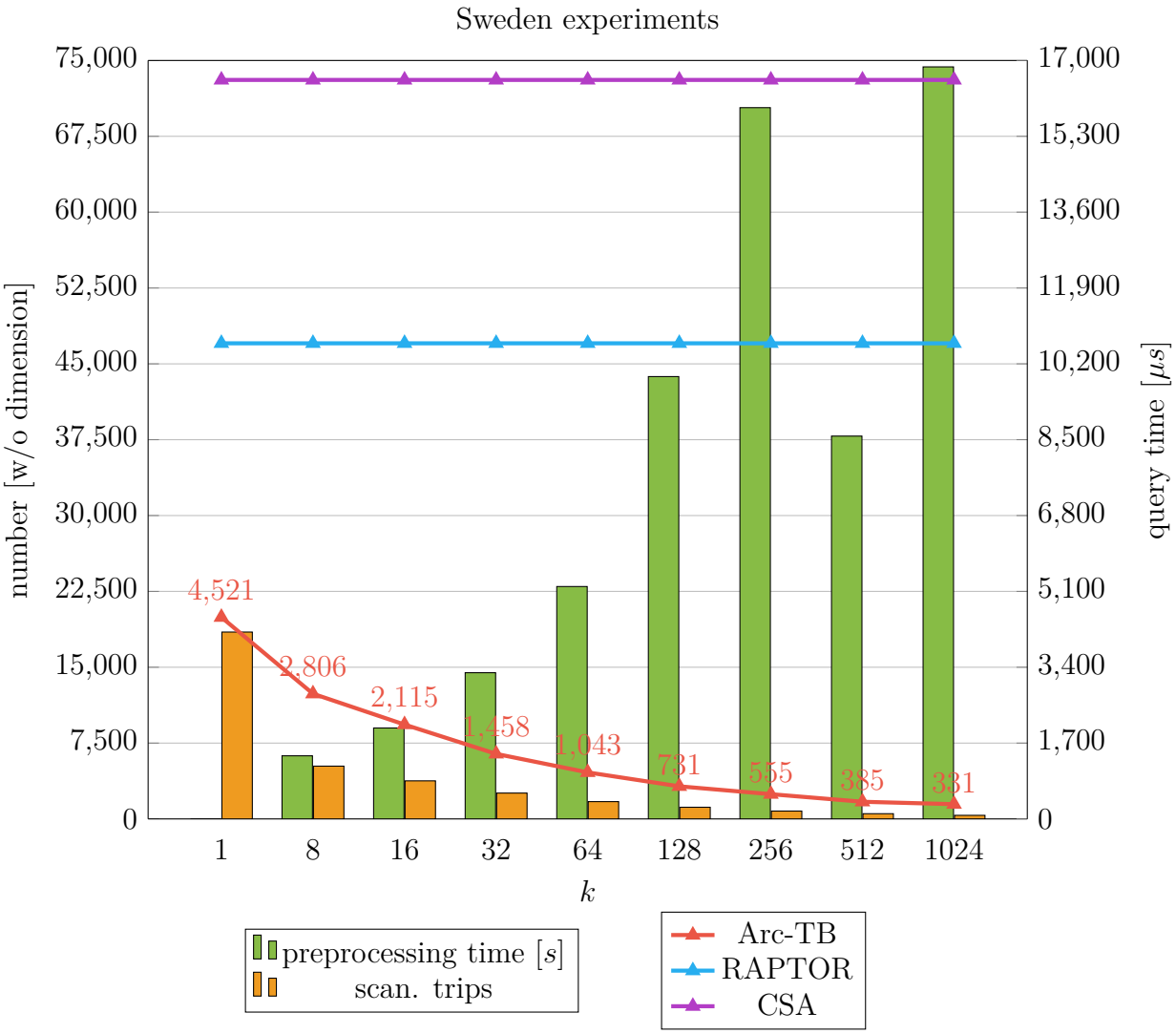


Figure 5.4: Plot of the experiments of the Sweden data set. On the abscissa the number of $k \in \mathbb{N}$ is plotted, query times (denoted by ▲) are plotted against the right axis and pre-computation as well as the number of scanned trips per query (of the Arc-TB) against the left. **Note**: For $k < 512$, the pre-computation has been done on the A machine with 32 threads, for $k \geq 512$, the B machine with 128 threads has been used.

CHAPTER 6

Future Work

It would be desirable to reduce the pre-computation time, as with our implementation, large networks are not yet manageable. There is a more efficient algorithm, which is expected to set better flags and has significantly faster pre-computation times. This will be a project for the near future. Furthermore, it would be interesting to think about delay-resistant Arc-Flags, as models that consider delays are of great importance in reality, as well as using other speed-up techniques, which generate even better runtimes, especially on large national networks.

Abstract (German)

In dieser Arbeit geht es um Routenplanung in öffentlichen Verkehrsnetzen, genauer gesagt um die Verbesserung eines bestehenden Algorithmus in Bezug auf die Abfragezeit. Auf landesweiten Netzen sind andere state-of-the-art Algorithmen eindeutig zu langsam, während unser Algorithmus auf solchen Netzen die größte Geschwindigkeitssteigerung aufweist und daher die anderen Algorithmen um einiges übertrifft. Der neue Arc-TB Algorithmus besteht aus dem Trip-Based Public Transit Routing (TB) Algorithmus und einer angepassten Version der Arc-Flags Beschleunigungstechnik. Es wird beschrieben, wie die beiden zugrundeliegenden Algorithmen funktionieren und wie Arc-Flags auf den TB-Algorithmus angewendet werden kann.

Bibliography

[1] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. *CoRR*, abs/1504.05140, 2015. URL `http://arxiv.org/abs/1504.05140`.

[2] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 19–80. 2016. doi: 10.1007/978-3-319-49487-6_2. URL `https://doi.org/10.1007/978-3-319-49487-6_2`.

[3] RICHARD BELLMAN. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958. ISSN 0033569X, 15524485. URL `http://www.jstor.org/stable/43634538`.

[4] Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller-Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In Jens Clausen and Gabriele Di Stefano, editors, *9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, volume 12 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-11-8. doi: 10.4230/OASIcs.ATMOS.2009.2148. URL `http://drops.dagstuhl.de/opus/volltexte/2009/2148`.

[5] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. In Lasse Kliemann and Peter Sanders, editors, *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 117–158. 2016. doi: 10.1007/978-3-319-49487-6_4. URL `https://doi.org/10.1007/978-3-319-49487-6_4`.

[6] Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More recent advances in (hyper)graph partitioning. *CoRR*, abs/2205.13202, 2022. doi: 10.48550/arXiv.2205.13202. URL `https://doi.org/10.48550/arXiv.2205.13202`.

[7] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. Public transit labeling. In Evripidis Bampis, editor, *Experimental Algorithms*, pages 273–285, Cham, 2015. Springer International Publishing. ISBN 978-3-319-20086-6.

[8] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-based public transit routing. *Transp. Sci.*, 49(3):591–604, 2015. doi: 10.1287/trsc.2014.0534. URL `https://doi.org/10.1287/trsc.2014.0534`.

[9] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection scan algorithm. *ACM J. Exp. Algorithmics*, 23, 2018. doi: 10.1145/3274661. URL `https://doi.org/10.1145/3274661`.

[10] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[11] Andrew V. Goldberg. The hub labeling algorithm. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933 of *Lecture Notes in Computer Science*, page 4. Springer, 2013. doi: 10.1007/978-3-642-38527-8_2. URL `https://doi.org/10.1007/978-3-642-38527-8_2`.

[12] Pierre Hansen. Bicriterion path problems. In Günter Fandel and Tomas Gal, editors, *Multiple Criteria Decision Making Theory and Application*, pages 109–127, Berlin, Heidelberg, 1980. Springer Berlin Heidelberg. ISBN 978-3-642-48782-8.

[13] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 13-14, 2006*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 41–72. DIMACS/AMS, 2006. doi: 10.1090/dimacs/074/03. URL `https://doi.org/10.1090/dimacs/074/03`.

[14] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering multi-level overlay graphs for shortest-path queries. In Rajeev Raman and Matthias F.

Stallmann, editors, *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments, ALENEX 2006, Miami, Florida, USA, January 21, 2006*, pages 156–170. SIAM, 2006. doi: 10.1137/1.9781611972863.15. URL https://doi.org/10.1137/1.9781611972863.15.

[15] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, Berlin, 2008. ISBN 3540779779. URL http://www.amazon.de/Algorithms-Data-Structures-Basic-Toolbox/dp/3540779779%3FSubscriptionId%3D192BW6DQ43CK9FN0ZGG2%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D3540779779.

[16] Henning Meyerhenke, Peter Sanders, and Christian Schulz. Parallel graph partitioning for complex networks. *IEEE Trans. Parallel Distrib. Syst.*, 28(9):2625–2638, 2017. doi: 10.1109/TPDS.2017.2671868. URL https://doi.org/10.1109/TPDS.2017.2671868.

[17] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *ACM J. Exp. Algorithmics*, 11:2.8–es, feb 2007. ISSN 1084-6654. doi: 10.1145/1187436.1216585. URL https://doi.org/10.1145/1187436.1216585.

[18] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable information: Models and algorithms. In Frank Geraets, Leo Kroon, Anita Schoebel, Dorothea Wagner, and Christos D. Zaroliagis, editors, *Algorithmic Methods for Railway Optimization*, pages 67–90, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74247-0.

[19] Matthias Müller-Hannemann and Karsten Weihe. On the cardinality of the Pareto set in bicriteria shortest path problems. *Annals of Operations Research*, 147(1):269–286, October 2006. doi: 10.1007/s10479-006-0072-1. URL https://ideas.repec.org/a/spr/annopr/v147y2006i1p269-28610.1007-s10479-006-0072-1.html.

[20] Peter Sanders and Christian Schulz. Think locally, act globally: Highly balanced graph partitioning. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933, pages 164–175. Springer, 2013.

[21] Ben Strasser and Dorothea Wagner. Connection scan accelerated. In Catherine C. McGeoch and Ulrich Meyer, editors, *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2014, Portland, Oregon, USA, January 5, 2014*, pages 125–137. SIAM, 2014. doi: 10.1137/1.9781611973198.12. URL https://doi.org/10.1137/1.9781611973198.12.

[22] Gerth Stølting Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. *Electronic Notes in Theoretical Computer Science*, 92:3–15, 2004. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2003.12.019. URL `https://www.sciencedirect.com/science/article/pii/S1571066104000040`. Proceedings of ATMOS Workshop 2003.

[23] Sascha Witt. Trip-based public transit routing. In Nikhil Bansal and Irene Finocchi, editors, *Algorithms - ESA 2015 - 23rd Annual European Symposium, Patras, Greece, September 14-16, 2015, Proceedings*, volume 9294 of *Lecture Notes in Computer Science*, pages 1025–1036. Springer, 2015. doi: 10.1007/978-3-662-48350-3_85. URL `https://doi.org/10.1007/978-3-662-48350-3_85`.

Algorithms

Algorithm 5: Pseudocode for bidirectional Dijkstras Algorithm

Input : graph $G = (V, E)$, reverse graph $G_{\text{rev}} = (V, E_{\text{rev}})$, edge weight function $l : E \to \mathbb{R}_0^+$, reverse edge weight function $l_{\text{rev}} : E_{\text{rev}} \to \mathbb{R}_0^+$, source node s, target node t

`/*` $\pi_{s,t}$ `is the shortest path,` $d_f[v]$ `keeps track of the distance from` s `to` v`,` $d_b[v]$ `keeps track of the distance from` t `to` v`,` $s_f[v]$ `marks if node` v `has already been scanned by the forward search,` $s_b[v]$ `analogously for the backward search` `*/`

1 **foreach** $v \in V$ **do**
2 $\quad d_f[v] \leftarrow \infty$, $d_b[v] \leftarrow \infty$
3 $\quad s_f[v] \leftarrow 0$, $s_b[v] \leftarrow 0$
4 **end**
5 $\pi_{s,t} \leftarrow \infty$
6 $d_f[s] \leftarrow 0$, $d_b[t] \leftarrow 0$
7 $Q_f \leftarrow \{(s,0)\}$, $Q_b \leftarrow \{(t,0)\}$
8 **while** $Q_f \neq \emptyset \wedge Q_b \neq \emptyset$ **do**
9 $\quad u \leftarrow Q_f$.deleteMin(), $v \leftarrow Q_b$.deleteMin()
10 $\quad s_f[u] \leftarrow 1$, $s_b[v] \leftarrow 1$
11 $\quad$ **foreach** $e = (u, w) \in E$ **do**
12 $\quad\quad$ **if** $s_f[w] = 0 \wedge d_f[u] + l(e) < d_f[w]$ **then**
13 $\quad\quad\quad d_f[w] \leftarrow d_f[u] + l(e)$
14 $\quad\quad\quad$ **if** $w \in Q_f$ **then** Q_f.decreaseKey$(w, d_f[w])$
15 $\quad\quad\quad$ **else** Q_f.insert$(w, d_f[w])$
16 $\quad\quad$ **end**
17 $\quad\quad$ **if** $s_b[w] = 1$ **then** $\pi_{s,t} \leftarrow \min\left\{\pi_{s,t}, d_f[w] + d_b[w] + l(e)\right\}$
18 $\quad$ **end**
19 $\quad$ **foreach** $e = (v, w) \in E_{\text{rev}}$ **do**
20 $\quad\quad$ **if** $s_b[w] = 0 \wedge d_b[u] + l_{\text{rev}}(e) < d_b[w]$ **then**
21 $\quad\quad\quad d_b[w] \leftarrow d_b[u] + l(e)$
22 $\quad\quad\quad$ **if** $w \in Q_b$ **then** Q_b.decreaseKey$(w, d_b[w])$
23 $\quad\quad\quad$ **else** Q_b.insert$(w, d_b[w])$
24 $\quad\quad$ **end**
25 $\quad\quad$ **if** $s_f[w] = 1$ **then** $\pi_{s,t} \leftarrow \min\left\{\pi_{s,t}, d_f[w] + d_b[w] + l(e)\right\}$
26 $\quad$ **end**
27 $\quad$ **if** $d_f[u] + d_b[w] \geq \pi_{s,t}$ **then** `return`
28 **end**

Algorithm 6: Earliest arrival query-pseudocode is from Witt's paper [23]

Input : Timetable, transfer set T, source stop p_{src}, target stop p_{tgt}, departure time τ
Output: Result set J

$J \leftarrow \emptyset$, $\mathcal{L} \leftarrow \emptyset$
foreach $k = 0, 1, \ldots$ **do** $Q_k \leftarrow \emptyset$
foreach *trip* t **do** $R(t) \leftarrow \infty$
foreach *stop* q *with* $\Delta\tau_{\text{fp}}(q, p_{\text{tgt}}) < \infty$ **do**
    if $q = p_{\text{tgt}}$ **then** $\Delta\tau \leftarrow 0$
    else $\Delta\tau \leftarrow \Delta\tau_{\text{fp}}(q, p_{\text{tgt}})$
    foreach $(L, i) \in \mathbf{L}(q)$ **do** $\mathcal{L} \leftarrow \mathcal{L} \cup \{(L, i, \Delta\tau)\}$
end
foreach *stop* q *with* $\Delta\tau_{\text{fp}}(q, p_{\text{src}}) < \infty$ **do**
    if $q = p_{\text{src}}$ **then** $\Delta\tau \leftarrow 0$
    else $\Delta\tau \leftarrow \Delta\tau_{\text{fp}}(q, p_{\text{src}})$
    foreach $(L, i) \in \mathbf{L}(q)$ **do**
        $t \leftarrow$ earliest reachable trip of line L with $\tau + \Delta\tau \leq \tau_{\text{dep}}(t, i)$
        `ENQUEUE(`t, i, 0`)`
    end
end
$\tau_{\min} \leftarrow \infty$
$n \leftarrow 0$
while $Q_n \neq \emptyset$ **do**
    foreach *trip segment* $p_t^b \rightarrow p_t^e \in Q_n$ **do**
        foreach $(L_t, i, \Delta\tau) \in \mathcal{L}$ *with* $b < i \wedge \tau_{\text{arr}}(t, i) + \Delta\tau < \tau_{\min}$ **do**
            $\tau_{\min} \leftarrow \tau_{\text{arr}}(t, i) + \Delta\tau$
            update result set and $J \leftarrow J \cup \{(\tau_{\min}, n)\}$
        end
        if $\tau_{\text{arr}}(t, b+1) < \tau_{\min}$ **then**
            foreach *transfer* $p_t^i \rightarrow p_u^j \in T$ *with* $b < i \leq e$ **do** `ENQUEUE(`u,j,$n+1$`)`
        end
    end
    $n \leftarrow n + 1$
end
procedure `ENQUEUE(`*trip* t, *index* i, *number of transfers* k`):`
    if $i < R(t)$ **then**
        $Q_n \leftarrow Q_n \cup \left\{ p_t^i \leftarrow p_t^{R(t)} \right\}$
        foreach *trip* u *with* $t \preceq u \wedge L_t = L_u$ **do** $R(u) \leftarrow \min\{R(u), i\}$
    end