Efficient Enumeration of All Reasonable Journeys in Public Transport Networks

Diploma Thesis of

David Weiß

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Dipl.-Inf. Ben Strasser
	M.Sc. Tobias Zündorf

Time Period: March 16th, 2015 – September 13th, 2015

Statement of Authorship

Karlsruhe, 14th September 2015

Abstract

Recent research on route planning in public transport networks mostly focused on customer perspective of answering individual queries for specific journeys in realtime. We shift this focus towards the viewpoint of traffic planners, who need to enumerate all reasonable journeys throughout the network over a given timespan at once, to e.g. compute demand data down to specific lines and vehicles. While realtime is not an issue, faster algorithms that incorporate realistic footpath models, are required to make all-to-all journey enumerations feasible for large networks. Also, the extraction of concrete journey representations, often regarded as trivial, becomes a substantial and time consuming part.

We present an efficient algorithm for searching and enumerating all journeys that are optimal in the Pareto sense in regard to travel time and number of transfers between all pairs of destinations as specific sets of train connections. The algorithm is capable of handling realistic transfer models as well as reasonable footpath networks.

For our real world test data incorporating 1,154 distinct destinations, we manage to enumerate 401,521,153 optimal journeys, averaging 302 journeys per destination pair, in less than 43 minutes or about 6 microseconds per journey. Exploiting the embarrassingly parallelizable structure of our algorithm, concurrent computation using 8 threads with a speed up of 5.5 brings down computation time below 8 minutes.

Deutsche Zusammenfassung

Bei aktuellen Forschungsarbeiten über Routenplanung in Öffentlichen Verkehrsnetzen liegt der Fokus zumeist auf Anfragen durch Fahrgäste an ein Fahrplanauskunftssystem, die in Echtzeit beantwortet werden sollen. Wir verschieben die Sichtweise auf den Standpunkt von Verkehrsplanern, die nicht nur eine, sondern alle sinnvollen Reisen über eine gewisse Zeitspanne in einem Netz auflisten wollen, um beispielsweise Nachfragedaten auf einzelne Linien oder Fahrzeuge umzulegen. Echtzeitanfragen spielen hier keine Rolle, trotzdem sind schnelle Algorithmen, die auch realistische Fußwegnetze einberechnen, wichtig, um solche umfangreichen Abfragen auf großen Netzen durchzuführen. Durch die Menge wird die Extraktion einer konkreten Reise mit allen Abschnitten ein gewichtiger und zeitintensiver Schritt des Algorithmus, der bisher oft als trivial angesehen und nicht näher betrachtet wurde.

Wir beschreiben einen effizienten Algorithmus, der alle nach Fahrzeit oder Umsteigehäufigkeit optimalen Reisen zwischen allen Quell-Ziel-Paaren in einem Netz als konkrete Folge von Zugverbindungen auflistet. Dabei werden realistische Umstiegsmodelle und Fußwegnetze berücksichtigt.

Für unser reales Testnetz mit 1.154 Zielen extrahiert der Algorithmus 401.521.153 Reisen, durchschnittlich ca. 302 pro Quell-Ziel-Paar, in weniger als 43 Minuten, was ca. 6 Mikrosekunden pro Reise entspricht. Der Algorithmus ist außerdem trivial parallelisierbar und erreicht im Testszenario mit acht Threads einen Speedup von 5,5, was die Laufzeit auf weniger als 8 Minuten reduziert.

Contents

1. Introduction

When looking on public transport networks, there are two natural viewpoints: the customer, i.e. passenger on the one side uses the network to travel between distinct locations. He likely seeks to find the optimal journey to reach his destination with respect to criteria like travel time, number of transfers or price. On the other hand, the service provider, including traffic planning staff, is not interested of a single journey, but the performance of the whole network under his administration. To make informed statements and decisions about the performance of the network and to be able to identify and plan further improvements or expansions, reliable usage data is needed.

Such data in the form of a demand model is expressed as a square matrix and a scoring function. The matrix defines the number of passengers traveling between any two destinations in the network in a given timespan, e.g. during a workday. The scoring function gives a statistical model of which journey the passengers prefer, if they have a choice between multiple journeys to reach their destination.

Mapping this data onto a list of journeys in the network, thus assigning a concrete number of passengers to each one, is called *transit assignment*. This mapping then can be further broken down into load factors of single vehicles, and can be used to e.g. identify bottlenecks.

Enumerating all *possible* journeys is a combinatorial problem of exponential complexity, and as such not feasible. Of course it is not necessary to enumerate all journeys, but only those that matter for the assignment. Acquiring a *reasonable list of journeys* for the assignment is a non-trivial task and the topic of this thesis.

An exemplary implementation of the full demand assignment toolchain is given in PTV Visum [1], a commercial transport modeling software. We use Visum as a reference for the performance of our algorithm and to evaluate the results in regard to assignment quality. Our goal is to minimize the search space to acquire reasonable speed while not dropping too much journeys that would have passengers assigned.

1.1 Problem definition

We define the *reasonable journeys enumeration problem* as follows. Given a public transport network and a fixed query time interval, enumerate all journeys for any source-target pair, that are optimal in the Pareto sense for departure time, arrival time and the number of

[1] `http://vision-traffic.ptvgroup.com/en-uk/products/ptv-visum/`

transfers. Our model of the transport network is defined in detail in section 2.2. A formal definition for a valid and optimal journey is given in section 2.3.

Limiting the search to time and transfer Pareto optimal journeys is reasonable in our view, because these are two of the most important criteria for customers and thus for demand models as well. Incorporating fares as a third, very important criterion is not covered in detail by this work, but we give some hints for possible extensions in section 3.5.

1.2 Related work

Until the introduction of the timetable based algorithms RAPTOR [DPW12] and CSA [DPSW13], research has focused on two approaches to the journey search problem in public transport networks: modeling them as shortest path problem in an either time-dependent or time-expanded graph [DMS08], [BSS13]. A lot of work has been done in adapting and evaluating graph-based algorithms and speed-up techniques from the better known world of road vehicle routing, but without as much success, as an overview study shows [MHSWZ07]. RAPTOR and CSA shifted the focus from graph traversal to simple table-based scanning algorithms, proving that using algorithm engineering to tap the full potential of modern computer architectures may result in better performance even if the algorithmic complexity suggests otherwise.

CSA has been augmented to advanced issues like multi-criteria profile queries and delay robustness [DPSW13]. The latter introduces decision graphs, which are used to enumerate alternative journeys in case of delays, but does still not expand the journeys themselves.

Some work has focused on answering large numbers of individual queries very fast for usage on web services, making use of preprocessing techniques [Gei10]. Our approach differs in that it does not try to optimize for individual queries but for answering all of them at once. Also, as all of these publications focused on the search algorithm itself, none of them looked into efficiently enumerating concrete journeys from the search results. We found that for our all-to-all scenario, enumerating the journeys themselves dominates the search in respect of both computation time and memory consumption, thus yielding the most potential for optimization.

The algorithm described in [DKP12] comes closest to our work. It solves all-to-all queries for fastest journeys in a whole network in parallel, using an Dijkstra-based one-to-all approach as base algorithm. Unlike their solution, our algorithm not only optimizes for travel time but for number of transfers as well. Also, they use only a very basic footpath model for local interconnections, where we support full networks.

Traffic assignment itself has been researched by Friedrich et al. [FHW01], where they introduced a branch-and-bound algorithm to solve the multi-criteria journey search problem for an arbitrary number of criteria. While being highly customizable and quite accurate, in practice this algorithm takes a lot of computation time for reasonably sized networks. It also is the main algorithm used in PTV Visum. Our approach is restricted to travel time and number of transfers as criteria, though emits a reasonably large number of journeys and takes much less computation time.

1.3 Contribution

We first adapt the multi-criteria profile connection scan algorithm (mcp-CSA) from [DPSW13] to our network model, adding fully dynamic handling of different footpath modalities. Smaller modifications are made to improve journey extraction speed later on.

While for one-to-one searches journey extraction is not much of an issue, it becomes the dominating one for our all-to-all approach. Our main contribution is a depth-first search algorithm to extract all Pareto optimal journeys in the network, exploiting the profile information computed with the connection scan algorithm to minimize the amount of visited branches. In fact, our algorithm never visits a dead end.

Both, connection scan and our new journey extraction algorithm work as all-to-one search algorithms. Running an all-to-all search is as simple as running independent searches for all destinations.

At last, this makes the whole approach embarrassingly parallelizable. We evaluate the speedup of using multi-core machines in section 4.3.

1.4 Outline

In the next chapter, we define the basic terms and our data model in detail. In chapter 3 we describe the underlying algorithms. We begin with an overview of the bicriteria profile connection scan algorithm and describe our modifications to handle our network model. After that, we describe the new journey extraction algorithm in detail. Also, we briefly discuss modifications for speedup and extensions to different queries. In chapter 4 we present the results of the performance tests we ran to evaluate the whole algorithm. At the end of the work, we summarize our results and give a short outlook to future work.

2. Preliminaries

First, we define some basic terms to ease explanations in the following sections and chapters.

2.1 Transport systems

For our work, we look upon several transport systems. We distinguish between *public transport* and *footpath* networks.

2.1.1 Public transport

Public transport incorporates all timetable-bound transport like railways, subways, trams and public bus services. Though we could probably model airplanes and ferries in a similar way, they are not of interest for this work and thus we did not look at their specifics. Whether the described algorithms do or don't work with such different transport systems has not been evaluated. To simplify explanations and because our data model abstracts away from that distinction, all public transport vehicles shall be called *trains*, regardless whether they represent trains, buses or other transport in reality. Trains always operate on fixed routes, halt at predefined stops only and follow a strict timetable.

2.1.2 Footpaths

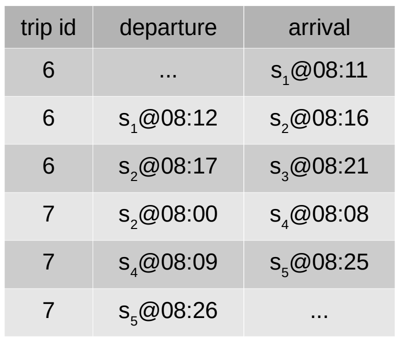
Public transport networks already implicitly include change times, i.e. the minimum time at a specific stop it takes to leave one train and enter another. But for large networks it is also necessary to look at longer walking distances, e.g. between close stops or inside a large station. In our work, we added support for three kinds of footpath networks, to model different walking modalities.

2.2 Network model

In this section we describe the structure of the network data on which the algorithms in chapter 3 operate. The model is based on the one in [DPSW13], but augments it with several ways to model walking distances.

2.2.1 Trips and connections

A *connection* is the most basic building block of a timetable, as depicted in figure 2.1. One connection describes the movement of a specific train from one stop to the next in a fixed timespan. This is described by a departure stop and time and an arrival stop and time. A formal definition of stops follows in 2.2.2. To identify all connections served by the same train, i.e. where no transfer is required in between, each connection is assigned a trip id. We first give a formal definition for connections and then for trips.



trip id	departure	arrival
6	...	s_1@08:11
6	s_1@08:12	s_2@08:16
6	s_2@08:17	s_3@08:21
7	s_2@08:00	s_4@08:08
7	s_4@08:09	s_5@08:25
7	s_5@08:26	...

Figure 2.1: Example of a timetable with six connections (rows) from trips 6 and 7 between stops s_1 to s_5.

Definition 2.1. *A* connection *is a tuple*

$$c := (s_{\text{dep}}, \tau_{\text{dep}}, s_{\text{arr}}, \tau_{\text{arr}}, \text{trip})$$

where $\tau_{\text{dep}} \leq \tau_{\text{arr}}$ *and* $s_{\text{dep}} \neq s_{\text{arr}}$.

The complete movement of one train from the first departure stop to the last arrival stop is called a trip. In our data model, each trip is implicitly defined by all connections that share its id.

Definition 2.2. *A* trip *is a sequence of connections*

$$\{c_0, c_1, \ldots\}$$

where $\tau_{\text{arr}}(c_i) \leq \tau_{\text{dep}}(c_{i+1})$ *and* $s_{\text{arr}}(c_i) = s_{\text{dep}}(c_{i+1})$ *for all* c_i.

No two connections in one trip overlap in time. Our algorithm however is able to handle trips correctly, if one or more connections take zero time, i.e. its departure time equals the arrival time. Also stop times, that is the difference between the arrival time of one connection in the trip and the departure time of the next connection, are allowed to be zero. Traveling back in time is prohibited.

2.2.2 Stops and transfers

As already mentioned, trains operate between *stops*. Each connection has an explicit departure and arrival stop, while each trip has an implicit one (the one of the first/last connection). Every stop s has a *minimal change time* $\tau_{\text{ch}}(s)$ assigned, which is the time it takes to transfer between two trips at the same stop.

Multiple stops connected through a clique of transfer edges are called a *station*. Figure 2.2 shows an example, where two stops are connected to form a station. Each *transfer edge* defines the minimal change time $\tau_{\text{ch}}(s_i, s_j)$ it takes to transfer from a trip arriving at s_i to a trip departing from s_j inside the station. Transfer edges always form a clique, but are directed, as depending on the infrastructure the change time may depend on the direction. Imagine e.g. a long stairway, or escalator in a subway station. Change times can be zero but not negative.

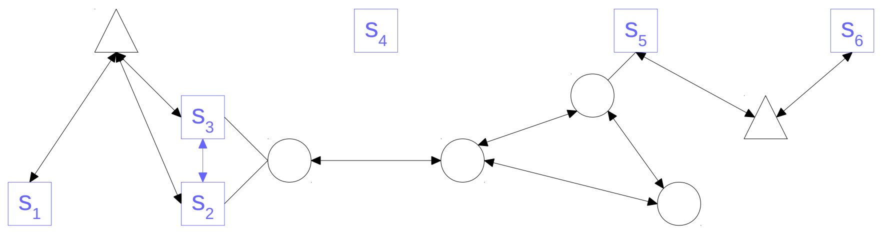


Figure 2.2: Several stops (s_1 to s_6) in a network. s_2 and s_3 form a station.

2.2.3 Zones and attachments

Like described in section 1.1, we subdivide the network into different zones. *Zones* serve as endpoints, i.e. source z_{src} and destination z_{dst}, for each journey exclusively. They cannot be traversed, and no other entity in our network can serve as an endpoint.

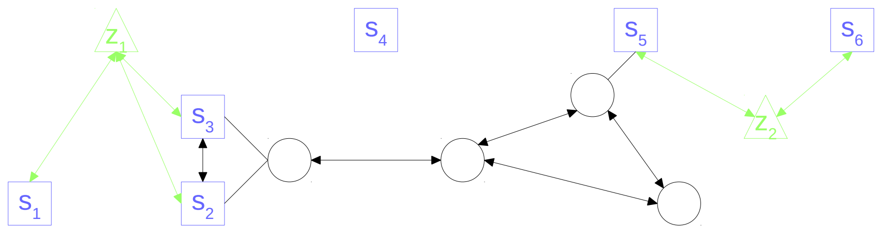


Figure 2.3: Zone z_1 is attached to stops s_1, s_2 and s_3, z_2 to s_5 and s_6.

Each zone is linked to one or more nearby stops through directed edges called *attachments*. Destination attachments determine the stops through which the zone can be reached, while source attachments define the stops which can be used to leave a zone. We denote the travel time for source attachments by $\tau(z_i, s_j)$ and for destination attachments by $\tau(s_i, z_j)$. Attachments cannot have zero or negative travel time, but they do not need to be symmetric.

2.2.4 Places and ways

The third, most general footpath network is modeled by a directed, not necessarily connected graph. We call its nodes *places* and its edges *ways*. Like with attachments, ways with zero or negative weight are prohibited.

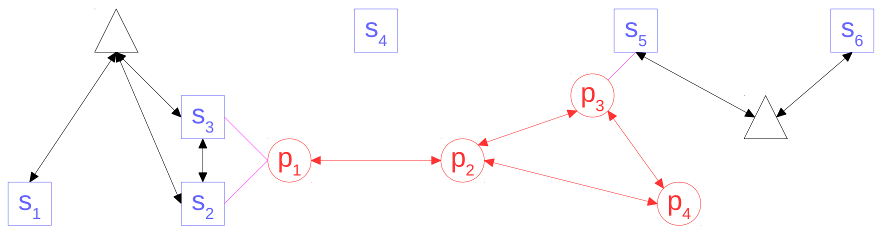


Figure 2.4: Footpath network with places p_1, p_2, p_3 and p_4. p_1 is an entrance for s_2 and s_3.

Entrances

The public transport network is connected to the footpath network by assigning places to stops as *entrances*. Each stop can have exactly one entrance assigned. If a stop does not have an entrance, there is no way to switch from it into the footpath network.

While one place can be the entrance of more than one stop, this should only be done for stops of the same station. As the relation between a stop and it's access node has no weight, it could be used to travel from one stop to another with zero time, and thus give flawed results. Traveling between stops in our network is only possible using exactly one transfer edge, if such exists, or using a footpath with at least one footpath edges, if such a path exists between the access nodes of the two stops.

This network model requires some care when defining the network, we describe some pitfalls in section 2.2.5.

Symmetry

Footpaths do not underly direction dependent restrictions like one-way access, turn restrictions or similar. Thus, for every directed edge, the counter direction shall exist. As with transfer edges, symmetry of weights is not enforced and may not be suitable, if travel time depends on e.g. the slope of the way. After preprocessing (described in section 3.1), the existence of both directions for each way is still likely but not enforced. The algorithms do not rely on it.

2.2.5 Pitfalls

The complicated footpath model using three different types of nodes (stops, zones and places) and edges (transfers, attachments and ways) easily causes trouble when the network modeling is not done carfully. Our algorithm does not allow to directly change between any of the footpath models, i.e. leaving the way graph through an entrance, then crossing a station by a transfer edge and then exiting back into the way graph is not allowed in our model. If crossing a station is wanted behaviour, a shortcut edge has to be added to the way graph. This can be done automatically in a preprocessing step, but we did not take effort to do so.

2.3 Journeys

Journeys begin at a zone z_{src} and end at a different one z_{dst}. In between, trips are entered, left and transfered in between. We first define legs to formalize traveling by train. Then we define how we transfer between trips through the different footpath modalities. Finally, we reach a formal definition for a journey.

Journey legs

A journey leg defines the segment of a trip, or the two connections through which it is entered and left, that is taken in the journey.

Definition 2.3. *A* journey leg *is a tuple*

$$l := (c_{\text{dep}}, c_{\text{arr}})$$

of two connections where $\text{trip}(c_{\text{dep}}) = \text{trip}(c_{\text{arr}})$ *and* $\tau_{\text{dep}}(c_{\text{dep}}) \leq \tau_{\text{dep}}(c_{\text{arr}})$.

Note that $c_{\text{dep}}(l) = c_{\text{arr}}(l)$ is the case where a trip is only taken for one one stop.

Intermediate transfers

Leaving a trip at connection c_{arr} means arriving at the stop s_{arr} at the time τ_{arr}. From there, there are up to three possibilities to transfer to a different trip.

- Transfer at s_{arr}.
- If s_{arr} is part of a station, transfer inside the clique.
- If s_{arr} is linked to an entrance, transfer through the way graph to distant stops.

Transfering localy at s_{arr} reaches all connections c_l departing at s_{arr} no earlier than $\tau_{\text{arr}} + \tau_{\text{ch}}(s_{\text{arr}})$.

If s_{arr} is part of a station, all connections c_n departing from a neighbour stop s_n in the station are reachable if they depart no earlier than $\tau_{\text{arr}} + \tau_{\text{ch}}(s_{\text{arr}}, s_n)$.

A distant stop s_d is reachable if it is linked to an entrance in the same connected component of the way graph as the entrance of s_{arr}. The minimal change time to transfer to s_d is given by the length of the shortest path $\tau_{\text{sp}}(e_a, e_d)$ from the entrance e_a of s_{arr} to the entrance e_d of s_d. Thus, connections departing at s_d are reachable if they depart no earlier than $\tau_{\text{arr}} + \tau_{\text{sp}}(e_a, e_d)$.

Formal definition

Now we can define a journey.

Definition 2.4. *A* journey *is a tuple*

$$j := (z_{\text{src}}, z_{\text{dst}}, \{\underbrace{(c_{\text{dep}}, c_{\text{arr}}), \ldots}_{k+1 \text{ legs}}\})$$

defining a sequence of legs to travel from z_{src} *to* z_{dst} *with* k *transfers.*

For each two consecutive legs (l_i, l_{i+1})*, the following restrictions apply:*

$$\tau_{\text{arr}}(l_i) \leq \tau_{\text{dep}}(l_{i+1})$$

$$\text{trip}(l_i) \neq \text{trip}(l_{i+1})$$

Two journeys are equal exactly if their source and destination zones and all legs are equal. The source departure (destination arrival) time of the journey is implicitly defined through its first (last) leg and the respective matching source (destination) attachment:

$$\tau_{\text{src}}(j) := \tau_{\text{dep}}(l_0) - \tau(z_{\text{src}}(j), s_{\text{dep}}(l_0))$$

$$\tau_{\text{dst}}(j) := \tau_{\text{arr}}(l_k) + \tau(s_{\text{arr}}(l_k), z_{\text{src}}(j))$$

This implies, that a source (destination) attachment between the first departure (last arrival) stop and the source (destination) zone must exist. Transfering between two consecutive legs must be possible as described in the previous paragraph.

3. Algorithms

We divide our algorithm in three main steps, as depicted in algorithm 3.1.

Algorithm 3.1: FindAllJourneys

```
   Input: Network data $N$
   Output: Journeys $J$
1  $N \leftarrow$ Preprocess $(N)$
2  $J \leftarrow \emptyset$
3  forall $z_{dst} \in \text{Zones}(N)$ do
4      $P_{dst} \leftarrow$ ConnectionScan $(N, z_{dst})$
5      $J_{dst} \leftarrow$ JourneyExtraction $(N, P_{dst})$
6      $J \leftarrow J \cup J_{dst}$
```

First, a preprocessing stage simplifies the network; we describe the procedure in section 3.1. Then, for each destination zone we run the two main steps of the algorithm. A customized variant of the multi-criteria profile connection scan algorithm (mcp-CSA) from [DPSW13] computes profile information, as described in section 3.2. Based on these profiles, we introduce our new journey extraction step in section 3.3, which enumerates all journeys through the network that satisfy our problem definition from section 1.1. A simple but significant speedup technique is described in section 3.4. We briefly discuss some possibilities to adjust the algorithms for related problems in section 3.5.

3.1 Preprocessing

The main goal of our preprocessing step is to minimize the size of the way graph. We do this by removing ways and nodes in two steps, that keep our definition of valid transfers through the way graph from section 2.3.

Shortest paths

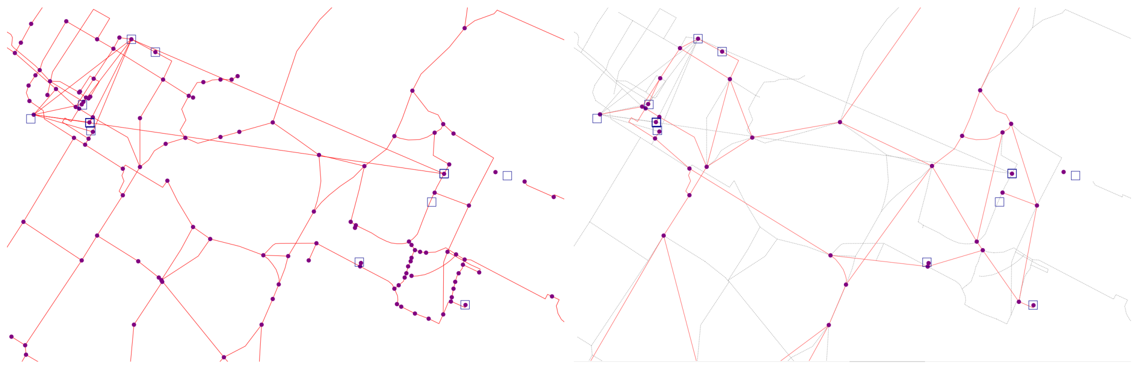
In section 2.3 we defined that to transfer from a stop s_i to a distant stop s_j through the way graph, if possible, is done through the shortest path between the entrances of s_i and s_j. Therefore we run an all pairs shortest path search between the entrances. All ways that are not a part of a shortest path between any two entrances are dropped.

This step dropped most ways in our test instance, especially those in components that are not connected to entrances at all. Also, it gets rid of fine grained inner city networks and parks, where redundant ways and detours occur frequently. A lot of places then remain with low degrees, which sets the stage for our second step.

Node contraction

We now remove unnecessary places from the network by using a simple contraction algorithm. In order to keep the correctness of the network we do not contract entrances. Places of degree two are removed and their adjacent ways are replaced by shortcuts between the adjacent places. Multiedges are handled by keeping only the shortest one.

Finally, we drop all places and entrances, that have no adjacent ways. Figure 3.1 shows a section of the way graph of our test instance before and after preprocessing. The red lines depict ways, the purple dots depict places and the blue rectangles depict stops.



(a) Before preprocessing (b) After reduction

Figure 3.1: Sample section of the way graph in Visum

We do a brief evaluation of the preprocessing for our test instance in section 4.1. There is room for improvement, but it is not of further interest for this work.

3.2 Augmented CSA

The mcp-CSA introduced in [DPSW13] propagates earliest arrival time information through the network backwards in time. Doing so, it computes piecewise linear step functions for each stop in the network. Before we go into detail with the algorithm, we give a formal definition ot these profile functions and define basic operations on them, which serve as building blocks for CSA.

3.2.1 Arrival time profiles

The core data structure of CSA are arrival time profiles. Given a source stop s and a destination, the earliest arrival time profile p_s of s is a piecewise linear function, that returns the earliest arrival time τ_{dst} at the destination for every departure time τ_{dep} at s. The *entries* of p_s are pairs $(\tau_{\text{dep}}, \tau_{\text{dst}})$, and p_s only contains Pareto optimal entries.

$$p_s(\tau) = \begin{cases} \tau_{dst,0}, & \tau \le \tau_{dep,0} \\ \tau_{dst,1}, & \tau_{dep,0} < \tau \le \tau_{dep,1} \\ \dots, & \dots \\ \infty, & \text{otherwise} \end{cases}$$

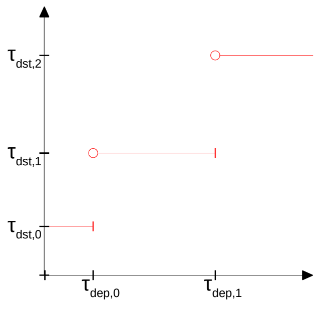


Figure 3.2: Visualization of a plain profile function

Profiles are implemented efficiently as vectors of entries, sorted by departure time. Using contiguous memory, this yields a cache friendly structure, which is critical for fast execution on current computer architectures.

Bicriterial profiles

To incorporate optimization for number of transfers, profiles have to be augmented with a third, discrete dimension. Instead of one, each entry now stores a *bag* of multiple arrival times.

Observing that the number of transfers in a real world public transport network has an upper bound of how many transfers passengers will likely be willing to take, we use a fixed size for the bag. A bag will carry the arrival time for i entries at index i. Because the index starts at zero, a bag size of k (indices 0 to 7) limits the maximal number of transfers to $k-2$ ($k-1$ entries). All experiments in chapter 4 have been run with a bag size of 8, i.e. a maximum number of 6 transfers or 7 legs in a journey. We evaluated the impact of different bag sizes on the total runtime of the algorithm in section 4.5.

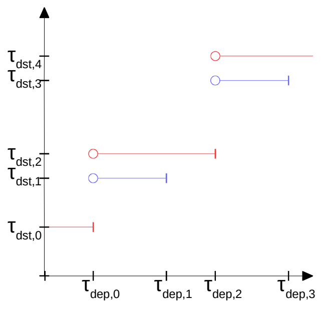


Figure 3.3: Visualization of a bicriterial profile function

Basic profile operations

CSA takes its speed from the cache friendly structure of the profiles. Entries are never removed from, and only added to the front of a profile, because CSA moves monotonic backwards in time. Therefore, the three basic operations we do on profiles share a simple, linear access pattern with high spatial locality.

Evaluating a profile for a given earliest departure time is a simple linear scan from the front. In most cases, this should only run over few entries; in our tests doing a binary search instead yielded no measurable speedup.

Adding an entry only happens at the front of the profile. Therefore, the arrival time bag of the new entry is minimized componentwise with the one of the current front entry. We only keep one entry per departure time. This is also necessary to support the constant evaluation optimization described in section 3.4.

Finally, *merging* two profiles is implemented using the merge step of merge sort to combine the entries into a new, sorted profile. If merged by departure time only, entries have to be dominated from back to front, which is a second, linear run over the profile. We use this linear merging procedure to efficiently build source zone profiles in section 3.2.4.

3.2.2 Original algorithm

We use the connection scan algorithm presented in [DPSW13] and extend it for our different footpath models. For reference, we cite the original procedure using pseudocode in algorithm 3.2.

Algorithm 3.2: ConnectionScan

```
1  $\forall s \in$ Stops: $P[s] \leftarrow \{\infty, (\underbrace{\infty, \dots, \infty}_{m})\}$
2  $\forall t \in$ Trips: $T[t] \leftarrow (\underbrace{\infty, \dots, \infty}_{m})$
3  forall $c \in$ Connections decreasing by $t_{dep}(c)$ do
       // exit at arrival stop
4      $t \leftarrow$ EVALUATE $P[s_{arr}(c)]$ AT $t_{arr}(c)$
       // exit at destination stop
5      if $s_{arr}(c) = s_{dest}$ then
6          $t \leftarrow$ MIN $(t, (t_{arr}(c), \dots, t_{arr}(c)))$
       // stay on the trip
7      $t \leftarrow$ MIN $(t, T[\text{trip}(c)])$
8      $T[\text{trip}(c)] \leftarrow t$
       // enter at departure stop
9      SHIFT $(t)$
10     $t \leftarrow$ MIN $(t, P[s_{dep}(c)].\text{LAST})$
11     if $t$ dominates $P[s_{dep}(c)]$.LAST then
12         APPEND $(t_{dep}(c) - t_{ch}(s_{arr}(c)), t)$ TO $P[s_{dep}(c)]$
```

For each connection c in order of descending departure time t_{dep}, the algorithm first evaluates the profile of the arrival stop $P[s_{arr}]$ at the arrival time t_{arr}. If s_{arr} is also the destination s_{dest}, t_{arr} is obviously the earliest possible arrival time and thus merged into the profile bag t. These steps model the behavior of leaving the trip of the current connection at s_{arr}. It is then evaluated whether not leaving the trip is faster by minimizing t with the trip bag T[trip(c)]. At last, the departure stop profile $P[s_{dep}]$ is queried and updated, if it makes sense to enter the trip there.

To incorporate for the minimum change time t_{ch} of a stop, it is subtracted from the connections' departure time t_{dep} when adding a new entry to $P[s_{dep}]$.

3.2.3 Trip profiles

The first addition we make to CSA is the introduction of trip profiles. CSA only maintains an arrival time bag for each trip, to account for entering or leaving a trip. At the end of the scan, this bag contains the earliest arrival times reachable when entering the trip with the first connection possible. However, when we search for all trips to take from a stop to enumerate all possible journeys, we want to know the arrival times for each trip when we enter from that exact stop, so we can rule out trips that go in the wrong direction.

Instead of only saving the earliest arrival times, we therefore maintain trip profiles. As we scan all connections backwards in time, evaluation of the trip profile for the current connection is simply reading the last entry. Updating is evenly simple by adding a new entry only if exiting through this connection dominates staying on the trip.

3.2.4 Handling zones and attachments

The next extension to CSA are source and destination attachments, defined in section 2.2.3. In [DPSW13], two hints were given as how to handle such initial and final footpaths efficiently. which we describe in more detail in this section. Destination attachments are converted into a *destination distance lookup table* in a short preprocessing step. This yields a simple lookup function $t_{dest}(s)$ given in 3.1.

$$t_{dest}(s) = \begin{cases} t_{dest}, & \text{iff a dest. att. } (s \xrightarrow{t_{dest}} z_{dest}) \text{ exists} \\ \infty, & \text{otherwise} \end{cases} \tag{3.1}$$

Source attachments are handled at the end of the connection scan run when all other profiles are finished. Source zone profiles then are computed by merging the profiles of the stops connected through source attachments. Note that merging is a linear operation due to the sorted order of the profiles. To account for the walk time t_{src} of the source attachment, the departure time axis of the stop profile has to be shifted by t_{src} before the merge step. Algorithm 3.3 depicts the necessary changes.

3.2.5 Handling intermediate footpaths

Adding support for our transfer cliques inside stations and the way graph connecting multiple stations through entrances and places is a lot more involved.

In [DPSW13] a method was presented to expand footpaths into pseudoconnections for each pair of an arrival and a departure at two adjacent stops. This of course is unsuitable for large graphs, as the amount of such pseudoconnections grows exponentially with the number of edges in the graph.

Departure events

We use a different approach to handle footpaths dynamically while scanning the connections. If the departure stop s_{dep} of the current connection c occurs to be part of a transfer clique or has an entrance e assigned, we delay updating its profile $P[s_{dep}]$ and those of the adjacent stops and places by generating *departure events*. These departure events propagate the new profile information through the different footpath networks.

Therefore each departure event is a tuple $((s_{dep}|p_{dep}), t_{dep}, (t_{arr}, \ldots))$ with the id of the stop s_{dep} or place p_{dep} of which to update the profile, the time t_{dep} at which the event occurs and the profile bag $(t_{arr}, \ldots)$. As depicted in algorithm 3.4, when running the connection scan, instead of only scanning through the connections, in each iteration we handle either the next connection or the next departure event.

The arrival part, i.e. evaluating the arrival stop profile, and updating the trip profile stays the same when handling a connection. For stops that have neither transfers nor an

Algorithm 3.3: ConnectionScanWithZones

```
1  $\forall s \in$ Stops: $P[s] \leftarrow \{\infty, (\underbrace{\infty, \dots, \infty}_{m})\}$
2  $\forall t \in$ Trips: $P[t] \leftarrow \{\infty, (\underbrace{\infty, \dots, \infty}_{m})\}$
3  forall $c \in$ Connections decreasing by $t_{dep}(c)$ do
       // exit at arrival stop
4      $t \leftarrow$ EVALUATE $P[s_{arr}(c)]$ AT $t_{arr}(c) + t_{ch}(s_{arr}(c))$
       // walk to target zone
5      if $t_{dest}(s) < \infty$ then
6          $t_{dest} \leftarrow t_{arr}(c) + t_{dest}(s)$
7          $t \leftarrow$ MIN $(t, (t_{dest}, t_{dest} \dots t_{dest}))$
       // stay on the trip
8      $t \leftarrow$ MIN $(t$, last$(P[$trip$(c)]))$
9      APPEND $(t_{dep}(c), t)$ TO $P[$trip$(c)]$
       // enter at departure stop
10     SHIFT $(t)$
11     $t \leftarrow$ MIN $(t$, last$(P[s_{dep}(c)]))$
12     if $t$ dominates last$(P[s_{dep}(c)])$ then
13         APPEND $(t_{dep}(c), t)$ TO $P[s_{dep}(c)]$
   // walk from source zones
14 forall $z \in$ Zones do
15     $P[z] \leftarrow \emptyset$
16     forall source attachments $(z \xrightarrow{t_{src}} s)$ do
17         $P'[s] \leftarrow$ OFFSET $(P[s], t_{src})$
18         $P[z] \leftarrow$ MERGE $(P[z], P'[s])$
```

entrance, the departure part stays the same, too. Else, instead of updating the stop profile directly, departure events are enqueued.

For each transfer $(s \xrightarrow{t_{ch}} s_{dep}(c))$, a departure event $(s, t_{dep}(c) - t_{ch}, (t_{arr}(c), \dots))$ is enqueued. Note that the local stop change time has to be handled this way, too.

The profile of the entrance e of $s_{dep}(c)$ is updated directly, because the assignment has no time difference. To propagate the arrival times through the way graph however, for each place p adjacent to the entrance e through a direct way $(p \xrightarrow{t_{way}} e)$ a departure event $(p, t_{dep}(c) - t_{way}, (t_{arr}(c), \dots))$ is enqueued.

Handling an event now is straight forward: Update the profile of the affected stop or place, and, if the profile bag was not dominated, continue propagation by creating new departure events in the same manner as before. In case of a non-dominated profile update to an entrance node, all assigned stop profiles have to be updated, too.

Changes to stop profiles

Because of the incorporation of local minimum change times into the stop profile, it's interpretation has to be taken with care. As we do not allow to switch directly between footpath modalities, the stop profiles are only valid when evaluated from a train arrival. This leads to errors in both creating the source zone profiles and if used as pruning rule in

Algorithm 3.4: ConnectionScanEventLoop

```
1  $\forall s \in$ Stops: $P[s] \leftarrow \{\infty, (\underbrace{\infty, \ldots, \infty}_{m})\}$
2  $\forall t \in$ Trips: $P[t] \leftarrow \{\infty, (\underbrace{\infty, \ldots, \infty}_{m})\}$
3  EventQueue $\leftarrow \emptyset$
4  while connections left do
5      $c \leftarrow$ next connection
6      if EventQueue $= \emptyset$ then
7          handleConnection ($c$)
8      else
9          $e \leftarrow$ top(EventQueue)
10         if $t_{dep}(e) < t_{dep}(c)$ then
11             handleConnection ($c$)
12         else
13             EventQueue.pop ()
14             handleDepartureEvent ($e$)
15 while EventQueue $\neq \emptyset$ do
16     $e \leftarrow$ top(EventQueue)
17     EventQueue.pop ()
18     handleDepartureEvent ($e$)
   // walk from source zones
19 forall $z \in$ Zones do
20     $P[z] \leftarrow \emptyset$
21     forall source attachments ($z \xrightarrow{t_{src}} s$) do
22         $P'[s] \leftarrow$ offset ($P[s], t_{src}$)
23         $P[z] \leftarrow$ merge ($P[z], P'[s]$)
```

the journey extraction. Therefore we manage a second profile for each stop, that carries arrival time information for all connections departing at that stop only. We call the first profile incorporating walk times *stop walk profile* and the second, incorporating departing trains only *stop train profile*.

3.3 Journey extraction

The destination zone z_{dst} is fixed for each iteration of the algorithm. To find all journeys that end at z_{dst}, we now loop over the source zones, and run a depth first search (DFS) through the network for each source zone z_{src}.

Iterating over the profile entries of the source zone gives us tuples $(z_{\text{src}}, \tau_{\text{src}}, z_{\text{dst}}, \tau_{\text{dst}}, k)$. From the construction of the profile we know, that these tuples describe exactly the optimal journeys we are searching for. We now need to find all possible sequences of $k+1$ legs, that match the given restrictions.

3.3.1 Source attachments

For each source profile entry $(\tau_{\text{src}}, \tau_{\text{dst}}, k)$ we initiate the DFS by scanning the source attachments (z_{src}, s_i) of z_{src}. Valid first departure stops s_i are those, where evaluating the stop train profile of s_i at $\tau_{\text{src}} + \tau(z_{\text{src}}, s_i)$ for k transfers gives exactly τ_{dst}.

3.3.2 Extracting journey legs

Beginning at the first departure stop we use the time and transfer information to iteratively evaluate profiles and find all possible series of connections that form such a path.

A path segment consists of two connections. The departing connection, where we depart from the current stop, thus enter a trip, and the arriving connection where we leave the trip again at some other stop. To find all suitable segments, we have to scan all departing connections at the current stop. We will look into footpath and transfer handling in a later section. For each departing connection we have to scan the subsequent connections in it's trip possible arriving connections.

Algorithm 3.5: ADDJOURNEYLEG

```
   Input: Departure stop $s_{\text{dep}}$, earliest departure time $t_{edt}$, remaining transfers $k$,
          target arrival time $\tau_{\text{dst}}$
   Output: List of segments $S$ to continue the current path
1  $S \leftarrow \emptyset$
   // possible departures from stop
2  $D \leftarrow$ findDepartureConnections $(s_{\text{dep}}, t_{edt} + s_{\text{dep}}.transfer\_time, k, \tau_{\text{dst}})$
   // possible arrivals from each trip
3  forall $c_{\text{dep}} \in D$ do
4      $A \leftarrow$ findArrivalConnections $(c_{\text{dep}}.trip, c_{\text{dep}}.\tau_{\text{dep}}, k-1, \tau_{\text{dst}})$
5      forall $c_{\text{arr}} \in A$ do
6          $S \leftarrow S \cup \{(c_{\text{dep}}, c_{\text{arr}})\}$
```

Departure connections

Finding departure connections is a very expensive operation, thus we describe and evaluate some speed-up techniques in chapter **??**. The basic algorithm is like follows. We scan over the connection array for connections that depart at the given stop no earlier than the given departure time and no later than the target arrival time. For each of those connections we then evaluate the trip profile to check whether we can reach the target arrival time with entering the trip at the current stop and time. If yes, it is a valid departure connection and we will find at least one arrival connection for it.

Algorithm 3.6: FINDDEPARTURECONNECTIONS

```
Input: Departure stop $s_{\text{dep}}$, earliest departure time $t_{edt}$, remaining transfers $k$,
       destination arrival time $\tau_{\text{dst}}$
Data: Connections $C$, trip profiles
Output: List of valid departures $D$ from the stop
1 $D \leftarrow \emptyset$
  // scan all connections
2 forall $c \in C$ do
      // filter those, that are not reachable
3     if $s_{\text{dep}}(c) = s_{\text{dep}}$ and $t_{edt} \leq \tau_{\text{dep}}(c) < \tau_{\text{dst}}$ then
          // evaluate the trip profile entering the connection
4         $\tau_{\text{arr}} \leftarrow$ EVALUATEPROFILE $(\text{trip}(c), \tau_{\text{dep}}(c), k)$
          // check if the trip reaches the targeted arrival time
5         if $\tau_{\text{arr}} = \tau_{\text{dst}}$ then
6             $D \leftarrow D \cup \{c\}$
```

Arrival connections

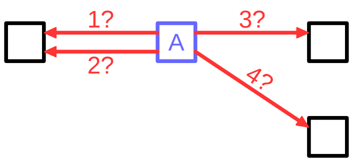


Figure 3.4: For departure stop A, outgoing con-

Finding arrival connections is similar to finding departure connections. Instead of the connections departing at a stop we now scan through the connections of a trip to search for valid exits. We therefore evaluate the profile of the connections arrival stop at the connections arrival time.

To account for final footpaths, if we have no more transfers left we check for a matching destination attachment instead of evaluating the profile.

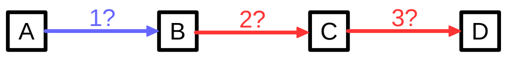


Figure 3.5: For departure connection 1, arrival connections 1, 2 and 3 are candidates. Profiles of stops B, C and D are checked to find out.

Partial connection datastructure

In the last chapter we described the algorithm to find journey legs by scanning all connections for departure and arrival candidates. Observing the patterns on how we select these connections leads us tthe definition of the intermediate step we mentioned briefly in the introduction of chapter 3.

To find valid departures from stop s, we only need to scan over connections departing at s. Sorting the connections after stop $s_{\text{dep}}(c)$ first and departure time $\tau_{\text{dep}}(c)$ second allows for a binary search instead of a linear scan to determine the first connection c departing at s no earlier than τ_{dep}. The same can be done for the arrivals, sorting by trip first and τ_{dep} second.

Changing the data structure from a one- to a two-dimensional array, we make the search for connections departing at s or being part of a trip a constant operation and also improve spatial locality. In the second dimension, two binary searches are used to omit connections outside the time interval $[\tau_{\text{dep}}, \tau_{\text{dst}}]$.

Instead of copying the full connections into the new two-dimensional structures, we copy only the information required to fulfill the correspondent query to further improve cache friendliness. As we described in section 2.2.1, a connection has five data members: departure

Algorithm 3.7: FindArrivalConnections

```
Input: Trip trip, earliest arrival time $t_{eat}$, remaining transfers $k$, destination
       arrival time $\tau_{\text{dst}}$
Data: Connections $C$, stop profiles
Output: List of valid arrivals $A$ from the stop
1  $A \leftarrow \emptyset$
   // scan all connections
2  forall $c \in C$ do
       // filter those, that are not reachable
3      if $\text{trip}(c) = \text{trip}$ and $t_{eat} \leq \tau_{\text{arr}}(c) < \tau_{\text{dst}}$ then
4          $\tau_{\text{arr}} \leftarrow \infty$
5          if $k = 0$ then
               // check for a final connector
6              if $\tau(s_{\text{arr}}(c), z_{\text{dst}}) \neq \infty$ then
7                  $\tau_{\text{arr}} \leftarrow \tau_{\text{arr}}(c) + \tau(s_{\text{arr}}(c), z_{\text{dst}})$
8          else
               // evaluate the trip profile entering the connection
9              $\tau_{\text{arr}} \leftarrow$ EVALUATEPROFILE $(s_{\text{arr}}(c), \tau_{\text{arr}}(c), k)$
           // check if the trip reaches the targeted arrival time
10         if $\tau_{\text{arr}} = \tau_{\text{dst}}$ then
11             $A \leftarrow A \cup \{c\}$
```

stop and time, arrival stop and time and the trip. We split each connection into two new *partial connections* c_{dep} and c_{arr}. The *departure connection* $c_{\text{dep}} = (\tau_{\text{dep}}(c), \text{trip}(c), \text{id}(c))$ inherits the departure time and the trip id and is stored in the bucket associated with the departure stop $s_{\text{dep}}(c)$. The *arrival connection* $c_{\text{arr}} = (\tau_{\text{arr}}(c), s_{\text{arr}}(c), \text{id}(c))$ inherits the arrival time and stop and is stored in the bucket associated with the trip $\text{trip}(c)$. Both partial connections also store the id of the original connection $\text{id}(c)$ to allow fast reconstruction of the journey with exact stops and times.

In the next section we introduce a fourth data member for each partial connection, the *profile lookup index*, that makes profile evaluation a constant time operation.

3.3.3 Intermediate transfers

Transfering at the local stop or station is trivial to evaluate. The stop train profile is evaluated to prune stops that will not yield matching departures.

Transfering through the way graph is more involved. We use a modified Dijkstra shortest path search algorithm to traverse the way graph. To check, whether search is continued from a place, its profile is evaluated. If τ_{dst} is not reachable anymore, that branch of the search is pruned. For each settled entrance, the stops' train profiles are evaluated to check whether transfering here is possible.

3.4 Constant time profile evaluation

The straight forward depth-first search based algorithm we presented in the last chapter does not perform well for large networks. Finding the possible legs to continue the current journey is expensive, given the amount of scans one has to perform.

As mentioned in the last section we can make profile evaluation in the leg search a constant time operation. Each time we scan a departure (arrival) connection candidate, we have to evaluate the associated trip (arrival stop) profile. While the result is fixed for each candidate, the evaluation is repeated every time the candidate is scanned. In fact, we already did that exact same evaluation when handling the connection in the connection scan run. Therefore, while scanning a connection, we save the index of the resulting profile entry when evaluating the arrival stop profile and the trip profile. We add these indices to the result dataset of the connection scan algorithm and copy them into the partial connections: departure connections get augmented with trip indices, arrival connections with stop indices.

3.5 Extensions

Enumerating journeys that are Pareto optimal for travel time and number of transfers is reasonable but cannot satisfy the requirements of demand assignment completely. We therefore briefly describe some extensions to our algorithm, that add support for different goals.

3.5.1 Epsilon-bound destination arrival time

The journey extraction algorithm evaluates profiles and only considers connections, where the destination arrival time $\tau_{\mathrm{dst}}(c)$ compares equal to the query target τ_{dst} for the current extraction run. If we relax this condition to an $\epsilon-$bound comparison, i.e. whether $\tau_{\mathrm{dst}}(c)$ lies within an interval $[\tau_{\mathrm{dst}} - \epsilon, \tau_{\mathrm{dst}} + \epsilon]$, we enumerate additional journeys that only differ slightly from the optimal ones, but may yield longer change times and thus better delay robustness.

3.5.2 Fare approximation using preprocessing

Incorporating fare information is quite difficult, as it is a third criterion that underlies hard to generalize irregularities like fixed tariff zones, discounts for groups and return trips, vehicle category dependent pricing, or Even in PTV Visum, exact fare information is not incorporated directly into the journey extraction but rather as a rough approximation, which hopefully yields a close enough set of journeys to apply exact calculations in the demand assignment.

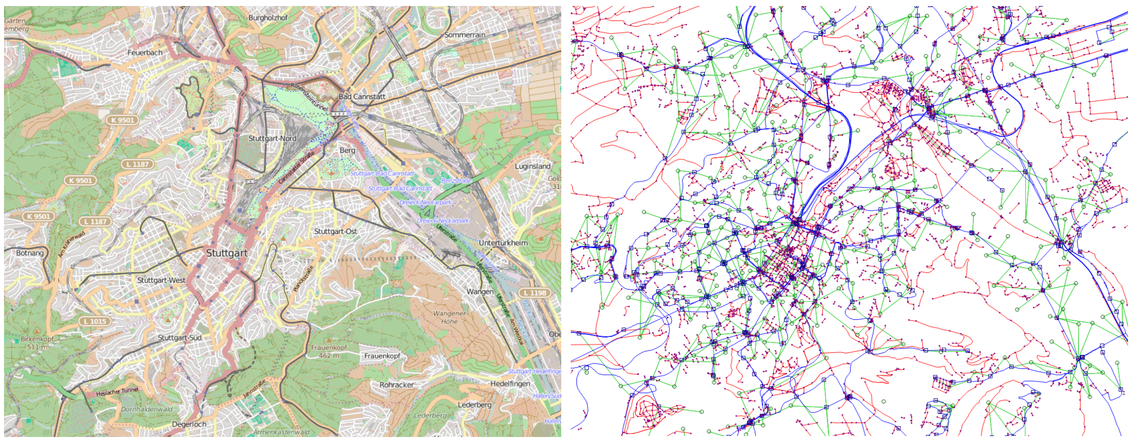
We can approximate at least some fare criteria by reducing the network in a preprocessing step. E.g. by filtering out trips that use vehicles of a higher price category, we can force the journeys to only take cheaper vehicles. The same can be done easily to filter for specific areas or time intervals. Repeating this procedure for different filter criteria yields sets of journeys optimized under different restrictions, which can afterwards be merged. As these runs operate on much smaller data, each one is faster than one run on the whole network.

4. Experiments

We ran extensive experiments on several machines with different CPU and RAM configurations to evaluate the performance and overall quality of our algorithm. In this chapter we describe our test data and discuss the results. Runtimes are given in seconds, unless noted otherwise. For the performance tests, we repeated every run at least thrice and quote the fastest one.

4.1 Test instance

The main instance for our evaluation is based on a model of the regional network around the German city of Stuttgart from 2009. Figure 4.1 shows an excerpt of the instance as visualized in Visum compared to the corresponding view from Openstreetmap[1].



(a) Map view from Openstreetmap.org

(b) Network visualization in Visum: Zones, stops, tracks and the way graph.

Figure 4.1: Excerpt of the area surrounding Stuttgart main station

The original data includes all kinds of information, and we extracted the model described in 2.2. To fix consistency errors, we recalculated all travel times in the data set. We also set travel times for ways to at least one second to fit our way graph model. For ways and attachments, travel times are rounded to one second. Departure and arrival times as well

[1]`http://openstreetmap.org`

as transfers are rounded to 30 seconds. The comparision run in Visum described in 4.4 was also executed on the fixed data.

Besides recalculating travel times, we filtered the following data from the original set:

- Connections within a stop, i.e. where $s_{\text{dep}}(c) = s_{\text{arr}}(c)$. They occur in the original dataset, but serve no good in our algorithm.
- Connections that depart before 1am or arrive after 11pm, to prevent issues with day change. Running the algorithm over multiple days should not be an issue, but requires some care when calculating departure and arrival times, which is beyond the scope of this work.
- Trips, if they do not contain any connection. Could occur because of the filter above.
- Stops, if there are neither departures nor arrivals.
- Ways and attachments, if they are not allowed for foot.
- Places and zones, if they have no adjacent way or attachment.

An overview of the final size of the instance is given in table 4.1.

	filtered	preprocessed
connections	769242	
trips	47542	
stops	12169	
transfer links	2574	
places (entrances)	29503 (11990)	620 (537)
ways	32190	986
zones	1154	
attachments	9502	

Table 4.1: Size of the Stuttgart instance

Preprocessing, as described in 3.1, removes the larger part of the way graph. The whole preprocessing step took about one or two seconds in our experiments, depending on the machine. Though there is probably room for improvement, we put no effort in this, because a few seconds do by no means hurt our overall performance, as long as correctness is guaranteed.

In the following paragraphs we look into the characteristics of the data set in more detail.

Trips

trip length	#connections	travel time [s]
min	1.0	0.0
median	16.0	780.0
max	64.0	30,630.0
mean	16.2	1,087.0

(a) Lengths of trips

travel time [s]	all	> 0
min	0.0	30.0
median	30.0	30.0
max	7,860.0	7,860.0
mean	67.1	71.5

(b) Travel times of connections

Table 4.2: Lengths of trips and connections

First, note, that we do allow connections to have zero travel time and thus, trips can have zero length, too. As shown in table 4.2 The lengths of the 47,542 trips are spread from one to 64 connections, i.e. reaching two to 65 stops, with a reasonable median of 16. Most of the 769,242 connections however take only 30 or 60 seconds travel time. The arrival and departure times are rounded to 30 seconds resolution.

Zones

	attached stops
min	1.0
median	3.0
max	34.0
mean	4.1

	attachment length [s]
min	1.0
median	480.0
max	7200.0
mean	456.9

Table 4.3: Zone connectivity

Table 4.3 gives an overview over the 1,154 zones and 9,502 attachments. On average, a zone is attached to about four stops.

Stations

station size	1	2	3	4	5	6	7	8	9
stations	10709	449	84	36	13	8	4	0	1
stops	10709	898	252	144	65	48	28	0	9

Table 4.4: Distribution of stops on station sizes

Table 4.4 gives an overview over the sizes of stations, i.e. transfer cliques in the network. Most of the 12,169 stops do not have directly linked neighbours, but station sizes up to 9 stops do occur.

change times [s]	in stop	in station	overall
min	0.0	0.0	0.0
median	60.0	120.0	60.0
max	300.0	480.0	480.0
mean	60.2	120.5	70.7

Table 4.5: Change times in and between stops

Change times are as expected, with an average of about 60 seconds local and 120 seconds between two stops of a station. 60 seconds is the resolution the change times are rounded to in the data set. With a maximum of 480 seconds or 8 minutes, table 4.5 reveals no excessive values.

Way graph

The footpath network after preprocessing consists of 620 places, of which 537 are assigned as entrance to at least one stop, and 986 ways in between these places. The graph is dissected into 199 connected components with an average size of only about three places per component, as shown in table 4.6.

	stops	places (entrances)	radius [s]	diameter [s]
min	2.0	2.0 (2.0)	17.0	17.0
median	2.0	2.0 (2.0)	145.0	175.0
max	28.0	87.0 (24.0)	6410.0	9649.0
mean	3.0	3.1 (2.7)	302.5	414.3

Table 4.6: Way graph connectivity

Also, the average diameter, i.e. the longest of all shortest paths through the component, is only about seven minutes. That means, most walking distances even through the way graph will be up to seven minutes only.

4.2 Result

In total, our algorithm extracts 401,521,153 journeys. On average, this gives about 347,938 journeys per destination or 302 journeys per source-destination pair.

As described in 3.2.1 we used a maximal number of transfers of 6, except for the experiments with different numbers in section 4.5.

Performance improvement with profile indexing

As described in section 3.4, we expect a huge gain from making profile lookups in the leg search step of the journey extraction a constant time operation. Our experiments confirm this expectation.

algorithm	base	w/ indexing
runtime [s]	1511.0	1244.9
time per journey [μs]	3.8	3.1

Table 4.7: Runtime with and without profile indexing

Table 4.7 shows the runtimes measured with and without profile indexing when running singlethreaded on the single Xeon machine, described in the next section. We used the faster version with profile indexing for all following experiments.

4.3 Multithreading performance

The whole algorithm is organized as one large loop over the destination zone, where each iteration only depends on the read only network data. Thus, it is embarrassingly parallelizable and we expect a good speedup running several iterations in parallel in separate, independent threads. As the algorithm itself is written in C++, parallelization is implemented using OpenMPs `#pragma parallel for` construct.

We evaluate the speedup on three machines featuring different CPU and RAM configurations. As mentioned in the previous paragraph, we enabled the profile indexing technique in all following tests.

Dual Xeon workstation

The first machine is a workstation powering two Intel Xeon X5550 CPUs and 48GByte RAM. Each CPU features 4 cores clocked at 2.67GHz and supports Intel Hyperthreading technology, so in sum there are 16 virtual cores available. The operating system installed is Windows 7 Pro SP1 64bit and we compiled the algorithm test tool using VisualStudio 2013 with optimization level /Ox.

threads	1	2	4	8	16
runtime [s]	2534.1	1443.5	757.7	463.0	345.9
time per journey [μs]	6,3	3,9	1,9	1,2	0,9
speedup	1.0	1.8	3.3	5.5	7.3

Table 4.8: Runtimes and speedup for the Dual Xeon setup

We observe a very good speedup with up to 8 threads. More than 8 threads use Intel Hyperthreading, which shares caches and memory buses among virtual cores. As we heavily rely on positive cache effects due to spatial locality of our data structure, we expected this behaviour.

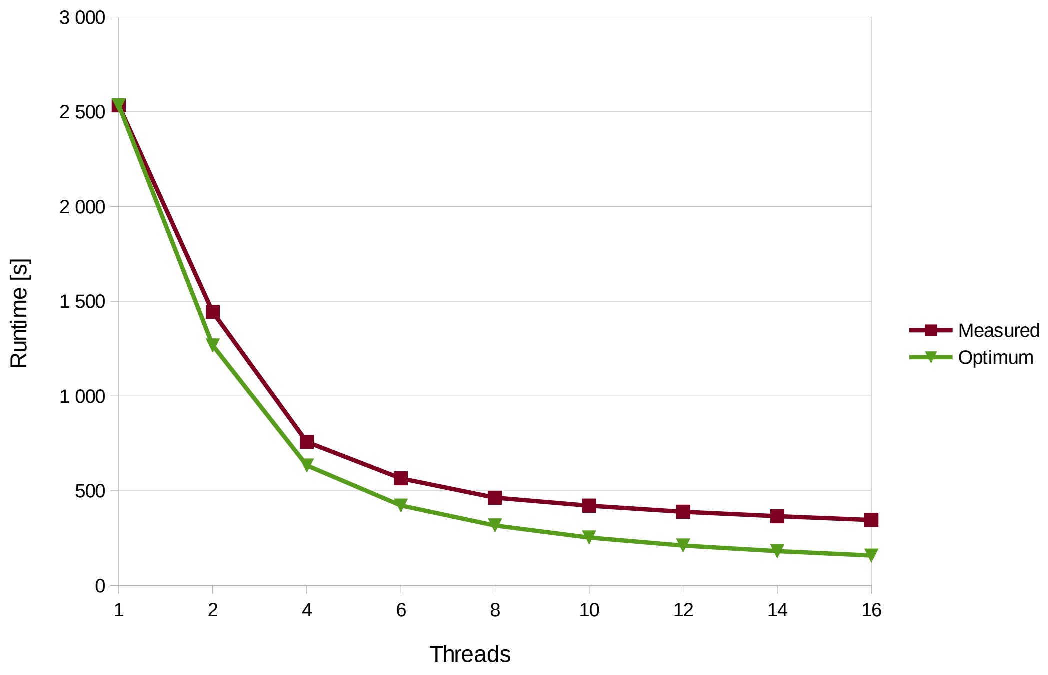


Figure 4.2: Runtime graph for the Dual Xeon setup

Quad Opteron server

The second test setup is a server powered by four AMD Opteron 6172 CPUs and 256GByte RAM. Each CPU features 12 cores clocked at 2.1GHz with six cores per die and two dies per CPU. The test tool is compiled using GCC 4.8 with -O3 and runs on a 64bit OpenSUSE installation. We used a binary counter like thread pinning pattern to keep the scheduler from flipping the threads between cores, which would render the cache hierarchy useless.

threads	1	2	4	8	16	32	48
runtime [s]	3465.4	2220.8	1540.9	1097.4	933.8	1466.9	2046.1
time per journey [μs]	8.6	5.5	3.8	2.7	2.3	3.7	5.1
speedup	1.0	1.6	2.2	3.2	3.7	2.4	1.7

Table 4.9: Runtimes and speedup for the Quad Opteron setup

Interestingly, the machine performs much slower than the first one. Even with 4 threads spread to 4 different CPUs the speedup is not nearly as good as with the dual Xeon setup. More than 12 threads seem to hurt the memory buses, as the speedup decreases until 48 threads perform nearly as slow as 2 threads.

Single Xeon server

The last machine we run our test tool on is a server powering a single Intel Xeon CPU with four cores clocked at 2.3GHz and 128GByte RAM. The test tool is compiled using GCC 4.8 with -O3 and runs on a 64bit OpenSUSE installation.

While beeing the fastest machine for the single threaded run with about half the runtime of the first system, the single Xeon setup does not scale very well with a speedup of only 2.4 with 4 threads.

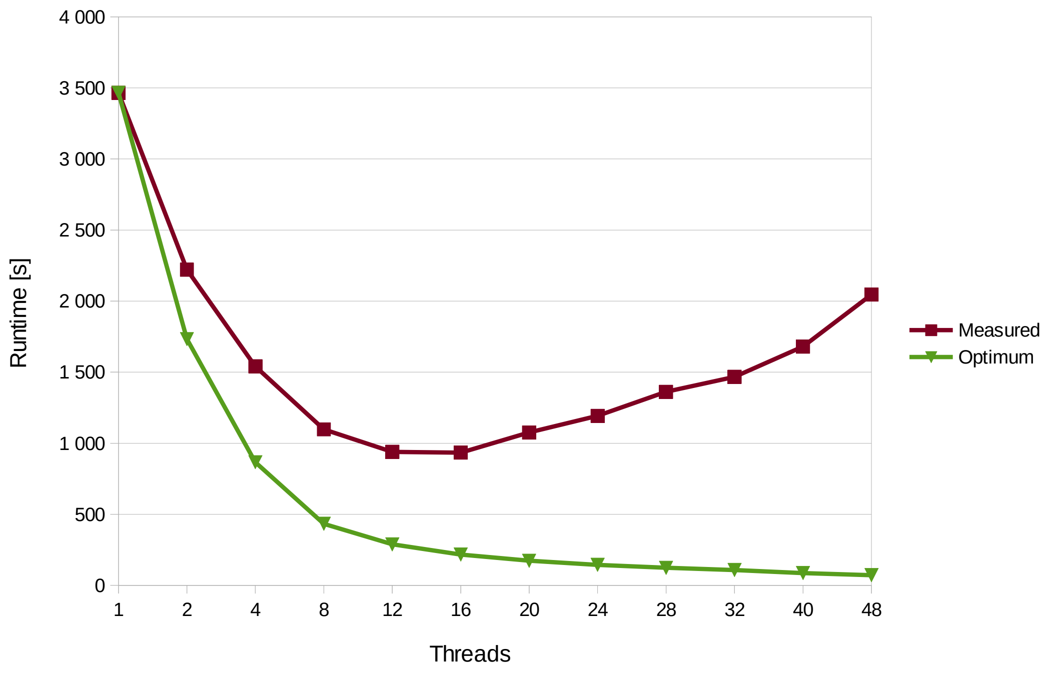


Figure 4.3: Runtime graph for the Quad Opteron setup

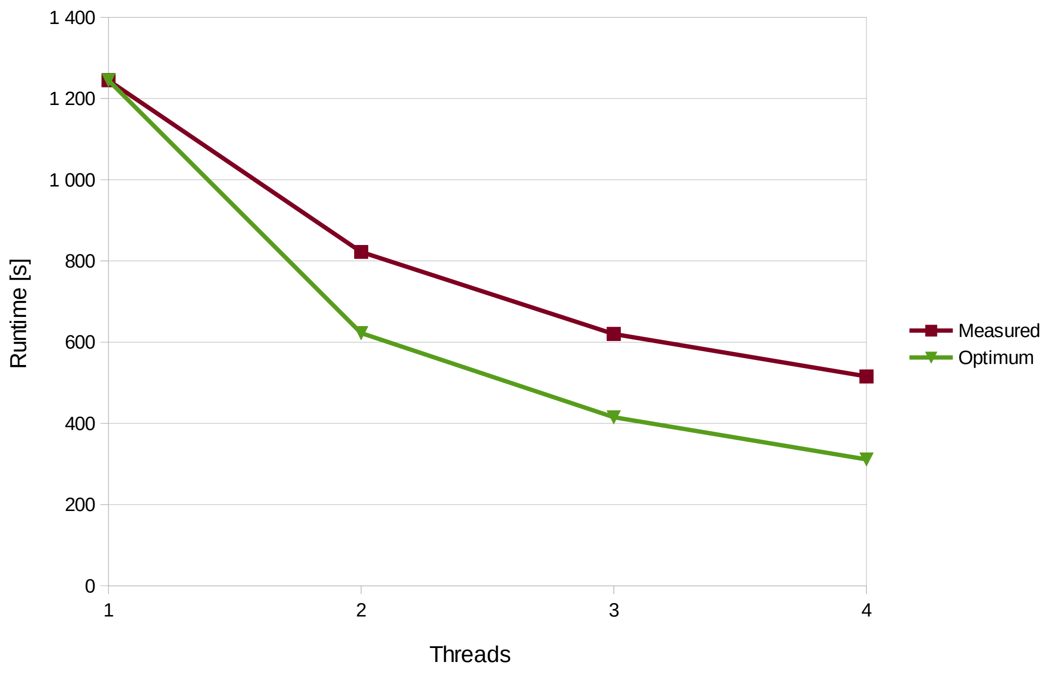


Figure 4.4: Runtime graph for the Single Xeon setup

threads	1	2	3	4
runtime [s]	1244.9	822.2	620.3	515.4
time per journey [μs]	3.1	2.1	1.6	1.3
speedup	1.0	1.5	2.0	2.4

Table 4.10: Runtimes and speedup for the Single Xeon setup

Parallelization summary

Overall, the speedup observed heavily depends on the machine. It seems that memory connection and cache organisations have a large impact.

The best speed we could observe is a total runtime of below $346s$ with 16 threads on the first machine, resulting in only $0.86\mu s$ per journey extraction time.

4.4 Comparision with Visum

Visum implements a bicriterial variant of the Dijkstra algorithm to find nearly the same set of journeys as we do. Configured to the same maximal number of transfers, as well as unrestricted walk times, the Visum implementation yields around 94 mio. journeys. The large difference to our over 400 mio. journeys comes from the issue, that the Visum implementation only searches for one alternative for each optimal journey, instead of all of them like we do. Also, there are slight differences in how Visum handles the complicated network model.

But even for the much smaller set of journeys found, Visum takes over 20 hours to complete the search using 8 threads on the dual xeon workstation we described in the first parallelization paragraph. Compared to that, we extract over four times the journeys in less than 8 minutes.

4.5 Maximal number of transfers

Our algorithm is compiletime configurable for the maximal number of transfers. Choosing the number is a tradeoff between quality of the result and runtime of the algorithm. The 6 transfers we choose for the most of our tests seems intuitively reasonable, as transfers are always unpleasant and increase the risk to suffer from delays. To evaluate the differences in runtime and result set, we did some tests with different maximal numbers of transfers. The tests were run on the single xeon machine described above, using four parallel threads.

max. # transfers	4	5	6	7	8
# journeys	311,557,913	370,529,623	401,521,153	419,837,896	433,438,274
# new journeys		58,971,710	30,991,530	18,316,743	13,600,378
runtime [s]	438.0	483.0	515.4	534.7	556.9
per journey [μs]	1.4	1.3	1.3	1.3	1.3

Table 4.11: Results with different maximal numbers of transfers

Table 4.11 shows the measured runtime and sizes of the result journey set. As journeys with more transfers never dominate those with less, increasing the number will probably add alternatives, but never remove journeys found with a lower number from the set. We give the number of newly found journeys in the second row of the table. Increasing the maximal number of transfers from 6 to 7 yields only about 4.5% more journeys.

More interestingly, the time per journey decreases with more transfers taken into account, rather than the oppposite. This is unexpected, as increasing the maximal number of transfers directly increases the profile bag size and thus the size of the profile data which is computed by CSA and scanned by the journey extraction step. As the parallelization experiments showed that the algorithm is memory bound, we expected negative impact on the runtime from the increased memory footprint of the profiles, which also hurts the caches.

5. Conclusion

We presented a new journey extraction algorithm to efficiently enumerate all journeys optimal to two criteria, time and number of transfers, in a given network. The algorithm incorporates handling for real world footpath networks.

Experiments proved the effiency of our approach. Especially in comparision to an algorithm that is currently available on the market, we perform a lot better. Speedup through parallelization is reasonably good.

5.1 Further work

While this algorithm is a good starting point, and a huge leap forward compared to a bicriterial Dijkstra based approach, there are a lot of open topics to research.

Footpath model

The footpath modeling we used in this work is very complicated and has pitfalls. A better unified model which incorporates transfers between stops as well as arbitrary footpath networks would drastically simplify the algorithm, as there are a lot of special cases right now. This would also get rid of some of the pitfalls.

Multi-criteria extensions

In section 3.5 we described how the algorithm can be extended to other criteria through simple pre- and postprocessing. These approaches need to be refined and evaluated. Also, other possibilities for multicriterial enumeration would be of interest.

Bibliography

[BSS13] Hannah Bast, Jonas Sternisko, and Sabine Storandt. Delay-Robustness of Transfer Patterns in Public Transportation Route Planning. In Daniele Frigioni and Sebastian Stiller, editors, *ATMOS - 13th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems - 2013*, volume 33 of *OpenAccess Series in Informatics (OASIcs)*, pages 42–54, Sophia Antipolis, France, September 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[DKP12] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel computation of best connections in public transportation networks. *J. Exp. Algorithmics*, 17:4.4:4.1–4.4:4.26, October 2012.

[DMS08] Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. Multi-criteria shortest paths in time-dependent train networks. In Catherine McGeoch, editor, *Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 347–361. Springer Berlin Heidelberg, 2008.

[DPSW13] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Intriguingly simple and fast transit routing. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54. Springer Berlin Heidelberg, 2013.

[DPW12] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-based public transit routing. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. Society for Industrial and Applied Mathematics, 2012.

[FHW01] Markus Friedrich, Ingmar Hofsaess, and Steffen Wekeck. Timetable-based transit assignment using branch and bound techniques. *Transportation Research Record: Journal of the Transportation Research Board*, 1752:100–107, 2001.

[Gei10] Robert Geisberger. Contraction of timetable networks with realistic transfers. In Paola Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 71–82. Springer Berlin Heidelberg, 2010.

[MHSWZ07] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable information: Models and algorithms. In Frank Geraets, Leo Kroon, Anita Schoebel, Dorothea Wagner, and Christos Zaroliagis, editors, *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*, pages 67–90. Springer Berlin Heidelberg, 2007.