Programming Internship: Connection Scan Algorithm (CSA)

PATRICK STEIL, Heidelberg University, Germany

Connection **S**can **A**lgorithm [1] (*CSA*) is an algorithm that efficiently answers queries to timetable systems, such as bus, rail, tram networks. Basic inputs consist of source and target points, departure times and further also time barriers, such as latest departure. Advantages over existing algorithms are, for example, the simple underlying data structure and the short preprocessing time. Various algorithm variants are implemented within the scope of this programming internship. In order to test the algorithms on real world scenarios, a GTFS [1] parser was also implemented that reads the RNV public data in such a way that the algorithms can work with it.

1 INTRODUCTION

In 2019, Deutsche Bahn transported 13.4 million passengers with approximately 23500 trains and 2300 buses per day [2]. At these scales, algorithms like CSA play an important role in journey planning. Travel planning simply has to be fast and efficient, because passenger expect subsecond response times for their queries.

This document is structured as follows: First, notation and terms are introduced, followed by an explanation of different problems and variants of the CSA. The GTFS format is then discussed in more detail, together with problems with the RNV dataset, and finally the results of the different CSA variants applied to the GTFS data are presented.

2 FORMALIZATION

The *timetable* consists of *stations* (denoted by $A, B, C, \dots$), *connections*, *trips* and *footpaths*.

- A *trip* t (identified with a *trip_id*) describes one train.
- A *connection* c describes a "real" train departing from A at timestamp π_{dep} and arriving at B at timestamp π_{arr} without intermediate stop, denoted by $(A, B, \pi_{dep}, \pi_{arr}, \text{trip\_id})$. To ensure correctness of CSA, $\pi_{dep} \leq \pi_{arr}$ and $A \neq B$.
- A *trip* can be seen as a sequence of connections c with the same *trip_id* $< c_1, c_2, \dots, c_j >$, such that

$$\forall i \in [1, j-1] : \; c^i_{\text{arr\_station}} = c^{i+1}_{\text{dep\_station}} \; \wedge \; c^i_{\text{arr\_time}} \leq c^{i+1}_{\text{dep\_time}}$$

- *Transfers* describe a possibility for the user to move between stations, written as triple (A, B, π_{dur}), with A being the departure station, B the arrival station and $\pi_{\text{dur}} \geq 0$ the transfer duration. Transfers could be walking (*footpaths*), or using a public bicycle or e-scooter (Bicycle or Scooter sharing system).
- A *journey* describes a "path" in the network, essentially a sequence of connections and footpaths (a user can hop on / off a train and walk between stations in order to transfer).

Example. The network in Figure 1 consists of 6 stations, two *trips* (with *trip_id* "red" and "green"), 4 connections (e.g. $(A, C, 10{:}00, 10{:}02, red)$) and 1 footpath $(C, C', 30)$. The *journey* e.g. from A to E would be the connection $(A, C, 10{:}00, 10{:}02, red)$, the *transfer* $(C, C', 30)$ and the connection $(C', E, 10{:}04, 10{:}06, green)$.

[1] General Transit Feed Specification - https://en.wikipedia.org/wiki/General_Transit_Feed_Specification
[2] https://www.deutschebahn.com/resource/blob/6066940/3d1c3864381befc7b3f3ea7b9a675922/DuF2020-data.pdf

Author's address: Patrick Steil, patrick.steil@stud.uni-heidelberg.de, Heidelberg University, Im Neuenheimer Feld 205, Heidelberg, Baden-Württemberg, Germany, 69120.

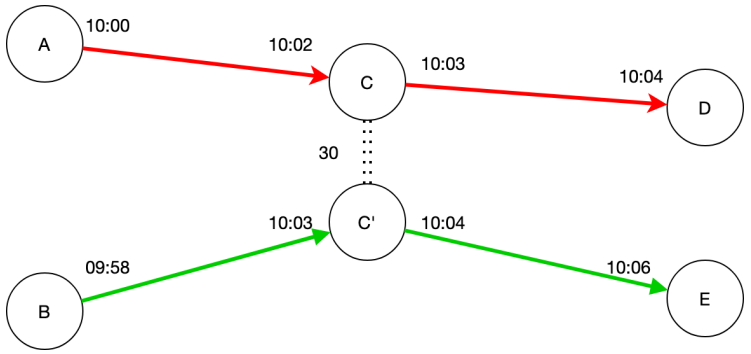

Fig. 1. Example network

3 CONNECTION SCAN ALGORITHM

In contrast to Dijkstra, A Star etc., CSA does not work on a graph structure, but rather on a list with all connections sorted by their departure time. The underlying principle for all variants of CSA is dynamic programming, i.e. the invariant that at any given time the current solution is always the optimal solution with respect to the previously scanned connections.

3.1 Earliest Arrival Problem

The **earliest arrival problem** describes the question of the earliest possible arrival time, i.e. the arrival time alone is optimised.

In the simplest variant, only connections are taken into account, meaning transfers and footpaths are not used. As input, the algorithm receives the departure time, the start stop and the destination stop. The algorithm returns the optimal arrival time.

The algorithm uses a hashmap S to identify stations with arrival times. The main idea is the following: We loop over all the connections in increasing departure time. When we scan a connection $c = (A, B, \pi_{\text{dep}}, \pi_{\text{arr}})$, we check whether the stored value $S[A] \neq \infty$. If this condition is true, it means that we were able to travel to station A with a previous connection (since we check connections in ascending departure time), and thus we can continue from A with the current connection. We then update the value in $S[c_{\text{arr\_stop}}]$ if the connection arrives earlier than currently stored. See pseudocode 1.

Algorithm 1 CSA - Earliest Arrival Problem - without transfers

```
1:  procedure EARLIEST_ARR(departure, target, dep_time)
2:      init map S <Station, Time>
3:      ∀A ∈ stations :  S[A] = ∞
4:      S[departure] = dep_time
5:      for connection c in sorted_connections do
6:          if c can be reached then
7:              S[c_arr_stop] = min {c_arr_time, S[c_arr_stop]}
8:          end if
9:      end for
10:     return time S[target]
11: end procedure
```

If we allow transfers between stations, we need to incorporate the duration of one transfer into S, if walking is faster than the current "best" arrival time. See pseudocode 2.

Algorithm 2 CSA - Earliest Arrival Problem - with transfers

```
1:  procedure EARLIEST_ARR(departure, target, dep_time)
2:      init map S <Station, Time>
3:      ∀A ∈ stations :  S[A] = ∞
4:      S[departure] = dep_time
5:      ∀ footpath from departure to B :  S[B] = dep_time+f_dur
6:      for connection c in sorted_connections do
7:          if c can be reached then
8:              S[c_arr_stop] = min {c_arr_time, S[c_arr_stop]}
9:              for footpath f from c_arr_stop do
10:                 S[f_arr_stop] = min {c_arr_time + f_dur, S[f_arr_stop]}
11:             end for
12:         end if
13:     end for
14:     return time S[target]
15: end procedure
```

Obviously, this algorithm is very inefficient, since we scan every connection. Thus, one can use the following techniques to drastically reduce the number of scanned connections.

(i) We do a binary search on the sorted connections to find the first one to depart at the given time.
(ii) We can stop as soon as we scan a connection departing later than the current "best arrival" time stored in S[to_id].
(iii) If our target is reachable via a footpath, the algorithm can finish after just one scan over all the outgoing footpaths from departure.
(iv) If we assume *transitivity*[3] of the transfer model, we only need to look at the outgoing footpaths from $c_{\text{arr}_s\text{top}}$, if (in line 8) we updated $S[c_{\text{arr\_stop}}]$. The proof can be found in [1].

In contrast to Dijkstra, CSA scans significantly more connections. But since it operates on an array, and not on a priority queue, access time is very low. The use of an array reduces cache-misses drastically, hence CSA is very fast.

One disadvantage is that parallelisation is difficult or impossible to realise. In addition, unnecessary connections that cannot be reached are scanned and thus take up valuable time.

Because a user is not only interested in the optimal arrival time, but rather in the journey on how to arrive at the target, we discuss a method to extract the journey information using the basic CSA. Different from the paper, I implemented the basic CSA using a hashmap S, which maps connections to stations. In other words, the value stored in the hashmap is the final connection, on how one can reach the station. Extraction is based on working your way "backwards" using the connections departure stations, starting from the target station until the departure station is found. See pseudocode 3.

3.2 Earliest Arrival Profile Problem

A stations' profile is a function, which maps departure time to arrival times (given departure and arrival station). See Figure 2 as an example. To store the profile function, a list with departure and

[3] $\forall A, B, C : A \xrightarrow{\text{transfer}} B \wedge B \xrightarrow{\text{transfer}} C \implies A \xrightarrow{\text{transfer}} C$

Algorithm 3 Journey Extraction

```
1: procedure EXTRACT_JOURNEY(departure, target, S)          ▷ S stores connections
2:     init result
3:     current_station = target
4:     while S[current_station]_dep_station is not departure do
5:         add S[current_station] to result
6:         update current_station with S[current_station]_dep_station
7:     end while
8:     add departure to result
9:     reverse result
10:    return result
11: end procedure
```

arrival times is used. The profile function is partially constant and monotonically increasing. If you want to insert new tuples, you have to check whether the new tuple may be inserted or whether other (already existing elements) have to be removed. This process is called "incorporate" in all pseudocodes. Queries with boundaries, such as a minimum departure time or the latest arrival time, can be solved using profiles.
The **earliest arrival profile problem** asks for the profile of the given departure station. The basic profile algorithm is shown in pseudocode 4.

Algorithm 4 CSA - Earliest Arrival Profile Problem

```
1: procedure EARLIEST_ARR_PROFILE(departure, target)
2:     init map S <Station, List<Time, Time>>      ▷ Hashmap with profiles mapped to stations
3:     ∀A ∈ stations : S[A] = {∞, ∞}
4:     init map T <Trip_ID, Time>
5:     ∀t ∈ trip_ids : T[t] = ∞
6:     for connection c in sorted_connections decreasing order do
7:         ▷ Calculate "best" arrival time
8:         τ_1 time when walking to the target
9:         τ_2 time when using the connection
10:        τ_3 time when transferring at c_arr_station
11:        τ_c = min{τ_1, τ_2, τ_3}
12:        ▷ Use τ_c to update all the stations' profiles, from which one can reach c_dep_stop
13:        Incorperate τ_c into the c_dep_station
14:        for footpath f with target c_dep_stop do
15:            Incorperate τ_c into S[f_dep_stop]
16:        end for
17:        T[c_trip_id] = τ_c                          ▷ Update trip arrival time
18:    end for
19:    return time S[target]
20: end procedure
```

CSA yields sometimes a "weird" journey, no one would take to just optimize the arrival time by 1 minute. The reason for this behaviour is simple: We only optimize the arrival time. But since users prefer journeys with fewer transfers but "similar" arrival times, we have to adapt the CSA.

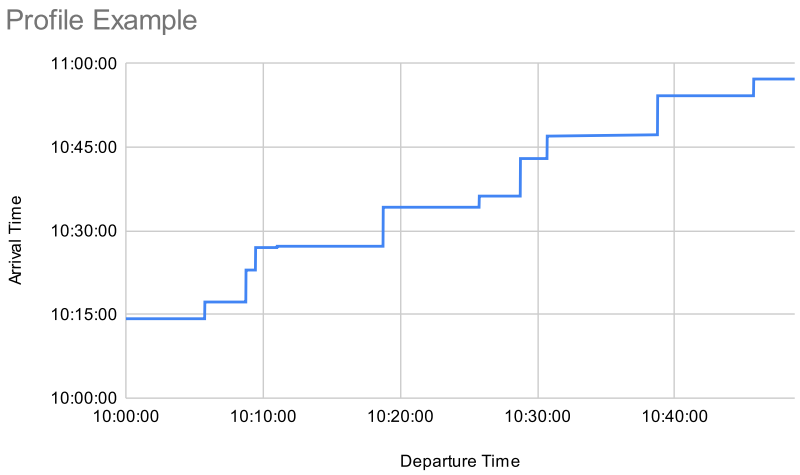

Fig. 2. Profile Function - This plot shows the profile for the query “Bismarckplatz” to “Bunsengymnasium” between 10:00 and 11:00 based on the RNV data (download date: 15.10.2021)

The basic idea is as follows: We round the arrival times at the destination to e.g. 5 minutes and choose the journey with the fewest changes. This way we get journeys that arrive at approximately the same time but require fewer transfers. However, this only works for journeys that are less than 5 minutes apart in arrival time.
We implement “rounding” by exploiting the properties of an integer representation. An integer is stored with 32 bits, and since we store a maximum of 48 hours as 172800 seconds, we only need 18 bits. I.e. we have 14 bits available to store the number of changes. Integers are internally compared from “left to right” (i.e. higher-order bit to lower-order bit). We take advantage of this to split the “arrival-time” integer as follows (see Figure 3).

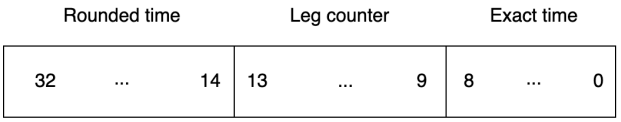

Fig. 3. Integer Bit Scheme

We use the 5 bits from 9-13 to encode the leg counter, the upper bits from 14 - 32 to save the rounded arrival time. This means that when a comparison is made, the rounded arrival time is compared first, and if this matches, the number of transfers is compared.
An important note: Only the arrival time at the destination is rounded, not intermediate arrival times. This way we maintain the correctness of the algorithm.
Another important aspect is the reconstruction of the exact arrival time, because a user wants to know when exactly the destination will be reached. But this is easy to realize with the use of the bit transformation. In order to extract the journey, not only are tuples of departure and arrival times stored in the profiles, but also additionally two connections. The first connection describes the “boarding” of a train, the second the “exiting”. In this way, we can reconstruct the journey in the same way as in the basic version (using arrival stations until we reach the departure).

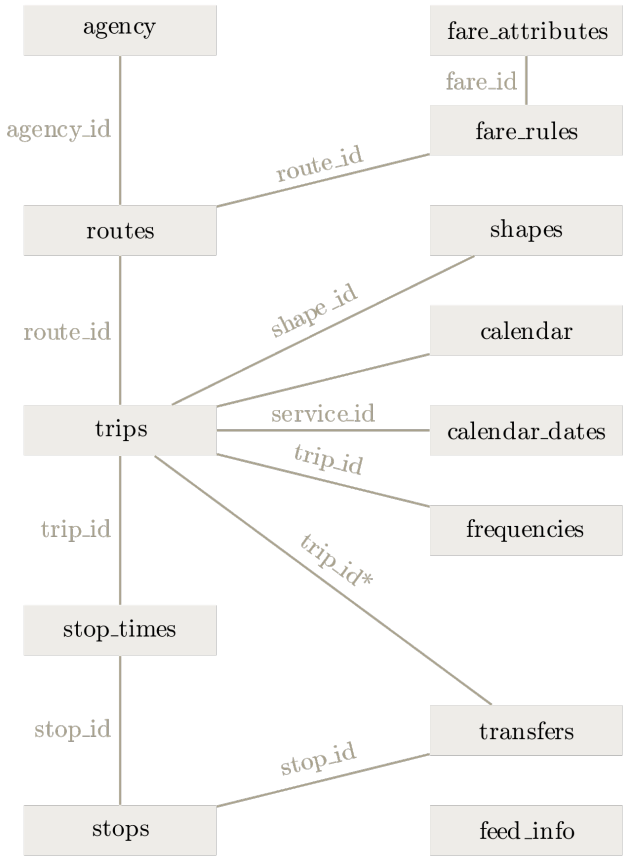

Fig. 4. Class diagram of GTFS - https://en.wikipedia.org/wiki/General_Transit_Feed_Specification

4 GTFS

General **T**ransit **F**eed **S**pecification (GTFS) is a format for public transportation. Google invented this file format, as there was previously no standardised file format for public transport in the USA (hence the original name *Google Transit Feed Specification*) [4]. GTFS consists of several txt files, such as "routes.txt" or "stops.txt". Each file has its own headers that defines the properties of the file's contents. For example, a header of "stops.txt" might look like this:

```
stop_id,stop_name,stop_lat,stop_lon
```

The relationship between the individual files is shown in Figure 4.

The structure of the "stop_times.txt", in which all connections are located, should be emphasised. One line contains, as defined by the header, the trip_id, stop_id, arrival time, departure time and additional information that CSA does not need. My parser reads line by line and generates a connection from 2 consecutive lines if they share the same trip ID. It is easier with the stops: A stop can be generated from one line.

[4] https://googleblog.blogspot.com/2006/09/happy-trails-with-google-transit.html

4.1 Creating missing tranfers.txt

Each GTFS dataset must contain the following files: agency, stops, stop_times, routes, trips, calendar (or calendar_dates). Since the file transfers is not mandatory, it is often not available in public data sets. The same applies to the data of Rhein-Neckar-Verkehr GmbH (rnv) [5]. Therefore, my parser creates a transfer file which is defined with the following header:

```
from_stop_id,to_stop_id,transfer_type,min_transfer_time
```

The first idea to create the transfer file from n stations is to calculate the distances to all other stations for each station and keep only the closest ones and calculate the transfer time with them. But this algorithm is in $O(n^2)$, and therefore not usable for large n. In order to efficiently generate the file from a list of stations, the parser uses a 2d tree, a space-partitioning data structure. See Figure 5 for an example of a partitioning of the 2 dimensional space.

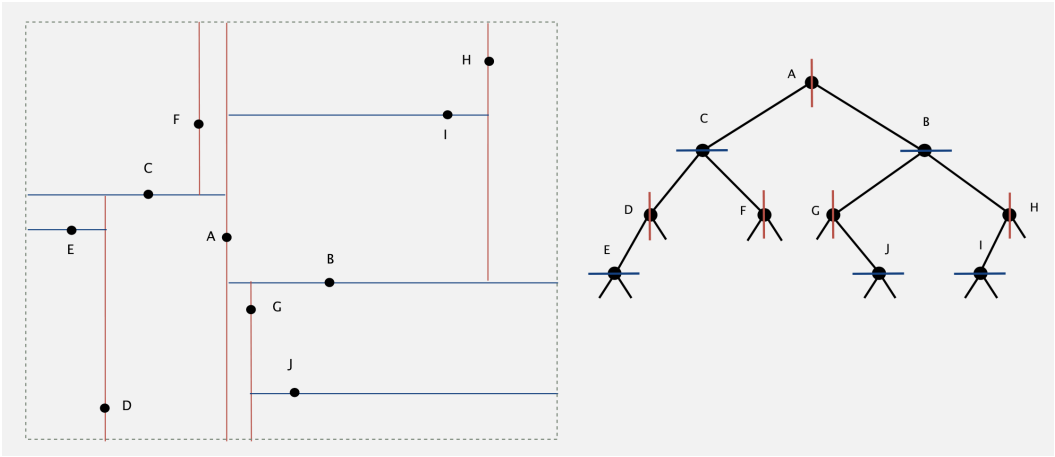

Fig. 5. Example of a 2d-Tree - Given 10 points $(A, B, C, \ldots, J)$ with two coordinates each, the 2d tree bisects the "space" by selecting point A as the pivot for the first coordinate (see the central red line). A suitable pivot is the median, because in this way an approximate halving of the data size is achieved in each step. In the next step, the 2d tree separates the half "right" of the red line (i.e. all points whose first coordinate is greater than A) at the median B using the second coordinate. In this way the tree constructs itself recursively. - See https://www.cs.princeton.edu/courses/archive/spring18/cos226/demos/99DemoKdTree.pdf#page=4 for more information and further operations

If no transfer file is available, a 2d tree is generated at the same time as the stop file is read in. Given the parameter distance and average walking speed $\delta, s \in \mathbb{R}$, nearest neighbour queries can be answered efficiently to find all stations within range of δ metres for a given station (see Figure 6). Together with the given average travel speed of $s\ \frac{m}{s}$, "optimistic" transfer times can be calculated. "Optimistic", since in this calculation the footpath is approximated by the air line. In reality, however, this is usually not true, as e.g. houses or roads could be in the way. Therefore, the actual footpath is longer than calculated here. Moreover, the footpaths are not transitively closed. Finding the neighbouring stations at a given radius is in $O(\log n)$, therefore the whole algorithm is in $O(n \log n)$. A significant improvement on the first idea.

One advantage of this method is the efficient parallelization. Since we need to find all reachable neighbours for each station and the process of finding the neighbours of two stations does not interfere with each other, we can split our list of stations and run the algorithm in parallel.

[5] https://opendata.rnv-online.de/dataset/gtfs-general-transit-feed-specification

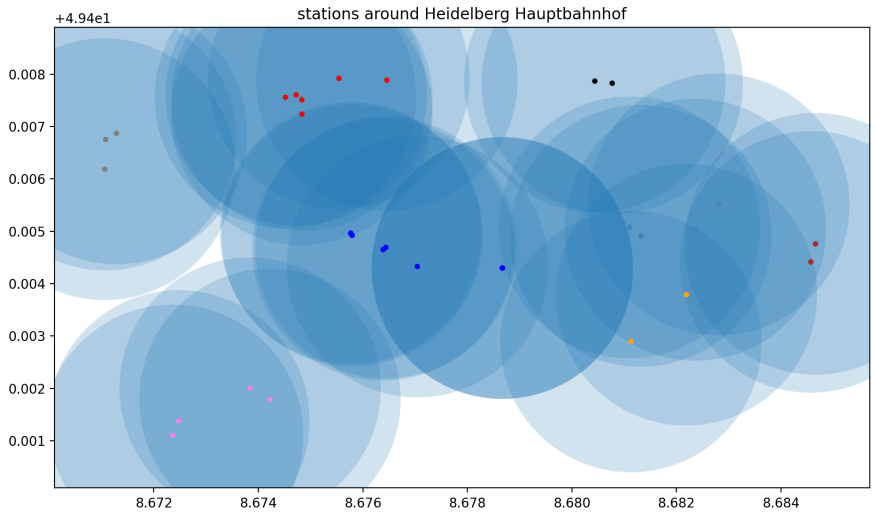

Fig. 6. Example plot of the area around Heidelberg Hauptbahnhof - Blue circles indicate the search radius of δ metres around every station. Same stations but different tracks have the same color. For each centre point of a circle (a station), a transfer is calculated to each station inside the blue circle. In this way, not only transfers between different tracks but the same stops are created, but also different stops (as can be seen on the far right of the picture with the red and grey dots).

5 EXPERIMENTS

The following experiments were conducted on the 107 Algorithm Engineering Server (16 cores and 100GB RAM), the underlying dataset is from the RNV (2101 stations and 291113 connections). The C++ code has been compiled using the g++ compiler with -O3 as optimizations option and -fopenmp for all parallelizations. To see the code, check out the Github repository [6].
Not only pure querying, but also journey extracting was carried out for all experiments.
First, the basic variant (only optimising the arrival time) was run 1000 times at random start and end stops. The result is shown in Figure 7, with an average running time of 0.847429 ms. As a comparison, the average runtime of the underlying paper was 1.3 ms on the London dataset of size 20843 stops and 4850431 connections over 100 random queries ($\approx$ 10 times more stations and $\approx$ 16 times more connections). *Note*: I use the RNV data because the London data used in the paper is not in GTFS format and no similar sized GTFS datasets were found from London.

[6] https://github.com/PatrickSteil/ConnectionScanAlgorithm

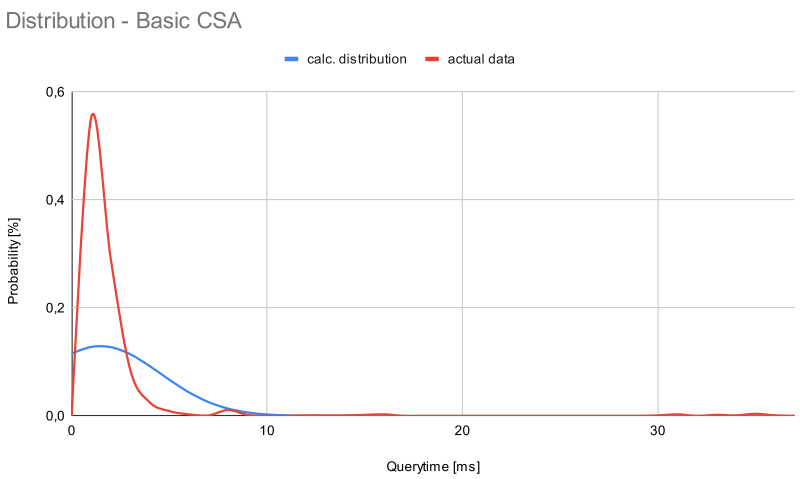

Fig. 7. Plot of the basic variant for 1000 random queries with departure time 00:00. The measured runtimes and the normal distribution characterised by them are shown.Minimum: 0.124548 ms, Maximum: 35.2495 ms

Next, the results of the profile variant (with footpaths) are considered (without leg optimisation). First, experiments were conducted on an 8-hour range (10:00 - 18:00), and then (to get a realistic picture) on a 4-hour range. The results of the first experiment are shown in Figure 8, with an average running time of 94.56 ms. The paper, however, achieves an average time of just under 9.4 ms on the London dataset, although it should be mentioned here that not all improvement techniques such as prefetch, limited walking or source domination were implemented.

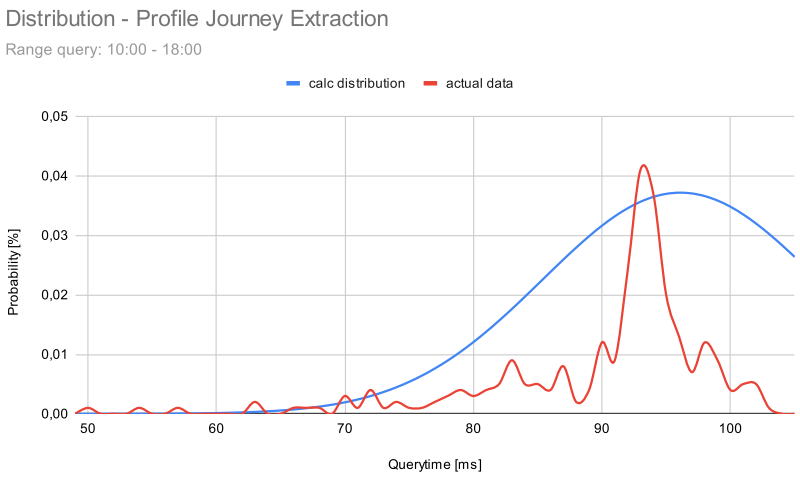

Fig. 8. Plot of the profile variant for 1000 random queries with a time range from 10:00 to 18:00. The measured data and the normal distribution characterised by it are shown. Minimum: 49.41 ms, Maximum: 116.75 ms

Using more realistic ranges of 4 hours, the variant achieves an improvement in runtime of just ≈ 43% to an average runtime of 42 ms.
Last but not least, the Leg-Optimization-Profile variant is tested, also once with an 8-hour range and once (again for realistic purposes) with a 4-hour range. The results are shown in Figure 9. Again, the average running time decreases by halving the range from 93.2 ms to 43.87 ms (≈ 47%). We see a slight improvement in the running time by optimising the changeovers. This is due to journey extraction: fewer connections need to be considered to recover the journey.

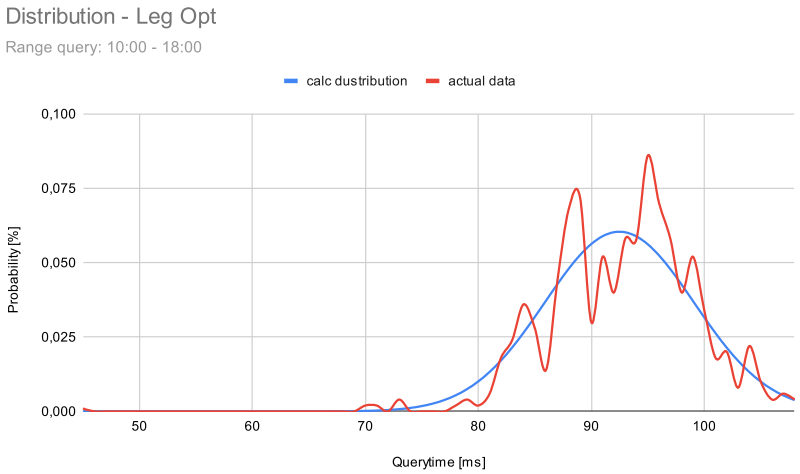

Fig. 9. Plot of 1000 random queries to the Leg-Optimisation-Profile variant. The measured data and the normal distribution based on the data are shown. The selected time range is from 10:00 - 18:00. Minimum: 42.38 ms, Maximum: 109.06 ms

Creating the transfer file for the RNV data took 12.6783 ms (remember: the dataset had 2101 stations). As you can see: parallelisation pays off.

6 FURTHER INFORMATION

Beyond this internship, improvement techniques should be implemented, which were described in the paper (prefetch, source domination, CSA-Basic & CSA-Profile combined, etc.). In addition, the journey extraction can be optimised by exploiting the fact that the profile function can be reduced when evaluating a profile, i.e. all entries in the list can effectively be deleted until the time found. This would drastically reduce subsequent queries of this profile. Another point is the filtering of connections that only run on certain days. This can be realised by storing flags for each connection that represent the availability on a day. This would ensure the correctness of the algorithm in a real context.

REFERENCES

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