

Undergraduate Thesis

PublicTransitSnapper: Map Matching Mobile Phones to Public Transit Vehicles in Real Time

Gerrit Freiwald

Examiner: Prof. Dr. Hannah Bast

Adviser: Dr. Patrick Brosi

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair of Algorithms and Data Structures

October 25th, 2022

Writing Period

25.07.2022 – 25.10.2022

Examiner

Prof. Dr. Hannah Bast

Adviser

Dr. Patrick Brosi

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I hereby also declare that my Thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Freiburg, 25.10.2022

Place, Date

Gerit Freiwald

Signature

Abstract

We match location data from a mobile phone to the most likely public transport vehicle, while using all information available in Google's public transit schedule format GTFS. We explain how we succeeded in creating a backend that is capable of performing a 'Public Transport Vehicle Matching' algorithm and how we built a user-friendly web- and mobile app. We analyze our conducted user study to test the app in Freiburg, Hamburg, Munich and Zurich. The results suggest that our app works especially well with busses, but has room for improvement with underground services. As there is a lack of publicly available real time data on public transport vehicles in Germany, we discuss its current state and improvement prospects conducted by German authorities.

Zusammenfassung

Wir ordnen Ortsangaben von Handys ihrem wahrscheinlichsten öffentlichen Transportmittel zu. Dafür benutzen wir sämtliche verfügbare Informationen in Googles GTFS-Format für Fahrplandaten. Wir erläutern, wie wir es geschafft haben ein Back-End zu erstellen, auf dem ein Algorithmus zur ‘Zuordnung zu öffentlichen Transportmitteln’ läuft. Außerdem erklären wir, wie wir unsere benutzerfreundliche App als Web-App und Handyapp gebaut haben. Wir analysieren unsere Nutzerstudie, bei welcher wir unsere App Testern in Freiburg, Hamburg, München und Zürich gegeben haben. Die Ergebnisse zeigen, dass unsere App besonders gut bei Bussen funktioniert, bei S- und U-Bahnen aber schwächelt. In Deutschland gibt es kaum öffentlich verfügbare Echtzeitdaten für öffentliche Transportmittel. Deshalb untersuchen wir den aktuellen Stand der Dinge und beschreiben geplante Verbesserungen durch Bund und Länder.

Contents

1	Introduction	1
2	Background	5
2.1	Introduction to GTFS	5
2.1.1	Term Definitions	5
2.1.2	Introducing the GTFS Tables Used by PublicTransitSnapper	5
2.1.3	Representing Shapes	8
2.1.4	GTFS Real Time Extension	8
2.1.5	Further Information on GTFS	8
2.2	Introduction to Public Transit Vehicle Matching	10
2.2.1	Great Circle Distance	10
2.2.2	Definition Map Matching	10
2.2.3	Map Matching to a Dynamic Map	10
2.2.4	Using a Graph to Represent the Shapes Network	10
2.2.5	Definition Markov Chains	11
2.2.6	Hidden Markov Models	11
3	Related Work	13
3.1	Other PTV Matching Applications	13
3.1.1	Google's Pigeon Transit Project	13
3.1.2	Citymapper	13
3.2	Map Matching to a Static Map	14
3.2.1	Hidden Markov Models in Map Matching	14
3.2.2	Real Time Map Matching	14
3.3	Public Transit Data in Germany	14
3.4	Working Together on PublicTransitSnapper	15
4	Building PublicTransitSnapper	17
4.1	Problem Definition	17
4.2	Project Structure	17
4.2.1	Used Programming Languages	17
4.3	The Frontend	20
4.3.1	App Layout	20
4.4	Communication Between Frontend and Backend	24
4.4.1	Handling HTTP Requests and Transferring Information on the PTV Matching	24

4.4.2	Communication Needed for the Connections Page	25
4.4.3	Communication Needed for Displaying the Shape	25
4.4.4	The Chat	25
4.5	The Backend	26
4.5.1	Docker	27
4.5.2	Connecting Shapes, Stops and Times	27
4.5.3	Preparing and Using the GTFS Files	29
4.5.4	TripsWithStopsAndTimes Methods	31
4.5.5	GTFS Container Methods	32
4.6	PTV Matching	34
4.6.1	Using a Hidden Markov Model to Find the Most Likely Edges	34
4.6.2	Assigning Weights to the HMM	36
4.6.3	Finding the Most Likely Trip	38
4.6.4	Avoiding Over-Matching	39
4.7	Fetching New GTFS Data	43
4.8	Using Fake GPS Data to Test the Project	43
4.8.1	Generating Noisified Points Along a Shape	44
4.8.2	Annotating the GPS Points With Timestamps	44
4.8.3	Using Selenium to Manipulate a Device's GPS Position	45
5	User Study	47
5.1	General Inquiry	47
5.2	Trips	48
5.3	Differences and Similarities	54
5.4	Conclusion of the User Study	54
6	GTFS in Germany	57
6.1	Availability of GTFS in Germany	57
6.1.1	DELFI	57
6.1.2	DEEZ - Real Time Data Throughout Germany	62
6.2	Prominent German Industrial Standards	64
6.2.1	HAFAS	64
6.2.2	DIVA	64
6.3	Concluding GTFS in Germany	65
7	Further Investigation	67
7.1	Improving the Frontend	67
7.1.1	Showing Real Time Updates	67
7.1.2	Showing Connection Issues	67
7.1.3	Offline Maps	67
7.1.4	Stop Features	68
7.2	GTFS Frequencies	68
7.3	City-Specific Information	68
7.4	Generating Real Time Data	69

7.5	The Debatable Use of Python	69
7.6	Testing PublicTransitSnapper on Long Distance Trips	70
8	Conclusion	71
9	Acknowledgments	73
10	Appendix	75
10.1	Data Structures	75
10.1.1	STRtrees	75
10.2	Algorithms	75
10.2.1	Dijkstra's Algorithm	75
10.3	GTFS Example	77
	Bibliography	82

List of Figures

1	<i>PublicTransitSnapper</i> Preview	1
2	Map Matching Illustration	2
3	Freiburg Shape Network	3
4	Relations Between GTFS Files	9
5	Markov Chain Example	11
6	Transition Probability Example	12
7	Front Page and Connections Page	21
8	Map Page	22
9	Chat Page and Settings Page	23
10	Example of Trip Segments	28
11	Example for <code>get_next_stop</code>	33
12	Looped Shapes	34
13	Example G_{network} and G_{markov} Graphs	35
14	Opposite Directed Shapes Example	37
15	Shared Shapes Example	38
16	Find the Most Likely Trip	39
17	Anti-Matching Illustration	41
18	Anti-Matching Counterexample	42
19	Time Interpolation Example	45
20	ControlDevTools Flowchart	46
21	Agencies Germany	58
22	‘Bayern Fahrplan’ Partner Agencies	59
23	DELFI Organigram	61
24	DEEZ Two Regio-Clusters	63
25	R-Tree	76

List of Tables

1	User Study Freiburg 1	48
2	User Study Freiburg 2	49
3	User Study Hamburg 1	49
4	User Study Hamburg 2	50
5	User Study Munich 1	51
6	User Study Munich 2	52
7	User Study Zurich 1	52
8	User Study Zurich 2	53

List of Listings

4.1	Content of a Chat Message	26
10.1	Pseudo-Code Dijkstra's Algorithm	77
10.2	GTFS Example routes.txt	78
10.3	GTFS Example trips.txt and calendar_dates.txt	78
10.4	GTFS Example stops.txt	78
10.5	GTFS Example stop_times.txt	79
10.6	GTFS Example calendar.txt	79
10.7	GTFS Example shapes.txt	80

1 Introduction

Public Transit Vehicle Matching (PTV Matching) is a contemporary issue of finding the most likely public transit vehicle (PTV) given a list of timestamped GPS points. As it currently stands, there are apps like Citymapper¹ and Moovit² that are presumably matching GPS points of their users to PTVs in real time. However, their algorithmic approaches are company-internal and, to the best of our knowledge, there are no open source projects on that matter. We aim to fill this gap.

Our app *PublicTransitSnapper* (see Figure 1) collects said timestamped GPS points and sends them to a backend, where a PTV Matching and other information requested by the app is calculated. We use GTFS (General Transit Feed Specification) to get information on the PTVs' schedules and moving trajectories called *shapes*.

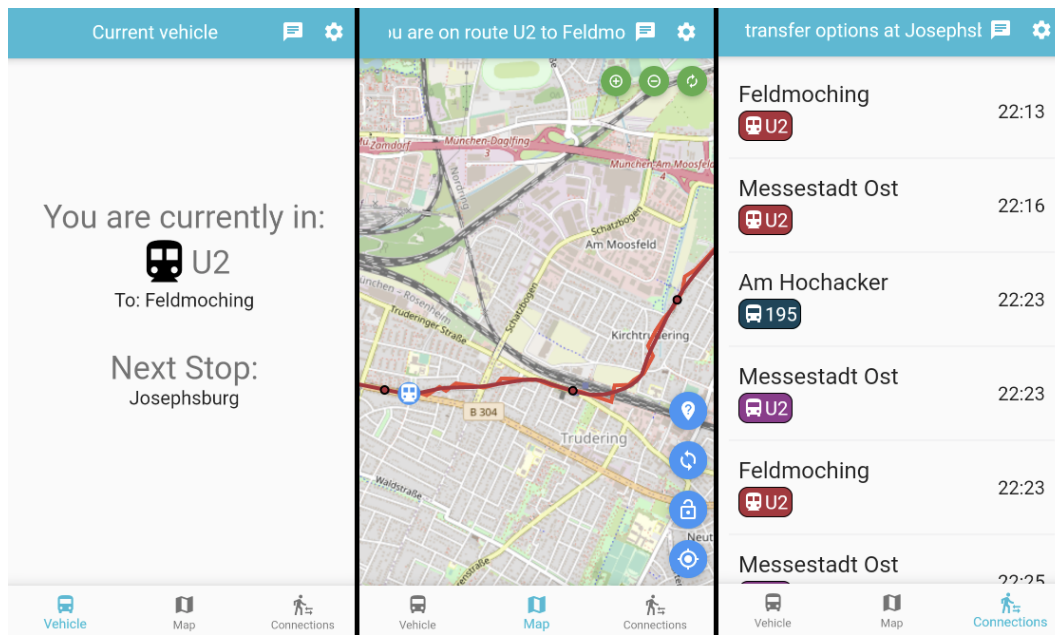


Figure 1: This figure shows a preview of the *PublicTransitSnapper* app. The user has currently been matched to the U2 in Munich.

¹<https://citymapper.com>

²<https://moovitapp.com>



Figure 2: This figure illustrates map matching. The left image shows GPS points with a natural inaccuracy. The image in the middle illustrates that a simple metric like always choosing the closest street is not a valid option when it comes to map matching. The image on the right shows a desirable map matching for the given coordinates.

After explaining the functionalities of the visual app, we discuss our approach on a PTV Matching implementation. In order to calculate the PTV Matching, we need to connect two bits of information: The geographic position of a PTV and the time at which the PTV is at that position, relative to the sent mobile phone coordinates. The geographic part can be solved with a map matching approach. Map matching is a well researched topic. Figure 2 illustrates the map matching of the GPS points received by a car. Whenever position measurements are too imprecise, we can use map matching to displace a measured position to a position within a network. This could be a road network like in Figure 2 or, like in our application, the network of shapes PTVs move on (see Figure 3 for an example). Imagine a collection of all shapes of currently active PTVs within a region close to the coordinates collected by a mobile phone user. Now, we only have check where the PTVs currently are along the shape. The closest vehicle moving along a shape that fits to the shape we found with our map matching is the most likely candidate.

For testing our real time GPS data based app, we wrote a testing tool that simulates a user device moving along a shape. The simulated GPS points are noisified and then given to *PublicTransitSnapper*.

We evaluate the usability of *PublicTransitSnapper* with our conducted user study. While the usability and bus matchings were praised, *PublicTransitSnapper* had certain issues with subways.

Many mismatches are induced by a lack of real time PTV data. Therefore, we discuss the current state of publicly available data on PTVs in Germany and give insight into projects that aim to improve the situation. One of these projects is called DEEZ, which translates to ‘real time data throughout Germany’.

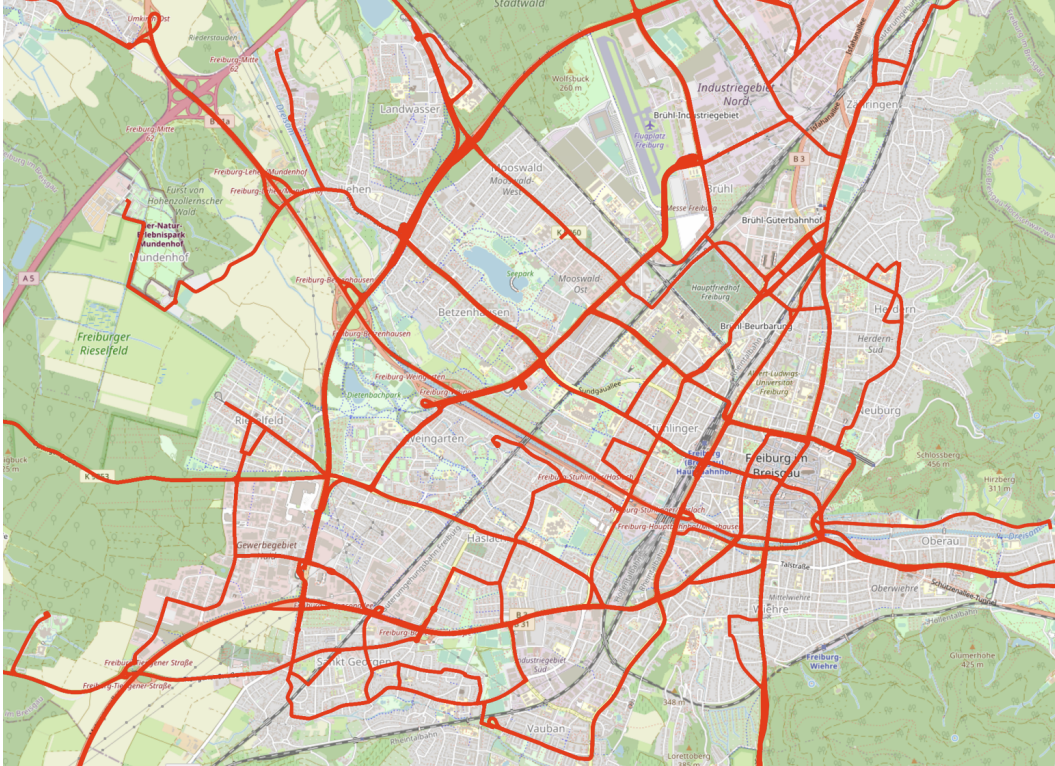


Figure 3: This Figure shows the shape network of PTVs in Freiburg as red lines. PTVs only move along the shapes. Thus, the street network is not relevant for map matching to the shape network.

In what follows, the ‘Background’ Chapter 2 introduces the GTFS format and other knowledge required for the reader to understand this thesis. The ‘Related Work’ Chapter 3 puts the content of this thesis into context with existing research and projects on related subjects. The ‘Building PublicTransitSnapper’ Chapter 4 gives detailed explanation of the functionality of the visual app (the frontend), as well as a thorough specification on the functionality of the PTV Matching and other required features (the backend). Here, we also introduce testing tools like the previously mentioned GPS simulator. The ‘User Study’ Chapter 5 discusses evaluation results collected by *PublicTransitSnapper* testers in Freiburg, Hamburg, Munich and Zurich, to have a comparison with a Swiss city. The ‘GTFS in Germany’ Chapter 6 elucidates the state of publicly available static and real time data on PTVs in Germany and discusses current projects aiming to improve the situation. The ‘Further Investigation’ Chapter 7 gives prospects on future work on mentioned problems. Chapter 8 concludes.

PublicTransitSnapper is an open source project which can be found on GitHub³.

³<https://github.com/TheRealTirreg/PublicTransitSnapper>

2 Background

In this chapter, we introduce Google’s ‘General Transit Feed Specification’ (GTFS), as well as the concept of map matching.

2.1 Introduction to GTFS

The ‘General Transit Feed Specification’ (GTFS)¹ was created by Google in 2005 [1]. A GTFS feed consists of at least six and up to 17 CSV files (with extension .txt) contained in a ZIP file. Each CSV file describes a table. All tables together describe a relational database². In a nutshell, this means that the tables are linked together, which we will explain in the following.

2.1.1 Term Definitions

To understand GTFS, we first need to learn about terms used in this context.

- **Field** - A column of a table. For example, ‘*route_id*’ is a field of the routes table
- **Field value** - An entry in a column of a table. For example, ‘*route001*’ would be an entry for the *route_id* field
- **Color** - Colors are encoded as a six-digit hexadecimal number. For example: ‘*FFFFFF*’ represents white, ‘*000000*’ represents black
- **Date** - Dates are written in a ‘YYYYMMDD’ format. For example, ‘20220805’ describes August 5, 2022
- **Time** - Times are written in a ‘HH:MM:SS’ format. For example, ‘14:30:00’ describes 14:30h. ‘25:35:00’ means 1:35h on the next day. I will further explain this feature in the next section.

2.1.2 Introducing the GTFS Tables Used by PublicTransitSnapper

In the following, we will name every required fields (column names) and the optional fields we used for the *PublicTransitSnapper*. You can see the relations between the tables in Figure 4.

¹<https://developers.google.com/transit/gtfs>

²<https://cloud.google.com/learn/what-is-a-relational-database>

routes.txt

This table contains every route. Every route has:

- A unique `route_id`
- A `route_short_name` which we mostly use
- A `route_long_name` which is irrelevant for the *PublicTransitSnapper*
- A `route_type` as a number, for example 0 describes a Tram, 1 a Metro, 2 Rails and 3 a Bus
- Optionally a `route_color` which we use in the frontend
- Optionally a `route_text_color` which we also use in the frontend

trips.txt

Every route has one or more trips. Every trip has:

- A unique `trip_id`
- A ‘parent’ `route_id`. Every route can have multiple trips, but every trip can only have one route
- A `service_id` which gets its meaning from the calendar table
- A `shape_id` which is representing a list of GPS points, defined in the shapes table

stops.txt

This table describes the geographical locations of each stop. Stops can have a ‘parent station’ comprising multiple stops that belong together. Every stop has:

- A unique `stop_id`
- A `stop_name`
- A `stop_lat` which describes the GPS latitude of the stop
- A `stop_lon` which describes the GPS longitude of the stop

stop_times.txt

This table links the trips to their stops. It tracks when a PTV should arrive at each stop according to schedule.

In the term definitions (see 2.1.1), we mentioned that times can overlap to the next day, for example ‘25:35:00’ would mean 1:35h on the next day. Let us consider the example date February 1st of 2022. Trips can start on that day at let us say 23:30h, and finish at 1:35h (‘25:35:00’) the next day. Trips can even start on ‘25:35:00’ for the February 1st of 2022 and end on ‘26:25:00’ for example. We will address this feature as ‘overtime’ from now on.

The table provides:

- A `trip_id`
- A `stop_id`
- A `arrival_time` which describes the arrival time of the PTV on the trip
- A `departure_time` which describes the departure time of the PTV on the trip
- A `stop_sequence` which describes the order in which the PTV on the trip visits the stops

calendar.txt

This table gives a meaning to the above mentioned `service_id`. Every trip has a service, but a service can provide information to multiple trips. Each service comprises:

- A `service_id`
- A Boolean telling if the corresponding trips are served on **Mondays**
- The same for **Tuesdays**
- **Wednesdays**
- **Thursdays**
- **Fridays**
- **Saturdays**
- and **Sundays**
- A `start_date` and an
- `end_date` which describe the validity period of the service

calendar_dates.txt

Calendar dates can provide exceptions to services. This can be used for reduced or extra service during a holiday for example. The table provides:

- A `service_id`
- A `date`
- An `exception_type` which is either 1 or 2. 1 means: Service has been added for the specified date. 2 means: Service has been removed for the specified date.

shapes.txt

This file is very important for the *PublicTransitSnapper*. Unfortunately, it is optional and many public transport associations choose not to provide the shapes file. Luckily, there is a tool for creating a shapes file called *pfaedle*. We will cover *pfaedle* in section 4.7. A shape describes the path that a PTV travels on. Every shape has multiple rows:

- A `shape_id`
- A `shape_pt_lat` describing the GPS latitude of a shape point
- A `shape_pt_lon` describing the GPS longitude of a shape point
- A `shape_pt_sequence` describing the order in which a PTV visits the points

2.1.3 Representing Shapes

Shapes are polylines, which is another word for a list of points. We can also think of polylines as linked edges. This representation is very useful for computation needed for the PTV Matching, which will be elucidated in the further course of this thesis.

If we take every edge from each shape, we can observe that there are many duplicate edges. We can eliminate these duplications by creating `edge` objects for each edge. An `edge` contains:

- A unique `edge_id`
- An `edge_start` GPS coordinate
- An `edge_end` GPS coordinate
- An `edge_length` containing a float of the length of the edge in meters
- A list that holds tuples of every `shape_id` this edge is part of, together with the sequence number of the edge in the shape: `(shape_id, edge_sequence_number)`

2.1.4 GTFS Real Time Extension

There is also a real time GTFS extension called GTFS-RT³. GTFS-RT is a stream providing real time information like delays, trip cancellations and more. GTFS-RT operates on top of a fitting static GTFS data set. Thus, other real time feeds are not compatible. As we will show later in the ‘GTFS in Germany’ Chapter 6, GTFS-RT is not common in Germany. This goes as far as GTFS-RT not even being mentioned on the German Wikipedia article on GTFS. However, Switzerland’s ‘open data platform mobility’⁴ does provide a GTFS-RT stream we could use for testing and implementing GTFS-RT support. Please see R. Wu’s thesis [2] for a more in-depth look on this subject.

2.1.5 Further Information on GTFS

We have visualized the relations between the tables in Figure 4. For an example GTFS feed, see this chapter in the appendix.

³<https://developers.google.com/transit/gtfs-realtime>

⁴<https://opentransportdata.swiss/en/cookbook/gtfs-rt/>

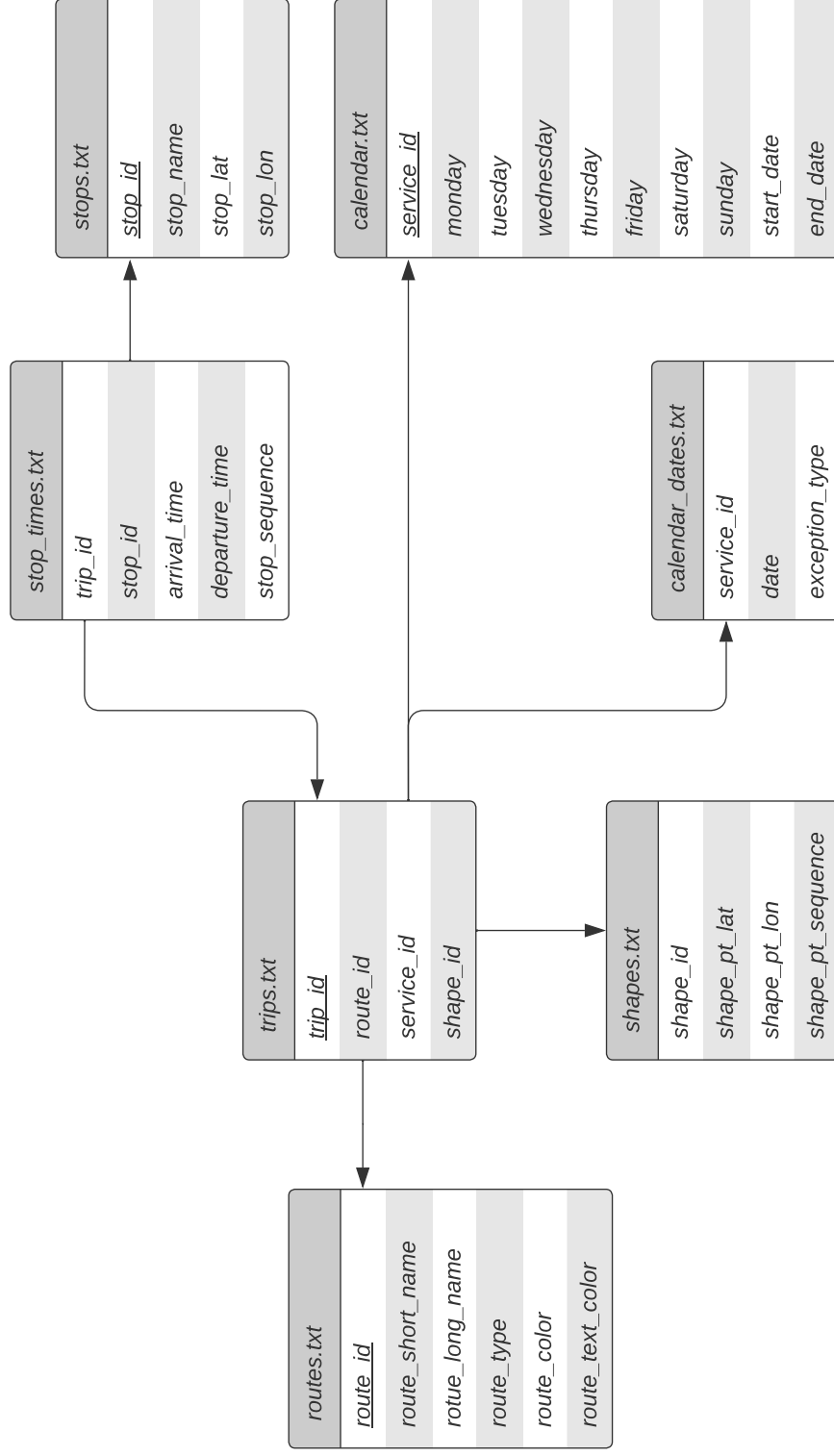


Figure 4: This info graphic shows the relations between the GTFS files. An arrow can be read as ‘describes’.
For example: “A **trip_id** describes a **trip_id**, a **route_id**, a **service_id** and a **shape_id**”

2.2 Introduction to Public Transit Vehicle Matching

2.2.1 Great Circle Distance

During this thesis, we are constantly dealing with geographical coordinates. As the earth is spherical, connecting two points while ignoring the earth's curvature can lead to inaccuracies. Therefore, we use the great circle distance (`gcd`) whenever we deal with distance between geographical coordinates or lengths of edges between two geographical coordinates. The distance between two geographical coordinates $p_1 = (\text{lat}_1, \text{lon}_1)$ and $p_2 = (\text{lat}_2, \text{lon}_2)$ can be calculated with the following formula:

$$\text{gcd}(p_1, p_2) = \arccos(\cos(\text{lat}_1) \cdot \cos(\text{lat}_2) \cdot \cos(\text{lon}_2 - \text{lon}_1) + \sin(\text{lat}_1) \cdot \sin(\text{lat}_2))$$

2.2.2 Definition Map Matching

With the help of map matching, we can assign geographical objects to locations on a map. Often, these geographical objects are positions received by sensors using positioning systems like GPS. In typical cases, the sensor is located in a car, a PTV or a mobile phone. Vehicles are often moving within a network made out of polylines that approximate roads or PTV paths. Map matching aims to project the generally inaccurate GPS positions onto the network's polylines.

2.2.3 Map Matching to a Dynamic Map

In our case, the digital map is not a static road network, but consists of independently moving public transit vehicles (PTVs).

The `shapes.txt` file from the GTFS data supplies us with polylines equivalent of those describing the center lines of the roads from a road network. Each trip from the GTFS data represents a PTV. The `stop_times.txt` file in connection with the `stops.txt` file give information on the location of the PTV at certain points in time (the stop times). We can interpolate the stop times to get an approximation on where the PTV could be along the shape at a given point in time. As PTVs can be early or delayed, we have to be tolerant when considering the PTV's position in time.

2.2.4 Using a Graph to Represent the Shapes Network

We represent the GTFS shapes as a directed graph G_{network} . Here, every GPS point of a shape is represented as a node and successive points of the same shape are connected with a directed edge. For a list C of GPS points with corresponding timestamp (`pt_lat`, `pt_lon`, `pt_time`), we want to find the most likely path in G_{network} that 'fits' to the points and timestamps.

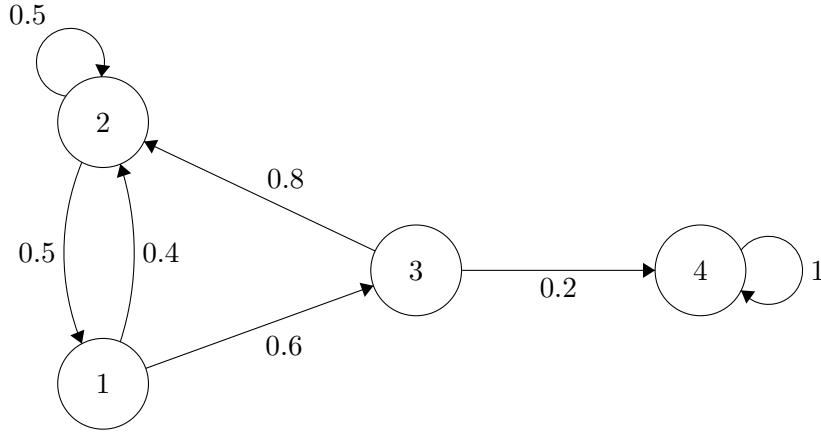


Figure 5: Each node in the graph represents a ‘state’. The numbers on the state-connecting arrows indicate the transition probability from one state to the next state. If a state does not have a transition probability arrow to another state, their transition probability is 0. The sum of outgoing transition probabilities must add up to 1 for every state. As an example: The probability to get from state 1 to state 2 in one step $P_{\text{OneStep}}(1 \rightarrow 2) = 0.4$.

2.2.5 Definition Markov Chains

A Markov chain is a sequence of possible states. Markov chains have the special characteristic that the probability of transitioning from one state to another is solely dependant on the current state. This probability is called ‘transition probability’.

Markov chains can be modelled by a directed graph, as can be seen in the example in Figure 5.

2.2.6 Hidden Markov Models

A Hidden Markov Model (HMM) is used when a process can likely be modeled by a Markov chain, but its states are unknown. These unknown states cause observable events. We can measure the observable events. As it is unclear, which of the unknown states has provoked the observable event, we introduce ‘hidden states’ for every observable event. The hidden states are linked by transition probabilities $P_{\text{transition}}$. The likelihood that a state generates the observed event is called ‘emission probability’ P_{emission} .

For example, we can imagine the states of the HMM as **edges**. The events would be represented by noisy GPS points. The transition probabilities can depend on many things. For example, edges that are not connected would have the transition

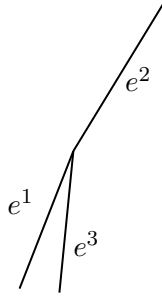


Figure 6: Example for a possible transition probability: $P_{\text{transition}}(e^1 \rightarrow e^2)$ should be higher than $P_{\text{transition}}(e^1 \rightarrow e^3)$, as it is unlikely, that a PTV makes such a harsh turn.

probability 0. Edges that are connected but point in opposite directions could have a lower transition probability than two directly connected edges pointing in a similar direction (see Figure 6). As an example for the emission probability, we could value the distance from a GPS point to an **edge**. The longer the distance between GPS point and **edge**, the lower the emission probability.

3 Related Work

3.1 Other PTV Matching Applications

We assume that matching mobile phones to PTVs in real time has been performed by multiple apps for a few years now. Sadly, none of the projects are open source, so we cannot be certain on this assumption.

3.1.1 Google’s Pigeon Transit Project

In 2019, Google’s Area 120¹ introduced Pigeon Transit [3]. The general idea of the project was to share real time information about changes in local public transit connections. They considered weather, delays, crowdedness, conditions of the vehicles and other information based on crowd-sourced data.

In order to calculate the delay of a PTV, it is likely that the mobile phones of the users have been matched to the individual PTVs. Delay and crowdedness could be derived based on the positions and the number of matched mobile phones.

Google decided to take down the project at the beginning of the coronavirus pandemic, as there were fewer commuters and many cities had to change their public transit system. At this point in time, Google has no plans to re-launch Pigeon Transit.

3.1.2 Citymapper

Citymapper² is a public transit app that displays transport options with real time data like PTV delay in over 150 supported cities. They use GTFS data³ and other sources for their PTV information. According to their website, they have a ‘unique approach to generating the most accurate mobility information globally’ [4].

Citymapper also provides a routing service which not only includes ‘classic’ public transit options like busses, trams and trains, but also taxis, e-bikes and scooters.

¹<https://area120.google.com>

²<https://citymapper.com>

³<https://opendata.rnv-online.de/node/61/revisions/211/view>

Citymapper even expanded beyond the app, as the company introduced the ‘Citymapper Pass’, a subscription to all public and private mobility in London, as an alternative to other weekly passes.

There are similar apps like TransitApp⁴ and Moovit⁵ providing comparable services.

3.2 Map Matching to a Static Map

3.2.1 Hidden Markov Models in Map Matching

The PTV Matching used by *PublicTransitSnapper* makes use of a Hidden Markov Model, similar to the approach described by Newson and Krumm in 2009 [5]. Newson and Krumm matched GPS positions of a car to a static street network. For our PTV Matching, we have to find the most likely GTFS shape in combination with a PTVs position in time. Newson and Krumm’s HMM approach to match the GPS positions to a street network is very similar to our shape matching algorithm.

3.2.2 Real Time Map Matching

As *PublicTransitSnapper* uses real time data, it requires a fast response time. Newson and Krumm used the Viterbi algorithm [6] to find the most likely path through the HMM, taking into account the transition and emission probabilities. The Viterbi algorithm provides a very high quality map matching, but can be too slow for real time data. Koller, Widhalm, Dragaschnig and Graser introduced the alternative use of a bidirectional Dijkstra algorithm in 2015. Their “test results show that [their] suggested solution can avoid up to 45% of the costly routing operations and has no negative effect on the quality of the map-matching result” [7]. *PublicTransitSnapper* uses the bidirectional Dijkstra algorithm as well.

3.3 Public Transit Data in Germany

In this thesis, we will elucidate the current state of publicly available public transit data in Germany, in particular GTFS. In 2014, S. Kaufmann wrote a thesis on this subject [8]. Kaufmann specifies the HAFAS and DIVA data models used by many German public transit agencies. Kaufmann then focuses on translating HAFAS and DIVA to GTFS, before giving an outlook on publicly available data. In eight years, many things have changed. For instance, Kaufmann did not mention DELFI yet, the German ‘association for continuous electronic schedule information support’, which is

⁴<https://transitapp.com>

⁵<https://moovitapp.com>

currently playing a very important role in the publication of public transit data in Germany.

3.4 Working Together on PublicTransitSnapper

PublicTransitSnapper was developed together with my colleague R. Wu. Read his thesis [2] for a more in-depth explanation on the PTV Matching, our GTFS-RT implementation and an evaluation of the *PublicTransitSnapper*. Throughout this thesis, I will refer to R. Wu's thesis whenever he covered subjects in a more detailed way.

4 Building PublicTransitSnapper

4.1 Problem Definition

Given a stream of timestamped GPS coordinates, static schedule data (GTFS) and optional real time schedule data (GTFS-RT), it was our task to build an app with the following features:

- (I) Show the user if they are matched to a public transit vehicle (PTV). If so, show the type and the name of the vehicle, as well as the next stop.
- (II) Visualize the trip of the user on a map. When matched to a PTV, show the shape of the current trip.
- (III) If matched to a PTV, show possible transfer options at the next stop. In the following, we will call these ‘connections’.
- (IV) If matched to a PTV, the user can chat to all users matched to the same PTV.

In order to solve these problems, we needed a backend to manage data derived from the GTFS data set, process requests from the app and provide the chat.

See R. Wu’s thesis [2] for information on how we implemented GTFS-RT real time updates.

4.2 Project Structure

Our *PublicTransitSnapper* consists of both a backend and a frontend. In the following Sections, we are going to introduce the frontend, how it connects to the backend using an API, and then describe the backend.

4.2.1 Used Programming Languages

We use three programming languages in total.

Choosing Flutter as Framework for the Frontend

Flutter¹ is an open source framework by Google for building multi-platform applications from a single code base. The programming language used for Flutter projects is called Dart². It has also been developed by Google. While coding, Flutter allows for the very handy feature called ‘hot reload’³, with which the programmer can modify code while running the application and see the changes without having to recompile the whole project.

A perk of Flutter is that Flutter apps can be built as Android and iOS mobile phone apps, but also as a web-app. We used both the mobile app and the web-app during development and testing.

Flutter natively contains two sets of widgets⁴: The Material Design⁵ from Google and Cupertino⁶ widgets known from iOS apps. We mostly used Material Design, which we preferred because of its familiarity.

Choosing Python for the Backend

Using Python entails the big advantage that there is a package for everything. As I will explain later, we chose to use Flask⁷ and flask-CORS⁸ to communicate with the frontend. Python’s datetime library⁹ comes in very handy, as PTV Matching comes with a lot of time-based calculations. There is also PyYAML¹⁰ to read YAML configuration files. As the backend will be running on linux, we can use the subprocess library¹¹ to execute unix commands. This can be very useful to fetch up-to-date GTFS files from the internet for example. More on that in Section 4.7.

For geographical data handling, we use the shapely library¹². Shapely uses the very efficient GEOS library¹³, which is written in C and C++ . Shapely also uses multithreading by releasing Python’s Global Interpreter Lock (GIL)¹⁴: ‘Normally in Python, the GIL prevents multiple threads from computing at the same time. Shapely functions internally release this constraint so that the heavy lifting done by

¹<https://flutter.dev>

²<https://dart.dev>

³<https://docs.flutter.dev/development/tools/hot-reload>

⁴<https://docs.flutter.dev/development/ui/widgets-intro>

⁵<https://material.io/design/introduction>

⁶<https://docs.flutter.dev/development/ui/widgets/cupertino>

⁷<https://flask.palletsprojects.com/en/2.2.x/>

⁸<https://flask-cors.readthedocs.io/en/latest/>

⁹<https://docs.python.org/3/library/datetime.html>

¹⁰<https://github.com/yaml/pyyaml>

¹¹<https://docs.python.org/3/library/subprocess.html>

¹²<https://shapely.readthedocs.io/en/stable/manual.html>

¹³<https://libgeos.org/>

¹⁴<https://realpython.com/python-gil/>

GEOS can be done in parallel, from a single Python process' [9]. A main benefit of shapely is the provision of its 'STRtree', which is a data structure for storing and accessing GPS data in an efficient manner (see Section 10.1.1 in the appendix for a more detailed explanation). We also use the NetworkX library¹⁵, which allows us to build graphs for the PTV Matching.

The combination of shapely and NetworkX alone convinced us to use Python, but during the development of our *PublicTransitSnapper*, it turned out to be even more useful. Firstly, it turned out that GTFS-RT data can be received very easily with Python. Google even provides a Python library¹⁶ for that purpose. Secondly, we had to test the matching of our *PublicTransitSnapper* somehow without having to travel in a PTV for coding and debugging. For that, we wrote a handy tool using Python's Selenium¹⁷ that creates fake GPS locations along a GTFS shape and passes them to a Python-controlled Google Chrome instance that runs our frontend.

Choosing C++ to Process GTFS Files

During the first few months of development, we gradually needed more and more data like dictionaries and graphs from the GTFS files for our *PublicTransitSnapper* to efficiently fetch information on GTFS during runtime. Some of this data was only built sloppily for debugging purposes, containing duplicates and thus filling the backend server memory with unnecessary information. The generation process of the data was often slow and inefficient, built with our at that time poor understanding of the powerful Python pandas library¹⁸. While that worked for smaller cities like Freiburg and Augsburg, we had memory and runtime issues when experimenting with larger GTFS data sets like Hamburg and even the entire Switzerland, for which there is one whole data set. So we decided to re-write the entire GTFS processing, this time efficiently and without duplicates, reading every GTFS file only once. We chose to use C++ for even faster runtimes.

C++ has a very fast library to handle JSON files titled 'JSON for modern C++'¹⁹. We use JSON to save C++ maps (the C++ equivalent of a Python dictionary) to load them in Python memory when starting the backend. There is also a C++ library for CSV parsing²⁰ which we use to parse the GTFS files.

¹⁵<https://networkx.org/>

¹⁶<https://developers.google.com/transit/gtfs-realtime/examples/python-sample>

¹⁷<https://selenium-python.readthedocs.io/>

¹⁸<https://pandas.pydata.org/>

¹⁹<https://github.com/nlohmann/json>


²⁰<https://github.com/ben-strasser/fast-cpp-csv-parser>



4.3 The Frontend






In this Section, we are going to delve into the implementation and functionality of features (I) to (III).


4.3.1 App Layout

We designed the app to be as very user-friendly as possible. There are only nine buttons in total. Using the three buttons on the bottom navigation bar, the user can switch between three main pages that fulfill features (I), (II) and (III).

The ‘front page’ (see Figure 7) can be reached by clicking on the lower left button with the -icon and the ‘Vehicle’ subtitle. It is also the first page that appears when the user opens the app. The page shows the user if they are matched to a PTV. If so, it shows the type and the name of the vehicle, as well as the final destination and the next stop, thus fulfilling feature (I). If not matched to a PTV, the front page will tell the user just that.

The ‘map page’ (see Figure 8) can be reached by clicking on the lower middle button with the -icon and the ‘Map’ subtitle. The page shows a map from OpenStreetMaps²¹. We used SfMaps²², which is a package capable of creating interactive maps. On the map, there is an icon. If not matched to a PTV, the icon is  and its location is corresponding to the last GPS position. If matched to a PTV, the icon corresponds to the route type of the matched PTV and its location will be clipped to the shape of the route, which is calculated in the backend. Also, when traveling in a PTV, the shape on which the user has traveled along will be displayed in the route color given in the GTFS files.

We embellished the map with four more buttons. The -button displays the latest GPS coordinates as a red line. Pressing it again hides the red line. The -button resets the received GPS coordinates. If the user does not get matched for some reason, they can press this button. The -button makes the camera follow the user’s position. When the camera is locked, the button displays a -icon. The -button zooms to the users current position. The user can press it multiple times, the zoom toggles between zoom levels.

The ‘connections page’ (see Figure 7) can be reached by clicking on the lower right button with the -icon and the ‘Connections’ subtitle. When matched to a PTV, it shows a list of the possible transfer options at the next stop. Each list entry contains the destination of the route, the route type as an icon next to the route name, as well as the departure time at the next stop.

²¹<https://www.openstreetmap.org/>

²²<https://help.syncfusion.com/flutter/maps/getting-started>

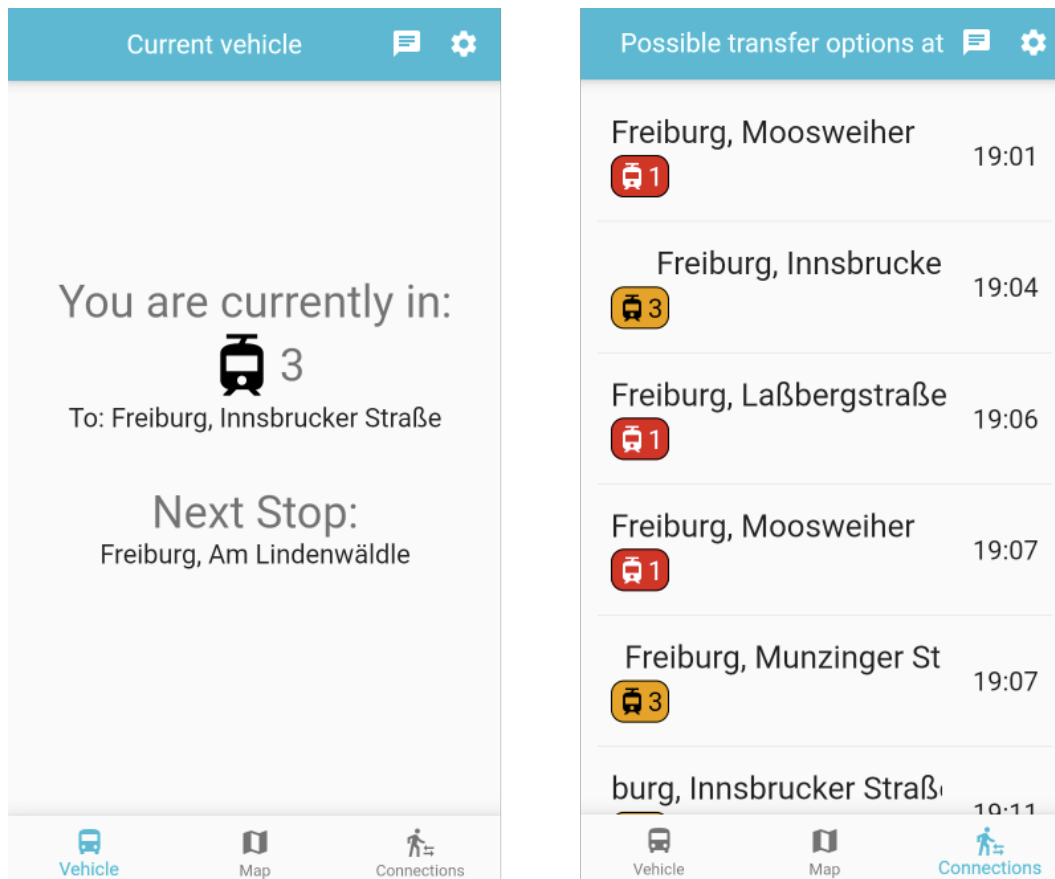


Figure 7: The 'front page' on the left displays information on the current route type, route name, the route destination, as well as the next station. The 'connections page' on the right shows all possible transfer options at the next stop if matched to a PTV. Else, this is empty.

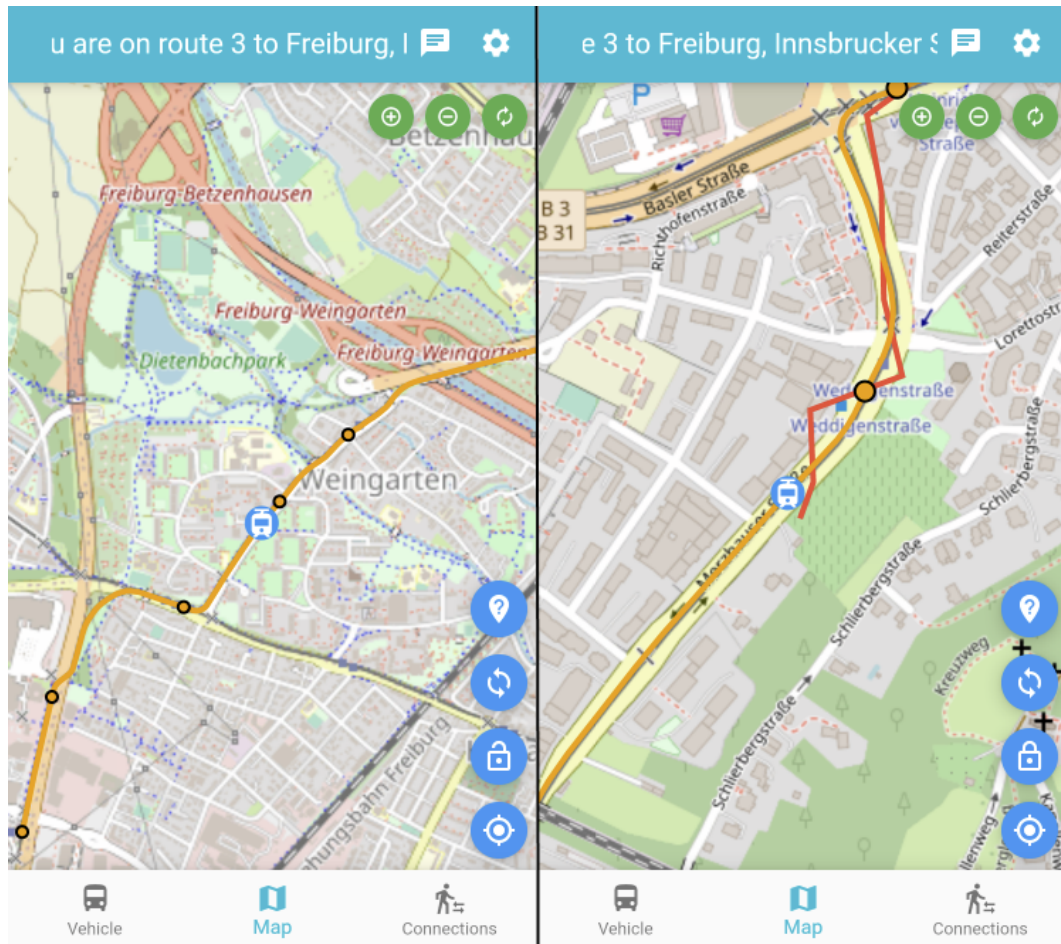


Figure 8: The ‘map page’ shows a map from OpenStreetMaps. The left image shows that the user is matched to route 3. The text in the header is scrolling, as the screen of the mobile device is not wide enough to display the whole destination name. On the right image, we can see that the user has pressed the button with the ‘?’ icon. This shows the red line, which displays the GPS points tracked by the device. We can see that the vehicle icon position has been matched to the orange line, which indicates the shape the vehicle is moving along on. The vehicle icon also indicates that the user has been matched to a tram. The line is orange, as specified in the GTFS field for `route_color`. We can also see the stops of the vehicle.

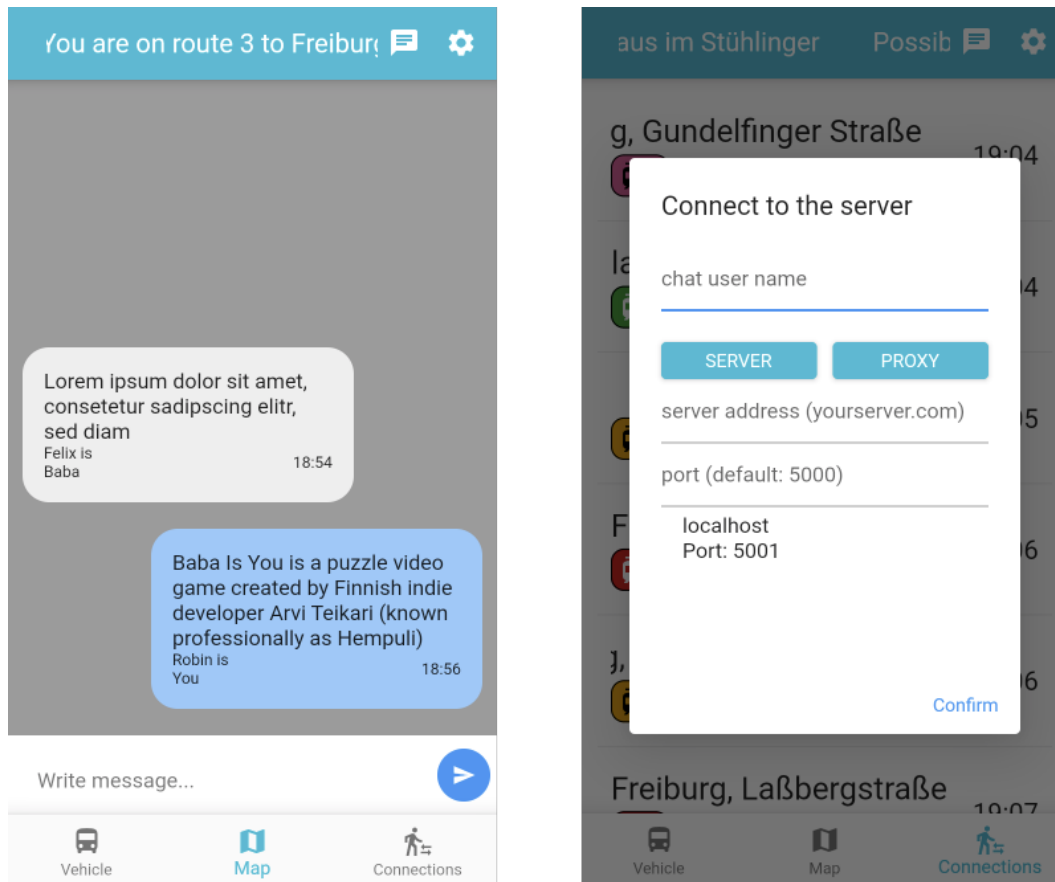




Figure 9: ‘Chat page’ on the left: When matched to a vehicle, every user matched to the same trip can chat with all the users in the same PTV. Messages from another user are displayed in white on the left hand side, while own messages are displayed in blue on the right. The name of each user is displayed on the bottom left side of each bubble. The time the message has been sent is displayed on the bottom right side of each bubble. ‘Settings page’ on the right: Here, the user can change their chat user name and the server the frontend is communicating with. The ‘SERVER’ and ‘PROXY’ buttons fill the server address and port text fields to preset values which can be specified in a ‘config.yml’ file (explained in subsequent section 4.7). Below the port text field, the user can choose from up to three recent server-address-port-combinations.

The ‘chat page’ (see Figure 9) can be reached by clicking on the -icon in the top right corner. When matched to a PTV, the user is able to chat with other users in the same PTV. This fulfills feature (IV).

The ‘settings page’ (see Figure 9) can be reached by clicking on the -icon in the top right corner. It is also displayed once when the app starts. The user can change the server address and port to communicate with the backend. They can also change their user name which is displayed when chatting. The last three saved server addresses and ports are saved below in a scrollable list, in case the user wants to switch server or their displayed name.

4.4 Communication Between Frontend and Backend

As the calculations to match a mobile phone to a PTV are computationally rather expensive, it would make no sense to calculate everything on the users device. Also, the GTFS files are too large for a mobile phone’s memory. Therefore, we need a server (the backend), which receives GPS coordinates and other information, calculates, and then sends the results back to the app (the frontend).

4.4.1 Handling HTTP Requests and Transferring Information on the PTV Matching

Whenever the frontend device detects that it has moved 15 or more meters in one direction (‘distance filter’), it sends a HTTP request containing a JSON string to our API (Application Programming Interface) on the backend server. The JSON contains the last ten GPS points with a corresponding timestamp. For the PTV matching request, we use the endpoint ‘/map-match’. It is called this way for historic reasons.

On the server side, we handle the HTTP requests and responses with Flask²³. There, the GTFS derived data is already loaded in the memory, so it can quickly calculate the PTV Matching when the API registers a request. After calculating whether the sent GPS points and timestamps match to a PTV trip, we return information for displaying the matched path to the mobile device. The JSON we return to the frontend contains the following information:

- The route name, corresponding to the `route_short_name` from the trip’s route
- The `route_type`, namely an integer to describe what type of PTV the user is travelling with
- the `route_color`
- the next stop of the trip

²³<https://flask.palletsprojects.com/en/2.2.x/>

- the destination, namely the last stop of the trip
- the GPS location to display the icon on the map
- the `trip_id` and the `shape_id`, which are not used in the frontend directly, but are needed for other requests

4.4.2 Communication Needed for the Connections Page

To display the connections page (recall Figure 7) in the frontend, we choose to send different requests to a different endpoint (`/connections`), as there is no need to fetch the connections every time the user device moves 15 meters. Instead, we just ask the backend for the connections at the next stop periodically when the connections page is loaded. The frontend knows the next stop name from the requests to endpoint `/map-match`. We can send the next stop name together with the time on the user device to the backend. The backend will calculate the connections at that stop and time and send back a JSON object containing a list of the result. If the next stop name is empty, the frontend displays an empty page.

4.4.3 Communication Needed for Displaying the Shape

Whenever the user gets matched to a new PTV, we send the `shape_id` together with the `trip_id` to the backend at endpoint `/shapes`. The backend calculates the shape and the stops along the shape and send them back.

4.4.4 The Chat

As we need the backend to differentiate between users, each user receives a unique `user_id` the first time they communicate with the backend using the API endpoint `/chat`. The frontend stores the `user_id` together with the current server address on the users device, so the next time the user starts the app and connects to the same server, they have the same `user_id` as before.

If the server should restart for some reason, the `user_ids` will be reset. To make sure that the user devices recognize they have to get a new `user_id`, we send a timestamp of the server start time (`server_start_timestamp`) back and forth with every request. If the `server_start_timestamp` the backend receives from a request is smaller (older) than the actual `server_start_timestamp` from the server, we need to distribute a new `user_id` to the user.

Every chat message contains a user name (`user_name`), a time when the message was sent (`time_sent`), a message content (`message_content`), as well as the `user_id`. Whenever the `user_id` of a message fits to the `user_id` stored on the users device, we can display the message in blue on the right side of the screen, as these messages

have been sent from the same device. All the other messages are displayed on the left side of the screen in a grey tone.

Because we want the chat page to display recent messages, we send a request to the backend containing a Boolean `'just_fetch = True'` every 10 seconds. When we just want to fetch new messages, we send the `user_id`, the `server_start_timestamp` and the current `trip_id` to the backend.

When the backend server starts, a Python dictionary `'trip_id_to_chat_messages'` is created. Whenever the backend receives a `'just_fetch'` request, we can check if the dictionary has stored any messages for the given trip and return a JSON with a list of the chat messages to the frontend. The list is empty if no chat messages have reached the server yet.

When a user sends a message, their frontend device will send a request to `'/chat'` with `just_fetch = False`, the `user_id`, the `server_start_timestamp` and the `trip_id`, as well as the message containing their `user_name`, `user_time` and `message_content`. Over on the server side, because `just_fetch` is `False`, we can add a new chat message to the dictionary. If the message sent was the first message of the trip, we add a new entry `{trip_id = [message]}`. Else, we can append the newly sent message to the already existing list behind the `trip_id`. See Listing 4.1 for a visualization.

```
chat_dictionary = {
    "trip_id": [
        (
            user_id,
            "user_name",
            "message_content",
            "time_sent"
        ),
        ...
    ],
    ...
}
```

Listing 4.1: Content of a Chat Message

As trips will end at some time and new trips with the same `trip_id` can potentially start every day, we want to delete chat conversations once the trip finishes. This presumes that trips do not last longer than 23h. So, trips that are not active will be removed from the dictionary hourly.

4.5 The Backend

The backend is running on a server. This is the place where the actual PTV Matching and many more features are being calculated and processed. In this Section, we introduce and explain the building blocks of the backend.

4.5.1 Docker

Our *PublicTransitSnapper* uses multiple programming languages, needs to manage GTFS files and executes linux commands. In order to facilitate this, we chose to use Docker²⁴. We can run a Docker Container²⁵ that runs the backend in the background of the server. The Docker Container automatically fetches and installs dependencies needed for the backend and then runs the Flask server. This works on many operating systems like Windows, Linux, macOS, and other Unix-like systems. Using Docker has proven to be great for debugging and is a handy and prevalent way to provide open-source projects on GitHub²⁶.

In short, the main benefit of using Docker is to run a command like ‘`docker run`’ and the whole backend sets itself up and is ready to be used.

4.5.2 Connecting Shapes, Stops and Times

The code described in the following Sections is all Python. In case you forgot about ‘overtime’, reconsider Section 2.1.2. The same applies to the term ‘edge’ (Section 2.1.3).

In order to make the PTV matching as efficient as possible, we decided to represent each trip as an object that has information on the shape it is on, the stops it uses and when it is where. This would make it easy to calculate frontend-requested information during runtime, as we could preprocess a lot of information, such as linking the trip to the shape it is on, knowing the next stop for each section of the trip and more.

TripsWithStopsAndTimes Attributes

In the end, we settled on an object called `TripsWithStopsAndTimes` that represents a trip with the following attributes:

- Its own `trip_id`
- Its `service_id` to have access to the trip’s service information
- Its `time_interval`, which is a tuple (`start_time`, `start_overtime`, `end_time`, `end_overtime`). The `start_time` and `end_time` are datetime²⁷ objects holding the `departure_time` when the trip leaves the first stop and the `arrival_time` when the trip arrives at the final stop. The two Booleans `start_overtime` and the `end_overtime` describe whether the `start_time` or `end_time` are in overtime or not.

²⁴<https://www.docker.com/>

²⁵<https://docs.docker.com/get-started/overview/>

²⁶<https://github.com/TheRealTirreg/PublicTransitSnapper>

²⁷<https://docs.python.org/3/library/datetime.html>

- Its `active_hours`, which is a set of tuples. Each tuple consists of two values. The first describes the weekday as an integer from 0 to 6, where 0 maps to Monday, 1 to Tuesday and so on. The second describes an hour of the day as an integer from 0 to 23. For every hour where the trip is active, we add a tuple to the set. For example: For a trip with a service that is active on Sundays from 23:05h to 01:15h, the `active_hours` would be $\{(6, 23), (0, 0), (0, 1)\}$
- A `trip_segment_hash`. This hash value is an integer which can be used to access a dictionary '`hash_to_edge_to_trip_segments_dict`', which we describe in the following Paragraphs. It is used to have access to every '`trip_segment`'. As there are many trips sharing the same shape and the same stops, we only need to store this information once.

Trip Segments

In order to find the next stop once we match a GPS device to a trip, we divide every trip into their `trip_segments`. A `trip_segment_id` is an integer describing what part of a trip's shape the user is on. See Figure 10 for an illustration.

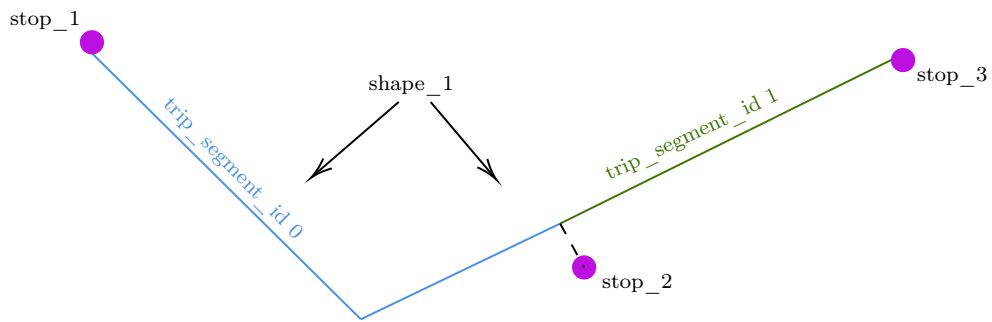


Figure 10: This Figure shows an illustration of trip segments. A trip segment describes the part of a shape between two stops. Here, `trip_01` uses `shape_1` and halts at `stop_1`, `stop_2` and `stop_3`, where `stop_1` is the starting station and `stop_3` is the destination of the trip. Stops do not have to be on the shape polyline, but can be a few meters off. However, we can still project the stop's position onto the shape. In our example, the segment between the closest point to `stop_1` on `shape_1` and the closest point to `stop_2` on `shape_1` would get `trip_segment_id 0`, and the segment between the closest point to `stop_2` on `shape_1` and the closest point to `stop_3` on `shape_1` would get `trip_segment_id 1`.

If we match a user device to a trip and are on the i -th trip segment, we can derive the next stop if we have a list of the trip's stops in halting order. The next stop

would be on the $(i + 1)$ -th position of the list. In the following Paragraph, we will introduce lists and other data structures we used in our project.

4.5.3 Preparing and Using the GTFS Files

In order for the PTV Matching algorithm to be fast enough for real time calculations, we need a representation of the GTFS files in the memory of the server. We tried using Python's `shelve` library²⁸, which works like a dictionary stored on a hard drive. This would drastically reduce the memory usage. Sadly, `shelve` is too slow for our purposes, as we have many look-up operations on GTFS related information. Therefore, when starting the program, we create an object called *GTFS Container*, which is always in the memory and contains the following objects:

`trip_id_to_route_and_stop_times`

A dictionary mapping the `trip_id` as key to a tuple containing its `route_id`, as well as a list of stops on the trip. The list contains the `stop_id`, a tuple (`arrival_time`, `is_arrival_overtime`) and a tuple (`departure_time`, `is_departure_overtime`) for every stop on the trip.

`route_id_to_route_info`

A dictionary mapping the `route_id` as key to a tuple containing information on the given route, namely the `route_short_name`, the `route_type`, the `route_color` and the `route_text_color`.

`stop_id_to_stop_info`

A dictionary mapping the `stop_id` as key to a tuple containing information on the given stop, namely its `stop_name` and its GPS coordinates `stop_lat` and `stop_lon`.

`stop_id_to_trips_and_departure_times`

A dictionary mapping the `stop_id` as key to a list of all the trips that halt at the stop, together with the `departure_time`: (`trip_id`, `departure_time`)

`stop_name_to_stop_ids`

A dictionary mapping each `stop_name` as key to a list of all the `stop_ids` that are bound together by a parent station (see Section 2.1.2).

²⁸<https://docs.python.org/3/library/shelve.html>

`shape_id_to_first_edge_and_trip_info`

A dictionary mapping each `shape_id` as key to the shape's first edge as four-tuple and to a list of all the `trip_ids` using the shape with their respective `service_id` and `route_id`: (`trip_id`, `service_id`, `route_id`)

`service_id_to_service_info`

A dictionary mapping each `service_id` as key to a tuple containing a list of 'active weekdays', the `start_date` and `end_date` of the service, as well as a list of the 'extra dates' and a list of the 'removed dates'. 'Active weekdays' describe the weekdays where there is service as integers, where 0 maps to Monday and 6 maps to Sunday. The 'extra dates' list contains all the dates from *calendar_dates.txt* where the service has additional service, whereas the 'removed dates' contain all the dates where the service has been removed.

`edges_STRtree`

A shapely STRtree²⁹ containing every `edge`. We use STRtrees for fast spacial queries. See Section 10.1.1 in the appendix for an explanation on STRtrees.

`edges_DiGraph`

A directed graph from NetworkX³⁰ containing every `edge`.

`trip_id_to_TripsWithStopsAndTimes`

A dictionary mapping each `trip_id` as key to its corresponding `TripsWithStopsAndTimes` object.

`hash_to_edge_to_trip_segments_dict`

A dictionary used by `TripsWithStopsAndTimes` objects, which maps one `trip_segment_hash` for each `TripsWithStopsAndTimes` object as key to a dictionary. This dictionary maps every `edge_id` of the trips shape to its `trip_segment`.

We generate the objects above by reading the GTFS files using our C++ program *ParseGTFS*. *ParseGTFS* generates C++ maps. These maps are either translated to Python dictionaries or are converted into Python-specific objects like STRtrees or NetworkX graphs. We save these C++ maps as JSON files. When generating the *GTFS Container*, we read the JSON files in Python.

If there are no JSON files yet, or they are outdated because new GTFS files have been fetched (see Section 4.7), we can run *ParseGTFS* using Python's subprocess

²⁹<https://shapely.readthedocs.io/en/stable/manual.html>

³⁰<https://networkx.org/>

library³¹ as if we ran the program from a command line interface. This is guaranteed to work, as the backend is running in a linux environment provided by Docker.

4.5.4 TripsWithStopsAndTimes Methods

A `TripsWithStopsAndTimes` object has two methods.

`is_trip_active`

`is_trip_active` calculates whether there is currently traffic on the trip, given a datetime object and a reference to the *GTFS Container*. Because each `TripsWithStopsAndTimes` object has a set of its own `active_hours`, we can check in $\mathcal{O}(1)$ whether the trip is active in the given hour. By using the `service_id_to_service_info` dictionary with the `TripsWithStopsAndTimes` object's `service_id` as key, we can check whether the given date is a removed date or an extra date. This look-up is also in $\mathcal{O}(1)$ on average. Therefore, `is_trip_active` has a constant runtime in most cases.

`get_active_trip_segment_ids`

`get_active_trip_segment_ids` returns a list of `trip_segment_ids`. Given a timestamp, an `edge_id`, an allowed delay, an allowed earliness, as well as a reference to the *GTFS Container*, it calculates every `trip_segment_id` where there could be a PTV within the allowed delay and the allowed earliness. This method makes use of the `TripsWithStopsAndTimes`' `trip_segment_hash` attribute, as we can use the hash as a key to find a dictionary that maps `edge_ids` to `trip_segment_ids` within `hash_to_edge_to_trip_segments_dict`. We call this dictionary `edge_id_to_trip_segment_id_dict`.

`get_active_trip_segment_ids` uses `service_id_to_service_info` in combination with the `TripsWithStopsAndTimes`' `service_id` attribute to find out whether the trip is within the validity period of the service. In order to check if the trip we want the active `trip_segment_ids` from is currently active, we can use the `is_trip_active` method. Furthermore, we need `trip_id_to_route_and_stop_times` from the *GTFS Container*. In order to find all the active `trip_segment_ids`, we can loop over all the `trip_segment_ids` yielded by the `edge_id_to_trip_segment_id_dict` with the input `edge_id` as key. Using `trip_id_to_route_and_stop_times`, we can get the `departure_time` and the `arrival_time` for each `trip_segment_id`. If the input `user_datetime` falls within the time span, the trip segment is active. The method runs in $\mathcal{O}(n)$, where n is the number of `trip_segment_ids` on the given edge. In theory, there can be $n - 1$ stops on one edge. Usually, $n = 1$ or $n = 2$ (see Figure 10, where the right edge is 'separated' by a stop). Only in rare cases is $n > 2$. Therefore,

³¹<https://docs.python.org/3/library/subprocess.html>

`get_active_trip_segment_ids` is not dependant on the size of the GTFS data set and has a constant run time in most real world instances.

4.5.5 GTFS Container Methods

The *GTFS Container* is capable of finding simple GTFS information like a `trip_ids` destination (its last stop), or a `route_ids` `route_short_name`, `route_color` or `route_type` in averagely $\mathcal{O}(1)$ using the dictionaries.

`get_active_trips_information`

Given a `shape_id`, an `edge_id` and a datetime object, this method returns all active `trip_ids` with a list of their active `trip_segment_ids`, their `service_id` and `route_id`. Using `shape_id_to_first_edge_and_trip_info`, we can get all `trip_ids` with their respective `route_id` and `service_id` in $\mathcal{O}(1)$ on average. For every `trip_id`, we can find its respective `TripsWithStopsAndTimes` object using `trip_id_to_TripsWithStopsAndTimes`. Then, we can get the wanted return attributes in $\mathcal{O}(n)$ using the `get_active_trip_segment_ids` method. Remember that $n < 2$ in most real world instances. Thus, `get_active_trips_information`'s runtime is $\mathcal{O}(nm)$, where m describes the number of trips sharing the given shape.

`get_next_stop`

`get_next_stop` calculates the most likely next stop given a position, a `trip_id`, the last matched `edge_id` and a list of its `trip_segment_ids`. Usually, there is only one `trip_segment_id`. In that case, we can get a list of the stops of the `trip_id` using `trip_id_to_route_and_stop_times`. Then, we can get the `stop_id` of the $(\text{trip_segment_id} + 1)$ th stop in the list of stops, as explained in the lower part of Paragraph 4.5.2. Finally, we can use `stop_id_to_stop_info` to get the desired `stop_lat`, `stop_lon` and `stop_name`. All of this has a constant runtime, as dictionary look-ups and index-based list access are constant on average. In case there is more than one `trip_segment_id`, we need to decide which stop to pick. See Figure 11 for an example. Here, the runtime scales with the number of `trip_segment_ids` on the given `edge_id` and is therefore in $\mathcal{O}(n)$, where n describes the number of trip segments on the given `edge_id`. Again, $n < 2$.

`find_transfer_possibilities`

`find_transfer_possibilities` calculates the trips shown on the connections page (recall Figure 7). Given a `stop_name`, a `trip_id` and a datetime object, this method returns the next up to 20 trips departing at the given stop. The input `trip_id` serves the purpose of preventing the user's trip from showing up in the connections list. Here, we can use the benefits of the active hour system used by `TripsWithStopsAndTimes` objects (see Paragraph 4.5.2).

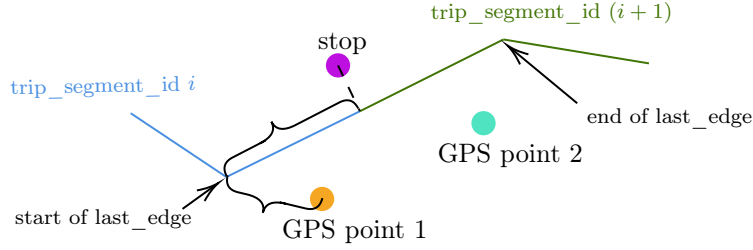


Figure 11: Example for `get_next_stop`: For GPS point 1, `get_next_stop` would return the stop at the end of trip segment i , as GPS point 1 is closer to the start of last_edge than the projected stop. For GPS point 2, the projected stop is closer to the start of last_edge, therefore `get_next_stop` would return the stop at the end of trip segment $(i + 1)$.

We can get a list of the `TripsWithStopsAndTimes` objects passing the given stop using the `stop_id_to_trips_and_departure_times` dictionary in combination with the `trip_id_to_TripsWithStopsAndTimes` dictionary.

For each `TripsWithStopsAndTimes` object, we can calculate in $\mathcal{O}(1)$ whether the trip is active using the object's `is_active` method. For active trips, we need to check if their departure time at the given stop is later than the given datetime. In that case, we have found a possible connection. We can now add `route_short_name`, `route_color`, `route_text_color`, `route_type` and the destination of the trip to the list of connections using `trip_id_to_route_and_stop_times` and `stop_id_to_stop_info`.

As some trips might not yet be active during the hour of the given datetime object, we can repeat the algorithm k times and each time add an hour to the datetime object. The runtime of `find_transfer_possibilities` is in $\mathcal{O}(nk)$, where n describes the number of trips passing the given stop.

`get_shape_polyline_and_stops`

`get_shape_polyline_and_stops` serves the purpose of calculating the shape and the stop positions of the given `trip_id`. Having to calculate the shape is a compromise to the memory usage of the project. As the memory load can be quite heavy for larger GTFS data sets, we decided not to load the shapes into memory, especially since we already have a duplicate representation of the shapes in the form of edges. Edges are both needed for the STRtree `edges_STRtree` and the directed graph `edges_DiGraph`.

In order to find out the shape, we traverse the directed graph `edges_DiGraph` starting from the first edge of the shape. We can get the first edge of the shape with `shape_id_to_first_edge_and_trip_info`. As each edge in the graph knows its own `sequence_id`, we do not have to deal with loops and double edge traversals

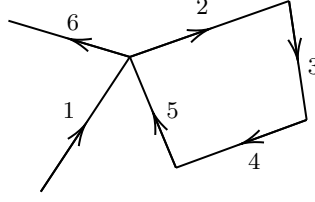


Figure 12: This figure illustrates the shape within the directed graph (I). When searching the neighbors of **edge 1**, there are two candidate edges 2 and 6. Because we save edges with their **shape_sequence_number** (remember section 2.1.3), we can always recreate the shape from the **edges** by choosing the next **edge** in the graph. In this case, we would choose **edge 2**.

(see Figure 12). Traversing the graph has a runtime of $\mathcal{O}(n)$, where n describes the number of **edges** on the shape.

We find the stops of the given **trip_id** by using **trip_id_to_route_and_stop_times**. Then, we can get each stop's position using **stop_id_to_stop_info**. The runtime for finding the stops is in $\mathcal{O}(m)$, where m describes the number of stops along the trip's shape. The total runtime for **get_shape_polyline_and_stops** is in $\mathcal{O}(n + m)$.

4.6 PTV Matching

The *GTFS Container* has more methods that are only used for the direct PTV Matching. For the implementation details on the PTV Matching, as well as our GTFS-RT functionalities, see R. Wu's thesis [2]. In this Section, we explain our PTV Matching approach. In case you forgot about G_{network} or HMMs, in particular transition and emission probabilities, reconsider Sections 2.2.4 and 2.2.6.

4.6.1 Using a Hidden Markov Model to Find the Most Likely Edges

The map matching part of PTV Matching can be solved using a HMM. We can model its states by creating another directed graph $G_{\text{markov}} = (N, T)$ that consists of nodes and transitions.

The nodes N describe edges from G_{network} $N = \{e_{c_k}^i\}$, with c_k describing the k -th GPS point and $e_{c_k}^i$ being the i -th edge from G_{network} within a certain radius of c_k .

The transitions $T = \{(e_{c_k}^i \rightarrow e_{c_{k+1}}^j)\}$ describe the transition likelihood from one node to another.

See Figure 13 for an example of G_{network} and G_{markov} .

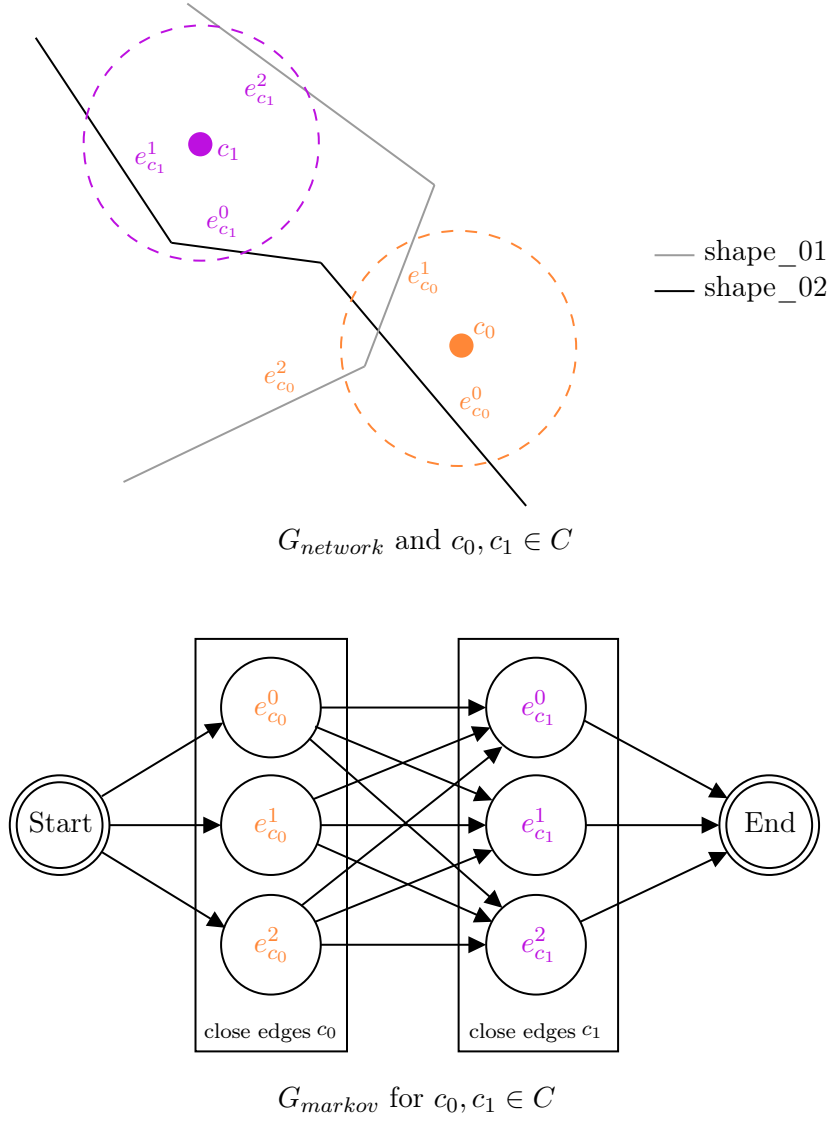


Figure 13: This Figure illustrates $G_{network}$ and G_{markov} . c_0 and c_1 are two GPS coordinates. We call **edges** within the circle around them ‘close edges’. For c_0 , the close edges are $e^0_{c_0}$, $e^1_{c_0}$ and $e^2_{c_0}$. Analogously for c_1 . We can build G_{markov} by adding a layer with all $e^i_{c_k}$ for each GPS coordinate k , assuming there is a currently active trip on both shapes.

To build G_{markov} , we create a single starting node and then add all the edges e^i from G_{network} , that fulfill two requirements. Firstly, the edge must belong to an active trip. For this, we use the `get_active_trips_information` method from the *GTFIS Container*. Secondly, the edge must be within a certain radius to the first GPS point $c_0 \in C$. We can add all these edges $e_{c_0}^i$ as nodes to the G_{markov} . We then connect the starting node with each node $e_{c_0}^i$. After that, we add all active edges that are close to the second GPS point as nodes $e_{c_1}^j$ and connect these with the nodes of the previous GPS point $e_{c_0}^i$. We repeat this for each $c_k \in C$. In the end, we add an end node and connect it to the nodes of the last GPS point.

If we assign meaningful probabilities to the edges and nodes in G_{markov} , we can find the shortest path from the start node to the end node. This will give us edges that are on a path that fits best to the GPS points. However, it is not apparent how to assign meaningful probabilities to the nodes and transitions in G_{markov} . Instead, we calculate weights $\mathcal{W}_{\text{emission}}$ and $\mathcal{W}_{\text{transition}}$ to replace P_{emission} and $P_{\text{transition}}$.

We use a bidirectional Dijkstra's algorithm³² (see 10.2.1 in the appendix) to find the shortest path in G_{markov} . Dijkstra's algorithm sums up all the weights on a path and prefers choosing edges with low weights. As a consequence, if a path has a higher probability, its weight needs to be smaller. As Dijkstra's algorithm is not guaranteed to work on negative numbers, weights need to be non-negative.

4.6.2 Assigning Weights to the HMM

In the following two Sections, we are going to assign weights to G_{markov} . Each node $e_{c_k}^i$ has an emission weight $\mathcal{W}_{\text{emission}}$, each transition between nodes ($e_{c_k}^i \rightarrow e_{c_{k+1}}^j$) has a transition weight $\mathcal{W}_{\text{transition}}$.

Generating Emission Weights

The emission weight of edge $e_{c_k}^i$ is dependant of the distance to the GPS point c_k :

$$\mathcal{W}_{\text{emission}}(e_{c_k}^i) = \text{gcd}(e_{c_k}^i, c_k)$$

gcd describes the great circle distance introduced in 2.2.1. The further away $e_{c_k}^i$ is from c_k , the less likely it is that $e_{c_k}^i$ is part of the most optimal path.

Generating Transition Weights

Transition weights describe the relation between two nodes.

³²<https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/>

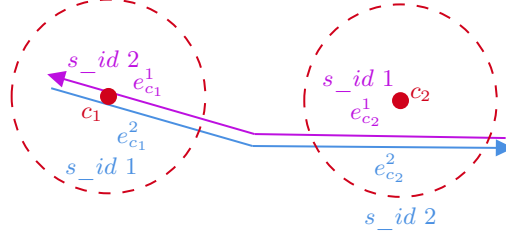


Figure 14: The two illustrated shapes are directed in opposite directions: The blue shape starts on the left and goes to the right, the purple shape starts on the right and goes to the left. This is represented by each shape's **sequence_number** (**s_nr**). Assuming GPS point c_1 was received before c_2 , we want to match to the blue shape, as we want to match to the shape fitting to the travel direction.

If two edges $e_{c_k}^i$ and $e_{c_{k+1}}^j$ are not on the same shape, we penalize the transition with a high weight:

$$\mathcal{W}_{\text{transition}}^{\text{shape}}(e_{c_k}^i, e_{c_{k+1}}^j) = \begin{cases} 0 & \text{if } e_{c_k}^i \text{ and } e_{c_{k+1}}^j \text{ are on the same shape} \\ \infty & \text{else} \end{cases}$$

If $e_{c_k}^i$ and $e_{c_{k+1}}^j$ are not connected to each other in G_{network} , we penalize the transition with a high weight. If they are connected, we prefer edges with as little path between them as possible. To calculate the length path between $e_{c_k}^i$ and $e_{c_{k+1}}^j$, we use a bidirectional Dijkstra's algorithm on G_{network} :

$$\mathcal{W}_{\text{transition}}^{\text{path}}(e_{c_k}^i, e_{c_{k+1}}^j) = \begin{cases} \infty & \text{if } e_{c_k}^i \text{ and } e_{c_{k+1}}^j \text{ are not connected in } G_{\text{network}} \\ \text{bi_dijkstra}(G_{\text{network}}, \text{start}(e_{c_k}^i), \text{end}(e_{c_{k+1}}^j)) & \text{else} \end{cases}$$

There are often two GTFS shapes directed in opposite directions. We had the common problem that the user got matched to the shape going in the wrong direction (See Figure 14). Therefore, we introduced a 'direction penalty'. Each **edge** holds an **edge_sequence_number** (**s_nr**) for that purpose (recall Section 2.1.3):

$$\mathcal{W}_{\text{transition}}^{\text{direction_penalty}}(e_{c_k}^i, e_{c_{k+1}}^j) = \begin{cases} 0 & \text{if } \text{s_nr}(e_{c_k}^i) < \text{s_nr}(e_{c_{k+1}}^j) \\ \text{extra_weight} & \text{else} \end{cases}$$

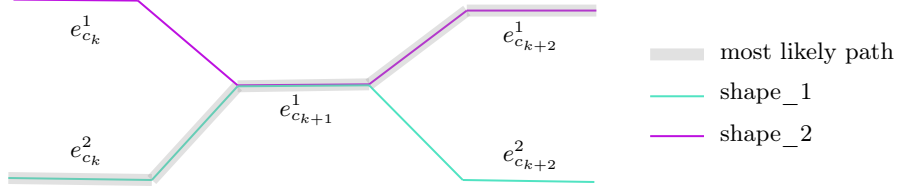


Figure 15: Even though $e_{c_k}^2$ and $e_{c_{k+1}}^1$ as well as $e_{c_{k+1}}^1$ and $e_{c_{k+2}}^1$ share the same edges, the most likely path does not cover a shape.

4.6.3 Finding the Most Likely Trip

So far, we have used the HMM to find a most likely path. This path only consists of edges. We chose our weight functions in a way that each pair of edges share at least one shape. However, this does not guarantee that the most likely path fits to one shape. For example, edges $e_{c_k}^a$ and $e_{c_{k+1}}^b$ could share shape_1, but edges $e_{c_{k+1}}^b$ and $e_{c_{k+2}}^c$ could share shape_2 (see Figure 15). For our PTV Matching, we just choose the shape that fits to most of the chosen edges.

We now need to determine the most likely trip that fits best with the chosen shape. Using the *GTFS Container* dictionaries `shape_id_to_first_edge_and_trip_info` and `trip_id_to_TripsWithStopsAndTimes`, we can find all `TripsWithStopsAndTimes` objects that are on our shape. We can now select the currently active trips using the `is_trip_active` method.

Next, we project the latest GPS point c_l onto the shape. The projected point is called p_l . With `hash_to_edge_to_trip_segments_dict`, we can find the `trip_segment` i that encloses p_l . Once we have the `trip_segment`, we can look up the stops s_i and s_{i+1} at the `trip_segment`'s ends using `trip_id_to_route_and_stop_times`. This also gives us the stops' `arrival_times` and `departure_times`.

Using the `departure_time` of stop s_i and the `arrival_time` of stop s_{i+1} , we can interpolate an approximated position $p_{\text{trip_id}}$ of each currently active trip. The $p_{\text{trip_id}}$ that is closest to p_l gives us the most likely trip. See Figure 16 for an illustration.

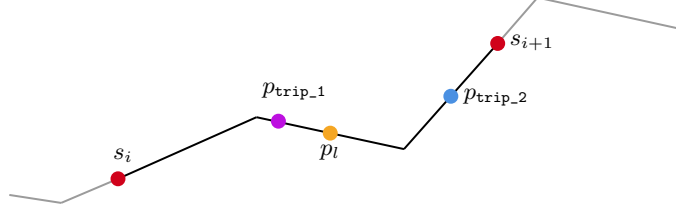


Figure 16: The projection of the latest GPS point p_l is closer to the approximated position p_{trip_1} than to the approximated position p_{trip_2} . Therefore, `trip_1` is our most likely trip.

4.6.4 Avoiding Over-Matching

During the development of *PublicTransitSnapper*, we had the issue that users would get matched to a PTV even if they were just next to a shape. They could be walking, standing or sitting in a café 50 meters next to a tram stop. In order to avoid this issue, which we will call ‘over-matching’ from now on, we introduced two functions.

`are_last_n_coordinates_timewise_too_far_apart`

This function takes a list of timestamped GPS points and two integers n and m .

`are_last_n_coordinates_timewise_too_far_apart` looks at the last n timestamps. If any of them are m minutes or more apart, we can assume that the user is not in a PTV, maybe they are waiting at a stop or just standing next to the shape. The frontend only sends new GPS coordinates to the backend, if a movement of more than 15 meters has been detected. Often, GPS is inaccurate enough that GPS points are sent even if the user has not moved at all. If the GPS is accurate for m minutes or more and thus not sending any coordinates, we do not match when the next request reaches the backend. As we need to loop over the last n timestamps, the runtime of this function is in $\mathcal{O}(n)$.

`are_last_gps_points_close_to_each_other`

This function also takes a list of timestamped GPS points L and two integers n and m . Let R be a set containing the last n points of L .

`are_last_gps_points_close_to_each_other` calculates an average GPS point p :

$$p = \left(\frac{1}{n} \sum_{q \in R} q_{\text{lat}}, \frac{1}{n} \sum_{q \in R} q_{\text{lon}} \right)$$

Now, we remove the the point from R that is the furthest away from p . We do this to soften the impact of outlier GPS points.

Finally, we check if every point left in R is within m meters of p . If this is true for every point left in R , we assume that the user has not moved far enough to be matched and we return **True**. As we need to loop over the last n timestamps, the runtime of this function is in $\mathcal{O}(n)$. See Figure 17 to see an illustration of the problem. Sadly, this method is not a perfect way to avoid over-matching. See Figure 18 for an explanation.

Both functions are called directly after receiving the timestamped GPS points from the frontend. If any of them is **True**, we can inform the frontend that no PTV has been matched.

A problem arising with the use of both functions is the impact of traffic jams or traffic lights. GPS inaccuracies lead to more GPS points collected than anticipated by the 15 meter distance filter implemented in the frontend. A user sitting in a PTV affected by a traffic jam or waiting in front of a traffic light for a longer time would produce GPS points like in the left image of Figure 17. However, this problem is hard to circumvent, as it is difficult to decide whether the user is in a PTV stuck in a traffic jam or if they are in a café next to the road.

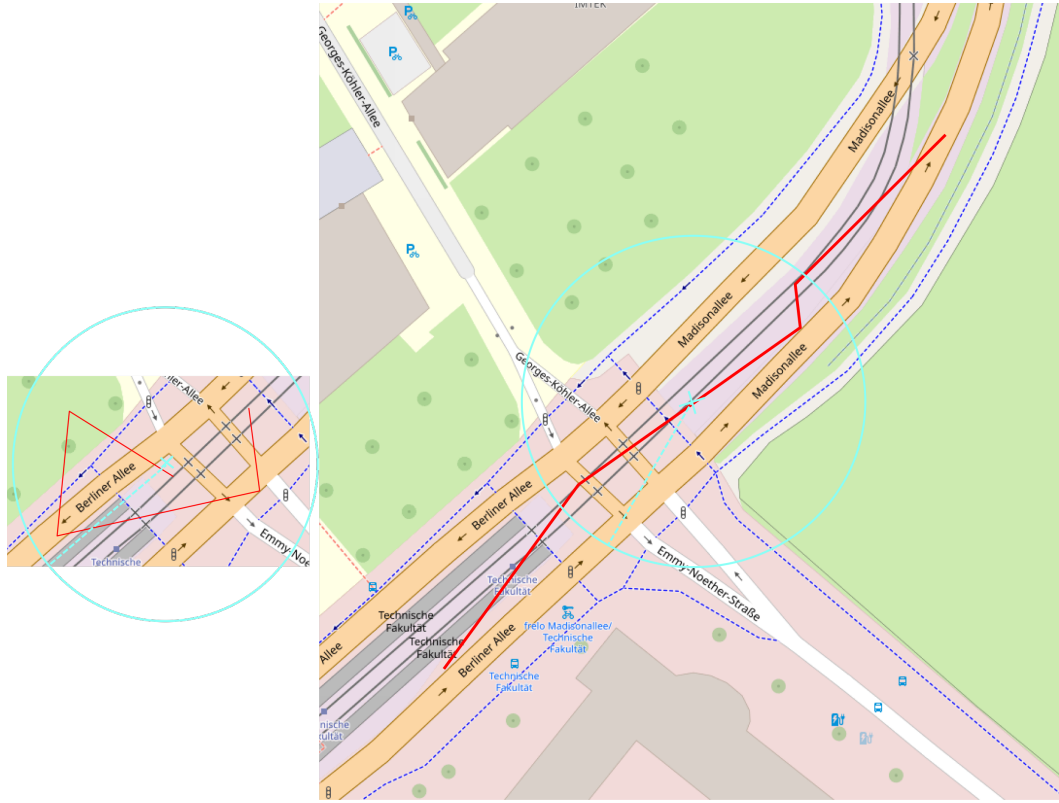


Figure 17: Examples for `are_last_gps_points_close_to_each_other`: Let us say the radius of the turquoise circle is $m = 35$ meters. In the left image, the function would return **True**, as all GPS points are within the radius. Here, we would not try to match to a PTV. In the right image, the function would return **False**, as two GPS points are not within the radius. Thus, we would try to match to a PTV.



Figure 18: `are_last_gps_points_close_to_each_other` is not perfect: Let us assume that there is a shape with a bus drives along the painted GPS point sequence. In this case, the function would remove the GPS point in the upper right corner from L to soften the impact of outliers. Then, none of the remaining points in L would be further away than the radius allows and the function would return `True`. Cases where a shape follows an acute angle are therefore susceptible to failures.

4.7 Fetching New GTFS Data

GTFS data has to be updated regularly. For example, there can be construction sites leading to detours or rail replacement transport. Updating the GTFS data manually can be very annoying, which is why we wrote a tool that undertakes this task for the server host.

Before starting the server, the server host can edit a ‘cities_config.yml’ YAML file and add their desired GTFS data set, as can be seen in the listing below.

```
Freiburg:
  generate-new-shapes: True
  path-to-GTFS: GTFS/Freiburg/VAG
  path-to-OSM: GTFS/Freiburg/OSM
  GTFS-link: link_to_freiburg_GTFS_data.zip
  OSM-link: link_to_openstreetmaps_data.osm.bz2
```

There is also a ‘cities_config.yml’, where the server host can choose the options ‘UPDATE_GTFS’, set an ‘UPDATE_TIME’ or an ‘UPDATE_FREQUENCY’. The tool will then stop server and API to pull the latest GTFS and OSM files from the links in the cities_config.yml. As most public transit agencies do not provide the shapes we need, we use *pfaedle*³³ to generate the shapes for us if specified by generate-new-shapes.

pfaedle takes a GTFS data set and an OpenStreetMaps file³⁴ and calculates the shapes using the OpenStreetMaps data. Without *pfaedle*, the regions covered by *PublicTransitSnapper* would be strongly diminished, considering how highly dependant it is on the GTFS shapes. After fetching the GTFS data and potentially generating the shapes, the server starts again.

Sadly, some agencies like the public transport agency for Hamburg (HVV³⁵) do not provide static links for their GTFS data, but release their new data sets under a new link. In these cases, our GTFS fetching tool is powerless, as it would always fetch the same, eventually outdated data.

4.8 Using Fake GPS Data to Test the Project

In order to test the PTV Matching and the functionality of the frontend, we had to come up with a tool to facilitate handling the GPS data. Without a test tool, we would have to sit in PTVs with a laptop for most of the developing.

Our tool mimics a device traveling along a random shape from the loaded GTFS data set. To give a brief overview, we first generate points along the shape, then

³³<https://github.com/ad-freiburg/pfaedle>

³⁴<https://download.geofabrik.de>

³⁵<https://suche.transparenz.hamburg.de>

‘noisify’ the GPS points. Now we add timestamps to the points so they fit to a trip traveling along the shape. Lastly, we use our web-app in combination with Python’s Selenium³⁶ and the Google Chrome devtools to visualize the PTV Matching with the faked journey.

4.8.1 Generating Noisified Points Along a Shape

For every trip in the GPS data, we know the exact shape it moves on. However, the GPS can be quite inaccurate. So, in order to simulate a device moving along a shape, we need to ‘noisify’ the polyline from the shape.

Firstly, we need to define an average moving speed v in $\frac{m}{s}$, a GPS signal frequency p in $\frac{1}{s}$ and an average GPS accuracy acc in meters. We can also calculate the length len of the shape polyline in meters.

Now, we can calculate the total time needed to travel along the polyline t :

$$t = \frac{len}{v}$$

The total number of GPS signals that we need to simulate the whole trip can be derived as follows:

$$numSignals = t \cdot p$$

As the simulated vehicle is moving with speed v and gets a signal every $\frac{1}{p}$ seconds, the average travelling distance between two signals can be calculated as:

$$s = \frac{v}{p}$$

The next step is to generate points along the polyline. We generate these points for every signal by going along the polyline, one average travelling distance step s at a time.

In order to simulate the GPS inaccuracy, we use a normally distributed deviation with $\mu = 0$ and $\sigma = acc$ for each point.

4.8.2 Annotating the GPS Points With Timestamps

In this step, we annotate the generated points along the polyline with timestamps that fit to the trip we want to simulate. We know the departure times t_k for each stop k of the trip from the *stop_times.txt*. In order to estimate the time at each signal, we can again use the `trip_segments` introduced in Paragraph 4.5.2 (recall

³⁶<https://selenium-python.readthedocs.io/>

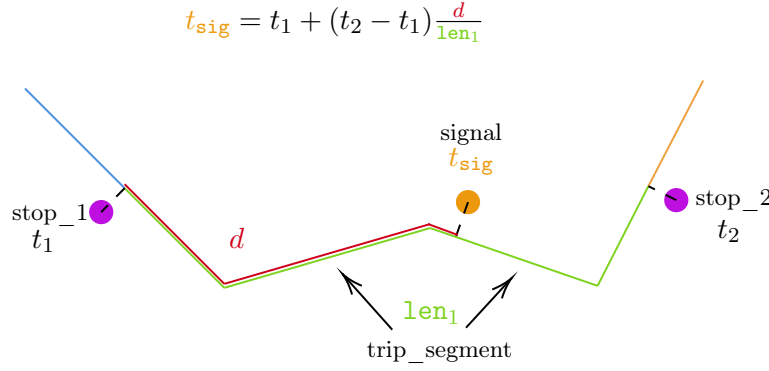


Figure 19: Example of a time stamp interpolation for a signal. We can approximate the time at which a simulated PTV is at the signal t_{sig} by using the shown formula.

Figure 10). In the following, we mean the projections of stops and signals onto the polyline when talking about stops and locations. The time frame (t_k, t_{k+1}) describes the time period the simulated device moves along the current `trip_segment`. We further denote len_k as the length of the `trip_segment` from stop k to stop $(k + 1)$. Let d be the distance from stop k to the signal `sig` along the polyline. The estimated time at the signal t_{sig} can be solved with an interpolation:

$$t_{\text{sig}} = t_k + (t_{k+1} - t_k) \cdot \frac{d}{\text{len}_k}$$

See Figure 19 for an illustrated example.

4.8.3 Using Selenium to Manipulate a Device's GPS Position

In Google Chrome's devtools, one can use the location sensor to manipulate one's position. These sensors are also accessible via Selenium³⁷.

Our tool `ControlChromeDevTools` generates a list of faked GPS points with their approximated timestamps and starts a Google Chrome browser controlled by Selenium. Every n seconds, we can alter the location sensor to the next fake GPS point. In the background, the frontend is running on localhost, so the Selenium browser can access the web-app. Sadly, Selenium cannot control the browser's time settings. Therefore, we need to send the timestamped GPS points to a proxy server that starts when running `ControlChromeDevTools`. The proxy forwards requests to `/connections`, `/shapes` and `/chat` to the backend without changes, but alters the timestamp for each GPS point sent to the `/map-match` endpoint. This way, the backend receives

³⁷https://www.selenium.dev/documentation/webdriver/bidirectional/chrome_devtools/

the precalculated timestamps and not the real timestamps captured by the Selenium browser. As the backend calculations do not depend on the server time, the answers can be displayed on the Selenium browser's web-app just as if the device just moved along a GTFS shape. See Figure 20 for an illustration of the process.

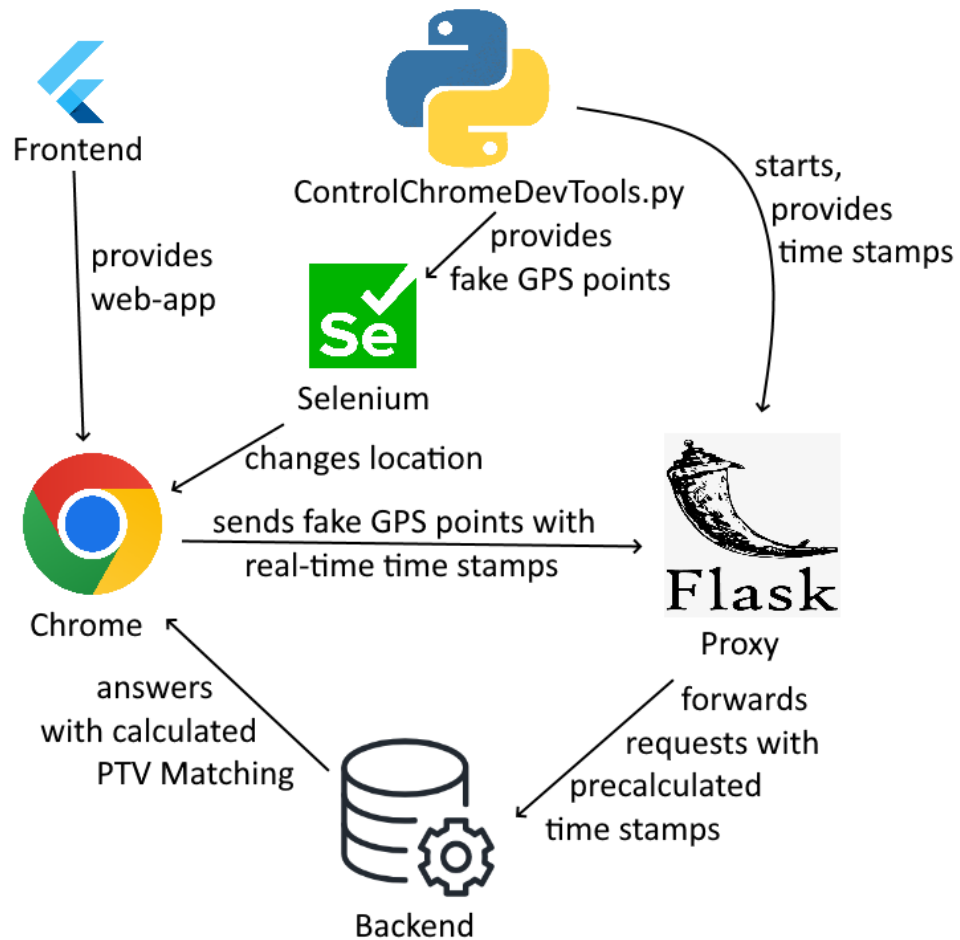


Figure 20: Flow chart explaining the functionality of `ControlChromeDevTools.py`

5 User Study

Over the course of one week, we gave *PublicTransitSnapper* to four testers in Freiburg, Hamburg, Munich and Zurich. We will first look at general feedback given by the testers. Then, we list and recapitulate the individual trips taken in each city. Finally, we will conclude the results.

5.1 General Inquiry

We asked the testers the following general questions:

- Were there any bugs?
 - ⇒ No bugs were found
- Please rate the usability of the app on a scale from 1 to 5, where 1 is bad and 5 is good
 - ⇒ The average usability of the app is 4.5. However, we know the testers personally and the answer might be biased
- Were there unclear elements within the app?
 - ⇒ No unclear elements within the app were mentioned.
- Were there matches when you were not in a PTV?
 - ⇒ In Zurich, the tester noticed that they got matched to their bus once while walking next to the street. Also, while standing at a stop and waiting for the bus.
- Any other remarks?
 - ⇒ The matching gets lost too often while halting at a stop
 - ⇒ In Zurich, the tester noticed that the stops shown on the map were a bit misplaced.
 - ⇒ When the app is closed, no GPS points are gathered.
 - ⇒ No real time information is provided.
 - ⇒ Why is it not possible to click on a stop to see possible connections?
 - ⇒ Show the next stops while waiting at a stop.
 - ⇒ A smooth movement of the vehicle icon would be nice.
 - ⇒ The app needs too much internet (100MB in two days)
 - ⇒ In Munich, no data for S-Bahnen was available

- ⇒ In Hamburg, some stops did not show bus connections on the connections page.
- ⇒ GPS can be very inaccurate

5.2 Trips

Furthermore, we asked the testers to document the following information for each trip:

- Route name?
- Vehicle type (bus, tram, ...)?
- Date and time?
- Name of the boarding station?
- Name of the exit station?
- Was the next stop shown correctly?
- Was the matching correct, and if not, what was the problem?
- Were there wrong matches?
- Were there periods where nothing was matched?

See Tables 1, 2, 3, 4, 5, 6, 7 and 8 for the results of the individual trips.

Freiburg	Route Name	Vehicle Type	Time & Date	Boarding Station	Exit Station
Trip 1	4	Tram	21:31h 10.10.2022	Messe	Stadttheater
Trip 2	5	Tram	21:53h 10.10.2022	Stadttheater	Europaplatz
Trip 3	4	Tram	21:57h 10.10.2022	Europaplatz	Messe

Table 1: Information on the trips in Freiburg

Freiburg	Was the next stop shown correctly?	Was the matching correct?	Were there wrong matches?	Were there periods where nothing was matched?
Trip 1	Yes	$\frac{2}{3}$ of the route was matched correctly	Routes 2 and 1 were matched in the city center	At some stations
Trip 2	Yes	Matching was correct	No	No
Trip 3	Yes	Only after Killianstraße, $\sim \frac{1}{2}$ correct	Routes 2 and 3 in the city center	At some stations

Table 2: Remarks on the trips in Freiburg. In the inner city, up to four trams frequent a two-way track in an every minute cycle. Apparently, they are not coinciding with the static schedule enough for *PublicTransitSnapper* to give an accurate matching. Less frequented tracks were very well matched. The next stop was always displayed correctly. The tester observed that *PublicTransitSnapper* did not match while waiting at some stations.

Hamburg	Route Name	Vehicle Type	Time & Date	Boarding Station	Exit Station
Trip 1	62	Ferry	14:45h 15.10.2022	Finkenwerder	Neumühlen
Trip 2	S3	Subway	15:18h 15.10.2022	Altona	Landungsbrücken
Trip 3	S1	Subway	15:16h 17.10.2022	Othmarschen	Blankenese
Trip 4	S1	Subway	15:30h 17.10.2022	Blankenese	Wedel
Trip 5	S1	Subway	20:58h 17.10.2022	Wedel	Blankenese

Table 3: Information on the trips in Hamburg

Hamburg	Was the next stop shown correctly?	Was the matching correct?	Were there wrong matches?	Were there periods where nothing was matched?
Trip 1	Only once	Only once at a pier	Matched to bus 250 while going over the Elbtunnel	Most of the time. Very inconsistent GPS signal.
Trip 2	No	Never	No, as there was no GPS signal in the tunnel	Always
Trip 3	Mostly	Matched to wrong S1. Displayed end station Wedel, but correct destination was Blankenese	Wrong S1 matched. One time, a bus was matched when tracks close to Osdorfer Landstraße	At stops
Trip 4	Yes	Correct	No	Parts of the tracks were covered by a big wall, which lead to a bad GPS signal. There, nothing was matched
Trip 5	Yes	Yes	No	Some times. GPS was very far from the tracks at times.

Table 4: Remarks on the trips in Hamburg. The tester found the matching on a ferry to be very inaccurate. Also, travelling in tunnels seemed to disable *PublicTransitSnapper*, as there is no GPS signal. Trips above ground seemed to work, although the wrong destination station was matched twice, which effectively means that the wrong trip was matched. Even though the area around Blankenese is highly frequented with different busses close to the tracks, only one bus was matched incorrectly for a short time.

Munich	Route Name	Vehicle Type	Time & Date	Boarding Station	Exit Station
Trip 1	192	Bus	08:39h 18.10.2022	Am Hochacker	Quiddestrasse
Trip 2	U5	Subway	08:50h 18.10.2022	Quiddestrasse	Stachus
Trip 3	U6	Subway	08:01h 20.10.2022	Odeonsplatz	Universität
Trip 4	68	Bus	11:56h 20.10.2022	Universität	Königsplatz
Trip 5	58	Bus	13:32h 20.10.2022	Königsplatz	Siegestor
Trip 6	153	Bus	09:57h 21.10.2022	Universität	Odeonsplatz

Table 5: Information on the trips in Munich

Munich	Was the next stop shown correctly?	Was the matching correct?	Were there wrong matches?	Were there periods where nothing was matched?
Trip 1	Yes	Correct, except first few seconds	Matched bus 195 for the first few seconds	No
Trip 2	No	Incorrect	No	No GPS signal because of the tunnel
Trip 3	Yes	Interestingly correct matching, even though there was a tunnel	No	No
Trip 4	Yes	Most of the time	Briefly matched U6 and a bus incorrectly	No
Trip 5	Yes	Correct	No	No
Trip 6	Yes	Correct	No	No

Table 6: Remarks on the trips in Munich. The tester found that trips below ground worked half of the time. There were incorrect matchings on two trips for a short time. *PublicTransitSnapper* performed very well over all.

Zurich	Route Name	Vehicle Type	Time & Date	Boarding Station	Exit Station
Trip 1	80	Bus	14:33h 14.10.2022	ETH Hönggerberg	Bhf. Oerlikon Nord
Trip 2	31	Bus	16:14h 14.10.2022	Neumarkt	Letzipark
Trip 3	80	Bus	19:12h 14.10.2022	Max-Bill-Platz	ETH Hönggerberg
Trip 4	89	Bus	20:52h 14.10.2022	Kappeli	Bahnhof Altstetten

Table 7: Information on the trips in Zurich

Zurich	Was the next stop shown correctly?	Was the matching correct?	Were there wrong matches?	Were there periods where nothing was matched?
Trip 1	Yes	Correct	No	Some stops or slowly driven curves lead to a non-matching
Trip 2	Yes	Correct	No	Some stops or slowly driven curves lead to a non-matching
Trip 3	Yes	Correct	No	Some stops lead to a non-matching
Trip 4	Yes	Correct	No	One intermediate stop lead to a non-matching

Table 8: Remarks on the trips in Zurich. Whenever *PublicTransitSnapper* matched a PTV, it was the correct choice. The tester remarked that some stops, intermediate stops and even curves lead to a non-matching.

5.3 Differences and Similarities

In this section, we are going to analyze the differences and similarities of the trip records. The testers recorded 18 trips in total. Out of those, there were eight bus, three tram and six subway rides, as well as one ferry ride.

PublicTransitSnapper failed to match to most subways, as there was no GPS signal in the tunnels. Apparently, the location data provided by mobile data does not provide enough information. Interestingly, the U6 in Munich was matched correctly even though the tester was in a tunnel. We cannot explain why.

Another interesting trip was the ferry ride in Hamburg. Sadly, we cannot recreate the GPS path of the testers. They mentioned that the GPS signal on the river Elbe was inconsistently spiking a lot. Thus, it is likely that the anti-over-matching algorithm denied a matching attempt (recall Figure 17).

Overall, in Freiburg, Hamburg and Zurich, the testers mentioned that some stops and in some cases even intermediate stops and curves lead to a non-matching of a trip. Surprisingly, this did not seem to be an issue in Munich, even after interrogating the tester on this subject again. Again, we cannot give a definitive explanation on this open question, as not enough testing with more than one tester per city has been done.

With the certainty of a sample size of eight, we can say that bus matches were very accurate. Only on two out of eight bus rides did the testers record a wrong matching.

Trams and the subways that were not limited by poor GPS reception scored less well. There were wrong matches whenever there was high traffic on or next to the tracks. In Hamburg, even though the right name of the PTV was shown, the destination station was incorrect, which lead to confusion. Testing with real time data would be needed for further clarification on the issue, which is a project for the future when such data exists publicly.

5.4 Conclusion of the User Study

A sample size of 18 is not enough to give significant results. Further testing is required for all kinds of vehicle types, especially ferries. Still, we can see some trends.

We were surprised that busses scored a lot better than railed vehicles, even though some of the bus shapes especially in Munich were very highly frequented by other routes.

About the general feedback; Testers complained that the matching gets lost too often while halting at a stop. This has to do with the anti-over-matching too. We would need more time to balance the anti-over-matching functions.

Apparently, the stops shown on the map were displaced in Zurich. This is an issue with the Zurich GTFS data set, which is a subset of the GTFS data set for the whole Switzerland. It is possible to write software that recognizes and corrects errors in GTFS data sets. This is beyond the scope for this thesis however.

The same applies to the fact that ‘S-Bahnen’ were not matchable in Munich. The agency MVV does not publicly provide the data, an issue which we cannot circumvent.

The fact that no GPS points are gathered while the app is closed or even when the phone’s screen is off is on purpose. We did not want to drain battery unnecessarily.

We cannot do much about the inaccuracy of GPS data, which can be very annoying when dealing with PTVs in tunnels or on water bodies for example.

We assume that the fact that some subway stops in Hamburg did not show bus connections on the connections page has to do with the GTFS data set too.

We will write about requested features by the testers and the internet usage issue in the later chapter ‘Further Investigation’.

6 GTFS in Germany

In this chapter, we will look at the availability of publicly available data on public transit in Germany. In particular, we analyze the availability of GTFS, both static and real time versions. Furthermore, we will give insight into other prominent standards in Germany.

Before the next sections, we need to introduce required German vocabulary: The German federal government is called ‘Bund’. The Bund is subdivided into 16 federal states called ‘Bundesländer’ (singular Bundesland). Each Bundesland is subdivided into several communes.

6.1 Availability of GTFS in Germany

As of 2021, there are over 100 public transit agencies in Germany (see Figure 21). They all use different systems to provide static and real time data on their PTVs. Every Bundesland in Germany has its own umbrella organisation. For Bundesländer like Hamburg, Bremen or Berlin and Brandenburg, there is one centralised agency (HVV¹, VBN² and VBB³ respectively). For other Bundesländer like Baden-Württemberg and Bayern, the umbrella organisations (bwegt⁴ and Bahnland Bayern⁵ respectively) have to comprise many small agencies. See Figure 22 for an illustration.

6.1.1 DELFI

Fortunately, Germany is on the right track to having centralised publicly available PTV data. The ‘Verein zur Förderung einer Durchgängigen Elektronischen Fahrgastinformation e.V.’ (DELFI)⁶, meaning ‘association for continuous electronic schedule information support’, became a registered association in 2016. The original version of DELFI was initiated in 1994 by the German Federal Ministry of Transport, Building and Urban Development. As local public transit is federal in Germany, it is DELFI’s

¹<https://www.hvv.de/>

²<https://www.vbn.de/>

³<https://www.vbb.de/>

⁴<https://www.bwegt.de/>

⁵<https://bahnland-bayern.de/>

⁶<https://www.delfi.de/>

Verkehrs- und Tarifverbünde in Deutschland

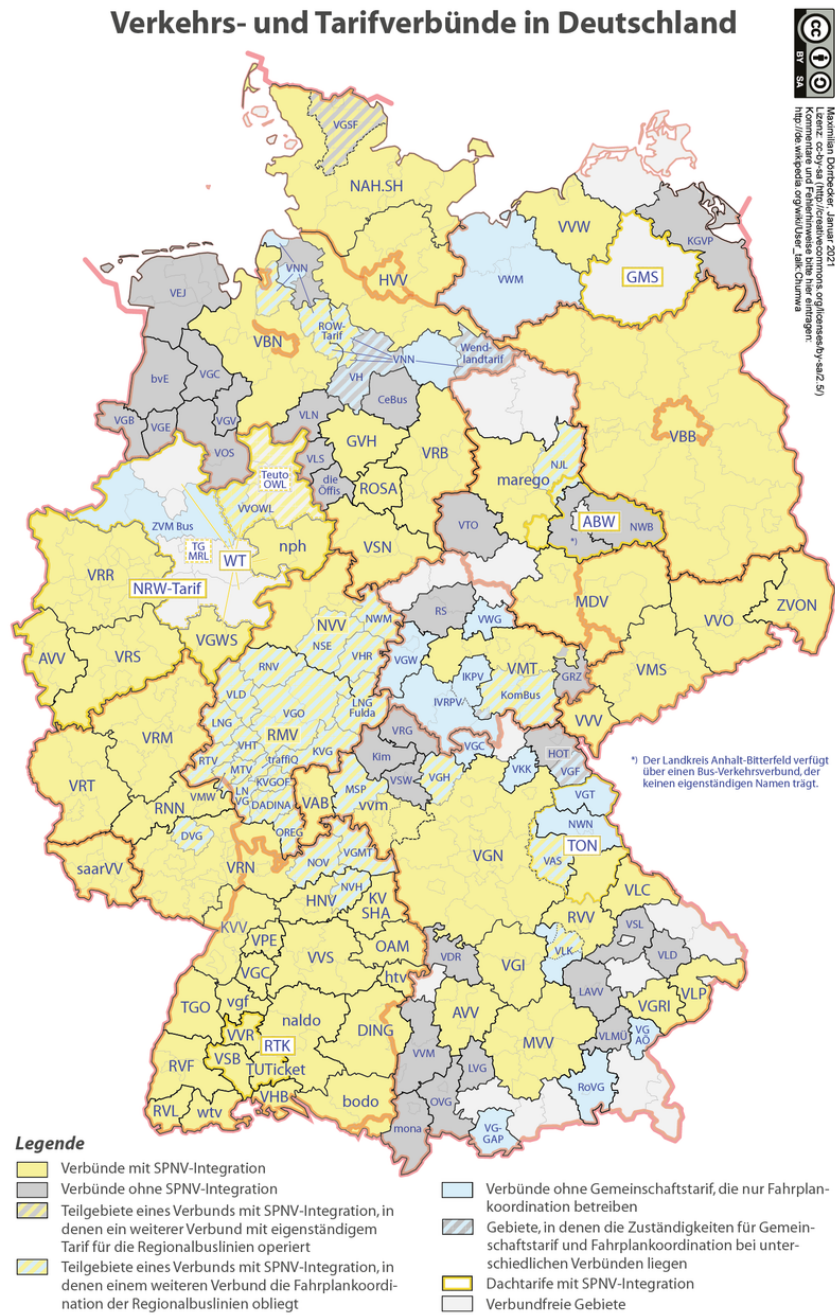


Figure 21: This map shows public transit agencies in Germany 2021 [10].

VERBÜNDE

Finden Sie hier die Ticketseite der Verbünde in Bayern direkt auf den Seiten unserer Partner.

Mehr erfahren

DIREKT ZU UNSEREN PARTNERN

- > AVV – Augsburger Verkehrsverbund
- > Bodo – Bodensee-Oberschwaben Verkehrsverbundgesellschaft
- > DING – Donau-Iller-Nahverkehrsverbund
- > VGI – Ingolstädter Verkehrsgesellschaft
- > MVV – Münchner Verkehrs- und Tarifverbund
- > RVV – Regensburger Verkehrsverbund
- > SVV – Salzburg Verkehr verbindet
- > VAB – Verkehrsgesellschaft Untermain
- > VGN – Verkehrsverbund Großraum Nürnberg
- > VGRI – Verkehrsgemeinschaft Rottal-Inn
- > VLC – Verkehrsgemeinschaft Landkreis Cham
- > VLP – Verkehrsgemeinschaft Landkreis Passau
- > VVM – Verkehrsunternehmens-Verbund Mainfranken

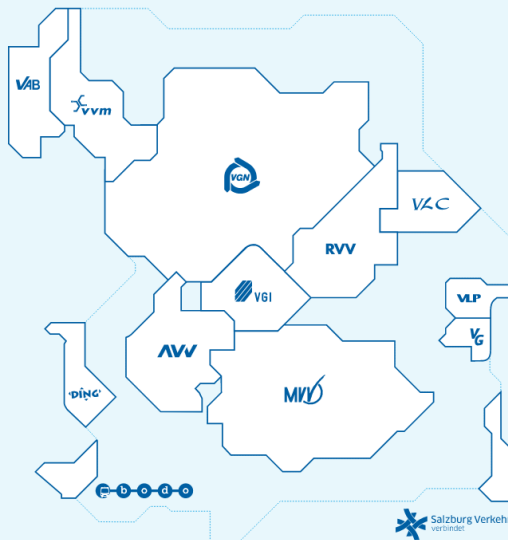


Figure 22: This map shows the partner agencies of Bayern Fahrplan, the umbrella organisation for Bayern. We can see a list of the partner agencies on the left. We can see that some regions of Bavaria are not covered by Bayern Fahrplan [11].

main task to connect the individual local public transport data sets and to enable the planning of trips across agency and Bundesland borders.

In Germany, local public transit can be subdivided into the ‘Schienenpersonen-nahverkehr’ (SPNV) and the ‘öffentlichen Straßenpersonenverkehr’ (ÖSPV). SPNV stands for ‘regional rail transport’ and ÖSPV translates to ‘public road transport’. SPNV with up to 50 kilometers range or SPNV with trip duration under an hour is up to the Bundesländer. ÖSPV comprises busses, subways and trams and is up to communes within the Bundesländer [12].

DELFI’s members are declared by the Bund and the Bundesländer⁷. There can be up to 16 members⁸, one for each Bundesland. Currently, there are 15 members, as the VBB covers two Bundesländer Berlin and Brandenburg. For this reason, the VBB representative can vote twice, once for each Bundesland they are representing. The DELFI association consists of the general meeting comprising all members and the steering committee consisting of one management director and two deputies. See Figure 23 for DELFI’s organisation chart.

DELFI is a non-profit association. They are funded by membership fees and money for research projects from the Bund.

As there are so many agencies in Germany, there needs to be a way to assign unique `route_ids`, `trip_ids`, `shape_ids`, `service_ids` and `stop_ids` overarching all agencies. For the `stop_ids`, DELFI has helped building the central stop register ‘Zentrales Haltestellenverzeichnis’ ZHS⁹ [14]. Within the ZHS, every stop in Germany has a unique ID. As of now, there are over 274,000 registered stops. Each stop serves as a parent station, covering its sub-stations with their individual IDs, exact locations, names and organisations. Shapes and services are also easy to distinguish, so we can generate unique `shape_ids` and `service_ids`. For `route_ids`, DELFI managed to provide over 26,300 unique routes in their data set. The `trip_id` can be derived from the `route_id` by adding characters to a trips corresponding `route_id`.

Unfortunately, not all routes in Germany are publicly available yet. For example, the ‘S-Bahn’ (parts of the subway) in Munich does not appear in any GTFS data set we could find. On their website, they proclaim that they do not have permission to make their GTFS data set publicly available for subway, tram and metro busses [15].

The static GTFS data set¹⁰ provided by DELFI is being updated on a daily basis. It is able to do routing across Bundesländer borders. However, the *shapes.txt* only consist of direct connections from stop to stop for each trip and is thus not eligible for *PublicTransitSnapper* or PTV Matching in general.

⁷https://www.delfi.de/media/satzung_delfi_verein_160906_1.pdf

⁸<https://www.delfi.de/de/strategie-technik/mitglieder/>

⁹<https://zhv.wvigmbh.de/>

¹⁰<https://www.opendata-oepnv.de/ht/en/organisation/delfi/start>

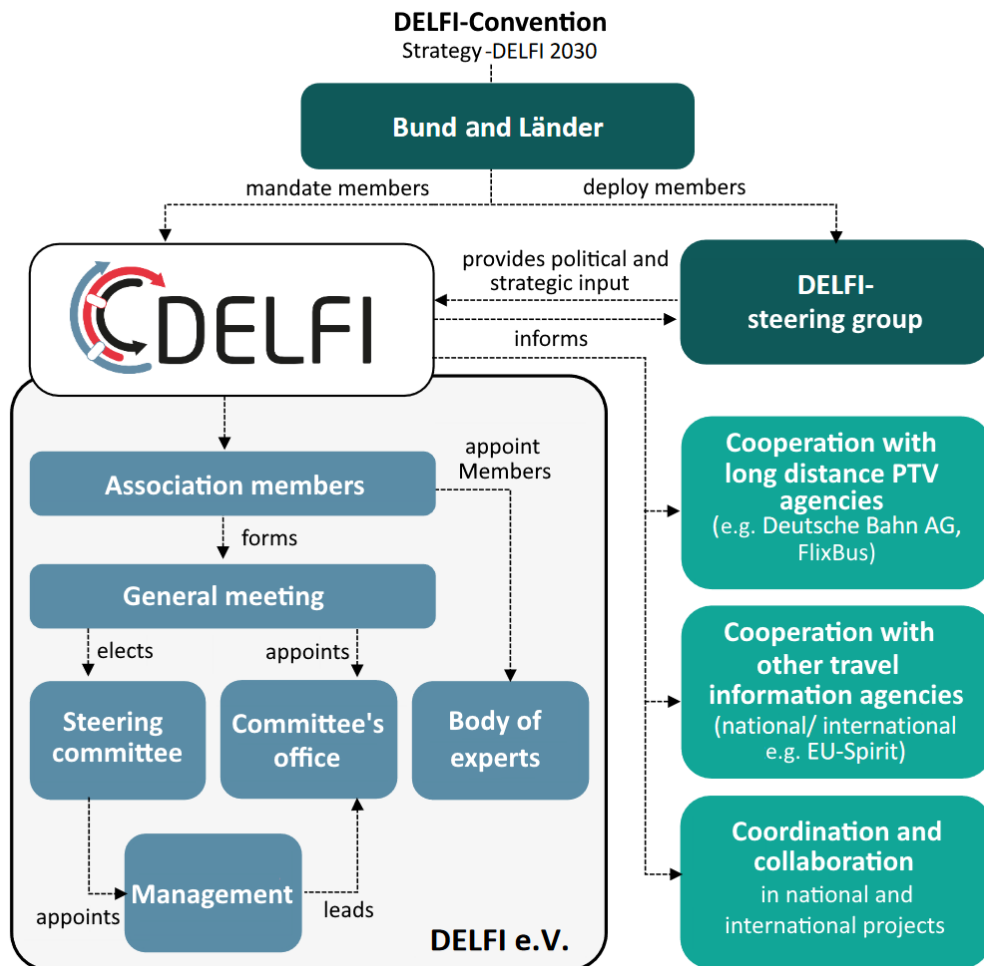


Figure 23: This organigram shows the structure of the DELFI e.V. [13].

In 2020, DELFI's general meeting agreed on their new ten year plan 'Strategie DELFI-2030'¹¹. They decided to maintain and improve the current static schedule data sets like GTFS. Furthermore, DELFI planned to launch a project to make real time data publicly available. The project is called DEEZ ('Deutschlandweite Echtzeitdaten'), meaning 'real time data throughout Germany'. More resolutions discuss a generalised ticketing service, connectivity with international partners and a more thorough quality assurance.

6.1.2 DEEZ - Real Time Data Throughout Germany

Real time data throughout Germany is a very desirable progress. A goal could be to provide real time data for the whole country like the 'open data platform mobility Switzerland'¹² does for Switzerland.

Even though some local public transport agencies provide real time data in a different format, there are no official GTFS-RT feeds from German public transit agencies available. There are some unofficial GitHub projects^(13,14) providing GTFS-RT for certain cities by translating other formats. The first step to an official generalised GTFS-RT feed is a generalised real time data stream. This is the main task of DEEZ. It is their plan to create two 'Regio-Cluster' (see Figure 24), one for northern Germany, one for southern Germany. Each Bundesland can then send their accumulated real time feed to their corresponding 'Regio-Cluster'. It is each Bundesland's task to provide accurate and quality tested real time data.

DEEZ will not only include information on PTVs like delay and crowdedness, but also on the stations. For example, a broken escalator would be listed in a real time request about a certain station.

The DEEZ project is scheduled to be finished towards the end of 2022. If everything went according to plan, DEEZ are in a trial operation and maybe already in normal operation at this point in time (October 2022).

The northern cluster will use the data format DIVA from the Mentz company¹⁵, whereas the southern cluster will use a HaCon¹⁶ format called HAFAS. This means that an official GTFS-RT stream is likely not a priority for DELFI.

While the Bundesländer provide real time data on local public transit (SPNV and ÖSPV), there is also an API for long-distance public transport maintained by the Deutsche Bahn¹⁷.

¹¹<https://www.delfi.de/de/strategie-technik/aufgaben/>

¹²<https://opentransportdata.swiss/en/cookbook/gtfs-rt/>

¹³<https://github.com/derhuerst/berlin-gtfs-rt-server>

¹⁴<https://github.com/derhuerst/hamburg-gtfs-rt-server>

¹⁵<https://www.mentz.net/en/solutions/>

¹⁶<https://www.hacon.de/en/portfolio/information-ticketing/>

¹⁷<https://data.deutschebahn.com/dataset/api-fahrplan.html>

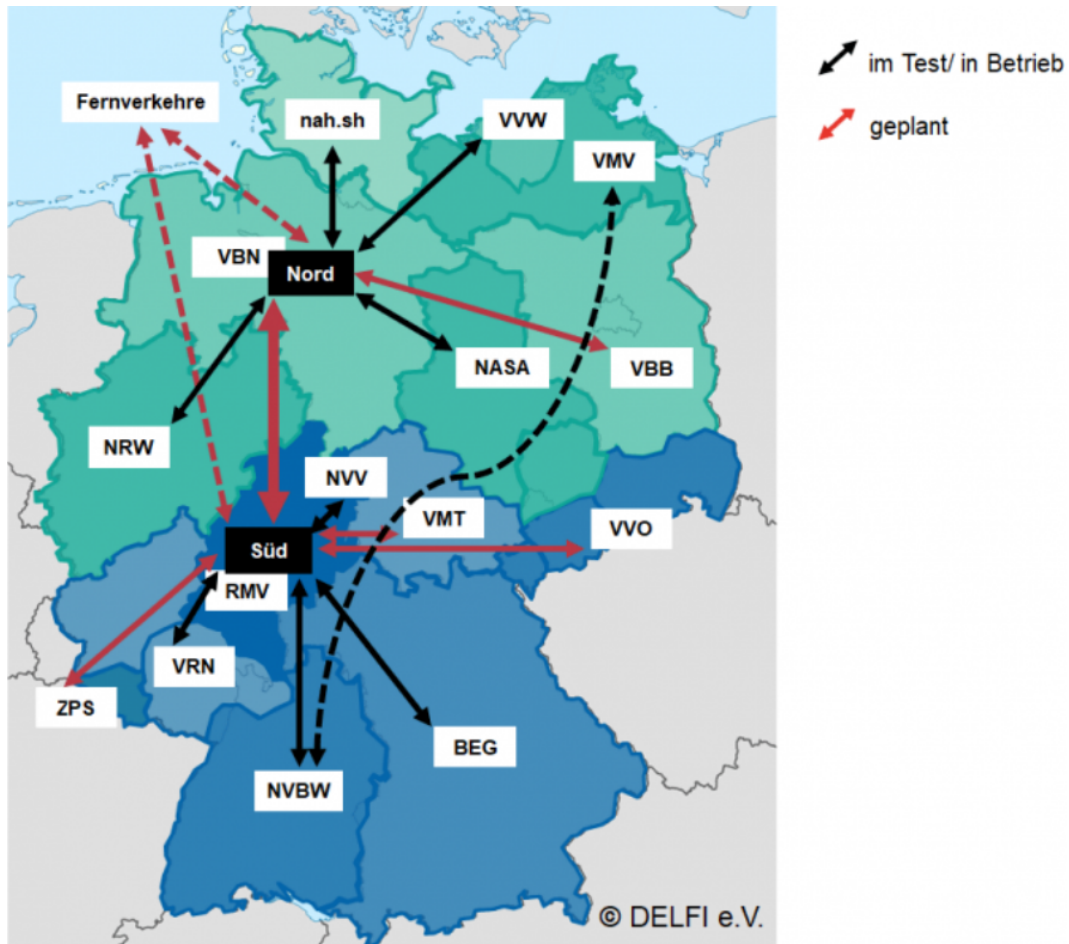


Figure 24: This map shows the two ‘Regio-Cluster’s bundling their Länder’s real time feeds. The arrows indicate the state of the connections. Red arrows are planned connections, black arrows indicate connections that are either being tested or that are already active. This graphic was created on the 15.05.2021. As of now (October 2022), the DEEZ project is likely more advanced. Source: [16]

6.2 Prominent German Industrial Standards

In Germany, many public transport agencies use software by certain companies to manage their PTVs, fares and more. Often consulted companies are HaCon, Mentz, HBT and IVU Berlin. In the following, we are briefly giving an overview on the most common software products HAFAS, developed by HaCon, and DIVA, created by Mentz.

6.2.1 HAFAS

HAFAS (‘HaCon Fahrplan-Auskunfts-System’) is an abbreviation for ‘schedule information system by HaCon’. HaCon is a subsidiary of the Siemens AG. HAFAS is a widely used industrial standard. Companies like the Deutsche Bahn (DB)¹⁸, the Österreichische Bundesbahnen (ÖBB)¹⁹, the Schweizerische Bundesbahnen (SBB)²⁰ and more important public transit agencies from more than 150 regions in 25 countries [17] use HAFAS. For the DB, HAFAS has been industry standard since the beginning of the 1990s on the occasion of DB’s ‘Kurs 90’ project [18].

Today, HAFAS is capable of providing scheduling information with geography data like shapes, real time data, ticketing and fare information. For exchanging data between HAFAS installations, HaCon developed the HAFAS RohDatenFormat (HRDF)²¹. It is very hard to read in contrast to the GTFS CSV files.

6.2.2 DIVA

DIVA (‘Dialoggesteuertes Verkehrsmanagement- und Auskunftssystem’) is short for ‘traffic management and information system steered by dialogue’. It is developed by the Mentz company.

DIVA was first released in 1979. Since then, the Mentz company has constantly developed new features and improvements. Today, DIVA is a complete management system for public transport agencies. With DIVA, agencies can create and optimize timetables and schedule PTVs, manage their employees using a duty scheduler and monitor their PTVs live. They even provide a service to schedule charging for electric busses. DIVA also comes with a vehicle monitoring app called ‘AVM Light’ that can be used by displays in PTVs or at stations. AVM Light can parse real time data feeds like GTFS-RT and update the information displayed accordingly.

¹⁸<https://bahn.hafas.de/>

¹⁹<https://oebb-live.hafas.de/>

²⁰<https://opentransportdata.swiss/en/cookbook/hafas-rohdaten-format-hrdf/>

²¹<https://opentransportdata.swiss/wp-content/uploads/2016/10/hrdf.pdf>

6.3 Concluding GTFS in Germany

As of now, public transport in Germany is still quite obscure, but the situation is improving. The Bund is cooperating with the Bundesländer to bundle static data sets and live feeds from local public transport agencies to make them publicly available. The two companies HaCon and Mentz share this task in a ‘2-cluster-system’. DELFI already provides an unfinished static GTFS data set which they want to improve. Providing a generalised GTFS-RT feed is likely not a priority.

7 Further Investigation

In this chapter, we discuss possible extensions of *PublicTransitSnapper*.

7.1 Improving the Frontend

7.1.1 Showing Real Time Updates

Even though *PublicTransitSnapper* can support a GTFS-RT stream if provided by the agency and use the data to calculate a more accurate PTV Matching, we do not show the user delays or other streamed events in the app. For example, we could show the delay of the vehicle on the front page and on the map in a small box next to the vehicle icon. On the connections page, we could annotate the individual connections with a red ‘+ n minutes’ label. We assume that an implementation would take less than a week.

7.1.2 Showing Connection Issues

Currently, the only indicator to see whether the frontend is properly linked to the backend is by trying to send a chat message. If it appears, the frontend and the backend are linked. Implementing good looking messages for successful connections and connection errors while covering all edge cases would probably take up to one week.

7.1.3 Offline Maps

While the communication with the backend does not use a lot of mobile data, downloading the map can cost a lot of cellular data. As SfMaps, the Flutter package we used for the map, does not provide built-in offline maps, we would have to create an offline maps feature. This should take less than two weeks.

7.1.4 Stop Features

The user study testers requested features where the user could press on a stop on the map, which would show the connections at the stop. This would be very easy to implement, as most necessary functions are already in use. We assume an implementation would take two to three days.

Another requested feature is that if a user is not matched to a vehicle, the connections page could show the connections of the geographically closest stop. The implementation would probably also be relatively easy, although there are some corner cases to address. Firstly, we would set a limit for a maximum distance allowed to the closest stop. Also, there can be multiple stops close to each other. Here, we could also use the average position of the last n stops, as we did for the anti-over-matching methods. We assume that an implementation would not take more than five days.

7.2 GTFS Frequencies

Some public transit agencies provide the optional *frequencies.txt* file¹ in their GTFS data set. Using this file, agencies can reduce the length of their *stop_times.txt* file by specifying a time between departures from the same stop in the *frequencies.txt* file.

PublicTransitSnapper does not support GTFS data sets relying on *frequencies.txt*. We assume that an implementation would take about one to two weeks.

7.3 City-Specific Information

PublicTransitSnapper could be optimized so that it works very well for one particular city or GTFS data set. For example, we could tweak the anti-over-matching functions (see section 4.6.4) in a way that we would not over-match in Freiburg. This might decrease the accuracy of the functions for other GTFS data sets though. Furthermore, GTFS data sets can have optional files like *transfers.txt*, *fare_rules.txt* and more files that provide additional information that could be displayed in the app.

We assume that applications like Citymapper² (recall section 3.1.2) tweak their algorithms so they fit very well to their supported individual cities.

Focusing on optimizing one data set could take months, maybe years to implement, considering the amount of testing that has to be performed and depending on the size of the GTFS data set. For this issue, we would again have to code and test from aboard the PTVs, which is expensive and time consuming.

¹<https://developers.google.com/transit/gtfs/reference#frequencies.txt>

²<https://citymapper.com/>

7.4 Generating Real Time Data

We could use the PTV Matching to generate an estimated delay or earliness based on devices matched to the same PTV.

The basic idea is rather simple. Take all devices matched to the same trip. Based on an average location of the devices, we can interpolate the position of the PTV along its shape and thus calculate the position in time relative to the scheduled time.

Collecting every device matched to the same trip is the same process we used for the chat (see section 4.4.4). We assume that calculating the average location and the interpolated position in time is simple. However, testing and balancing this feature sounds like a very hard task to accomplish for two people.

As future investigation, we propose an extension of our fake GPS data tool (recall section 4.8). The extended version would have to simulate n devices and have the option to be given a delay/earliness. We assume this would take about a week.

However, as R. Wu's evaluation [2] and the user study showed, the PTV Matching can often be wrong in more frequented areas. This issue could also have a workaround that is even harder to test as a team of only two developers. We can be quite confident in our matching for trips that start in a less frequented area. *PublicTransitSnapper* users that start their journey in these spots could be given a high 'certainty' value c . Users that travel together for a longer time could be linked in a data structure D , so the users can pass their certainty to other users. The data structure D could combine currently active trips with their scheduled position and their linked user devices positions with their interpolated position in time.

Implementing and polishing this idea could take months of development and testing. Also, we would probably need more advanced tools and/or more people for testing.

7.5 The Debatable Use of Python

As elucidated in section 4.2.1, we used Python for most of our backend. R. Wu's evaluation shows, that the PTV Matching requests take less than 0.5 seconds on the tested servers [2]. This is fast enough, as PTVs usually need longer than 0.5 seconds to travel 15 meters ($30\frac{m}{s} \approx 108\frac{km}{h}$). The 15 meters relate to the 'distance filter' explained in section 4.4.1. Still, we assume that this time could be improved by implementing the whole backend in C++ . Furthermore, we strongly assume that the needed memory could also be diminished, as we would have to manage the memory manually instead of having to use Python's garbage collector.

7.6 Testing PublicTransitSnapper on Long Distance Trips

As the estimate in the section above shows, it is very well possible that long distance trips like a train ride with an ICE ('inter city express') could overcharge the backend server by sending too many requests. We have not tested this issue. Testing and potentially fixing this issue would probably not take more than a week.

8 Conclusion

We built an application called *PublicTransitSnapper* that is able to match a users timestamped location coordinates to a PTV based on GTFS data in many situations. For that, we created a frontend for the user and connected it to a backend.

The frontend is a Flutter app and can thus be run as a web-app or as a mobile phone app. The app shows the user if they are matched to a vehicle. If so, it provides information on the current trip like the destination station and the next stop. Possible transfer possibilities can be displayed too. Furthermore, it is possible to chat with users that have been matched to the same vehicle.

The backend is capable of calculating a PTV Matching. We used a HMM based approach in combination with bidirectional Dijkstra’s algorithm to compute the most likely path through the HMM. The HMM contains only edges from active PTVs. After calculating the most likely path through the HMM, we just need to find PTVs that are close to sent GPS points. In order to solve this problem, we estimated the positions of PTVs using the stop times from the GTFS files. As for data structures, we used STRtrees and directed graphs in combination with dictionaries to achieve fast run times. To avoid over-matching, we introduced functions to prevent matching whenever the GPS points are too close to each other or if the time in between sent GPS points is too long.

Furthermore, we built a helpful tool to simplify the management of GTFS data. A server running the backend can automatically fetch up-to-date GTFS files from the provider. Another tool spares testers and developers to travel in PTVs. We used a combination of Python’s Selenium, Flutter’s ability to build web-apps and noisified GPS points to simulate a trip along a GTFS shape.

The user study has revealed gaps in the functionality of the *PublicTransitSnapper*. While bus matchings seem to have a high chance of success, subway trips still have issues. A common cause of failure are tunnels or spots with poor GPS connection in general. Many problems require more testing, like balancing anti-over-matching or adjusting the HMM’s weight functions.

Our PTV Matching could be more accurate with real time data. While *PublicTransitSnapper* supports GTFS-RT streams, there are no official usable GTFS-RT streams available in Germany. This could change in the future however, as DELFI, the German ‘association for continuous electronic schedule information support’ is planning to improve the amount of publicly available data on PTVs. They plan

to bundle the real time data streams of regional public transport associations in a project called DEEZ, meaning ‘real time data throughout Germany’. We hope that this will lead to a publicly available GTFS-RT stream in the future.

9 Acknowledgments

First and foremost, I would like to thank Dr. Patrick Brosi. As my adviser, you helped me with suggestions, server troubles, knowledge on the behavior of public transit vehicles and stories. I would also like to thank Prof. Dr. Hannah Bast for letting Robin and me work together on this project. Many thanks to my parents Tat and Tom, as well as my grandma Traudl for sponsoring my studies. Thanks to everyone who participated in the user study, in particular Tat, Tom, Dome and Robin. Thank you Julian and Felix for letting me use your server for user study purposes and helping me in the middle of the night. To Tat, Mario, Dome, and Robin, many thanks for proofreading, you have improved this thesis by a lot. Lastly, thank you Robin for working with me on this odyssey of a project.

10 Appendix

10.1 Data Structures

Every domain specific data structure mentioned in this thesis will be specified here.

10.1.1 STRtrees

For the PTV Matching implementation, we used shapely's STRtree¹ for quick access to spatial data like `edges` and GPS coordinates. An R-tree is a tree data structure used for storing spatial data indexes in an efficient manner. R-trees are very useful for spatial data queries². They store objects in leaf bounding boxes. The leaves are again packed into bounding boxes. R-trees are especially fast on intersection queries and nearest neighbor queries, which we commonly use for the PTV Matching. 'STR' stands for 'Sort-Tile-Recursive', which is a simple and efficient algorithm for the packing of the bounding boxes [19].

See Figure 25 for an example R-tree.

10.2 Algorithms

10.2.1 Dijkstra's Algorithm

Dijkstra's algorithm [20] is a greedy dynamic algorithm that computes the lengths of the shortest paths from start node s to every other node in a weighted graph G . G 's weights cannot be negative.

The algorithm maintains a priority queue Q of nodes that have not been processed yet, and a set S of nodes that we already know the minimal distance to. For every node $n \in G$, the algorithm keeps track of the approximated shortest path length $d(n)$. Initially, $d(n) = \infty$ for all $n \neq s$ ($d(s) = 0$). Every iteration, Dijkstra's algorithm picks the first element u off priority queue Q , which is initialized with starting node s . We can now add u to S . For every neighbor $v \in \text{adj}(u) \setminus S$ of u , we can update

¹<https://shapely.readthedocs.io/en/stable/manual.html>

²<https://www.geeksforgeeks.org/introduction-to-r-tree/>

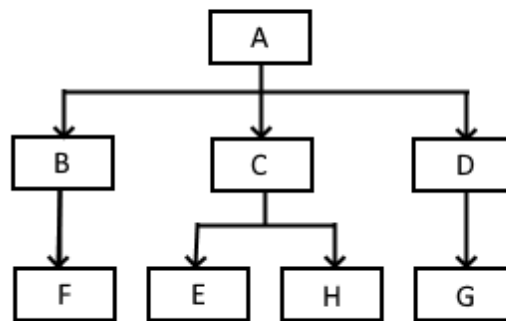
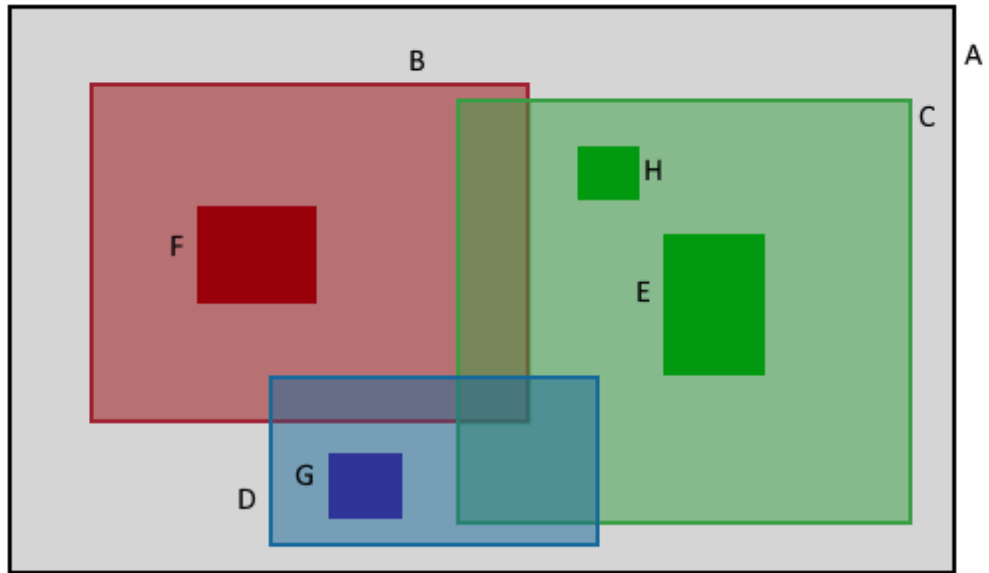


Figure 25: Example of an R-tree. The solid boxes represent actual spatial data. The tree below shows how the boxes are interlinked within the data structure.

the weights $d(v) = \min(d(u) + w(u \rightarrow v), d(v))$. If the new weight of v has become smaller, we can add v to Q .

We can visualize the algorithm as pseudo-code in Listing 10.1.

```
while Q is not empty:
    u = Q.pop()
    S.add(u)
    for v in adj(u):
        if v not in S:
            old_weight = d[v]
            d[v] = min(d[u] + w(u, v), old_weight)
            if d[v] < old_weight:
                Q.add(v, priority=d[v])
```

Listing 10.1: Pseudo-Code Dijkstra’s Algorithm

Dijkstra’s algorithm computes the lengths of the shortest paths from start node s to every other node in a weighted graph G . However, if we are just interested in the length of the shortest path between start node s to end node e ($s, t \in G$), we can speed up the calculation by using a bidirectional Dijkstra.

The bidirectional Dijkstra approach can save many iterations over the basic Dijkstra’s algorithm³. If every $t \in G$ has an average outdegree m , and the shortest path from s to e has length n , we can expect m^n iterations when using the basic Dijkstra’s algorithm. If we run two alternating searches, one starting at s , the other starting at e , we can stop when one node $k \in G$ has been reached by both searches. If k is in the middle of s and e , we can expect $2m^{\frac{n}{2}}$ visited nodes⁴.

10.3 GTFS Example

See Listings 10.3.1, 10.3.2 and 10.3.3 for a GTFS example.

³www.youtube.com/watch?v=1oVuQsxkhY0

⁴www.homepages.ucl.ac.uk/~ucahmto/math/2020/05/30/bidirectional-dijkstra.html

GTFS Example: Page 1

routes.txt					
route_id	route_short_name	route_long_name	route_type	route_color	route_text_color
route01	1	Berlin - Hamburg	2	A9A9A9	FFFFFF
route02	N66	Central Station - Airport	3	5DBB63	FFFFFF

trips.txt			calendar_dates.txt		
trip_id	route_id	service_id	service_id	date	exception_type
route01_trip01	route01	service01	shp_01	20220605	1
route01_trip02	route01	service01	shp_02	20220612	1
route01_trip03	route01	service02	shp_03	20220619	1
route02_trip01	route02	service03	shp_04	20220626	2
route02_trip02	route02	service03	shp_05		

stops.txt		
stop_id	stop_name	stop_lat, stop_lon
stop01	Airport	48.0133918118, 7.8339651653758
stop02	University	48.01339181171813, 7.833965165375644
stop03	School	48.01339181170815, 7.83396516534867
stop04	Bakery	48.01339181178438, 7.8339651653458

		stop_times.txt				
trip_id,	stop_id,	arrival_time,	departure_time,	stop_sequence		
route01_trip01,	stop01,	07:38:00,	07:38:00,	0		
route01_trip01,	stop02,	07:45:00,	07:50:00,	1		
route01_trip01,	stop01,	08:01:00,	08:05:00,	2		
route01_trip02,	stop01,	19:38:00,	19:38:00,	0		
route01_trip02,	stop02,	19:45:00,	19:50:00,	1		
route01_trip02,	stop01,	20:01:00,	20:05:00,	2		
route01_trip03,	stop01,	07:38:00,	07:38:00,	0		
route01_trip03,	stop02,	07:45:00,	07:50:00,	1		
route02_trip01,	stop03,	08:12:00,	08:12:00,	0		
route02_trip01,	stop04,	08:38:00,	08:38:00,	1		
route02_trip02,	stop03,	16:12:00,	16:12:00,	0		
route02_trip02,	stop04,	16:38:00,	16:38:00,	1		

		calendar.txt						
service_id,	monday,	tuesday,	wednesday,	thursday,	friday,	saturday,	sunday,	start_date, end_date
service01,	1,	1,	1,	1,	1,	1,	0,	20220101, 20221231
service02,	1,	1,	1,	1,	1,	1,	0,	20220101, 20221231
service03,	0,	0,	0,	0,	0,	0,	1,	20220101, 20221231

shapes.txt				
shape_id,	shape_pt_lat,	shape_pt_lon,	shape_pt_sequence	
shp_01,	48.0133918117,	7.8339651653758,	0	
shp_01,	48.01339181171814,	7.833965165375645,	1	
shp_01,	48.01339181171,	7.833965165376,	2	
shp_01,	48.01339181175,	7.83396516537678,	3	
shp_02,	48.01339181171814,	7.833965165375645,	0	
shp_02,	48.01339181171,	7.833965165376,	1	
shp_02,	48.01339181175,	7.83396516537678,	2	
shp_02,	48.0133918117,	7.8339651653758,	3	
shp_03,	48.0133918117,	7.8339651653758,	0	
shp_03,	48.01339181171814,	7.833965165375645,	1	
shp_04,	48.01339181170815,	7.83396516534868,	0	
shp_04,	48.01339181176228,	7.83396516523458,	1	
shp_04,	48.01339181178438,	7.8339651653458,	2	
shp_05,	48.01339181178438,	7.8339651653458,	0	
shp_05,	48.01339181176228,	7.83396516523458,	1	
shp_05,	48.01339181170815,	7.83396516534868,	2	

Bibliography

- [1] T. T. Avichal Garg, Product Manager, “Public Transit via Google,” *Google Official Blog*, p. 1, 2005.
- [2] R. Wu, “PublicTransitSnapper: Dynamic Map-Matching to Public Transit Vehicles,” Bachelor’s Thesis, Albert-Ludwigs-Universität Freiburg, 2022.
- [3] Google, “Commute better with Pigeon, the crowdsourced transit app.” url: blog.google/technology/area-120/pigeon-transit-app-new-cities/, Nov 5, 2019.
- [4] Citymapper, “The Ultimate Technology for Mobility in Cities.” url: citymapper.com/company.
- [5] P. Newson and J. Krumm, “Hidden Markov Map Matching through Noise and Sparseness,” in *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS ’09, (New York, NY, USA), p. 336–343, Association for Computing Machinery, 2009.
- [6] G. Forney, “The Viterbi Algorithm,” *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.
- [7] H. Koller, P. Widhalm, M. Dragaschnig, and A. Graser, “Fast Hidden Markov Model Map-Matching for Sparse and Noisy Trajectories,” in *2015 IEEE 18th International Conference on Intelligent Transportation Systems*, pp. 2557–2561, 2015.
- [8] S. Kaufmann, “Opening Public Transit Data in Germany.” 2014.
- [9] shapely, “Shapely GitHub: README.rst.” url: github.com/shapely/shapely/blob/main/README.rst, visited on: 30.09.2022.
- [10] M. Dörrbecker (Chumwa), “Verkehrs- und Tarifverbünde in Deutschland.” url: https://de.wikipedia.org/wiki/Liste_deutscher_Tarif-_und_Verkehrsverb%C3%BCnde#/media/Datei:Karte_der_Verkehrsverb%C3%BCnde_und_Tarifverb%C3%BCnde_in_Deutschland.png, January 2021.
- [11] bahnland-bayern, “VERBÜNDE.” url: <https://bahnland-bayern.de/de/tickets>, visited on 14.10.2022.

- [12] Forschungsinformationssystem Mobilität und Verkehr, “Zuständigkeiten für die Verkehrsinfrastrukturfinanzierung im föderalen System.” url: <https://www.forschungsinformationssystem.de/servlet/is/516068/?clsId0=276646&clsId1=276651&clsId2=276890&clsId3=0>, created: 07.01.2021.
- [13] DELFI e.V., translated by G. Freiwald, “DELFI Convention.” url: https://www.delfi.de/media/delfi-organigramm_10.2021_2.pdf, translated on 18.10.2022.
- [14] J. R. René Maier, Marco Felix Gennaro, “MF014: Moderne Informationsdienste im ÖPNV – mit DELFI e.V..” url: <https://mobilitaetsfunk.de/mf014-moderne-informationsdienste-im-oepnv-mit-delfi-e-v/>, created on: 28. April 2021.
- [15] MVV München, “Soll-Fahrplandaten (GTFS).” url: <https://www.mvv-muenchen.de/fahrplanauskunft/fuer-entwickler/opendata/index.html>, visited on: 16.10.2022.
- [16] rms, “Zwei Drehscheiben bündeln Daten hunderter Unternehmen.” url: <https://www.rms-consult.de/news/projekt-deez-deutschlandweit-minutengenau-in-bus-und-bahn-informiert/>, created: 31.05.2021.
- [17] HACON, “Referenzen.” url: <https://www.hacon.de/unternehmen/>, visited on: 17.10.2022.
- [18] Computerwoche, “Bahn will offenen Rechner-Verbund schaffen.” url: <https://www.computerwoche.de/a/bahn-will-offenen-rechner-verbund-schaffen>, 1157261, written on: 11.11.1988.
- [19] S. Leutenegger, M. Lopez, and J. Edgington, “STR: a simple and efficient algorithm for R-tree packing,” in *Proceedings 13th International Conference on Data Engineering*, pp. 497–506, 1997.
- [20] E. W. Dijkstra, “A note on two problems in connexion with graphs,” *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

