Karlsruhe Institute of Technology

# On the Complexity of Public Transit Profile Queries

Bachelor Thesis of


Florian Grötschla

At the Department of Informatics
Institute of Theoretical Informatics


| | |
|---|---|
| Reviewers: | Prof. Dr. Dorothea Wagner |
| | Prof. Dr. Peter Sanders |
| Advisors: | Tobias Zündorf, M. Sc. |
| | Moritz Baum, M. Sc. |

Time Period: 1st July 2017 – 31st October 2017

**Statement of Authorship**

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 31st October 2017

**Abstract**

More and more commuters and travellers are dependant on public transport every day and finding journeys that fit their time schedules becomes increasingly important. While the problem of computing a set of best journeys in a given time-interval is well researched and meets those needs, only few bounds on the theoretical complexity are known. The incorporation of additional time-independent-networks, enabling a traveller to take a car or walk for segments of his journey, especially turned out to be more complex and impractical in comparison to single-mode ones, even with modern acceleration techniques. This thesis focuses on the theoretical complexity of such queries, abstracting from the implementation in practice, thus exploring its foundation. Different approaches, such as using combinations of established algorithms or trying recursive computation attempts are analysed and tested for viability, giving or discarding possible starting points for future works. These include the family of Connection Scan Algorithms and Dijkstra-based approaches on different graph-models. Aside from finding asymptotic runtime boundaries of these well-known algorithms, this thesis takes a look at partitions of networks and how they might be used for efficient algorithms.

**Deutsche Zusammenfassung**

Immer mehr Berufspendler und Reisende sind tagtäglich auf öffentliche Verkehrsnetze angewiesen und es wird immer wichtiger Verbindungen zu finden, die in ihre Zeitpläne passen. Obwohl das Finden der besten Verbindungen in einem bestimmten Zeitintervall gut erforscht ist, gibt es wenige theoretische Aussagen über die Komplexität des Problems. Insbesondere die Betrachtung von zusätzlichen zeitunabhängigen Netzwerken, die beispielsweise die Nutzung eines Autos oder das Laufen auf Zwischenabschnitten der Reise ermöglichen, haben sich als sehr komplex erwiesen und sind selbst unter Zuhilfenahme von modernen Beschleunigungstechniken häufig impraktikabel. Diese Arbeit beschäftigt sich mir der theoretischen Komplexität solcher Anfragen und abstrahiert damit von der praktischen Implementierung. Unterschiedliche Ansätze, wie die Nutzung einer Kombination aus etablierten Algorithmen oder rekursive Berechnungsarten werden analysiert und auf Realisierbarkeit geprüft, wobei Ansatzpunkte für zukünftige Arbeiten geschaffen oder verworfen werden. Insbesondere werden die Familie von Connection Scan Algorithmen und Dijkstra-basierte Ansätze auf unterschiedlichen Graph-Modellen untersucht. Neben dem Finden von asymptotischen Laufzeitgrenzen von diesen Algorithmen legt diese Arbeit einen zusätzlichen Fokus auf partitionierte Netwerke und wie sie eventuell für effiziente Algorithmen verwendet werden könnten.

# Contents

# 1. Introduction

This thesis takes a look at the complexity of some questions concerning profile queries for public transit networks. The profiles contain information about all optimal journeys leading from a source stop to a target stop and have broad applications for journey planning. One catchy example is the online service of the German railroad company Deutsche Bahn. A user can input the time he wants to leave at a specified stop and where he wants to travel to. The service then outputs the best train connections for the next hours. Computing such profiles has been done in these use cases for longer now and a set of algorithms was developed to solve this problem efficiently. First approaches tried to use algorithms made for route planning in graphs with weighted edges, like the famous algorithm of Dijkstra. However, it soon turned out that they can not unfold their full potential on the discrete structures given by connections that leave and arrive at fixed times. New algorithms that do not work on graphs but on the underlying timetable information were developed for this. One of them is the Route Based Public Transit Router (RAPTOR) [DPW12], which uses the structures of routes and trips taken by trains. It solves the problem more effectively and showed its applicability on instances like the public transit network of London. Other algorithms followed and maybe most importantly, the Connection Scan Algorithm (CSA) [DPSW17] was invented. It only has to scan over an array of connections, sorted by departure time. This causes it to be very cache-efficient and well suited for practical purposes. There have been little major breakthroughs in terms of completely new algorithms since then and research concentrates on accelerating CSA [BCE$^+$10] [BHS16] [BGMH10] and other Dijkstra-based approaches [BDGMH09]. Furthermore, other variations of this problem are discussed now. While CSA can handle transitively closed sets of footpaths, arbitrary ones are problematic [WZ17]. Combining time-dependent transit networks and time-independent ones like footpaths or streets (used by cars) seems very difficult. It can be classified as a multi-modal query, which is in focus of current research [Paj09][DDP$^+$13].

## 1.1 Related Work

The model for public transit networks used in this thesis can not sufficiently represent every aspect of real transit networks. The most important things missing here are minimum change times for stops and the subdivision of connection sets into trips and routes [BDG$^+$15]. Trips are sequences of consecutive connections which are served by a single train. This means that a traveller can remain seated at the intermediate stops and does not have to change the train. It only has an effect on the validity of journeys when using minimal

change times for stops. These are times that are needed if one changes a train at a stop and have to pass between the arrival and departure of the connections. The time is not needed when staying with the same trip. Routes partition the trips into sets containing trips that serve the same stops in the same order.
The model was trimmed for a reason in this thesis, as the work is focused on the complexity in asymptotic terms and not the exact runtime. Trips and minimum change times have little effect on this. The algorithms used in the following chapters can be expanded to incorporate these aspects, without loss of asymptotic efficiency. However, considering them would make the explanations of algorithms more complex, shifting the focus to less important elements. So in an attempt to streamline things, they are left out.
Other related work includes the Round Based Public Transit Router (RAPTOR) [DPW12], which is not included for various reasons. It heavily depends on the fact that in practice shortest journeys do not contain many transfers between vehicles. Of course, in theory that has not to be the case and in an extreme example every pair of stops can involve two routes (one in each direction) and every connection its own trip. Then the number of transfers for a shortest journey is the number of connections taken minus one. Another aspect that makes it hard to compare RAPTOR (or the version for profile queries: rRAPTOR) to other algorithms for the profile calculation in public transit network is the inherent multiple criteria approach of it. It calculates a Pareto-set of shortest journeys optimizing the number of transfers and the arrival time. There are other publications which deal with those multi-criteria queries [DMHS08] [Wit15].
The time-model used in this thesis models connections with one fixed departure and arrival time. However, in reality a route is taken every few minutes by a train which can be represented by a repetition rate. There is work that uses this compressed information [BS14].
The holy grail of route-planning are multi-modal queries. These enable a traveller to take different modes of transportation for one journey. They can include means of transportation like trains, streets, itineraries, ships, taxis or even planes. Some of them use time-dependent vehicles, while others are time-independent [Paj09] [DDP$^+$13].

## 1.2 Contribution

The first part of the thesis takes a look at the problem of public transit routing and analyses established algorithms and possible boundaries. Those include variations of Dijkstra and versions of CSA tailored for different use cases. One acceleration technique for CSA is considered too, namely Connection Scan Accelerated ([SW14]). Then the problem of arbitrary footpaths is addressed. All algorithms are treated on a theoretical level, working out asymptotic runtimes. The worst case scenarios used for this expose difficulties the algorithms struggle with.
The second part tries to reveal the problems for finding better algorithms. After showing some boundaries for one-to-one queries, a simple recursive algorithm that uses some basic operations and solves one-to-one queries without solving the equivalent all-to-one query simultaneously, is presented. Extending on this idea an algorithm using graph-separators is presented. It is only applicable for very limited instances, but the results can still be used to identify problems and improving upon those.

# 2. Preliminaries

For understanding the rest of the thesis, basic notions are introduced first. These are structured by first giving definitions for public transit networks, profiles and the different kinds of queries. Afterwards some graph models will be shown and explained. After giving some operations on profiles, the algorithms used to solve the problems are presented. Some parts of them will be elaborated in following chapters where they are needed.

## 2.1 Definitions

***Public Transit Network.*** A public transit network is defined as a 3-tuple $(P, C, F)$ containing a stop set $P$, a set of connections $C$ and footpaths $F \subseteq \{\{p_1, p_2\} \mid p_1, p_2 \in P\}$. Further, the sizes of those sets are denoted as $|P| = n$, $|C| = k$ and $|F| = m$. Additionally, there are the following functions to model times and stops for connections and footpaths:

| **Function** | **Meaning** |
|---|---|
| $\pi_{dep} : C \to P$ | Departure stop of a connection |
| $\pi_{arr} : C \to P$ | Arrival stop of a connection |
| $\tau_{dep} : C \to \mathbb{R}$ | Departure time of a connection |
| $\tau_{arr} : C \to \mathbb{R}$ | Arrival time of a connection |
| $\tau_{dur} : F \to \mathbb{R}_+$ | Walking duration of a footpath |

In the context of public transit networks only such connections are reasonable that arrive after they depart, or a train would travel back in time and enable an equivalent of negative cost cycles in weighted graphs. Although times are modelled by real numbers, often times timestamps like 14:25 will be used to give the intuition of a point in time. It should be clear that one can be mapped to the other. The reason real numbers are used is the fact that it would be tedious to define a set with properties of an algebraic field with appropriate comparators.
Footpaths can be transitively closed while fulfilling the triangle-inequality so that they divide the stop set $P$ into equivalence classes. This ensures that all minimal paths only containing footpaths are of length one as multiple successive footpaths can be replaced by a single footpath that is guaranteed to exist and have smaller or equal length. It simplifies some algorithms because the search for shortest paths through footpaths can be limited to

a search radius of one. The set of connections leading from a stop $p_1$ to another stop $p_2$ have to fulfil the FIFO-property, i.e. connections that depart earlier, arrive earlier, too. If this is not the case, then a connection $c$ can be dominated in the sense that there is a connection $c^\star$ departing later and arriving earlier. Then $c$ is not needed because a traveller can always use $c^\star$ instead. Therefore $c$ can be left out of the problem formulation.
The thesis will use the convention of drawing footpaths as dotted lines and connections as arrows leading from the departure to the arrival stop.
***Journeys.*** A journey of length $l$ is a sequence of connections and footpaths $(d_0, \dots, d_{l-1})$ with $d_i \in C \cup F$. A journey is only valid if a traveller can get every connection in time, i.e. if for every subsequence $(c_0, f_0, \dots, f_j, c_1)$ with two connections and an arbitrary amount of footpaths fulfils

$$\tau_{arr}(c_0) + \sum_{a=0}^{j} \tau_{dur}(f_a) \leq \tau_{dep}(c_1)$$

***Pareto Optimality.*** A set of tuples $S \subset A \times B$ is called Pareto optimal for two binary equivalence relations $\leq_1 \subset A^2$ and $\leq_2 \subset B^2$ if no element is dominated in regards to both of its elements, i.e. there are no $(a_1, b_1), (a_2, b_2) \in S$, so that $a_1 \leq_1 a_2 \wedge b_1 \leq_2 b_2$.
***Profiles.*** An $u$-$v$-profile $profile_u^v$ is a Pareto optimal ordered (for departure times) sequence of departure-time/arrival-time tuples $((t^1_{dep}, t^1_{arr}), (t^2_{dep}, t^2_{arr}), \dots)$ describing the arrival time of a shortest journey from $u$ to $v$ for any departure time and an additional walking duration $w \in \mathbb{R}_+ \cup \infty$ for the time it takes to walk from $u$ to $v$. The Pareto relations are $\leq_1 := \leq_{\mathbb{R}}$ and $\leq_2 := \geq_{\mathbb{R}}$. Profiles can be represented as a set of tuples as public transit networks only allow for connections to leave at discrete points in time. Every journey that contains at least one connection has a fixed latest departure time at $u$ where it is possible to reach the first connection of the journey. Only journeys that contain no connection can not be discretised this way. Those journeys are a sequence of footpaths and there is only one shortest time for that. It is independent of the departure time and can be stored in the walking duration. An empty profile can be represented as the empty set, which can cause the need for special treatment. Instead, guard entries $(\infty, \infty)$ can be inserted at the end. Profiles can also be understood as a partial function $f_u^v : \mathbb{R} \to \mathbb{R}$ mapping departure time to arrival time, although this is more abstract and makes defining minimal profiles difficult. The tuples will sometimes be referred to as profile entries.
***Queries.*** A time-query asks for the earliest possible arrival of any journey to a target stop $t$ after a specified time at a start stop $s$. A profile query not only asks for the arrival of an earliest journey, but for the whole profile $profile_s^t$ containing all optimal journeys from the source stop $s$ to the target stop $t$.

## 2.2 Depiction of Profiles

As mentioned before, profiles can be represented as a partial function mapping departure time to arrival time. Depicting them as such is more intuitive than giving a list of tuples. The following section makes clear how those drawings have to be understood. The profile shown in figure 2.1 has one entry and the according stop is connected to the target by a series of footpaths. Shifting the identity function by the walking duration gives a linear function describing the arrival time if only footpaths are taken. A constant part is induced by the tuple, which itself is just a point in the graph. The resulting function is drawn in blue and takes the minimum of both. Assuming that the x- and y-axis intersect in the origin $(0, 0)$ of the coordinate-system this offset is also the intersection of the shifted identity-function with the y-axis. This convention can be used for the coordinate systems but restricts all lines to run above the identity function as points under it would represent a journey that departs after it arrives. That is why the coordinate systems will not be scaled this way in the following chapters. The depictions are only meant to visualize profiles and not to give quantitative values.

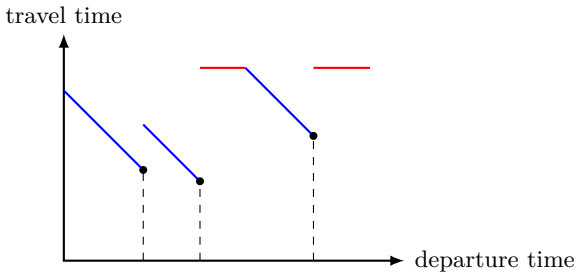



Figure 2.2: Profile depicted as a function mapping departure to travel time

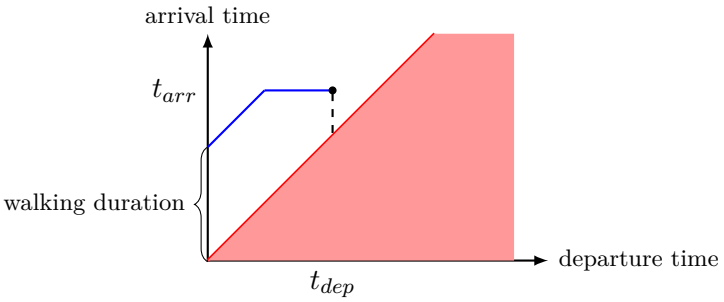



Figure 2.1: Profile with one entry and a walking duration not $\infty$. The red part can not be used as journeys arrive before they depart in this area.

Another alternative way of depicting profiles is by mapping the departure time to the travel time. This leads to linear parts of gradient $-1$ for leaving connections and to constant parts if the optimal journey is a sequence of footpaths. The two different functions can be converted into each other: Let $f_1 : \mathbb{R} \mapsto \mathbb{R}$ be a function mapping departure time to arrival time and $f_2 : \mathbb{R} \mapsto \mathbb{R}$ a function that maps the departure time to the travel time. Then the following equation holds:

$$f_1(x) = f_2(x) + x, \qquad \forall x \in \mathbb{R}$$

This is true because the arrival time is the departure time of the journey, plus the time the journey takes. Figure 2.2 shows such a function. Blue parts are times where a connection is taken at some point in the journey. For the red parts the shortest journey only takes footpaths to the destination.

## 2.3 Graph Models

Public transit networks were introduced as a set of stops, connections and footpaths. However, there are various ways to model them as a graph. Two ways for doing so are the time-dependent and time-expanded model [PSWZ08]. The time-dependent one uses the notion of profiles introduced before and models edges by profiles between two stops. For the time-expanded one a new graph is constructed and used for the query. The models will be explained in the following sections.

### 2.3.1 Time-Dependent Model

The goal of this approach is to transform the network into a simple graph with only a maximum of one simple directed edge running between stops. In contrast to the time-expanded approach the topology of the network is preserved and functions describing the travel-time are introduced as edges. These functions are profiles. An edge with a profile only has to be added if there is any footpath or connection between the stops. Figure 2.3

shows a network with connections as arrows and dotted lines for footpaths. The resulting graph with time-dependent edges is shown on the right. The colours of profile-segments indicate the originating connection or footpath.

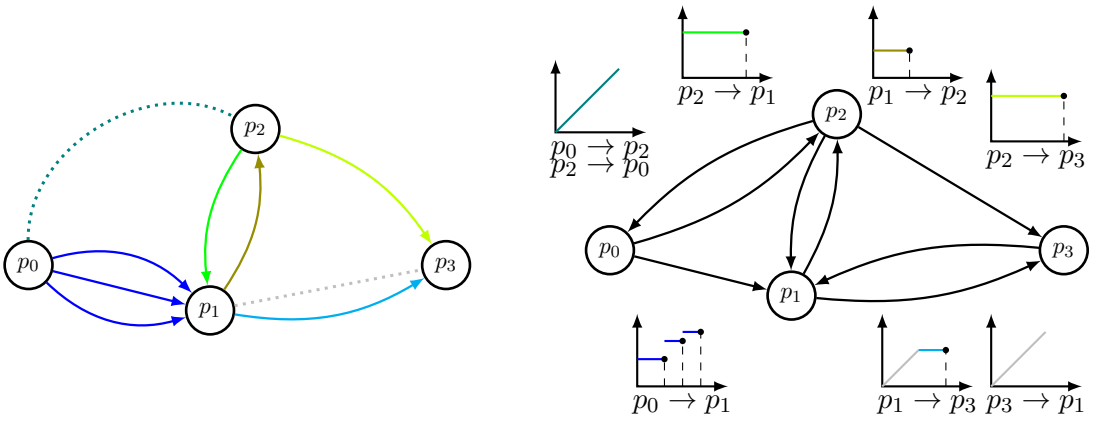



Figure 2.3: Network (left) modelled as a time-dependent graph (right). Yellow nodes symbolize departure events and blue ones arrival events

### 2.3.2 Time-Expanded Model

Instead of modelling the network as a graph with time-dependent edges it is also possible to translate the problem instance to a graph with constant weights. It enables the usage of the standard Dijkstra algorithm and all its speed-up techniques. On the downside, the resulting graph is bigger and footpaths can not be incorporated that easily, especially when they are not transitively closed.
Figure 2.4 shows how the transition is done in an informal way. For every stop the departure- and arrival-events of connections are unrolled in time. What that means is that one node is created for every every pair $(s, t)$ of arriving or departing connection $c$ at $s$ and associated time of the event $t$. In the graphic the vertically aligned nodes are the event-nodes for the respective stop denoted above them. Nodes created by an arrival event are coloured blue and such created by a departure of a connection in yellow. Time passes from top to bottom and the dotted horizontal lines symbolize the points in time written next to them. The nodes of each stop are connected downwards. This enables waiting at a stop for the departure of a later connection. Furthermore, the sets of nodes are connected by one edge for every connection. They run from their according departure event to the arrival event. Intuitively, they make it possible to travel from one stop (represented by a set of nodes) to another. The weight of an edge is the difference between the time at the arrival and the time of the departure. This guarantees that a shortest path in this graph represents a shortest journey in the transit network. As directed edges are only added from higher to lower nodes (or from earlier to later events) in the figure, the graph is acyclic.
The figure should give a vague intuition what is done and why it works, however there is potential for confusion and the construction will be explained more formally here. The new directed graph $G = (V, E)$ with the weight function $w : E \to \mathbb{R}$ is defined as follows:

$$V = \{(s,t) \mid s \in P, c \in C : (\pi_{dep}(c) = s \wedge t = \tau_{dep}(c)) \vee (\pi_{arr}(c) = s \wedge t = \tau_{arr}(c))\}$$

$$\begin{aligned} E = \{((p,t_1),(p,t_2)) \in V^2 \mid t_2 > t_1 \wedge \nexists (p,t_i) \in V : t_1 < t_i < t_2\} \\ \cup \{((p_1,t_1),(p_2,t_2)) \in V^2 \mid \exists c \in C : \pi_{dep}(c) = p_1 \wedge \pi_{arr}(c) = p_2 \\ \wedge \tau_{dep}(c) = t_1 \wedge \tau_{arr}(c) = t_2\} \end{aligned}$$

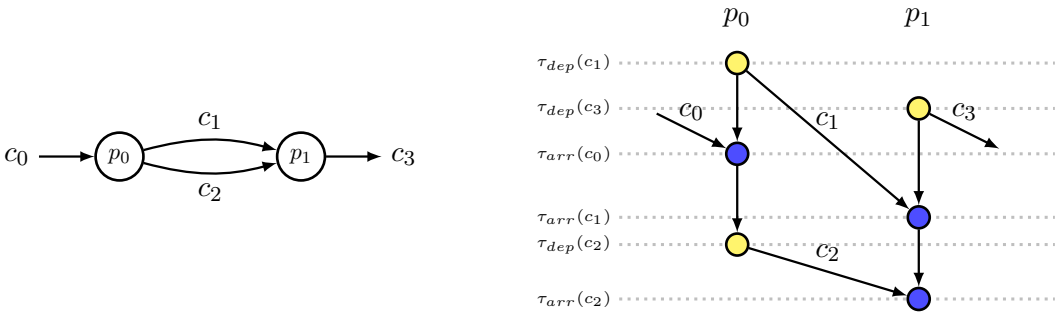



Figure 2.4: Network (left) modelled as a time-expanded graph (right)

$$w : ((p_1, t_1), (p_2, t_2)) \mapsto t_2 - t_1$$

As the construction uses the limited set of discrete points in time given by the arrival or departure of a connection it is not possible to directly incorporate footpaths without changing the model. They can be taken at any time, so it is more difficult to map them to nodes that represent a discrete event.

## 2.4 Operations On Profiles

Profiles will be handled by the algorithms presented later and therefore need some elementary operations which will arise in different context throughout the thesis. To get some basic understanding of those notions they will be presented here. In particular those are:

- Shifting: Adding a constant offset to the profile function
- Merging: Taking the union of two profiles
- Linking: Connecting two profile functions

The implementation for all of the operations will be given together with the asymptotic runtime in chapter 3.

### 2.4.1 Shifting

This operation's goal is to add an offset to the function, resulting in a new profile. The formal definition for a shift by $x \in \mathbb{R}$ is as follows:

$$profile^t_{s+x} : \mathbb{R} \to \mathbb{R}, \qquad a \mapsto profile^t_s(a - x),$$

where $profile^t_{s+x}$ is the shifted profile. Shifting can be used when walking times between stops are incorporated and lead to all profile entries of the next stop to be shifted in negative x-direction by the walking duration as one has to leave earlier to get the corresponding connections. Figure 2.5 demonstrates this.

### 2.4.2 Merging

Sometimes it is easier to break up the computation of a profile and compute two or more profiles which all describe times of possible journeys. The minimum over all those functions is the desired output of this operation. More formally:

$$(\underline{profile^t_s} + \overline{profile^t_s})(x) := min\{\underline{profile^t_s}(x), \overline{profile^t_s}(x)\}, \forall x \in \mathbb{R}$$

Merging is denoted by a simple $\cup$ and because it is associative and commutative it can be uniquely extended to any number of profiles.

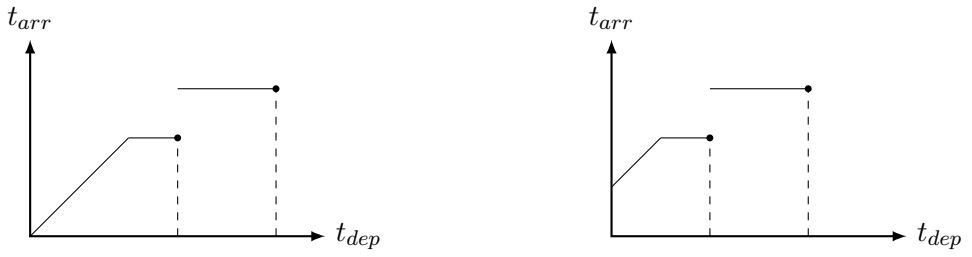



Figure 2.5: Profile before shift (left). Profile after shift (right).

### 2.4.3 Linking

For two profiles $profile_u^v$ and $profile_v^t$ the linking operation $\oplus$ is defined as the concatenation of the profiles:

$$profile_u^v \oplus profile_v^t := profile_v^t \circ profile_u^v$$

and results in a new profile. If the stops are on a path with stops $\{p_0, p_1, \ldots, p_{n-1}\}$ the following holds:

$$\forall p_i, p_j, p_k \in \{p_0, p_1, \ldots, p_{n-1}\}, i < j < k : profile_{p_i}^{p_k} = profile_{p_i}^{p_j} \oplus profile_{p_j}^{p_k}$$

because of simple concatenation of the operation.
This operation will be used to calculate new profiles from already existing ones. The notation is the other way around when comparing it to the concatenation of functions, making it more intuitive to work with in algorithms. Figure 2.6 shows the execution of the operation on two profiles.

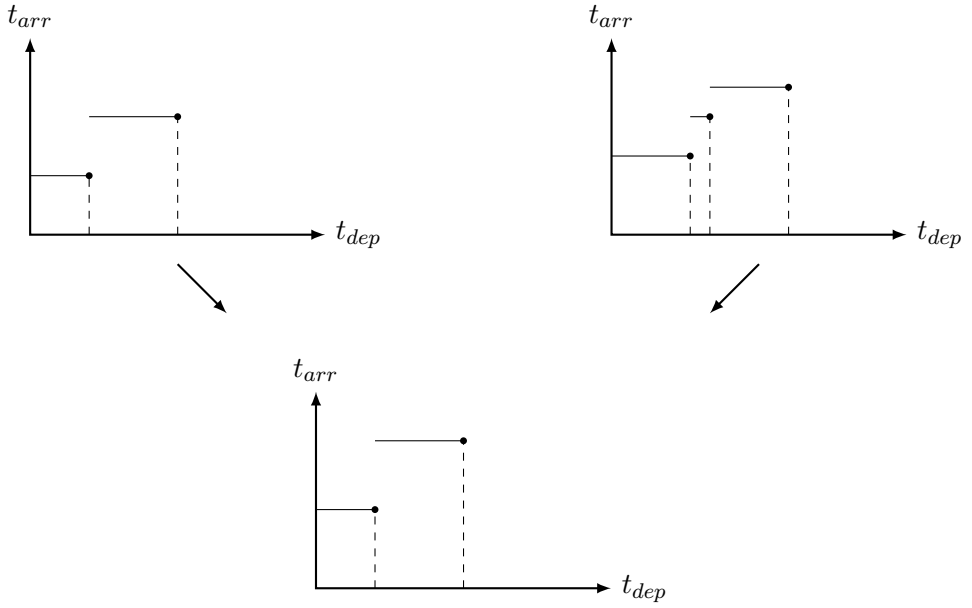



Figure 2.6: Two profiles which are linked, resulting in a third one

## 2.5 Dijkstra's Algorithm

Dijkstra's algorithm [Dij59] solves the problem of finding shortest paths in a graph $G = (V, E)$ with a weight function $w : E \mapsto \mathbb{R}_+^0$. It starts at a node $s$ and computes distances to all reachable nodes. Keeping track of tentative distances is implemented with a priority

queue $Q$, where nodes are inserted with their respective distance as the key. In the beginning, $Q$ only contains $s$ and distances are set to 0 for $s$ and $\infty$ for all the others. The main loop of the algorithm takes the minimal node out of the queue and settles it. Settling is done by relaxing all outgoing edges of the node. The relax-operation computes the sum of the distance to the node and the weight of the edge and compares it to the tentative value at the target node. If the computed distance is shorter, the target node will be inserted into $Q$ or the key will be decreased (if it is already contained in $Q$). The main invariant of the algorithm is that only such nodes are settled, where the tentative distance is already optimal. If a target node $t$ is specified the algorithm can therefore stop when taking it out of the queue, knowing that the distance to it can not improve. The runtime depends on the priority-queue and can be expressed as

$$|V| \cdot (T_{insert} + T_{deleteMin}) + |E| \cdot T_{decreaseKey},$$

where $T_{insert}$ and $T_{deleteMin}$ are the times needed to insert an element into the queue and extract the minimum and $T_{decreaseKey}$ the time for decreasing the key of an already contained element. The insertions and deletions are done once for every node, whereas every edge can cause a decrease-operation. For a Fibonacci-heap all queue-operations are constant except for the deletion of the minimal element, which is $\mathcal{O}(log(n))$.

## 2.6 Connection Scan Algorithm

The Connection Scan Algorithm [DPSW13] does not work directly on a graph structure, but scans the set of connections in order and keeps labels for stops. The time-query computes the earliest arrival time from one stop $s$ to all other stops in the network while profile-queries compute profiles from all stops to one target stop $t$. Footpaths can be incorporated but have to be transitively closed for the algorithm to stay correct. In the following, both kinds of queries will be introduced while not going too much in depth. In chapter 3 the runtime of the different variants will be analysed and pseudocode with more elaboration will be given for them.

### 2.6.1 Time-Query

As the name suggests the Connection Scan Algorithm takes the sorted array of connections and scans them in increasing departure time. For every stop $p$ it holds a label $l(p)$ containing the earliest arrival time of all journeys from $s$. It is initialized to $\infty$ for all stops that are not $s$ as with the lack of any connection taken into consideration none of them is reachable. The label of $s$ is set to the starting-time passed in the query. When scanning a connection $c$ the algorithm checks whether it is reachable, i.e. if $l(\pi_{dep}(c)) \leq \tau_{dep}(c)$. If this is the case and $\tau_{arr}(c) \leq l(\pi_{arr}(c))$ the label at $\pi_{arr}(c)$ is set to $\tau_{arr}(c)$. This ensures that at any point of the execution the labels are correct regarding journeys only containing connections which were already scanned. Footpaths are handled by relaxing ones at the destination stop of a connection that is scanned.

### 2.6.2 Profile-Query

In contrast to the time-query, profile-queries are not answered one-to-all, but all-to-one by CSA. For a given target stop $t$ the query answers with complete profiles from all other stops to it. First, consider networks without footpaths, the algorithm will be expanded upon those later. It is no longer sufficient to store only the earliest arrival in a label, instead a tentative profile will be used. These tentative profiles are complete in the sense that they contain all tuples necessary to find the earliest journey for any time later than the the departure time of the last scanned connection. The correctness of the algorithm is based on this invariant. The

profiles are initialized as the identity profile for $t$ (this is a profile which is not represented by a list of tuples, but it can be handled as a special case) and the empty profile for all other stops. The algorithm no longer scans the connection array in ascending departure time, but in reverse. For any connection $c$ that is scanned, the algorithm evaluates $profile^t_{\pi_{arr}(c)}$ (the label at $\pi_{arr}(c)$) for the time $\tau_{arr}(c)$. This returns the earliest possible arrival at $t$ for a time later than $\tau_{arr}(c)$. The evaluation of the incomplete profiles is correct as only entries with later departure times are needed for the evaluation and these are already contained. If this time improves the profile at $\pi_{dep}(c)$ it can be inserted into it. Note that one has to specifically check if the profile is still made up of a Pareto optimal set or no insertion is made.

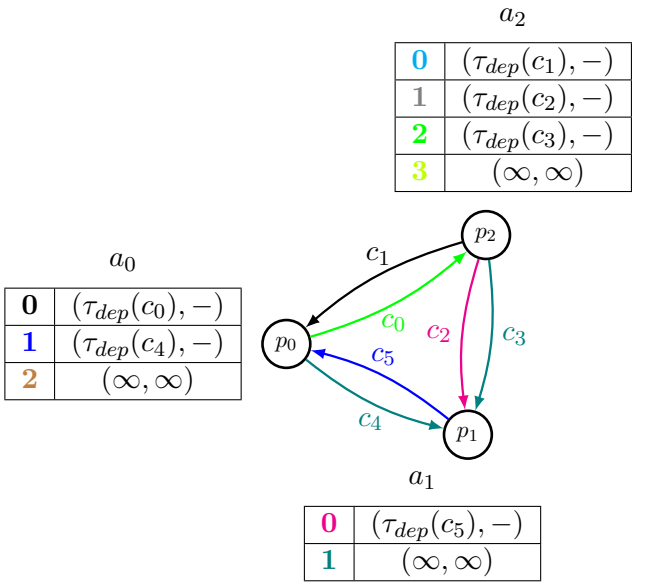



Figure 2.7: Network with profiles drawn next to their stops. Colors symbolize the entries that have to be evaluated for the scan of a connection.

In contrast to the time-query, the evaluation of a profile for every scan is no longer constant as the best entry of the profile at the destination stop has to be found. The fastest way to do so is by using a binary search. This problem is fixed by an adapted version, namely pCSA-C (where the C stands for constant evaluation of profiles). Of course, it is not possible to evaluate any profile in constant time, but one can use special properties of the scan to make it constant when scanning connections in descending departure time. In this case, the evaluation queries on the profiles are always in the same order and use the same times (the arrival times of the connections). They are the same for all target stops. Additionally, the algorithm is modified so that labels do not hold profiles, but sets of tuples that are not necessarily Pareto optimal. Every connection that is scanned then leads to the insertion of a tuple, even if it does not improve it. To stay correct, the newly inserted tuple is inserted with the arrival time of the minimum of the result of the profile evaluation and all arrival times the profile already contains. Every label can hold a fixed array for the tuples that has the size of the number of outgoing connections as these lead to insertions. At the beginning, arrival times are not given yet and the entries are templates that the algorithm will fill in. What this adaption enables is the preprocessing of pointers for every connection that indicate the place for the next insertion in the array of a connection's departure stop and pointers that indicate where the profile at the arrival stop has to be evaluated. Now every query only needs constant time to evaluate profiles. Figure 2.7 depicts a network with three stops and the arrays that were computed. The departure times of the entries are those of the connections leaving at that stop (in order). The colours

symbolize the entry of the array at the arrival stop that has to be evaluated to scan the connection. A small disadvantage is that for getting profiles instead of tuple-sets one last step of minimizing the sets has to be added.
Handling footpaths is possible, if those are transitively closed. The standard version of CSA without constant evaluation can be adapted to simply relax all footpaths at $\pi_{dep}(c)$ when scanning $c$. For pCSA-C things get more complicated as this version uses the discrete points in time given by the departure and arrival times of connections to create a template-entry for them. For every footpath $\{u, v\} = f \in F$, the difference $\tau_{dep}(c) - \tau_{dur}(f)$ is a possible entry for the label at $u$ if $\pi_{dep}(c) = v$. To fix this, footpaths are replaced by a set of pseudo-connections. These are computed in the following way: For a footpath between $p_1$ and $p_2$ all pairs of incoming and outgoing connections at different stops are identified. For every pair $(\tau_{arr}(p_1), \tau_{dep}(p_2))$ or $(\tau_{arr}(p_2), \tau_{dep}(p_1))$ a pseudo-connection is added if there is not already one with a later or equal departure time and an earlier or equal arrival time between the stops of the footpath, meaning that it is not dominated and therefore needed. Unfavourably this increases the number of connections. Footpaths at $s$ and $t$ have to be treated separately.

### 2.6.3 Connection Scan Accelerated

The basic Connection Scan Algorithm can be accelerated [SW14] by using similar principles to those of Multi-Level-Dijkstra [DGPW17]. The network is recursively partitioned into cells (the components of the partition) such that all connected components induced by footpaths are only part of one cell. It starts with a component for the whole network $z_{root}$ and subdivides it successively. In the preprocessing-phase the Transit Connection Set $T(z)$ of a cell $z$ is computed. It is a subset of all connections running inside of a cell $z$, i.e. they depart in $z$. Additionally, every optimal journey through a cell can be replaced by one that only contains transit connections. It should be noted that multiple correct sets can exist, although one is interested in a preferably small one. The Long Distance Connection Set $D(z)$ is the set of all interior connections iff $z$ is a bottom layer cell (it contains no subcell) or the union of the transit connection sets of all direct subcells. Moreover, let $z_i$ be the bottom cell containing the stop $i$.
The query for computing the earliest arrival journey from $s$ to $t$ uses the following property: There is an optimal journey only using connections from $D(z_s)$ and then from $D(z_s^{parent}), \ldots, D(z_{root}), \ldots, D(z_t^{parent}), D(z_t)$. The sets of connections can be processed in this order, i.e. the profile CSA can work on the sets and considers the connections in ascending departure time. It also means that the connections are not processed in ascending departure time in general, as a set occurring later can contain connections that depart earlier than the latest of the previous ones. In the setting where no routes and trips exist this is no problem, however adaptions have to be made if one wants to consider those. The transitively closed footpaths are handled just like CSA handles them, they are just not explicitly part of the connection sets because their components do not stretch amongst multiple cells. The profile algorithm of CSA can be used in the same way in the accelerated version by storing profiles for every stop instead of the earliest arrival time and using the Long Distance Connection Set in order.
As the footpaths are required to be transitively closed they do not have a bigger impact on the query as in the normal CSA. However, they give additional restrictions for the partitions because all transitively closed sets have to be part of one cell. The Arrival Time Transit Set $T_a(z)$ is sufficient for networks with those footpaths. If one wants to optimize the number of transfers (this only makes sense for trips and routes which are not considered here) or minimum change times, a bigger transit connection set is necessary. It is the Transfer Transit Set $T_t(z)$.
The original paper [SW14] contains more optimizations, such as only walking the hierarchy of components up to the lowest common ancestor cell $z_{lca}$ of $z_s$ and $z_t$ and descending

from there to $z_t$. However it is necessary to compute a set of loop connections $L(z_{lca})$, that contains connections to model all shortest journeys leaving and re-entering the cell. This approach is not pursued any further as it has no impact on the asymptotic runtime. There is always a pair of stops where the lowest common ancestor is the root cell and thus this optimizations does not accelerate the query asymptotically at all. Quite in opposition the loop connection set only makes it more complicated and causes more overhead.

# 3. Runtime Of Established Algorithms

In this chapter the complexity of known algorithms like Dijkstra for different graph models, the Connection Scan Algorithms and a mix of both for networks with arbitrary footpath sets will be analysed. The runtimes will give an estimate for the practicability of the algorithms and the worst case examples show weaknesses towards special networks. But first, implementations of the profile-operations will be given as they are needed for some of the following algorithms.

## 3.1 Implementation Of Profile-Operations

In the preliminaries some basic operations like shifting, merging and linking were introduced, however no algorithms and runtimes were given. The following section will present solutions with linear runtime. The operations will then be used in the following algorithms.

### 3.1.1 Minimizing Tuple-Sets

Although not being an operation on profiles itself, making a set of tuples Pareto optimal is an important component of other algorithms. The algorithm presented here requires a set of $(t_{dep}, t_{arr})$-tuples that is already sorted by departure time. It sweeps over the sequence and removes tuples that are not necessary. Those are the ones departing after another journey, but arriving earlier. Figure 3.1 shows a set of tuples with the last one dominating the rest.

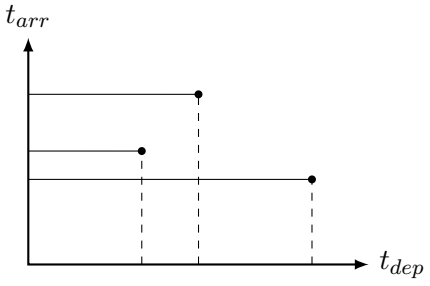



Figure 3.1: Set of tuples with dominated entries

Algorithm 3.1 solves the problem and is optimal as every departure and arrival time is processed once. It scans the sequence of tuples from back to front and evaluates if the

current tuple can be inserted while preserving Pareto optimality.
It has linear runtime in the number of tuples as every tuple is considered for insertion exactly once. The resulting set guarantees that arrival times are sorted in ascending order, too. The result represents a valid profile.

**Algorithm 3.1:** MINIMIZING TUPLE SETS

**Input:** Sequence of distinct departure-/arrival-time tuples $\mathsf{T} = ((t^0_{dep}, t^0_{arr}), (t^1_{dep}, t^1_{arr}), \ldots, (t^j_{dep}, t^j_{arr}))$, ordered in ascending departure time.

**Output:** Pareto-optimal profile $profile$, describing the same function.

```
1  profile ← ((t^j_dep, t^j_arr))
2  t_last ← t^j_arr
3  for i ← j − 1 to 0 do
4      if t^i_arr < t_last then
5          profile.pushFront((t^i_dep, t^i_arr))
6          t_last ← t^i_arr
```

### 3.1.2 Merging

Merging profiles is not much different than merging any two sorted sequences. The algorithm merges them, so that a list of sorted tuples results. The only thing that has to be kept in mind is the Pareto optimality of profiles. Merging the profiles does not keep this property intact, however they can simply be minimized with the algorithm shown before. The walking duration is simply the minimum of both. The runtime is dominated by scanning the two profiles with constant effort for both.

**Algorithm 3.2:** MERGING PROFILES

**Input:** Profiles $profile_1 = ((t^0_{dep_1}, t^0_{arr_1}), \ldots, (t^a_{dep_1}, t^a_{arr_1}), (\infty, \infty))$ and $profile_2 = ((t^0_{dep_2}, t^0_{arr_2}), \ldots, (t^b_{dep_2}, t^b_{arr_2}), (\infty, \infty))$, each terminated with a guard entry and walking durations $w_1$ and $w_2$ respectively

**Output:** The merged input with tuples $profile$ and walking duration $w$

```
1  i, j ← 0
2  while i ≤ a ∨ j ≤ b do
3      if t^i_dep1 < t^j_dep2 then
4          profile.pushBack((t^i_dep1, t^i_arr1))
5          i ← i + 1
6      else
7          profile.pushBack((t^j_dep2, t^j_arr2))
8          j ← j + 1
9  w ← min(w1, w2)
10 minimize profile // Make it Pareto optimal
```

### 3.1.3 Linking

The linking-operation for profiles takes two profiles, one from $u$ to an intermediate stop $v$ and one from $v$ to another stop $w$. There are some possibilities for shortest journeys from $u$ to $w$. They can start with a journey being described by an entry of $profile^v_u$ and then walk to $w$ or take another journey at $v$. To compute those journeys for every entry

of $profile_u^v$ the best journey at $v$ needs to be known. This is either a direct walk to the destination or an entry of $profile_v^w$. This is what the nested loops in the algorithm are for. They sweep over both profiles and compute the next entry departing at $v$ for every entry of $profile_u^v$. For every such combination the algorithm checks whether walking to the destination or taking the journey described by a tuple of $profile_v^w$ is better. The resulting sequence of tuples is stored in $profile_u^{w\star}$, which is only a tentative profile at first and is made into a Pareto optimal sequence at the end.
The only journeys that are not considered yet are ones that take a series of footpaths from $u$ to $v$ and then proceed to $w$. A profile for these journeys can be computed by shifting $profile_v^w$ by the time it takes to walk from $u$ to $v$. The shifting itself is done in negative x-direction as one needs to leave earlier to get the first connection of the journey. The implementation is rather simple, the algorithm only needs to adapt the departure time values of $profile_v^w$ and can be done in linear time. This shifted profile and $profile_u^{w\star}$ can now be merged to get the final result. The walking time of the resulting profile needs to be set to the sum of the walking times of the input profiles. The runtime for this operation is linear in the number of entries of both profiles.

**Algorithm 3.3:** $\oplus(profile_u^v, profile_v^w)$

**Input:** $profile_u^v$ and $profile_v^w$, each ending with the dummy entry $(\infty, \infty)$
**Output:** The profile $profile_u^w$

```
1  (t^u_dep, t^u_arr) ← first entry of profile^w_u
2  (t^v_dep, t^v_arr) ← first entry of profile^w_v
3  while (t^u_dep, t^u_arr) ≠ (∞, ∞) ≠ (t^v_dep, t^v_arr) do
4      while t^u_arr < t^v_dep do
           // Check if walking at v is faster
5          if t^u_arr + profile^w_v.walkingTime < t^v_arr then
6              profile^{w⋆}_u.INSERT(τ^u_dep, τ^u_arr + profile^w_v.walkingTime)
7          else
8              profile^{w⋆}_u.INSERT(t^u_dep, t^v_arr)
9          (t^u_dep, t^u_arr) ← next ordered entry of profile^v_u
10     (t^v_dep, t^v_arr) ← next ordered entry of profile^w_v
11 minimize profile^{w⋆}_u
12 x ← profile^v_u.walkingTime
13 profile^w_u ← profile^w_{v−x} ∪ profile^{w⋆}_v
14 profile^w_u.walkingTime ← profile^w_v.walkingTime + profile^v_u.walkingTime
```

## 3.2 Using Dijkstra's Algorithm

A first approach to one-to-one queries is to use existing well known algorithms like Dijkstra to find shortest journeys. For the representation of the network as a graph two models [PSWZ08] were introduced in the previous chapter: the time-dependent and the time-expanded model. For the time-expanded model Dijkstra's algorithm can be used without much adaption. For the time-dependent one small adjustments have to be made, changing the behaviour and runtime of the algorithm.

### 3.2.1 Time-Dependent Model

Figure 3.2 shows a simple network and the corresponding profiles. An edge with a profile only has to be introduced if there is any footpath or connection between the stops. To

relax an edge $(u, v)$ and compute the profile from $s$ to $v$ the tentative profile (a profile that does not consider all connections and footpaths yet) at $u$ and the profile of the edge $(u, v)$ can be linked:

$$profile_s^v \supseteq profile_s^u \oplus profile_u^v$$

The $\subseteq$-relation is supposed to symbolize that $profile_s^v$ can contain more entries for other shortest journeys which do not visit $u$. Moreover, all incoming edges $E_i$ at $v$ have to merged to cover all journeys:

$$profile_s^v = \bigcup_{(u,v)\in E_i} profile_s^u \oplus profile_u^v$$

Recall that $profile_u^v$ does not have to be computed as it is the value of the edge $(u, v)$ and part of the input graph. The $\cup$-notation is used for the merging of profiles. It is clear that this statement holds as every journey causing an entry in $profile_s^v$ has to pass through one of the incident edges.
For example getting $profile_{p_0}^{p_3}$ can be done by relaxing the edge $(p_2, p_3)$ and computing $profile_{p_0}^{p_2} \oplus profile_{p_2}^{p_3}$, where $profile_{p_2}^{p_3}$ is the profile of that edge. In contrast to Dijkstra's algorithm for shortest paths, an existing label at the target stop has to be merged with the newly generated one.

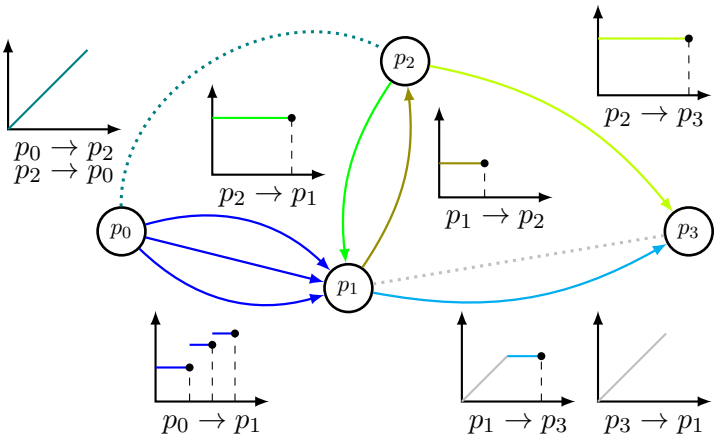



Figure 3.2: Time-dependent graph of a network with colours showing connections or footpaths causing the parts in the profiles.

One problem that still remains for the usage of Dijkstra's algorithm is the absence of a total order on profiles. Although they can dominate each other, they can also be incomparable. This raises the question for the value of the keys to use in the priority queue. Settling every node only once can be guaranteed for the shortest path Dijkstra because the minimal key of the queue is falling monotonously. When settling a node $v$ this ensures that all shortest paths that are smaller than the one to $v$ were already computed. Profile queries ask for multiple shortest journeys departing at different times from the source stop. Being monotonously falling means that all shorter shortest journeys for different departure times have to be computed when a stop is settled. This can not be guaranteed when settling stops as a whole. As shown in figure 3.2, when starting the search at $p_0$, then there are two journeys to $p_3$ (assuming that the profiles on the edges are chosen accordingly): $(p_0, p_2, p_1, p_3)$ and $(p_0, p_1, p_2, p_3)$. $p_2$ and $p_1$ can not be settled before one or another as information for one journey would be missing. Therefore it is not possible to maintain the label-setting property and the algorithm has to be label-correcting (meaning that stops can be settled multiple times). Algorithm 3.4 shows the pseudocode for the label-correcting algorithm which uses the linking operation to compute labels. It is worth mentioning that

**Algorithm 3.4:** LABEL-CORRECTING DIJKSTRA

**Input:** Public transit network, source stop $s$
**Output:** Label $l(p)$ for every stop $p$ with $profile_s^p$

```
1  for p ∈ P, p ≠ s do
2      l(p) ← ∞ − profile // profile with (∞, ∞) as only entry
3  l(s) ← id − profile // initialized with identity-profile
4  Q.INSERT(s, 0)
5  while Q ≠ ∅ do
6      u ← Q.DELETEMIN()
7      for edges (u, v) with respective value profile_u^v do
8          l(v) ← l(v) ∪ (l(u) ⊕ profile_u^v)
9          if l(v) was imroved then
10             t_arr^min ← minimal arrival time of l(v)
11             Q.INSERT(v, t_arr^min) // decrease the element if already contained
```

although the runtime changes the algorithm stays correct as long as a viable stop criterium is used. The algorithm presented here answers one-to-all queries and thus does not use any target pruning. It is therefore sufficient to stop when the queue is empty. The order for settling stops can be chosen arbitrarily, although this can impact the runtime. The minimal arrival time of a profile is used to get boundaries for the analysis in the next section.
Slightly altering the problem and using directed footpaths instead of undirected ones can lead to restricted graph classes where Dijkstra can remain label-setting. One of those are directed acyclical graphs (DAGs), regarding connections and footpaths as edges. A topological sorting can be obtained and used for the order of settling stops. This works because all stops that are on a shortest path from $s$ to a stop are settled before the stop itself. It also eliminates the need for a dynamic priority queue and the order can be computed in advance.
The approach is not the only way to run an adapted version of Dijkstra to solve profile-queries. Another one that exists is the Self-Pruning Connection Setting algorithm [Paj13]. It concurrently runs one indexed query for every connection leaving at $s$. The labels for stops contain the arrival time and this index. When settling a stop, a pruning rule can be applied: If the stop was already settled for a higher index (belonging to a connection that leaves later at $s$), the entry in the profile is dominated and the search can stop for this index at this stop. SPCS is therefore not too different from the algorithm presented here, only that this algorithm prunes journeys by keeping Pareto optimal profiles and throws those entries away if not needed and SPCS checks this with the indices. The main difference between the algorithms lies in the relaxation of edges, which does not have an impact on the runtime. The connection setting property, which determines the runtime, holds for the label-correcting Dijkstra, too. It will be proven in the next section. SPCS is interesting in practice as it can be parallelized very well.

### 3.2.2 Runtime of the Label-Correcting Algorithm

The runtime of the label-correcting algorithm is heavily impacted by the possibility of settling a stop multiple times. It depends on the chosen order and the toplogy of the network. Guarantees can be given when using the minimal arrival time of the profile's entries as the key. This is either the arrival time of a connection leading to the stop or a footpath arriving there.

**Lemma 3.1.** *If a stop p and its profile were taken out of Q and the minimal arrival time $t_{arr}^{min}$ used as key is the arrival time $\tau_{arr}(c)$ for any connection c, then the profile-entry with arrival time $\tau_{arr}(c)$ can not be improved in the future.*

This is because there are no other journeys having a smaller arrival time when removing the stop from $Q$. Otherwise another stop would have been taken out of the queue instead. Therefore the algorithm proceeds by considering only arrival times $> \tau_{arr}(c)$ and $c$ can not be improved. This also means that the profile can only be repeatedly inserted into $Q$ because a later entry improves the profile. In this sense, $c$ does not lead to a repeated insertion and can be regarded as settled.
For footpaths, this is not the case and they can contribute to multiple shortest journeys at any time. Finding good boundaries for the number of times they trigger improvements for better profiles and thus the need of resettling stops is very difficult. A footpath can be contained in each shortest journey, none or everything in between. There are $k + 1$ journeys at maximum, so a footpath could trigger $k$ settling-operations.

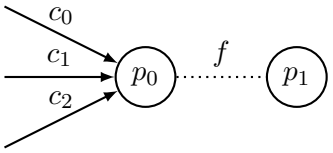



Figure 3.3: Network where a footpath leads to one settle-operation for every connection

Figure 3.3 shows a network where a footpath leads to a settle operation at $p_1$ for every connection. For this to work, the following inequality has to hold:

$$\tau_{arr}(c_0) + \tau_{dur}(f) < \tau_{arr}(c_1) < \tau_{arr}(c_2) - \tau_{dur}(f)$$

This way, the connections arrive with a margin of $\tau_{dur}(f)$ in between. Every time a connection is scanned, $p_0$ is taken out of the queue and $f$ leads to an insertion $p_1$. More footpaths can be added by connecting them to one new stop each and setting $\tau_{dur}$ low enough.
In contrast to Dijkstra's algorithm for normal graphs, edges (consisting of connections and footpaths) can be relaxed multiple times as this is done every time the incident stop is settled. Additionally, the cost of relaxing an edge is not constant any more as it kicks off one linking and one merging operation. The runtime for those operations depend on the number of connections represented by the edge $(u, v)$ and the size of the profile at the arrival stop $v$ of the edge. As $profile_s^u$ and $profile_u^v$ are linked and merged with the tentative profile $profile_s^v$, the linear linking- and merging-operations together take

$$\mathcal{O}(|profile_s^u| + |profile_u^v| + |profile_s^v|)$$

The cost of one settle-operation can therefore be as high as $\mathcal{O}(k + m)$. The size of the priority queue is bounded by the number of stops. One connection can cause one settle operation as stated above and a footpath can cause $\mathcal{O}(k)$, so there are $\mathcal{O}(m \cdot k)$ in total. This is also the number of insertions into (removal of minimal elements of) the priority queue. This gives the bound:

$$\mathcal{O}((m \cdot k) \cdot (T_{insert}(n) + T_{deleteMin}(n) + T_{decreaseKey}(n) + k + m))$$

The functions named with $T$ represent the queue-operations named the same way. $m \cdot k$ is the number of settle operations, each causing at most one insertion, a deletion and the

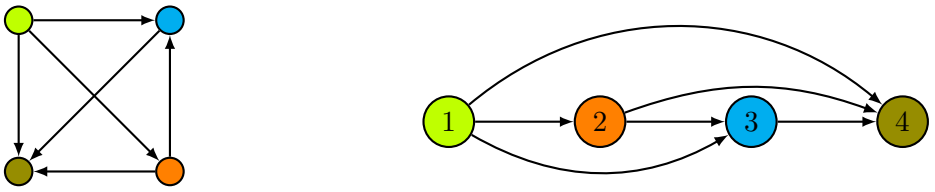



Figure 3.4: DAG (left) with nodes drawn in topological order on the right

effort for linking at merging can be at most $k + m$. For a Fibonacci-Heap the runtime becomes

$$\mathcal{O}((m \cdot k) \cdot (log(n) + k + m)).$$

and shows that in comparison to a normal execution time of Dijkstra's algorithm, slow times have to be expected.

### 3.2.3 Time-Expanded Model

Instead of modelling the network as a graph with time-dependent edges, it is also possible to translate the problem-instance to a graph with constant weights. It enables the usage of the standard Dijkstra and all its speed-up techniques. On the downside, the resulting graph is much bigger and the incorporation of footpaths is more complicated. Therefore the version introduced in the preliminaries will be used and no footpaths will be considered. It was already stated that the graph after the transformation is a directed acyclic graph (DAG) because the graph can be drawn with nodes arranged from top to bottom, representing times of departure and arrival events, and edges only running from higher nodes to lower nodes. The following result can be used to solve such instances.

**Theorem 3.2.** *Shortest paths for a directed acyclic graph $G = (V, E)$ can be found in time $\mathcal{O}(|V| + |E|)$ by using the topological order of the instance.*

Every directed acyclic graph $G = (V, E)$ induces a topological order on its nodes. An example is shown in figure 3.4. All nodes are on a path given by this ordering with edges running only from lower to higher nodes. In particular, every shortest path is a subpath of this path and the order of its nodes is strictly increasing. The following algorithm is therefore sufficient to find shortest paths.

**Algorithm 3.5:** SHORTEST PATHS IN DAG'S

**Input:** Graph $G = (V, E)$ with $V = (v_0, \ldots, v_{|V|-1})$ in topological order, starting node $v_s$, target node $v_t$ and weight function $w : E \to \mathbb{R}$

**Output:** Shortest path $p$, encoded in the parent array $p$

```
// Initialize parent-array to v_s for v_s and ⊥ for every other node
// Initialize distance array d to 0 for v_s and ∞ for others
for v ← v_s to v_{t-1} do
    for e = (v, w) ∈ E do
        if d(v) + w(e) < d(w) then
            d(w) ← d(v) + w(e)
            p(w) ← v
```

The algorithm is very similar to Dijkstra, with the crucial difference that no priority queue is needed to handle the nodes. Edges are relaxed in the same way. The topological sorting

used for determining the order can be computed in $\mathcal{O}(|V| + |E|)$ by a depth-first search. While transforming the public transit network into a DAG, for every connections two event-nodes and one edge are added. Furthermore, edges connecting the events of a stop are added. No more edges are added this way than there are nodes, making the sum of edges and nodes in the graph $\mathcal{O}(k + n)$. This means that the runtime of the shortest path algorithm is equal to $\mathcal{O}(k + n)$.
Note that those are time-queries and therefore barely comparable to the label-correcting Dijkstra above.

## 3.3 Asymptotic Runtime for the CSA-Family

The following section deals with the runtime of the Connection Scan Algorithms [DPSW17]. In the beginning they are analysed for time-queries with and without footpaths. Afterwards, the profile-queries are further separated into ones that use special adaptions to evaluate profiles in constant time (denoted with a C at the end of the name) and those which hold a Pareto optimal set (denoted by a P). To distinguish between profile and non-profile algorithms in their denotations, profile algorithms are prefixed with a small p. Table 3.3 gives an overview on the results, they are explained in the subsequent sections. All runtimes require that enough stops are contained in the network and are therefore not factored in.

| | Profile | Foothpaths | Preprocessing | Query |
|---|---|---|---|---|
| CSA | ○ | ○ | $k \cdot log(k)$ | $k$ |
| CSA | ○ | ● | $k \cdot log(k)$ | $k \cdot \sqrt{m}$ |
| pCSA-P | ● | ○ | $k \cdot log(k)$ | $k \cdot log(k)$ |
| pCSA-C | ● | ○ | $k \cdot log(k)$ | $k$ |
| pCSA-P | ● | ● | $k \cdot (log(k) + \sqrt{m})$ | $k \cdot \sqrt{m} \cdot log(\frac{k}{\sqrt{m}})$ |
| pCSA-C | ● | ● | $k \cdot \sqrt{m} \cdot log(\frac{k}{\sqrt{m}})$ | $k \cdot \sqrt{m}$ |

Table 3.1: Asymptotic worst case runtime for the Connection Scan Algorithms

### 3.3.1 Asymptotic Runtime for Time-Queries

The Connection Scan Algorithm needs a sorted connection array to scan them in increasing departure time. The preprocessing therefore sorts the connection and needs time $\mathcal{O}(k \cdot log(k))$.

Algorithm 3.6 shows the general pseudocode for the earliest arrival problem with footpaths. If the network does not contain any footpaths, lines 9 to 11 can be left away. In this case, the algorithm needs constant time for every connection it is scanning to compare arrival and departure times. The total runtime then becomes $\mathcal{O}(k)$. As the connections are scanned in order, only stops that are incident to a connection are considered and the runtime is not impacted by the number of stops.

If footpaths are incorporated, all incident footpaths of a connection's arrival stop have to be scanned when relaxing it. To get the worst case runtime for the algorithm one has to construct an instance where connections lead to a maximum number of footpath relaxations. The linear overhead for scanning the array always stays the same. To maximize the footpaths for every connection, they can be connected to the stop with the maximal number of incident footpaths. This works because the connections can be distributed independently on the stops and do not affect the number of relaxation for other ones. As figure 3.5 demonstrates, all connections can lead to one stop that has a maximum number of incident footpaths. In the example it is $p_3$. It therefore suffices to to maximize the

**Algorithm 3.6:** CSA WITH FOOTPATHS

**Input:** Timetable network, source stop $s$ and query time $t_{dep}$
**Output:** Label $l(p)$ for every stop $p$ with the earliest arrival time for any journey from $s$

```
// Initialize the data-structures
for p ∈ P, p ≠ s do
    l(p) ← ∞
l(s) ← t_dep
for (s, p) ∈ F do
    l(p) ← t_dep + τ_dur(f) // set times for footpaths from the source stop
// Main loop
for c ∈ C in increasing departure time do
    if τ_dep(c) > l(π_dep(c)) and τ_arr(c) < l(π_arr(c)) then
        l(π_arr(c)) ← τ_arr(c)
        // Check for footpaths at the target stop
        for {π_arr(c), p} = f ∈ F do
            if τ_arr(c) + τ_dur(f) < l(p) then
                l(p) ← τ_arr(c) + τ_dur(f)
```

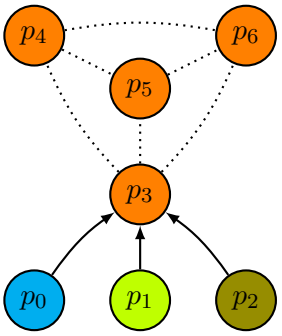



Figure 3.5: Network with worst case complexity for CSA with footpaths. Connected footpath components are drawn in the same colour.

number of footpaths incident to one stop. The result of the next theorem can be applied to get the runtime.

**Lemma 3.3.** *In an undirected graph $G = (V, E)$ with transitively closed edges $E$, the maximal degree of a node is $\Theta(\sqrt{|E|})$.*

*Proof.* First recall that it is not possible for one stop to be incident to all edges as it conflicts with the transitive closeness. Then observe that every connected component induces a complete subgraph $G' = (V', E')$ where $V'$ are all nodes incident to a footpath in the component and $E'$ all edges of the connected footpath component. If it was not complete, then there is an edge $\{v, w\} \notin E'$. This means that $G'$ is not transitively closed as $v$ and $w$ are connected (since $G'$ is connected) but the edge $\{v, w\}$ is not in $E'$. $\lightning$
Building one complete graph of maximum size with the edges available maximizes the node-degree. In a complete graph with $l$ nodes, every node is connected to $l - 1$ others.

Summing over the number of edges for every node counts each edge twice. Therefore the total number of edges is

$$|E| = \frac{l \cdot (l-1)}{2}$$

and $l \in \Theta(\sqrt{|E|})$. The degree of every node is $l - 1$ and the lemma is proven. □

Figure 3.5 shows all nodes of connected footpath-components in different colours. Orange ones are contained in the component of maximal size and contains $p_3$ where all connections lead to. With the lemma, one can conclude that the total runtime is $\mathcal{O}(k \cdot \sqrt{m})$, with effort of $\sqrt{m}$ for every scan of a connection.

### 3.3.2 Asymptotic Runtime for Profile-Queries

Next, the complexity of profile-queries will be investigated. In contrast to time-queries, here the query-mode is all-to-one, meaning that profiles are computed from every stop to the target stop $t$. Algorithm 3.7 shows the pseudocode for the query. It initializes all profiles except the one of $t$ with the guard $(\infty, \infty)$ as the only entry. The profile of $t$ gets special treatment as $t$ is the only stop that is reachable although no connections are considered yet. If a traveller departs at $t$ and wants to travel to $t$ he needs no time as he is already there. Therefore $profile_t^t$ is the identity profile. It could be implemented as a profile with a walking duration of 0. The only two lines where the runtime is not clear at first glance are

**Algorithm 3.7:** CSA FOR PROFILE QUERIES

**Input:** Timetable network, target stop $t$
**Output:** Label $l(p)$ for every stop $p$ containing $profile_p^t$

```
// Initialize the data-structures
```
1 **for** $p \in P, p \neq t$ **do**
2  $l(p) \leftarrow$ profile with the only entry $(\infty, \infty)$
3 $l(t) \leftarrow id$
```
// Main loop
```
4 **for** $c \in C$ *in decreasing departure time* **do**
5  $t_{arr} \leftarrow$ evaluate profile at $\pi_{arr}(c)$ for $\tau_{arr}(c)$
6  **if** *entry* $(\tau_{dep}(c), t_{arr})$ *improves* $l(\pi_{dep}(c))$ **then**
7  Add $(\tau_{dep}(c), t_{arr})$ to $l(\pi_{dep}(c))$

5 and 7. For the stop $p = \pi_{arr}(c)$ and the arrival time of the currently scanned connection $\tau_{arr}(c)$ the query returns the arrival time $t_{arr}$ of the entry $(t_{dep}, t_{arr})$ with $\tau_{arr}(c) < t_{dep}$ and $t_{dep}$ minimal. This is the arrival time of the next connection departing after $c$ arrives. As profiles are Pareto optimal this is the earliest arrival time at the destination. This operation can be implemented in different ways. The naive way is to do a binary search on the sorted list of tuples and find the next entry. Using this approach leads to the pCSA-P variant of CSA, while the constant evaluation discussed in the next section is used in pCSA-C. The time needed for the binary search depends on the number of entries in the profile at the arrival stop, which can be at most $k$ as every departure time of a tuple correspondes to a departing connection at the stop. Therefore the time for the evaluation is $\mathcal{O}(log(k))$.
Indeed, the evaluation can take that long for half of the connections. Consider the graph in figure 3.6 where each edge represents a single connection. Let $C = C_1 \cup C_2$ and let $C_1$ contain all connections from $p_1$ to $t$ and $C_2$ contain all connections from $s$ to $p_1$. If

$|C_1| = |C_2|$, all connections in $C_1$ depart at a distinct time and

$$\forall c \in C_1 : \tau_{arr}(c) - \tau_{dep}(c) = r_1, \tau_{dep}(c) < r_2 \text{ for } r_1, r_2 \in \mathbb{R},$$
$$\forall c \in C_2 : \tau_{arr}(c) > r_2,$$

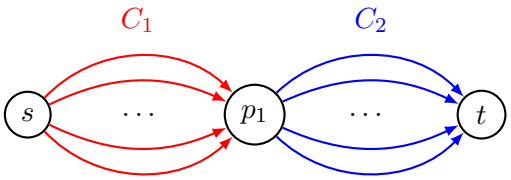



Figure 3.6: Network with worst case complexity for pCSA-P without footpaths

In this case, connections from $p_1$ all depart before any connection from $s$ arrives. As connections are scanned sorted by descending departure time all connections in $C_2$ are scanned before the ones in $C_1$. There are $\mathcal{O}(\frac{k}{2})$ of them which are all added to the profile at $p_1$ because they all take the same time and have a distinct departure time. Afterwards, the profile-queries at $p_1$ for the connections in $C_1$ take $\mathcal{O}(log(k))$ for the binary search. The other non-trivial operation is executed in line 7. The new tuple is added to the profile at the departure stop. As connections $c$ are scanned in decreasing departure time $\tau_{dep}(c)$ and all insertions are done with $\tau_{dep}(c)$ as the departure time of the tuple, departure times of the insertions are decreasing, too. This is why the new tuple can be appended at the beginning of the profile. To maintain Pareto optimal profiles, the new entry has to be checked for domination, but it is enough to only compare it to the now second entry $(t^2_{dep}, t^2_{arr})$. For any later tuple $(t^\star_{dep}, t^\star_{arr})$ the departure times are ordered: $t^2_{dep} < t^\star_{dep}$. Furthermore the profile was optimal before the insertion and $t^\star_{arr} > t^2_{arr}$. If $(t^\star_{dep}, t^\star_{arr})$ dominates the new entry $(t^1_{dep}, t^1_{arr})$ and $(t^2_{dep}, t^2_{arr})$ does not, then $t^\star_{arr} < t^1_{arr}$ and $t^2_{arr} > t^1_{arr}$. Together, this is a contradiction:

$$t^\star_{arr} < t^1_{arr} < t^2_{arr} < t^\star_{arr}$$

So, constant time is enough for the insertion of the new tuple and all lines except for 5 can be executed in constant time. The evaluation of the profile was shown to take $\mathcal{O}(k)$ and can be achieved with the network in figure 3.6. One can conclude that the algorithm takes $\mathcal{O}(k \cdot log(k))$.

### 3.3.3 Constant Evaluation of Profiles

pCSA-C still uses the same pseudocode as in algorithm 3.7, with the exception that the evaluation of profiles and insertion of tuples are handled differently.

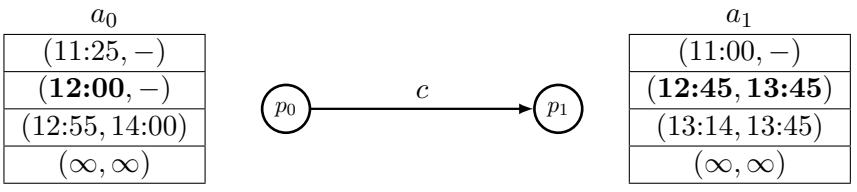



Figure 3.7: Excerpt of a network with arrays as labels. Bold entries are referenced via pointers of the connection.

Figure 3.7 depicts an excerpt of a network. $p_0$, $p_1$ and the connection $c$ are the only elements drawn, although there are more connections and stops in the network. They caused the

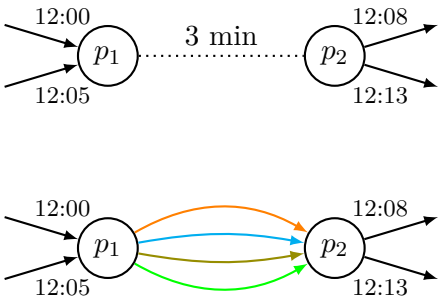
entries in the labels $a_0$ and $a_1$ of the stops. Furthermore, $c$ departs at $\tau_{dep}(c) = 12{:}00$ and arrives at $\tau_{arr}(c) = 12{:}30$. The algorithm has already scanned all connections that depart later than $c$ and is now beginning to scan $c$ itself. The labels contain placeholder-entries, one for every connection departing at the stop. They are filled in for all departure times after $\tau_{dep}(c)$. Something that is not possible when the profiles are held as a Pareto-set is the existence of dominated entries. $a_1$ contains such entries as pCSA-C does not hold Pareto-sets, but sets of tuples. The time for the entry (12:45,-) was not improved when the outgoing connection at $p_1$ with departure time 12:45 was scanned. Therefore the arrival time of a later tuple was put in so that the tuple still contains the optimal arrival time for the departure at 12:45.
For each connection, two template-tuples are referenced via pointers: One containing the departure time of the connection at $\pi_{dep}(c)$ and one that points at the tuple of $\pi_{arr}(c)$ that has the closest departure time after $c$ arrives. The ones for $c$ are highlighted in bold font in the figure. When $c$ is scanned, the tuple at the arrival stop already contains the optimal arrival time at that stop since the departure time of that tuple is later and the outgoing connection at 12:45 was scanned before. The time 13:45 can now be taken as the optimal arrival time if the connection $c$ is taken. Now the tuple at $a_0$ needs to be filled in. It is still possible that a later entry dominates that time and that $c$ is not part of the optimal journey at 12:00. The time has to be compared with the next tuple in $a_0$ and the minimum is taken.
In total, the algorithm only considers three entries for the scan of a connection and is constant for that. This means that the runtime for the query is $\mathcal{O}(k)$. In the preprocessing, the pointers have to be computed. This can be done by creating the arrays with template-entries for all departure times of connections at that stop (the arrival times are undefined at that point and a placeholder is inserted for them) and then execute a binary search for every connection $c$ to get the tuple at the arrival stop that has the closest departure time after $\tau_{arr}(c)$. This results in logarithmic time in $k$ for the binary search of every connection and therefore makes $\mathcal{O}(k \cdot log(k))$ for the preprocessing.

### 3.3.4 Profile Queries with Footpaths

Footpaths can be handled by replacing them with enough connections so that profile-queries in the new network provide the same result. In the preprocessing this is done by iterating over all footpaths $\{u, v\} = f \in F$ and looking at the connections departing and arriving at $u$ and $v$. If a connection $c_1$ arrives at $u$ and at $\tau_{arr}(c_1) + \tau_{dur}(f) = \tau_{dep}(c_2)$ a connection $c_2$ departs at $v$, then the footpath provides a way to get to the next stop in time to get $c_2$ after leaving $c_1$. Therefore a pseudo-connection $c_p$ can be added with $\tau_{dep}(c_p) = \tau_{arr}(c_1)$ and $\tau_{arr}(c_p) = \tau_{dep}(c_2)$. For every pair of incoming connection $c_1$ and outgoing connection $c_2$, it is tested whether $\tau_{arr}(c_1) + \tau_{dur}(f) \leq \tau_{dep}(c_2)$ and a pseudo-connection is added if it is the case. However doing so can add too many pseudo-connection and the FIFO-property is violated eventually. The FIFO-property stated that connections that depart earlier, arrive earlier, which means that the set of outgoing connections is Pareto optimal. Consider the footpath in figure 3.8.

The table lists the departure and arrival time of the connections which were added. $c_3$ dominates all other entries and it is sufficient to only include this connection as all successive connections at $p_2$ can be reached by taking it. To create an optimal set of pseudoconnections, the sweep algorithm 3.8 can be used.

It scans arrival and departure times in descending order and only adds connections which can not be dominated. The runtime is linear in the number of incoming and outgoing connections.
The pseudo-connections allow for connections that can be taken successively by using the connecting footpath to be reachable in an instance without footpaths, too. However,




| | $\tau_{dep}(\cdot)$ | $\tau_{arr}(\cdot)$ |
|---|---|---|
| $c_1$ | 12:00 | 12:08 |
| $c_2$ | 12:00 | 12:13 |
| $c_3$ | 12:05 | 12:08 |
| $c_4$ | 12:05 | 12:13 |

Figure 3.8: Network where a footpath is replaced by pseudo-connections

**Algorithm 3.8:** REPLACING FOOTPATHS WITH PSEUDOCONNECTIONS

**Input:** Sorted sequence of arrival times of incoming connections $(t^i_0, \dots, t^i_{a-1}, t^i_a = \infty)$ at stop $p_1$, sorted sequence of departure times of outgoing connections $(t^o_0, \dots, t^o_{a-1}, t^o_a = \infty)$ at stop $p_2$, footpath $f = \{p_1, p_2\}$

**Output:** Pareto-optimal sorted list $l$ of pseudoconnections replacing $f$

```
1 l ← b − 1
2 t_last ← ∞
3 for k ← a − 1 to 0 do
4     while t^i_k + τ_dur(f) ≤ t^o_{l+1} do
5         l ← l − 1
6     if t^o_l ≠ t_last then
7         l.PUSHFRONT(connection c with τ_dep(c) = t^i_k and τ_arr(c) = t^o_k)
8         t_last ← t^o_k
```

footpaths can also be taken at the beginning of the journey where no connections were taken yet and in the same way at the end of a journey. These are the footpaths incident to $s$ and $t$. To cover these journeys, CSA can be modified to ensure that the evaluation of profiles for stops that are connected to $t$ yield correct results. Profiles already contain a walking duration to the target which is used in the evaluation of a profile to check if walking to $t$ is faster than taking another connection. These walking times have to be initialized at the start of the query by iterating over all footpaths incident to $t$ and setting the times accordingly. This way, footpaths at the end of the journey are taken into consideration.
Footpaths at the start of a journey are incident to $s$ and after the CSA-query has finished, profiles at all adjacent (via footpaths) stops to $s$ are already correct for journeys that do not start with a footpath. Because of the transitive closeness of the footpaths, it suffices to shift the profiles at these stops by their walking duration to $s$ and merge them with the profile at $s$.
The effort for those two extra steps at the beginning and end of the query both are $\mathcal{O}(k \cdot \sqrt{m})$. There are $\mathcal{O}(\sqrt{m})$ stops adjacent to $s$ and each of the profiles at those stops have $\mathcal{O}(k)$ entries. As the merge-operation needs linear time this results in $\mathcal{O}(k \cdot \sqrt{m})$ together for all merge-operations at the end. The footpaths at the target only need one constant step for the evaluation of profiles. The setting of the walking time at the beginning can be done for $\mathcal{O}(\sqrt{m})$ at most and therefore needs $\mathcal{O}(\sqrt{m}) \subset \mathcal{O}(k \cdot \sqrt{m})$.
Now one can try to maximize the effort for the preprocessing and add a maximal number of new connections for a worst case query-time. Observe that every incoming connection can generate at most one pseudo-connection for a given footpath. If more pseudo-connections

than incoming connections exist, then one departure time has to occur more than once for the pseudo-connections replacing the footpath as all departure times of pseudo-connections are arrival times of incoming connections. This is not possible in a Pareto optimal set. The same holds for the outgoing connections. So the maximal number of pseudo-connections $P$ for the incoming connection-set $I$ and outgoing connection-set $O$ of stops incident to a footpath is

$$|P| \leq min\{|I|, |O|\}$$

Further, if every connection can only cause one pseudo-connection for every footpath at its arrival and departure stop, then every connection can generate a maximum of $\mathcal{O}(\sqrt{m})$ of them as this is the maximal amount of footpaths at these stops. Given that every connection creates this maximal number, a total of $\mathcal{O}(k \cdot \sqrt{m})$ pseudo-connections are added. Then this is also the maximal time the preprocessing needs additionally as every linear sweep of algorithm 3.8 adds a linear amount of pseudo-connections in this case.
Indeed, it is possible to realize the number of $\mathcal{O}(k \cdot \sqrt{m})$ pseudo-connections. Figure 3.9

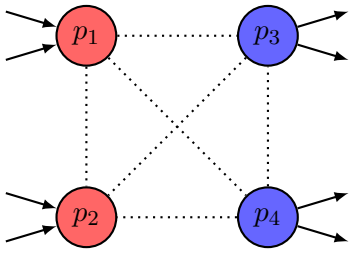



Figure 3.9: Network for a maximal number of added pseudo-connections

shows an example for this worst-case. It depicts the maximal footpath component which is logically split into red stops that only have incoming connections and blue stops where connections depart. There are $\mathcal{O}(\frac{\sqrt{m}}{2}) = \mathcal{O}(\sqrt{m})$ red and blue stops. Connections are distributed evenly amongst them, meaning that every stop has $\frac{k}{\sqrt{m}}$ connections. Between the sets of stops with incoming and outgoing connections there are $\mathcal{O}(\sqrt{m}^2) = \mathcal{O}(m)$ footpaths as every pair $(p_1, p_2)$ of stops with $p_1$ and $p_2$ out of different sets are connected with a footpath. The number of added pseudo-connections then is $\mathcal{O}(m \cdot \frac{k}{\sqrt{m}}) = \mathcal{O}(k \cdot \sqrt{m})$.
For this instance each binary search of pCSA-P in the preprocessing takes $\mathcal{O}(log(\frac{k}{\sqrt{m}}))$, so the runtime is $\mathcal{O}(k \cdot \sqrt{m} \cdot log(\frac{k}{\sqrt{m}}))$ if a maximal number of pseudo-connections is added. This always dominates the time of $\mathcal{O}(k \cdot log(k))$ the version without footpaths takes.
After the pseudo-connections are added to the connection-set, the profile algorithm with constant evaluation can be used without adaption. As stated above, its preprocessing consists of the execution of the profile algorithm without constant evaluation. This time is $\mathcal{O}(k \cdot \sqrt{m} \cdot log(\frac{k}{\sqrt{m}}))$. The query then scans over the connection-array with constant time for every scan. Now that there are $\mathcal{O}(k \cdot \sqrt{m})$ connections in total this takes $\mathcal{O}(k \cdot \sqrt{m})$.

## 3.4 Connection Scan Accelerated

In comparison to the standard CSA the accelerated version [SW14] extends on the idea of having a preprocessing and a query phase. In the preprocessing of the algorithm the network is first recursively partitioned into cells. Then the Arrival Time Transit Set $T_a(z)$ is computed for every cell $z$. To get the transit connection sets the interior and exterior border stops, i.e. the arrival stops of entry or exit connections, are identified and all profiles from interior to exterior ones are computed inside each cell. The runtime for this depends on the quality of the partitioning, the sizes of the cells, and the number of border stops.

Without any restrictions on the network even the best partitioning can give bad results. For example a network that induces a complete connection graph leads to partitions where every stop is a border stop. Another bad case where only little can be accelerated is a network that contains very big connected components of footpaths. These big components always have to be in the same cell and limit the sizes of them. Thus the complexity of the algorithm does not only depend on the number of stops, footpaths and connections, but also the following values:

- $s(z)$: the number of stops of a cell
- $c(z)$: the number of connections of a cell
- $b(z)$: the number of border stops (interior and exterior)
- $f(z)$: the number of footpaths in the cell
- $p$: asymptotic time for finding the recursive partitioning

Then the asymptotic time needed in addition to finding the partitioning is dominated by the computation of profiles, which takes $\mathcal{O}(T_{pCSA}(s(z), c(z), f(z))$ for the cell $z$ if $T_{pCSA}$ is the runtime of the profile algorithm in dependence of its parameters. The time needed to identify the border-stops is negligible for they can be found by scanning the connections of every stop and determining if one of its incident stops is in another cell, taking $\mathcal{O}(c(z))$. The profile queries are executed on every cell, which leads to this runtime for the preprocessing:

$$p + \sum_{z \in Z} b(z) \cdot T_{pCSA}(s(z), c(z), f(z))$$

where $T_{PCSA}(a, b, c)$ is the runtime of the profile CSA for $a$ stops, $b$ connections and $c$ footpaths. $Z$ is the set of all cells. Executing the all-to-one query of the profile CSA for all border-stops guarantees that profiles from every border-stop to every other border-stop are covered. This time is rather abstract but without any information about the partitioning it is complicated to get better bounds.
The same holds for the query-time. The number of connections which have to be scanned heavily depends on the quality of the preprocessing's result and the sizes of the transit connection sets. This number is hard to limit as every connection could possibly be necessary to cross a cell and no asymptotic improvement would be made. For this reason, this section only concentrates on the preprocessing of the algorithm.

### 3.4.1 Connection Scan Accelerated on Planar Networks

First it should be noted what a planar transfer-network is, as it contains footpaths and connections which could both be considered as some type of edges. In this context the graph given by the stops as vertices and connections as edges (replaced by undirected edges, multiple connections between the same stop are replaced by a single edge) suffices to be planar.
Now that those networks are narrowed down, some special properties can be used to fill in the placeholder functions and get better bounds. The following theorem found by Frederickson [Fed87] helps with this.

**Theorem 3.4.** *For a planar graph $G = (V, E)$ there exists an r-division, i.e. a division of a graph into $\Theta(\frac{|V|}{r})$ overlapping partitions with $\mathcal{O}(r)$ vertices and $\mathcal{O}(\sqrt{r})$ boundary vertices each. These can be found in time $\mathcal{O}(|V| \cdot log(|V|))$.*

Although the partitions (which are also called regions) are not disjunct, they only intersect on the boundary vertices, so the accelerated Connection Scan can be used if two regions

are joined successively to form a (balanced) hierarchy of cells. With these facts in mind a bound for the asymptotic runtime can be found.
The size of a cell depends on the level of it. The root cell contains all stops and is split into two asymptotically equally sized parts. Those are split equally again, thus a cell on level $i$ contains $\Theta(\frac{n}{2^i})$ stops and $\mathcal{O}(\sqrt{\frac{n}{2^i}})$ boundary stops. The number of connections can not be bounded that easily as in contrast to the standard definition of a planar graph there can be multiple connections between the very same stops. The time for finding the partitioning is $\mathcal{O}(k + n \cdot log(n))$ for replacing connections and using Frederickson's construction. Using runtimes of $T_{preproc}(k, m)$ for the preprocessing of the profile Connection Scan Algorithm and $T_{query}(k, m)$ for the profile queries executed on the boundary stops and further restrict that connections and footpaths are distributed evenly amongst cells adds up to:

$$\mathcal{O}(T_{preproc}(k, m) + k + n \cdot log(n) + \sum_{i=0}^{log(n)} 2^i \cdot \sqrt{\frac{n}{2^i}} \cdot T_{query}(\frac{k}{2^i}, \frac{m}{2^i}))$$

Although the number of footpaths of a cell is part of the parameter of the query-runtime, distributing footpaths evenly amongst the cells effectively means that there are no big footpath components or the graph can not be divided further at some point as footpaths of one component have to stay in the same cell. This is very restrictive and means that there are a lot of very small components. Using a runtime of $\mathcal{O}(k \cdot \sqrt{m})$ for the Connection Scan profile algorithm results in:

$$\mathcal{O}(\sqrt{n} \cdot \sqrt{m} \cdot k)$$

for the sum on the right counting the additional runtime for the executions of the profile algorithm on the boundary stops. This is only $\mathcal{O}(\sqrt{n})$ times bigger than the query-time and therefore not too bad. But although the preprocessing seems manageable, the query of Connection Scan Accelerated still depends on the number of connections in the Transit Set which is not bounded by the restriction made here. It is possible that all connections are necessary for all journeys from $s$ to $t$ and the query-time is the same as the normal CSA. It shows that speed-up techniques depend on heuristics of public transit network and are not that interesting for a look on complexity of the problem.

## 3.5 Algorithms for Arbitrary Footpath Sets

The previous Connection Scan Algorithms could not handle footpath-sets that are not transitively closed. Although every arbitrary footpath-set can be extended to be transitively closed, this comes with a lot of effort and a bigger number of additional footpaths. It is covered in section 3.5.4. On the other hand, arbitrary footpath-sets are interesting for applications. It is not too far fetched that one is interested in journeys where bigger parts can be taken by foot [WZ17]. The following sections will take a look at one approach to handle these graphs.

### 3.5.1 Functioning of the Algorithm

Algorithm 3.9 builds one the basic idea of the profile Connection Scan Algorithm, which is scanning connections in decreasing departure time while holding tentative profiles at every stop. Additionally, an interleaved Dijkstra search is executed on the footpaths for every (departure time, arrival time)-tuple to search for shortest paths for every departure time. This way all shortest journeys are found as every part of a journey where a series of footpaths is taken is appended by a connection that is taken afterwards (or it ends with a footpath to the destination). For all those connections and their best arrival times at $t$ the Dijkstra search is executed and will find the series of footpaths. The direct arrival at the destination is a special case that has to be handled separately by computing the walking

**Algorithm 3.9:** PROFILE ALGORITHM FOR ARBITRARY FOOTPATH SETS

**Input:** Public Transit Network, target node $t$
**Output:** Profiles from every node to $t$ departing between $minTime$ and $maxTime$

```
1  walkingTime(u) ← d(u,t) // Using Dijkstra's algorithm on the footpath
       graph starting at t
2  forall conn ∈ P ordered by descending departure time do
       // Interleaved Dijkstra on the footpath network
3      while Q is not empty and τ_dep(conn) ≤ Q.MAX() do
4          Settle the maximal queue element using algorithm 3.10
5      profile_arr ← profiles(π_arr(conn))
       // walk to the destination...
6      t_arrival ← τ_arr(conn)+ walkingTime(π_arr(conn))
       // ...or take another connection
7      t_arrival ← min{profile_arr.EVALUATE(τ_arr(conn)), t_arrival}
       // Insert new entry
8      profile_dep ← profiles(π_dep(conn))
9      if (π_dep(conn), t_arrival) improves profile_dep then
10         profile_dep.INSERT(π_dep(conn), t_arrival)
11         Q.UPDATE(profile_dep)
   // Empty the queue of the search
12 while Q is not empty do
13     Settle the maximal queue element using algorithm 3.10
```

time for every stop to the destination. It is done at the beginning of the algorithm in line 1. One global priority queue $Q$ (max heap) is maintained and used to search for shortest paths for every profile entry, meaning that the search is executed for every departure time of a profile entry. To achieve this, one can insert every (departure time, arrival time)-tuple with the associated stop into $Q$, although it has the drawback that $Q$ can hold a large amount of elements, making queue-operations slower. A small observation helps to solve this problem: The elements are taken out of the queue in descending departure time (this fact will be elaborated later) and if $(t_{dep}, t_{arr})$ is the last settled tuple, it is sufficient for $Q$ to only contain the tuple $(t^\star_{dep}, t^\star_{arr})$ of each profile with $t^\star_{dep} < t_{dep}$ and maximal $t^\star_{dep}$. It ensures that the global queue will always contain the best element while not holding all of them. Further, $(t^\star_{dep}, t^\star_{arr})$ has some special properties. It is always the the tuple with maximal departure time that was not settled yet for every profile. The tuple that was settled last has a higher departure time than the current $t_{dep}$ and the entry before it has a lower one, thus making it the element of interest.
This means that it is possible to insert profiles into the queue and use the last unsettled entry as the key. It guarantees $|Q| \leq |P|$, which can be used in the runtime analysis. To get the last unsettled entry of every profile in constant time, a pointer to the last entry can be stored at first, and is always moved to the previous entry when the profile is settled.
Algorithm 3.10 is executed to settle such a $(t_{dep}, t_{arr})$-tuple belonging to a stop $p$. It is a slightly adapted version of the settle operation of Dijkstra's algorithm with some modifications. For one, it searches backwards, meaning that when relaxing a footpath $f = \{u, v\}$ for a tuple belonging to $v$, it reduces $t_{dep}$ by $\tau_{dur}(f)$ to get the departure time for a journey that starts at $u$ and takes the footpath. The new tuple $(t_{dep} - \tau_{dur}(f), t_{arr})$ is only inserted into the profile of $u$, if it is not already dominated in the Pareto-sense. Furthermore, it is possible that the settled profile contains more unsettled entries and $Q$ has to be updated for it.

**Algorithm 3.10:** SETTLING ONE QUEUE ENTRY

**Input:** Public Transit Network and priority queue Q

```
1  profile_u belonging to node u ← Q.DELETEMAX()
2  (t_dep, t_arr) ← profile_u.NEXTUNSETTLEDENTRY()
   // Settle entry of u
3  forall f = {u, v} ∈ F do
4      profile_v ← profiles(v)
5      t_dep^new ← t_dep − τ_dur(f)
6      if entry (t_dep^new, t_arr) improves profile_v then
7          profile_v.INSERT(t_dep^new, t_arr)
8          Q.UPDATE(profile_v)
9  if profile_u.hasUnsettledEntries() then
10     Q.UPDATE(profile_u)
```

The interleaved Dijkstra searches are not executed until the queue is empty, but can be paused in between which does not affect correctness. However, it gives the surrounding profile Connection Scan Algorithm the possibility to scan connections to stops which can dominate entries generated by a footpath that leaves there. This optimization leads to less Dijkstra searches for unnecessary entries.

Figure 3.10 shows a network where this optimization is used. It shows the entries of

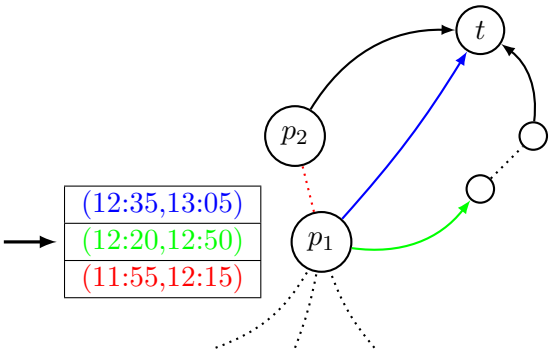



Figure 3.10: Network with connections and a footpath causing entries in the profile next to $p_1$. The arrow shows the last unsettled entry.

$p_1$'s profile and the arrow symoblizes the pointer to the last unsettled entry. The blue connection was scanned and created the first entry. Then some other connection at $p_2$ was scanned and the red footpath was relaxed, creating the red entry. Afterwards the green connection lead to the insertion of a tuple, which shows another property of the algorithm: Adding entries to a profile is not constant any more as footpaths can lead to other entries being inserted in between and it is not possible to use CSA with constant evaluation of profile. The red entry was not settled because its departure time was lower than the yet to scan green connection. The state of the image is right after the green connection was scanned.

Besides the interleaved Dijkstra during the scan of a connection, the necessary steps of the Connection Scan Algorithm have to be executed. These are:

- Check if the arrival stop is connected to the destination by a series of footpaths
- Check if another connection or footpath can be used at the arrival stop

After the new arrival time is calculated as the minimum of the cases shown above the new entry is added to the profile if it is not dominated. It has to be added to $Q$ in this case. Lastly, when all connections were scanned, the queue has to be emptied so that all journeys starting with footpaths are found.

### 3.5.2 Asymptotic Runtime

The asymptotic runtime is affected by the one of the profile Connection Scan Algorithm and the execution of the interleaved Dijkstra. As mentioned before, constant evaluation of profiles is not possible since footpaths lead to the insertion of entries with arbitrary departure times and no template-profile for pCSA-C can be computed in advance. Therefore the evaluation of a profile in line 7 of algorithm 3.9 takes logarithmic time to find the correct entry and the insertion in line 7 of algorithm 3.10 too. On the other hand, the pointer to the last unsettled entry can be held in constant time.
Dijkstra's algorithm can be executed multiple times, once for every profile-entry while generating new entries. It is even possible that the whole footpath network has to be looked at for every connection. Figure 3.11 shows an example where this is the case.

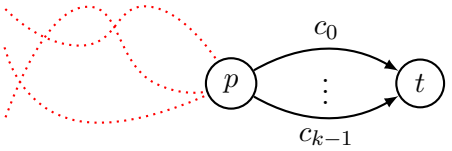



Figure 3.11: Worst case network for the algorithm. Red dotted lines symbolize a big footpath component while all connections run between $p$ and $t$.

Red dotted lines symbolize one huge complete graph $G$ consisting of stops and footpaths. All connections $c_0, \ldots, c_{k-1}$ run from $p$ to $t$. The departure and arrival times of them have to be chosen in a way that after a connection is scanned, the algorithm executes Dijkstra on the whole footpath subgraph. The search is paused if $\tau_{dep}(conn) > Q.MAX()$ and if $t_{max}$ is the maximal duration any shortest path takes in $G$, then the values of the connections can be chosen like this: $\tau_{arr}(c_0) = a$, where $a$ is an arbitrary number, $\tau_{dep}(c_i) = \tau_{arr}(c_i) - t_{max}$ for $i \in \{0, \ldots, k-1\}$ and $\tau_{arr}(c_i) = \tau_{dep}(c_{i-1})$ for $i \in \{1, \ldots, k-1\}$. This causes the Dijkstra search to never stop as for any connection $c_i, i \in \{0, \ldots, k-2\}$:

$$\tau_{dep}(c_{i+1}) \leq \tau_{dep}(c_i) - t_{max}$$

and $\tau_{dep}(c_i) - t_{max}$ is the minimal value $Q.MAX()$ can become. The asymptotic time needed for the search on $G$ is $\mathcal{O}(T_{Dijkstra}(n, m))$, where $T_{Dijkstra}(n, m)$ is the time needed for the execution of Dijkstra's algorithm on $n$ nodes and $m$ edges.
Additionally, the worst time for the CSA-part of the algorithm caused by the evaluation and insertion of entries into profiles can take $\mathcal{O}(log(k))$ per iteration as both operation are logarithmic in the number of connections. This was already discussed before with the worst case example in figure 3.6. In total, the evaluation and insertion then take $\mathcal{O}(k \cdot log(k))$. Together with the Dijkstra-part it makes

$$\mathcal{O}(T_{Dijkstra}(n, m) + k \cdot log(k))$$

The runtime can be refined by splitting the time needed for Dijkstra into the runtimes of the queue-operations $T_{deleteMin}$, $T_{insert}$ and $T_{decreaseKey}$. This results in

$$\mathcal{O}(k \cdot (n \cdot T_{deleteMin} + m \cdot T_{decreaseKey} + n \cdot T_{insert}) + k \cdot log(k)).$$

For a binary heap this is $\mathcal{O}(k \cdot (n + m) \cdot log(n) + k \cdot log(k))$ and for a Fibonacci heap $\mathcal{O}(k \cdot (m + n \cdot log(n)) + k \cdot log(k))$. This is clearly more than the usual runtime of the

Connection Scan profile algorithms. On a side note, although the network in figure 3.11 has this bad runtime, there is some potential for accelerating the algorithm in this case. The durations between stops in the footpath graph could be computed once. Then they would not have to be computed for every connection. However generalizing this idea is not trivial and it is not clear how this can be done in networks where the separation between footpath and connection components is not this strong.

### 3.5.3 Acceleration for Special Graph Classes

The runtime is pretty slow for normal networks and the pausing of the Dijkstra search can be nulled by setting the times between departing connections big enough as shown in previous section. Effectively, the search does not need to interleave between scanned connections and the same runtime can be achieved by doing a complete search for every connection. This has the advantage that normal shortest paths algorithms can be used without them taking the special structure of public transit networks into account. This can be done by executing them solely on the graph $G$ where stops are the nodes and footpaths edges. The length of the shortest path from $u$ to $v$ can be subtracted form the departure time of the connection departing at $v$ to get the maximal departure time when leaving at $u$. Some graph classes have better asymptotic runtimes for the shortest path problem. One of them is planar graphs, where the following theorem (proven in [HKRS97]) holds.

**Theorem 3.5.** *On a planar graph $G = (V, E)$ shortest paths can be found in time $\mathcal{O}(|V| + |E|)$.*

Setting $T_{Dijkstra}(n, m) = n + m$ in the runtime analysis seen before leads to

$$\mathcal{O}(k \cdot (n + m) + k \cdot log(k))$$

which is still slower than the Connection Scan profile algorithms. Constant evaluation for profiles is still not possible as new tuples with arbitrary times are inserted after every Dijkstra search.

### 3.5.4 Transitively Closing Footpath Sets

Although the Connection Scan Algorithms can only be used for public transit networks with transitively closed footpath sets, one can take an arbitrary footpath set and close it by adding enough footpaths. This enables the usage of CSA afterwards. The number of footpaths which can be maximally added to the network is significant for the query-time.

**Theorem 3.6.** *For a graph $G = (V, E)$ the minimal graph $G' = (V, E')$ with $E \subseteq E'$ and $E'$ transitively closed, has $|E'| = \mathcal{O}(|E|^2)$ edges. The upper bound is minimal.*

*Proof.* A transitively closed component is a complete subgraph and the number of its edges only depends on the number of nodes. It therefore suffices to take a connected graph on $V$ with a minimal amount of edges and observe how many edges must be added to get a complete graph. A tree is a minimally connected graph, meaning that it is connected but removing any edge leads to a graph with more than one component. It is the graph with the least amount of edges that connects $V$ and has $|V| - 1$ edges, while the complete graph on $|V|$ nodes has $\mathcal{O}(\frac{|V| \cdot (|V|-1)}{2})$ edges. Therefore $|E'| = \mathcal{O}(|E|^2)$ and the upper bound is minimal as it is realized if $V$ is a tree. □

With the theorem one can conclude that the number of footpaths in the new network is $\mathcal{O}(k^2)$. For the profile Connection Scan Algorithm with constant evaluation of profiles the query then is $\mathcal{O}(k \cdot m)$.

## 3.6 Comparison of Algorithms for Arbitrary Footpaths

Three approaches were introduced to handle arbitrary footpaths. These are:

- Label-correcting Dijkstra
  Query-time: $\mathcal{O}((m \cdot k) \cdot (log(n) + k + m))$ for a Fibonacci-heap
- Profile CSA with interleaved Dijkstra
  Query-time: $\mathcal{O}(k \cdot (m + n \cdot log(n)) + k \cdot log(k))$ for a Fibonacci-heap
- Making the footpath-set transitively closed and using a profile CSA
  Query-time: $\mathcal{O}(k \cdot m)$ for pCSA-C

It seems that the last approach dominates the other two, which is rather surprising as the transitive closure seems to add a lot of footpaths. The reason why the runtime is not that overwhelming is the fact that the footpaths are replaced by pseudo-connections. They are only added if there are connections arriving and departing at the two stops the footpath connects. Even if there are $\mathcal{O}(m)$ footpaths at every stop of the transitively closed network, still only $\mathcal{O}(k \cdot m)$ connections can be added. However, a significant difference between the first two and the last is the fact that the transitive closure needs to fulfil the triangle-inequality, whereas the Dijkstra-based approaches work with arbitrary non-negative values for the footpaths.
The second approach is better than the first, especially when considering that the number of stops is not that big in comparison to connections and footpaths (at least in practice), which is less surprising as the Profile CSA with interleaved Dijkstra was made with arbitrary networks in mind.

# 4. Speeding Up One-To-One Queries

Algorithms in previous chapters always computed one-to-all or all-to-one queries. Although they can be used for one-to-one queries as well, the question remains whether algorithms with better asymptotic runtime exist for the one-to-one case and where natural boundaries for the runtime lie.
One such boundary is the output size of the problem. The chapter starts by discussing it and proceeds with an analysation of the size a profile can have in a one-to-one query without footpaths. Afterwards, an algorithm that does not compute a one-to-one query by simultaneously solving the problem for all stops will be presented for restricted graph classes.

## 4.1 Runtime Boundaries

The runtime of an algorithm can be bounded by the output size of the underlying problem. Here, two different kinds of queries exist: one-to-one queries and all-to-one queries. They will be discussed for networks with and without footpaths.

### 4.1.1 Boundary for Networks without Footpaths

One-to-one queries only output one profile containing a set of entries. These entries all correspond to a distinct connection arriving at the target stop of the problem instance, namely the one having the same arrival time as the entry:

**Lemma 4.1.** *For a profile $profile_s^t$ in a public transit network without footpaths the departure times of all tuples are departure times of connections leaving at $s$. The arrival times of tuples are arrival times of connections arriving at $t$.*

*Proof.* Every tuple of the profile represents a journey from $s$ to $t$. Those journeys have to start with a connection leaving at $s$ and one arriving at $t$. Therefore the lemma has to hold. □

Furthermore, there can not be two or more entries having the same departure or arrival time as they are held in a Pareto-set. This gives a boundary for the size of one profile in the network, which is also the output-size of an one-to-one query:

$$|profile_s^t| \leq min\{|\{c \in C \mid \pi_{dep}(c) = s\}|, |\{c \in C \mid \pi_{arr}(c) = t\}|\} \quad \text{for } s \neq t$$

and because all connections can leave at $s$ and arrive at $t$, the asymptotic size of a profile is $\mathcal{O}(k)$. The output-size of the all-to-one query, i.e. $\sum_{p \in P} |profile_p^t|$ can be bounded with the following observation: Every connection $c$ can only cause one profile entry with its according departure time at maximum and the entry has to be part of $profile^t_{\pi_{dep}(c)}$. It is a direct result of lemma 4.1. Thus,

$$\sum_{p \in P} |profile_p^t| \leq k$$

and the asymptotic output size of all-to-one queries is $\mathcal{O}(k)$.

### 4.1.2 Boundary for Networks with Footpaths

When looking at footpaths, one can differentiate between transitively closed and arbitrary footpath sets. A boundary for transitively closed ones can be given by the query-runtime of pCSA-C for these instances, as it only uses a transformed network instance to work on and still outputs all profiles. It runs in $\mathcal{O}(k \cdot \sqrt{m})$ and outputs all profiles of the network. There are networks realizing this value and therefore $\mathcal{O}(k \cdot \sqrt{m})$ is a minimal upper bound. Figure 4.1 shows such a network for 5 footpaths and 3 connections. The basic idea is to run

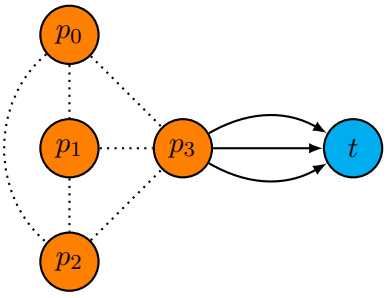



Figure 4.1: Network with maximal output-size for an all-to-one query. Different footpath-components have different colour.

all connections between two stops, in this case $p_3$ and $t$. $p_3$ is part of one large footpath component and therefore has $\mathcal{O}(\sqrt{m})$ neighbours. All the other stops in the footpath component have the profile of $p_3$, shifted by the duration of the footpath connecting them. The size of $profile^t_{p_3}$ is $\mathcal{O}(k)$ as outgoing connections fulfil the FIFO-property and are therefore Pareto optimal. In total, that makes a size of $\mathcal{O}(k \cdot \sqrt{m})$ for all profiles. Footpaths can be added to the example ba making the footpath component bigger and connections can be added between $p_3$ and $t$.

For arbitrary footpaths, a similar approach can be pursued. As footpaths do not have to be transitively closed any more, footpaths that run within the component of figure 4.1 and do not have $p_3$ as a node can be removed without changing the sizes of the profiles in the example. So, not only $\mathcal{O}(\sqrt{m})$ footpaths can be connected to $p_3$, but $\mathcal{O}(m)$. This makes a total output-size for all-to-one queries of $\mathcal{O}(k \cdot m)$.

### 4.1.3 Number of Different Profiles of Same Size

Proving lower bounds for algorithms turns out to be more complicated than upper bounds as one can not simply look at one algorithm and take its runtime as a lower bound. The argumentation has to include all algorithms that solve the problem and take the best of them. Another way to show lower bounds is by reducing an algorithm $A_1$ that is known to have a certain bound to the algorithm $A_2$ in question. This proves the lower bound of $A_1$ for $A_2$ as any faster one for $A_2$ induces a faster one for $A_1$ which contradicts with its

optimal runtime. One problem that is known to run in $\mathcal{O}(n \cdot log(n))$ is comparison-based sorting. It would be interesting to reduce a sorting instance to a profile query for a network without footpaths and $n$ connections to prove a lower bound for the profile Connection Scan Algorithm that does not use any preprocessing (excluding pCSA-C, which does more than sorting the array in the preprocessing), but solely works on the input and gets the connections sorted by departure time. To show why this approach fails, the number of possible profiles is calculated and compared to the number of permutations of a sequence. To limit the number of possible profiles lemma 4.1 can be used. It gives a discretization of the profile entries and makes it possible to place all tuples of a profile on a grid given by the departure and arrival times of connections as stated in the lemma. The value of the profile for some departure time $t_{dep}$ is

$$min(\{\tau_{arr}(c) \mid c \in C, \pi_{dep}(c) = s, \tau_{dep}(c) \geq t_{dep}\})$$

Intuitively, it is the value of the profile for the next point right to the queried departure time. As we are interested in profiles, which are Pareto optimal sets of tuples, not all placements of points on the grid represent a valid profile. In particular, if for two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, the values fulfil $x_2 \geq x_1$ and $y_2 \leq y_1$, then $p_1$ is dominated and the set is not optimal. This gives the restriction that only one point can exist per row and column. Otherwise they can be interpreted as $p_1$ and $p_2$ in the previous argumentation and one is dominated.

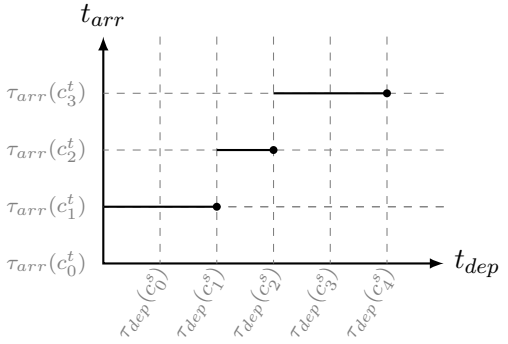



Figure 4.2: Profile with departure and arrival times on a grid

Figure 4.2 shows an example of a profile with departure and arrival times as an underlying grid where points can be placed on. The connections $\{c_0^s, \ldots, c_4^s\}$ are departing at $s$, $\{c_0^t, \ldots, c_3^t\}$ arrive at $t$. The number of departing and arriving connections is bounded by $k$. If the points could be placed freely there would be $\mathcal{O}(k^k)$ different profiles. In the following, the number $\eta : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}$ of different grids of size $a \times b$ with the restriction that the according profile is valid, will be calculated.
The case of $a = b = 0$ is special and $\eta(0, 0) = 1$ as there is only one profile with that property. The same applies to $\eta(1, 0)$, $\eta(0, 1)$ and $\eta(1, 1)$. All $2 \times 2$ grids are depicted in figure 4.3. There are six of them.

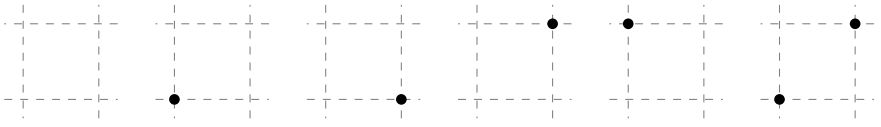


Figure 4.3: All possibilities for the $2 \times 2$ grid

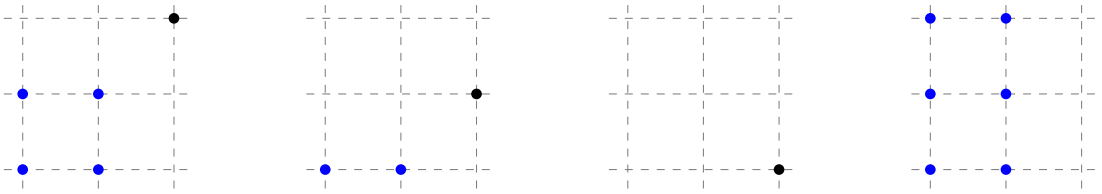


Figure 4.4: A $3 \times 3$-grid with the black dot positioned in the last column. Blue dots show the remaining grid that can be used.

**Lemma 4.2.** *The follwing recurrence relation holds for the number $\eta$ of valid $a \times b$-grids:*

$$\eta(a,b) = \sum_{i=0}^{b} \eta(a-1,i), \qquad \eta(0,0) = 1$$

*Proof.* A recurrence relation for $\eta$ can be found by constructing the grids from right to left and placing one dot (or none) in the last column. Placing more than one dot in a column would violate the Pareto-property. Depending on which row the dot is placed on the remaining grid that can be freely used is restricted more or less. Figure 4.4 shows an example of a $3 \times 3$-grid and all possible placements of the black dot in the last column. The blue dots show the remaining grid where dots can be independently placed. Those are all points that are to the left and under the black dot. Thus the size of the remaining grid is $(a-1) \times (i-1)$, if the dot is placed in row $i$ and the original grid had size $a \times b$. If no dot is placed the grid is of size $(a-1) \times b$. Using the addition principle the number of valid grids is the sum over all possibilities to place points on the remaining grid for every placement in the last column. This gives the recurrence relation $\eta(a,b) = \sum_{i=0}^{b} \eta(a-1,i)$. The initial value $\eta(0,0)$ is one, as there are is only one possibility to place dots on an empty grid.

□

**Theorem 4.3.** *The recurrence for $\eta$ is solved by $\eta(a,b) = \binom{a+b}{a} = \binom{a+b}{b}$.*

*Proof.* Proving $\binom{a+b}{b} = \sum_{i=0}^{b} \eta(a-1,i) = \sum_{i=0}^{b} \binom{a-1+i}{i}$ can be done with the help of Pascal's Formula. The initial value is correct as $\eta(0,0) = \binom{0}{0} = 1$. Now use that $\binom{n+1}{k+1} = \binom{n}{k+1} + \binom{n}{k}$:

$$\binom{a+b}{b} = \binom{a-1+b}{b} + \binom{a-1+b}{b-1} = \binom{a-1+b}{b} + \binom{a-2+b}{b-1} + \binom{a-2+b}{b-2} = \dots$$

The last term can replaced repeatedly using Pascal's Formula until the whole sum is written out and therefore the claim holds. □

With this knowledge it is easy to get the number of profiles for a network with $k$ connections. It is $\eta(k,k) = \binom{2k}{k} = \frac{(2k)!}{(k!)^2}$ and called the central binomial coefficient. It has some interesting properties like the following boundaries (for $n \geq 1$):

$$\frac{1}{2}\frac{4^n}{\sqrt{\pi n}} < \binom{2n}{n} < \frac{4^n}{\sqrt{\pi n}}$$

This implies that $\eta(n,n) \in \Theta(\frac{4^n}{\sqrt{n}})$ and therefore $\eta(n,n) \in o(n!)$. It means that there are asymptotically fewer possibilities for different profiles than there are for permutations of size $n$. This explains why the reduction of the sorting problem on the profile query is difficult. The sorting algorithm has to distinguish between $n!$ cases, while the profile query only distinguishes between $\binom{2n}{n}$. The reduction itself therefore has to do some computation

to distinguish between more cases.
Another approach for proving this bound that is closely related is to look at decision trees. For comparison-based problems the runtime can be bounded by the minimal height of a decision tree. In such a tree every profile that can possibly be gained is represented by a leaf. The section before showed that the number of those cases is $\Theta(\frac{4^k}{\sqrt{k}})$. The binary tree has a minimal height for a fixed number of elements only if it is balanced. This means that the last level contains $\Theta(\frac{4^k}{\sqrt{k}})$ nodes and therefore the number of nodes in the entire tree is $\Theta(\frac{4^k}{\sqrt{k}})$. With this knowledge the height is $log_2(\frac{4^k}{\sqrt{k}}) = 2k - log_2(\sqrt{k}) \in \Theta(k)$ and if there exists an algorithm that balances the decision tree it would have linear runtime. To prove the lower bound of $\mathcal{O}(k \cdot log(k))$ for the algorithm one would have to show that this is not possible.

## 4.2 Algorithm for Multipaths

Another special graph class are multipath-chains with connections and footpaths only between following stops on the chain. They can be formally described as a stop set $P = \{p_0, p_1, \ldots, p_{n-1}\}$ with connections $c \in C$ fulfilling $\pi_{dep}(c) = p_i$ and $\pi_{arr}(c) = p_{i+1}$. The footpath set $F$ must be a subset of $\{\{p_i, p_{i+1}\} \mid p_i, p_{i+1} \in P\}$. Figure 4.5 shows an example.

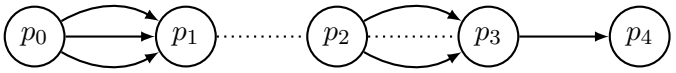



Figure 4.5: An example for a multipath

In contrast to general problem classes for these multipaths the profiles can be computed per stop from $t$ to $s$ as the profile of $p_i$ is dependent on the one of $p_{i+1}$. This means that one can bundle all outgoing connections of a stop and check for the earliest reachable connection at the next for each of the outgoing connections together via a sweep-algorithm. It will be discussed after some observations.
The size of $profile^t_{p_0}$ is bounded by the number of connections arriving at the target stop and by the same argumentation by any number of connections between adjacent stops on the path (because again every entry must take a distinct connection between adjacent stops or it would have the same minimal arrival time as another one). Summarizing:

$$|profile^j_i| \leq min_{k \in \{0,\ldots,n-1\}}\{|S_k|+1 : S_k = \{c \in C \mid i \leq k < j, \pi_{dep}(c) = p_k \wedge \pi_{arr}(c) = p_{k+1}\}\}$$

The maximal size of a single profile is $\mathcal{O}(k)$ as all connections can run between two stops.
An example for a worst case regarding the output size for all-to-one queries is given in

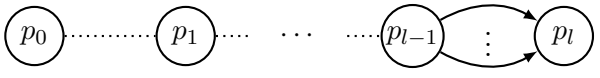



Figure 4.6: Worst case for all-to-one queries

figure 4.6. All connections run from $p_{l-1}$ to $p_l$. This means that $profile^{p_l}_{p_{l-1}}$ has $k$ entries and propagates them backwards through the network to every other stop. As a profile can have at most $k$ entries, this construction leads to every stop having a profile of maximum size and the output size is $\mathcal{O}(k \cdot n)$. This means that any algorithm computing all-to-one queries has runtime at least $\mathcal{O}(k \cdot n)$. In the following, an algorithm will be presented that solves those queries. It uses the linking-operation and is shown in algorithm 4.1.

**Algorithm 4.1:** CALCULATING $profile_0^{n-1}$

```
1 for i ← 2 to n − 1 do
2     profile_0^i ← profile_0^{i−1} ⊕ profile_{i−1}^i
```

The profiles between consecutive stops $i$ and $i+1$ can be obtained by taking the outgoing connections of $i$ and set the walking duration to the duration of the footpath between $i$ and $i+1$.

The runtime lies in $\mathcal{O}(k \cdot n)$ as it needs linear time in the number of profile entries at each stop because every operation needed for $\oplus$ runs in linear time in the number of profile entries involved. The worst case for the number of those entries is $k$.

### 4.2.1 Recursive Algorithm

The previously discussed algorithm still computes profiles for all stops on the path between the source and target node and therefore can not break the boundary of $\mathcal{O}(k \cdot n)$. To get a better runtime, the algorithm must not compute all profiles. Based on this observation algorithm 4.2 is introduced, which does not compute all profiles.
The algorithm divides a multipath into two paths only having one stop in common or computes the profile directly if there are two stops or less left.

**Algorithm 4.2:** $profile(i, j)$

**Input:** Multipath network, Indices $i$ and $j$ with $i < j$
**Output:** $profile_{p_i}^{p_j}$

```
1 if j − i > 1 then
      // At least one stop in between, split recursively
2     profile_{p_i}^{p_j} ← profile(i, ⌊(j−i)/2⌋) ⊕ profile(⌊(j−i)/2⌋, j)
3 else if j − i = 1 then
      // Only direct connections and footpaths between stops
4     profile_{p_i}^{p_j} ← outgoing connection-set of p_i and footpath
5 else
      // Single stop, profile is the identity function
6     profile_{p_i}^{p_j} ← id
```

Figure 4.7 shows it schematically splitting the path up into smaller ones until they are small enough to work on directly. The profiles are then combined to get $profile_{p_0}^{p_4}$.

### 4.2.2 Runtime

The runtime for the algorithm can be partitioned into recursion levels, describing the call-depth of the executed operations. Figure 4.7 depicts them. On layer 0 two profiles are linked, leading to two calls on layer 1. In general every call of the function leads to two calls on the next layer. The number of calls on each layer therefore increases exponentially, meaning that there are $2^i$ calls on layer $i$ and there are $\lceil log_2(n) \rceil$ layers.
The runtime of the algorithm depends on the effort to link the profiles on all layers except for the last one. Computing the profiles of successive stops on the last layer is not problematic, as the outgoing connections are already sorted and just have to be written into a new profile. The size of a profile is limited by the number of connections on the sub-path it covers. This also means that every connection causes at most one entry of a profile in each

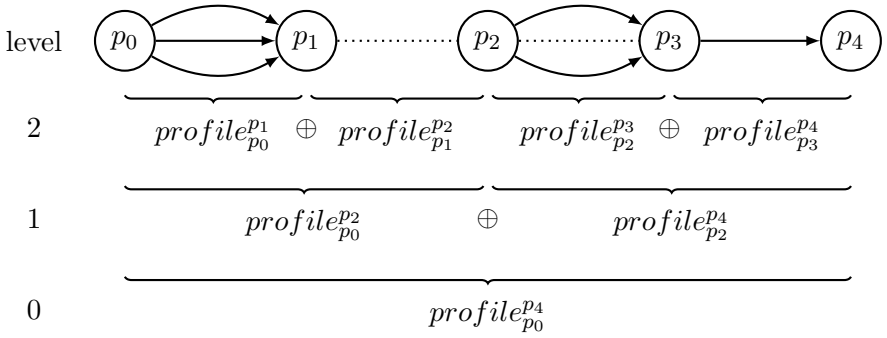



Figure 4.7: Depiction of the recursive profile calculation. The levels show the call-depth of the operation.

layer. The asymptotic size of all profiles on one layer is $\mathcal{O}(k)$. Every profile is linked with at most one other profile, meaning that the profile size has a linear impact on the runtime and because the number of entries is limited by the connections the total time needed on one level is $\mathcal{O}(k)$. There are $O(log(n))$ layers, so the total runtime for the handling of connections is $O(k \cdot log(n))$. In addition to the connections, footpaths have to be taken into consideration, too. They are represented by the walking duration of the profile, therefore only existing once per profile. The total number of profiles is $\sum_{i=0}^{\lceil log_2(n) \rceil} 2^i \in \mathcal{O}(n)$, so the stops have a linear part in the runtime equation. Together with the connections that makes $O(k \cdot log(n) + n)$

This assumes that the connections are sorted in the beginning so the generation of the profiles on the last layer takes linear time. If one drops this requirement the connections have to be sorted by the algorithm. This takes at least $\mathcal{O}(k \cdot log(k))$, raising the whole runtime to this value plus the number of footpaths as $n \in \mathcal{O}(k)$ for connected multipaths (paths can be checked for connectivity at first and the computation could be stopped earlier). In this case the runtime is indeed minimal since the footpaths have to be computed at least once and every sorting problem of size $n$ can be solved with the solution of a multipath instance having $n$ connections in linear time. This instance only has two stops which are connected by one connection for every number to sort. The network is defined by the following sets, assuming that the set $S$ contains the numbers to sort.

$$P = \{p_0, p_1\}, E = \emptyset,$$

$$C = \{c \mid \tau_{dep}(c) \in S, \tau_{arr}(c) = \tau_{dep}(c) + 1, \pi_{dep}(c) = p_0, \pi_{arr}(c) = p_1\}$$

For this network $profile_{p_0}^{p_1}$ contains an entry for every connection since there are no dominated ones. Furthermore, those entries are sorted by departure time, which correspond to elements of $S$. The sorting instances are somehow simplified as they can not contain duplicates but this restriction can be lifted and handled in linear time by first locating those and inserting them again after the profile returns the sorting of all distinct ones. This implies that an algorithm for multipaths that does not get the connections ordered can not be faster than $\mathcal{O}(k \cdot log(k))$ and the one presented here is optimal.

### 4.2.3 Recursive Algorithm for Trees

The recursive algorithm 4.2 only works for paths, but can be extended to trees quite easily by first searching for the unique multipath connecting the two stops and letting the algorithm run on this instance. To find the multipath, a breadth-first search from the departure stop is sufficient. It finds a path to the target node in $\mathcal{O}(n + k + m)$. On this directed path every footpath, stop and connection pointing towards the target node (meaning it leads from a stop closer to the departure node to one closer to the target node

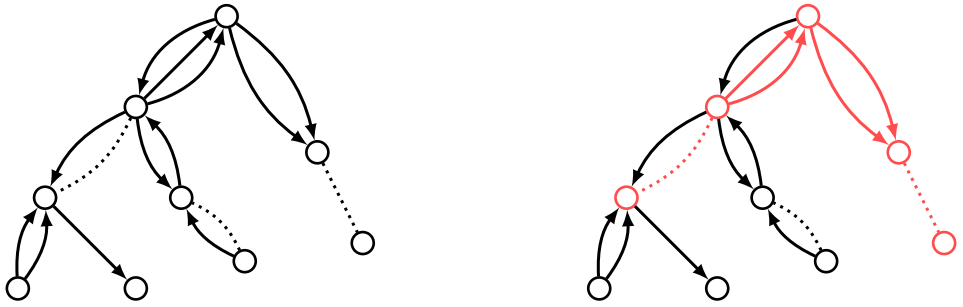


Figure 4.8: Finding a multipath between two stops in a tree

on the path) can be taken and added to the multipath. Figure 4.8 shows an example of such a network where the source node is to the left and the target node to the right. All elements that are marked in red are part of the resulting multipath. The runtime for the procedure is mostly still dominated by the recursive algorithm and in $\mathcal{O}(k \cdot log(n) + n + m)$.

## 4.3 Using Graph-Separators

The algorithm for multipaths is restricted to a very special graph-class. In general, not all shortest journeys have to be on one multipath where a stop $p$ can be taken and it suffices to join $profile_s^p \oplus profile_p^t$. Instead, it is possible to take a set of stops which separate the graph and use them to compute profiles. This leads to the following algorithm.

### 4.3.1 Algorithm Using Separators

**Algorithm 4.3:** $computeProfiles(G, P_1, P_2)$

**Input:** Transit network $G$, Stop sets $P_1$ and $P_2$
**Output:** $profile_a^b$ for all $a \in P_1, b \in P_2$

```
1  if G is small enough or no suitable separator exists then
2      calculate profile_{p1}^{p2} for all p1 ∈ P1, p2 ∈ P2 in network induced by G
3  else
4      S ← special separator splitting G into S1, S2 and P1 ⊆ S1, P2 ⊆ S2
5      calculateProfiles(S ∪ S2, S, P2)
6      calculateProfiles(S ∪ S1, P1, S)
7      for every pair (p1, p2) with p1 ∈ P1, p2 ∈ P2 do
8          initialize profile_{p1}^{p2} with walkingdistance = ∞ and no entries
9          for s ∈ S do
10             merge profile_{p1}^{p2} with profile_{p1}^{s} ⊕ profile_{s}^{p2}
```

The algorithm is formulated to take two sets and compute all pairwise profiles between elements of them. To get the result for a one-to-one query the sets can be chosen accordingly as $P_1 = \{s\}$ and $P_2 = \{t\}$. It takes a separator $S$ between $P_1$ and $P_2$. $S$ then has the property that any journey from an element in $P_1$ to one in $P_2$ has to pass one of the separators stops. Therefore the algorithm can recurse on both separated components each adding the separator itself and join them by considering all possible combinations of profiles through it.

Figure 4.9 shows the networks $S \cup S_1$ and $S \cup S_2$ the two recursive sub-searches are working on. Since $S_1$ and $S_2$ are the components separated by $S$ they are per definition disjunct.

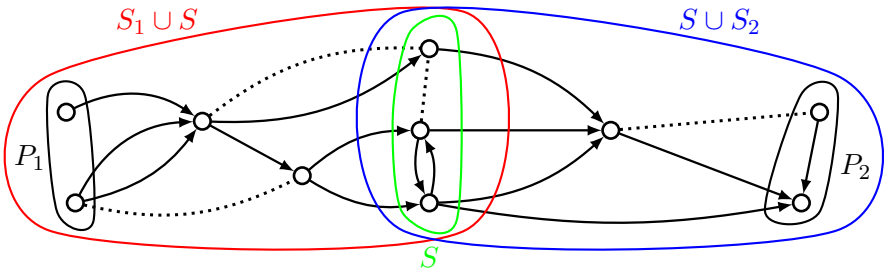



Figure 4.9: Network that is split by the separator $S$, leading to two components

The only part where the two networks overlap is the separator itself. Profiles are computed for all elements in $P_1$ to all in $S$ and for all in $S$ to all in $P_2$. As all journeys from $P_1$ to $P_2$ have to pass the separator the results can be joined to get $profile^{p_2}_{p_1}$ for fixed $p_1 \in P_1, p_2 \in P_2$ by taking every stop $s \in S$ and first linking $profile^{s}_{p_1}$ and $profile^{p_2}_{s}$ (the two profiles were computed in the two recursive calls before) and then merge all of them. One linked profile is only optimal for journeys that lead through the stop it is linked on and if the journey can be partitioned so that the first part only runs in $S_1 \cup S$ and the second part in $S_2 \cup S$. In this case $profile^{s}_{p_1}$ covers the first part of the journey and $profile^{p_2}_{s}$ the second. The linking thus yields the wanted result. The optimal profile can contain journeys through all of the stops of $S$, thus they have to merged into a Pareto-set. The only case where the algorithm does not cover all possibilities occurs when there is a shortest journey that enters the separator, leaves it in one direction and enters the separator again.

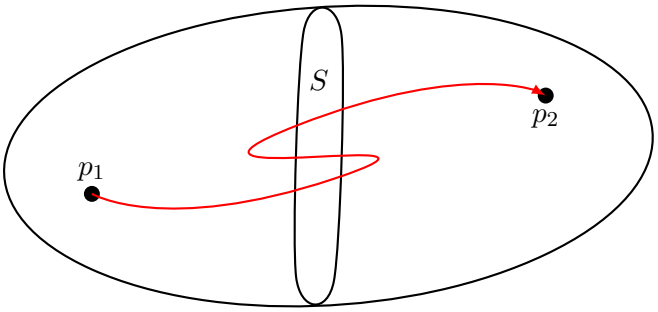



Figure 4.10: Shortest journey leaving the separator multiple times

This is depicted in figure 4.10. Here, simple joins on all stops of the separator may not be enough as the path can not be split on any stop so that one half only runs through stops in the first component and the separator and the second in the other one including the separator. It has to be avoided and can be done by forbidding separators that have connections leaving the separator in both directions, meaning that connections and footpaths only enter $S$ from $S_1$ and only leave to stops of $S_2$. Footpaths are only allowed to run from $S$ to one of the separated components.

In the network of figure 4.9 the algorithm recurses only once, although it would be possible to execute $computeProfiles(S \cup S_1, P_1, S)$ to get the profiles for elements on the left. If the query is not handled recursively other ways for the many-to-many queries have to found. This could be as simple as to use brute force or a Dijkstra based approach for footpath sets which are not transitively closed.

### 4.3.2 Complexity

To get any reasonable result for the algorithm's time complexity some assumptions have to be made. One of them is the size of the separator, which will be bounded by being at most the constant $c$. It should be noted that this is not possible for all graphs. Another one is the size of $P_1$ and $P_2$ for the initial query. Bounding them to be at most $c$ makes the analysis easier as the two sets are then bounded for every call of the algorithm. In the case of one-to-one queries the sets would only have one element, showing that this assumption may not be too far fetched. As the separators dissect the network based on the connectivity of stops the connections and footpaths could still be distributed unevenly to the separated components, especially because the network can contain multiple connections between two stops. Assuming that the distribution is always even (in regards to stops and footpaths and connections) makes the analysis easier and excludes cases where one component of the network contains the majority of connections but can not be split any further as it is already too small in terms of stops. The algorithm heavily profits from dividing the problem instances into smaller parts and the connections are one big factor for the size of an instance.
More general, the following recursive formula for the runtime $T$ holds:

$$T(G, P_1, P_2) = T(S \cup S_1, P_1, S) + T(S \cup S_2, S, P_2) + |P_1| \cdot |P_2| \cdot |S| \cdot k$$

The first two summands represent the recursive calls, the last one is the time it takes to join all paths on the separator. The equivalence is asymptotic. The number of $(p_1, p_2)$-pairs is $|P_1| \cdot |P_2|$ and for each pair all stops of the separator of size $|S|$ lead to two profiles being linked and merged. Linking and merging can be done in $\mathcal{O}(k)$ as shown before. The separators are meant to be part of the input or retrievable in constant time.
A drawback of this formula is that it is hard to estimate the size of the separated components and the recursion depth of the algorithm. The problem of separators not having an empty cut with $P_1$ and $P_2$ and the handling of this case does not lead to a higher runtime as many-to-many queries between the sets do not take more time than $\mathcal{O}(|P_1| \cdot |P_2| \cdot |S| \cdot k)$. A rough estimate for the runtime can be given with the restrictions in mind and some observations of the separators' build-up.

Figure 4.11 shows a network that is partitioned by multiple separators. Underneath, a tree is depicted representing components of the network. $C_1$ is the entire network and is split evenly into two components in one step of the recursion. Those components are $C_2$ and $C_3$ and only have $S_1$ in common. They are split the same way into more subcomponents. The area below a separator where two components overlap is the separator itself. Every layer of the tree from top to bottom represents the recursion depth in which the component is handled and they are always drawn beneath the separator they are split by. Every layer $i$ contains the whole network, split into $2^i$ parts. Each of the components joins the components below it which are connected via and edge on the separator they have in common. To get the number of steps needed for the join it is important how big the profiles are. As seen previously they are bounded by the number of connections of the network they are contained in. The size of those networks is different on every layer/recursion-depth of the algorithm. On layer 0 all connections of the original network can have an impact when joining on $S_1$ while on layer 2 only a fourth of the connections are joined on each of the separators. Because of the restrictions that the network is always split evenly, a layer contains $2^i$ separators on layer $i$ and the components on one layer build the whole network and not more (in fact the union of them is not disjoint as separators are part of multiple components, but this does not play a role in the asymptotical analysis) a join of a component on layer $i$ deals with $\frac{k}{2^i}$ connections and needs a time of $\mathcal{O}(|P_1| \cdot |P_2| \cdot |S| \cdot \frac{k}{2^i}) = \mathcal{O}(c^3 \cdot \frac{k}{2^i})$. The time needed for one layer can then be described as $\mathcal{O}(2^i \cdot c^3 \cdot \frac{k}{2^i}) = \mathcal{O}(c^3 \cdot k)$. Note that a join is only executed on layer $0 - 2$ where subcomponents have to be joined. On the

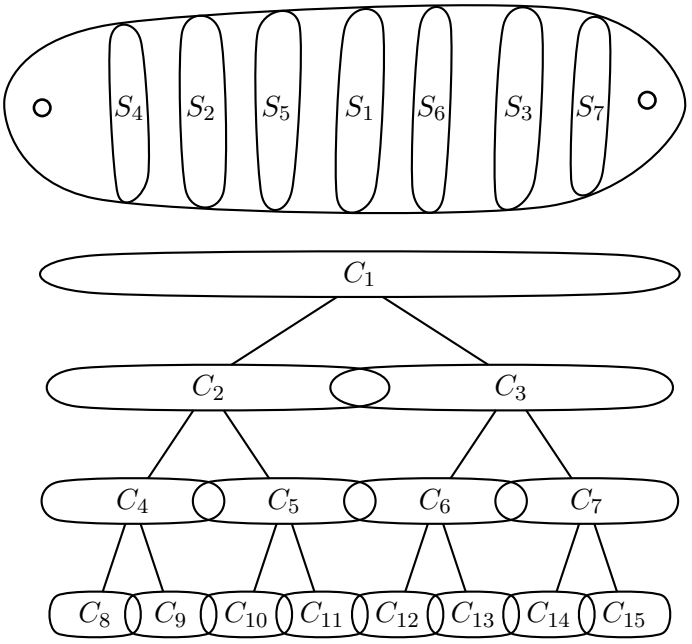



Figure 4.11: Network recursively split by the separators $S_1$ to $S_7$, resulting in the components depicted below.

last layer the algorithm does not split the network any further and executes the many-to-many queries instead. Because of the even splits of the network the tree representing the components is balanced and therefore has a height of $log(n)$. This makes a runtime of

$$\mathcal{O}(c^3 \cdot k \cdot log(n) + q),$$

where $q$ is the time needed for the many-to-many queries between separators. This is heavily dependend on the algorithm used for it though one can reason that every separator is used as a target and a source for the queries on the last layer. The number of separators is $\mathcal{O}(\frac{n}{c})$, so there are about that many queries on the network.

To get to this runtime it has taken a lot of restrictive prerequisites to the network. Those were:

- Between any two stop-sets of size $c$ there is a separator of size $c$ partitioning the network
- Shortest journeys can not leave the separator on both sides.
- The connections and footpaths are distributed nicely among the separated components
- The recursion tree is balanced in the sense that it has logarithmic height

Therefore practical feasibility is severely narrowed down. However, the goal of this section was not only to show a new approach that does not compute all profiles of the network, but to show where the problems of the approach lie.

# 5. Conclusion

Public transit networks do not behave like weighted graphs. The discrete structure given by departures and arrivals of connections call for special treatment and new algorithms. Modelling the network as a graph with different models is not only slower in practice, but also falls behind the Connection Scan Algorithms in terms of asymptotic runtime, which was to be expected. The need for a priority queue and the processing of nodes in no fixed order do not even have to be factored in to get this result. The Connection Scan Algorithms and in particular the ones for profile-queries are better suited for the task, but struggle when too much footpaths and even worse, footpaths not fulfilling the triangle inequality, are part of the network. Here, the combination of the two approaches where Dijkstra's algorithm handles footpaths and the Connection Scan Algorithm deals with connections seems to take best of the two worlds and combines them appropriately. As it turns out, this procedure does not have the edge over a plain profile CSA where footpaths are replaced by pseudo-connections. At least this is the case if the triangle inequality holds for the footpaths of the public transit network. This result is rather surprising and can be explained by the relatively small amount of footpaths that have to be added as only ones important for the reachability of successive connections are integrated.

The thesis gives information about the asymptotic complexity of algorithms used for profile queries in public transit networks. The analysis of the Connection Scan Algorithms shows where the runtimes can be ranked and includes worst-case scenarios for them.

The acceleration technique investigated showed that no improvement in runtime has to be made if bad instances are chosen. This is in line with the experience of route planning algorithms for road networks. The acceleration builds on inherent properties of the network and heuristically uses them to get better results.

While some algorithms could be proven to be optimal by regarding their output-size, others turned out to be more difficult to analyse. One nice result that was proven on the way is that the number of different profiles of size $n$ is $\mathcal{O}(\binom{2n}{n})$.

One major boundary for the runtime of profile algorithms was identified: Computing all profiles even if one is only interested in a one-to-one query, limits the runtime. A new approach was tested that uses separators to counter that problem and was successful for very restricted graph classes. The problems for a wider usage were pointed out, although it is not clear how they can be solved.

This is one of the aspects of future work in this field. Another starting point is the further testing of feasibility of computing transitive closures for profile queries in practice. There is some work to be done, especially towards more multi-criteria settings, but it seems that there is no intrinsic unfeasibility that makes it impossible.

# Bibliography


[BCE+10] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. *Fast Routing in Very Large Public Transportation Networks Using Transfer Patterns*, pages 290–301. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[BDG+15] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks, 2015.

[BDGMH09] 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.

[BGMH10] Annabell Berger, Martin Grimmer, and Matthias Müller-Hannemann. *Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks*, pages 35–46. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[BHS16] Hannah Bast, Matthias Hertel, and Sabine Storandt. Scalable transfer patterns. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 15–29. SIAM, 2016.

[BS14] Hannah Bast and Sabine Storandt. Frequency-based search for public transit. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 13–22. ACM, 2014.

[DDP+13] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. *Computing Multimodal Journeys in Practice*, pages 260–271. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[DGPW17] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, dec 1959.

[DMHS08] Yann Disser, Matthias Muller-Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. *Lecture Notes in Computer Science*, 5038:347–362, 2008.

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

[DPSW17] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection scan algorithm. *arXiv preprint arXiv:1703.05997*, 2017.

[DPW12] Daniel Delling, Thomas Pajor, and Renato Werneck. Round-based public transit routing. Society for Industrial and Applied Mathematics, January 2012.

[Fed87] Greg N. Federickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, dec 1987.

[HKRS97] Monika R Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *journal of computer and system sciences*, 55(1):3–23, 1997.

[Paj09] Thomas Pajor. Multi-modal route planning. *Universität Karlsruhe*, 2009.

[Paj13] Thomas Pajor. *Algorithm Engineering for Realistic Journey Planning in Transportation Networks*. PhD thesis, Karlsruhe, Karlsruher Institut für Technologie (KIT), Diss., 2013, 2013.

[PSWZ08] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *Journal of Experimental Algorithmics*, 12:1, jun 2008.

[SW14] Ben Strasser and Dorothea Wagner. Connection scan accelerated. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 125–137. SIAM, 2014.

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

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