Bachelor's Thesis

PublicTransitSnapper: Dynamic Map-Matching To Public Transit Vehicles

Robin Wu

Examiner: Prof. Dr. Hannah Bast
Adviser: Patrick Brosi

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

September 25th, 2022

Writing Period
25. 07. 2022 - 25. 10. 2022

Examiner
Prof. Dr. Hannah Bast

Adviser
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.

Place, Date	Signature

Abstract

PublicTransitSnapper is an application for matching a user to the public transit vehicle the user is currently traveling on. We base the matching to a public transit vehicle on a sequence of timestamped locations. Public transit vehicles travel along a specific path at a specific time within their public transit network. Since location measurements are noisy, only "snapping" to the closest road might not work. Map-matching can be used to match a sequence of locations to the most probable path in a static road network. For map-matching to a public transit vehicle we need to find a vehicle that follows a similar path to the locations and the vehicle must be at these locations at the measured times. In this thesis we present a solution to this dynamic map-matching problem. Similar to conventional map-matching, we utilize a Hidden Markov Model. Furthermore, we include time information from the schedule to find the most likely public transit vehicle.

Zusammenfassung

PublicTransitSnapper ist eine Anwendung, um zu bestimmen, in welchem öffentlichen Verkehrsmittel ein Benutzer gerade unterwegs ist. Hierfür verwenden wir eine Liste an zeitgestempelten Standorten. Öffentliche Verkehrsmittel fahren in ihrem Verkehrsnetz entlang eines bestimmten Weges zu einer bestimmten Zeit. Da Standortmessungen verrauscht sein können, genügt es nicht, jeden Standort auf die nächstgelegene Straße zu projizieren. Map-Matching kann verwendet werden, um mit einer Liste von Standorten den wahrscheinlichsten Weg in einem statischen Straßennetz zu bestimmen. Für das Map-Matching auf ein öffentliches Verkehrsmittel müssen wir ein Fahrzeug finden, das einen ähnlichen Weg zu den gemessenen Standorten zurücklegt und zusätzlich zu den gemessenen Zeiten an den Orten ist. In dieser Arbeit stellen wir eine Lösung für dieses dynamische Map-Matching-Problem dar. Ähnlich wie beim herkömmlichen Map-Matching verwenden wir ein Hidden Markov Modell. Darüber hinaus beziehen wir Zeitinformationen aus dem Fahrplan mit ein, um das wahrscheinlichste öffentliche Verkehrsmittel zu finden.

Contents

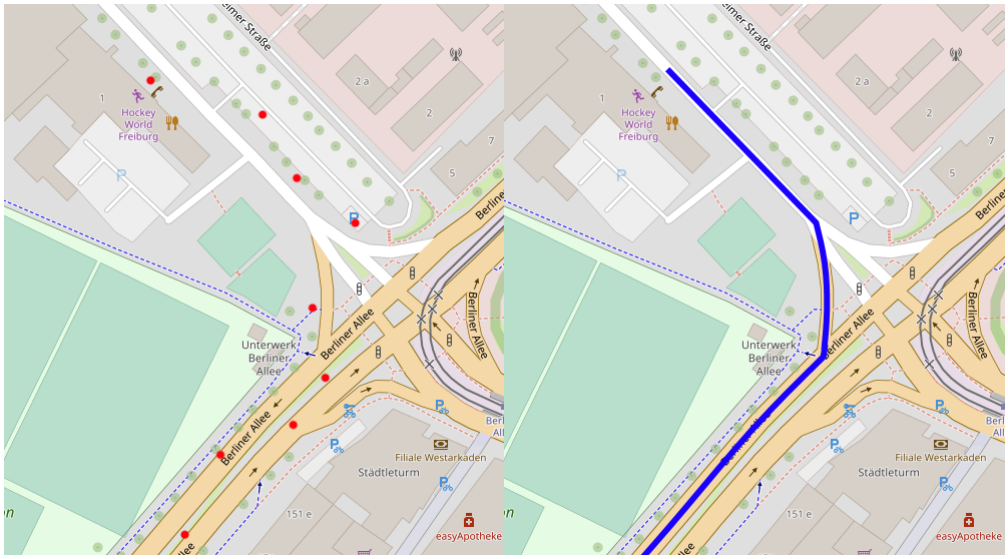
List of Figures

List of Tables

1 Introduction

Imagine sitting in a public transit vehicle and you want to know more information about the current vehicle on your mobile phone. This could be the name of the route, the next stop or transfer possibilities at the next stop. In order to correctly display any relevant information, we need to determine in which public transit vehicle you are currently traveling on. Our application *PublicTransitSnapper* can run on your mobile phone and collect location information on your device. Based on the last collected locations, we try to determine the current vehicle you are on.

This problem can be solved with map-matching. Figure 1 shows map-matching of a sequence of locations to a most likely path in a static road network. The main challenge is that location measurements are not infinitely accurate and may contain measurement noise. Simply ”snapping” every location to the closest street may not be the best possible solution. Solving this kind of map-matching problem is a well-researched topic.



(a) Location measurements **(b)** Matched path

Figure 1: Illustration of map-matching to a static road network.

In contrast to map-matching to a static road network, public transit vehicles are highly dynamic. Public transit vehicles follow a specific path at a specific time. We

cannot simply map-match the sequence of location measurements to the roads of a public transit network. Instead, we need to check for active public transit vehicles and only allow matching onto those.

Our project *PublicTransitSnapper* serves as a solution to this dynamic map-matching problem. *PublicTransitSnapper* is split into a *backend* and a *frontend* part. The *backend* performs a dynamic map-matching, that matches timestamped locations onto the most likely public transit vehicle. Most of the *backend* is implemented in Python[1]. The *frontend* is an application that runs on a mobile device. The *frontend* collects timestamped locations on the mobile device and sends these to the *backend* for processing. Once a public transit vehicle is matched, more information about the current vehicle is displayed to the user. Moreover, we provide a group chat for users sitting in the same vehicle. The *frontend* is built with the Flutter framework[2] which uses the Dart[3] programming language.

We provide *PublicTransitSnapper* as a public open-source project[4]. In the repository, there is a detailed readme for installation and running both the *backend* and *frontend*. Additionally, we provide a tool for simulating locations of a public transit vehicle to test the matching of our application.

[1] `https://www.python.org/`
[2] `https://flutter.dev/`
[3] `https://dart.dev/`
[4] `https://github.com/RobDaNub/PublicTransitSnapper`

2 Related Work

Map-matching tries to determine a most likely path through a road network given a sequence of location measurements. Our work is closely related to this problem.

In order to solve map-matching to a static road network a Hidden Markov Model (HMM) can be used. It was first introduced to map-matching in [1]. In [2], the authors improved the HMM approach, by tuning the emission and transition probability of the HMM. This was done by taking the measurement noise and the underlying road network into consideration. Their approach produced accurate results up to a sampling rate of 30 seconds. Since we are collecting location measurements with a mobile phone, we can expect modern devices to have a similar or higher sampling rate.

Our work is also closely related to map-matching based on schedule data.

With the stops of a public transit vehicle and the underlying road network, determining the most likely path a vehicle has taken was described in [3]. Depending on the type of vehicle the stops can have a high distance between each other. This leads to a highly sparse map-matching problem. In this case a HMM was used with additional inter hop turn restrictions based on [4].

Since we want to provide a responsive experience to the user, we require a fast calculation of the dynamic map-matching. Typically, the Viterbi algorithm [5] is used to calculate the sequence of states with the highest probability in a HMM. In [6], instead of using the Viterbi algorithm the probabilities were determined with a single execution of Dijkstra's algorithm [7] to reduce the run-time. With [8], an alternative using bidirectional Dijkstra was introduced. In most cases only a fraction of all nodes needs to be visited for finding the shortest path [9]. Therefore, we adapt the approach with bidirectional Dijkstra.

In our approach, the main difference to existing work in map-matching is the added dynamic element of public transport vehicles. Added to possible noise of location measurements and the road network of a public transit network we need to consider the schedule data of a public transit network.

In September 2018 Google launched *Pigeon*[1] for generating crowd-sourced real-time data. The idea was that users are able to report delays, incidents, "crowdedness", condition of vehicles or facilities and other information for other users to see. The

[1] `https://www.blog.google/technology/area-120/pigeon-transit-app-new-cities/`

expectation was that the crowd-sourced real-time data is more accurate than real-time data provided by public transport agencies. Moreover, information can be updated faster with many users. The app was launched as a test in six major cities across the USA. Unfortunately, the *Pigeon* project was canceled in 2020.

Another application for public transit vehicles is *transit*[2]. It provides many services, such as finding location of nearby public transport vehicles, finding transfer options and "rider-powered locations", which seems to be similar to crowd-sourced real-time data. In Germany it is only available in Berlin, Hamburg and Ulm. Thus, we were not able to try the features of this app.

Both *pigeon* and *transit* do not have publications available. Thus, we do not have any information about how these applications work in detail.

PublicTransitSnapper was developed together with G. Freiwald. His thesis [10] covers further aspects of *PublicTransitSnapper*, especially details related to the *frontend* part.

[2] `https://transitapp.com/`

3 Background

In this chapter we introduce a representation of a public transit network. Moreover, we introduce the GTFS standard that describes a public transit network. Lastly, we briefly describe location data and calculating distances.

3.1 Public Transit Network

A public transit network (PTN) provides transportation for passengers. It is intended to be used by the general public. Most PTNs operate based on a repeating weekly schedule and service routes according to this schedule. A public transit vehicle (PTV) that services a route follows a specific course of the road and stops at predetermined stops for passengers to board or get off. We refer to the course of the road as the shape of a PTV. There can be different kinds of PTVs within a PTN such as trains, buses, trams or ferries.

3.2 Network Graph

PTVs follow a shape which we can intuitively represent in a directed Graph. A directed Graph $G = (V, E)$ consists of a set of vertices $V = \{v_1, \ldots, v_m\}$ and a set of directed edges $E = \{(v_i, v_j) | v_i, v_j \in V\} = \{e_1, \ldots, e_n\}$. The shape of a PTV is a sequence of locations on a map. Each of the locations is represented as a vertex. We connect the vertices of consecutive locations in the shape with a directed edge in direction of travel. This way, we can construct G_{PTN} for a given PTN. In a PTN some shapes can have the same edge. We want to avoid duplicate edges in G_{PTN}. Therefore, we annotate each edge with the names of the shapes they are contained in. A weighted graph assigns a weight to every edge with weight function $w : E \to \mathbb{R}$. A path from vertex v_i to v_j in a graph is a finite sequence of edges $path(v_i \to v_j) = (e_1, \ldots, e_n), e_i \in E$ with e_1 starting from v_i and e_n ending at v_j. Shortest path algorithms find a path that minimizes the sum of edge weights. In our Python implementation we use the NetworkX[1] library for handling graphs.

[1] `https://networkx.org/`

3.3 GTFS Data Set

The General Transit Feed Specification (GTFS) [11] is an open standard released by Google in 2006. It is used for the distribution of geographic data and schedules of PTVs. It is the de facto standard that public transport agencies around the wold use to publish data D_{PTN} about their PTN.

A GTFS data set is a ZIP file with information stored in multiple comma-separated values (CSV) files. Currently there are 22 files with different content specified in the standard. At least six of these files are required for a minimal data set. For our application not all files in the GTFS standard are used. In the following, we briefly describe necessary files and the relevant attributes of their content.

- **routes.txt**
  Defines the transit routes. A route is a collection of trips, that display the same information to the user.
  Attributes: `route_id`, `route_short_name`, `route_color`, `route_text_color` and `route_type`.

- **trips.txt**
  Describes the trips for each route. A trip is defined as a sequence of at least two stops, that is active at specific time.
  Attributes: `trip_id`, `route_id`, `service_id` and `shape_id`.

- **stops.txt**
  The individual stops trips stop at.
  Attributes: `stop_id`, `stop_name`, `stop_lat` and `stop_lon`

- **stop_times.txt**
  Arrival and departure time of trips at every stop is stored. A time can be greater than 24 hours. This is needed for trips that start on a particular day, but finish after midnight on the next day. If this happens, we refer to this as *overtime*.
  Attributes: `trip_id`, `stop_id`, `stop_sequence`, `arrival_time` and `departure_time`

- **calendar.txt**
  The weekdays a service is active on in a weekly schedule.
  Attributes: `service_id`, `start_date`, `end_date`, `monday`, `tuesday`, `wednesday`, `thursday`, `friday`, `saturday` and `sunday`

- **calendar_dates.txt**
  Contains any exceptions to the weekly schedule. These exceptions dates can be added or removed services for a date.
  Attributes: `service_id`, `date` and `exceptions_type`

- **shapes.txt**
  A shape describes the course of the road a PTV takes. A shape consists of a sequence of locations and only has one direction of travel. Multiple trips may follow the same shape.
  Attributes: `shape_id`, `shape_pt_lat`, `shape_pt_lon` and `shape_pt_sequence`.

In the following we first assume, that this data is all available in D_{PTN}. In section 4.9 we describe how we store all necessary data in detail.

3.3.1 Missing Shapes

The **shapes.txt** contains the locations for building G_{PTN}. According to the GTFS standard, the **shapes.txt** file is optional. During our work, most data sets we have used, did not include the **shapes.txt**. In his thesis [10], G. Freiwald examined GTFS data sets for different PTN in Germany more closely in this regard.

As described in the related work section, in [3], the authors provide an approach to generate the shape given the stops and underlying road network of a PTN. An implementation *pfaedle*[2] is provided as an open-source project. With GTFS files and OpenStreetMap data, *pfaedle* generates a fitting **shapes.txt**.

3.3.2 GTFS Realtime Extenstion

The GTFS data set described above is a static schedule data. Even with some PTNs releasing weekly updated GTFS data sets, we do not get accurate time information for every trip. This is because PTVs can vary from the schedule caused by external factors. Our map-matching is highly dependent on accurate time information. Therefore, in addition to the static GTFS data set we can incorporate real-time data.

Public transport agencies publish real-time data with different standards. GTFS has an extension GTFS Realtime specified in the GTFS standard. In Germany some transport agencies (e.g., Hamburg/HVV, Berlin/VBB) use "HaCon Fahrplan-Auskunft-System" (HAFAS)[3]. The structure of HAFAS is different to GTFS. We found a project[4] that enables polling a HAFAS endpoint and then publishing a GTFS Realtime feed by matching the HAFAS information to the GTFS data set. Since GTFS Realtime is an open standard that is compatible with the static GTFS data set, we are going to use GTFS Realtime for fetching real-time updates. In the following, we are going to cover relevant parts of the GTFS Realtime standard for our approach.

[2] `https://github.com/ad-freiburg/pfaedle`
[3] `https://www.hacon.de/en/portfolio/information-ticketing`
[4] `https://github.com/derhuerst/hafas-gtfs-rt-feed`

The GTFS Realtime data is provided as a feed in a format that is defined with protocol buffers[5]. Protocol buffers is a standard provided by Google, that is used for serializing data independent of programming languages or platforms. The protocol buffer for GTFS Realtime data is defined here[6]. Bindings are provided for many different programming languages. For fetching a GTFS Realtime in Python, a module is provided here[7]. To access the data more easily, we found a project[8] that converts the feed into a Python dictionary. This way we can easily access the real-time information. Thus, we introduce the GTFS Realtime data as a dictionary.

First the converted dictionary represents a `FeedMessage`. A `FeedMessage` requires a `header` and can have a list of `entity`s.

The `header` contains the GTFS Realtime version. Currently there are version 1 and version 2. The main difference between version 1 and 2 is that in version 2 some fields have been converted from optional to required.

Each `entity` contains a unique `id` and a `trip_update`.

The `trip_update` contains real-time information of a specific trip. It has an entry `trip` and a list of `stop_time_update`s. The `trip` should contain enough information to distinctly identify a trip. In most cases, a `trip_id` is sufficient. Optionally, the `start_date` of the trip can be specified. This is necessary in cases where trips have enough delay to collide with a scheduled trip on the next day. For instance, if a trip has service on two consecutive days, but on the first day the trip has 24 hours of delay, then on the second day there are two trips at the same time. We can unambiguously distinguish between them with the `start_date`.

A `trip_update` also contains a list of `stop_time_update`s. A `stop_time_update` identifies a stop of a trip by either a `stop_sequence` or a `stop_id`. The delay of a trip is split into `arrival` and `departure`. They do not have to be specified at the same time. The delay can either be specified as a delay in seconds or as an absolute time.

In order keep an update message as small as possible, if a stop of a trip has a delay, then this delay applies to all following stops on the trip until the next stop that has a delay. For this next stop, the delay before will not be added. Figure 2 contains an example `FeedMessage`.

Some transport agencies publicly provide their real-time feed. With a link to the feed, we can fetch the real-time data with a HTTP GET request. Other transport agencies limit access and require authentication. For Switzerland we can create an account on their website (full explanation with access to API described here[9]). To access their

[5] `https://developers.google.com/protocol-buffers`
[6] `https://gtfs.org/realtime/proto/`
[7] `https://github.com/MobilityData/gtfs-realtime-bindings/blob/master/python/README.md`
[8] `https://github.com/kaporzhu/protobuf-to-dict`
[9] `https://opentransportdata.swiss/de/cookbook/gtfs-rt/`

```
{
  "header": {
    "gtfs_realtime_version": "1.0"
  },
  "entity": [
    {
      "id": "1.T0.10-46-I-j22-1.2.H",
      "trip_update": {
        "trip": {
          "trip_id": "1.T0.10-46-I-j22-1.2.H",
          "start_date": "20220816"
        },
        "stop_time_update": [
          {
            "stop_sequence": 1,
            "arrival": {
              "delay": 0
            },
            "departure": {
              "delay": 5
            },
            "stop_id": "de:08311:6508:3:14"
          },
          {
            "stop_sequence": 12,
            "arrival": {},
            "departure": {
              "delay": 10
            },
            "stop_id": "de:08311:30065:0:1"
          }
        ]
      }
    }
  ]
}
```

Figure 2: Example GTFS Realtime update for a single trip as a Python dictionary.

real-time feed, we need to pass a generated API-key in the header of the HTTP GET request. The server will respond with the real-time feed data only if the request is correct. Other transport agencies may have different requirements. Therefore, we currently cannot support fetching an arbitrary GTFS real-time feed. A function that fetches the feed must be implemented first if it differs from the two options described above.

Some transport agencies limit the amount of allowed requests within a certain time frame. In Switzerland it is twice every minute. Therefore, the update rate of the real-time data can be limited in a configuration file.

3.4 Hidden Markov Model

A Markov chain describes a sequence of possible events. Each event is modeled as a state and only depends only on the previous state. The probability of changing from one state to another state is the transition probability. If we assume that a process can be modeled by a Markov chain, but we do not know the states, we use a Hidden Markov Model (HMM). A HMM has observable events and hidden states. We are only able to perceive observable events and try to find the states that caused the observable event to happen. For every observable event we create hidden states. We give them a probability of generating the observable event. This is the emission probability. The transition probability measures the likelihood of transition from a hidden state to another hidden state. With the HMM, we can determine the most likely sequence of hidden states for the solution to our problem.

3.5 GPS Location and Great-Circle Distance

The location of a mobile phone is provided by the location services of the operating system. Most commonly the Global Positioning System (GPS) is used. GPS uses satellites around the globe to determine the location of a mobile phone. The satellites constantly broadcast a signal with their time and position. A mobile phone is able to accurately calculate the distance to satellites with the current time and the speed of light. Using signals from multiple satellites a mobile device can accurately determine latitude, longitude and altitude with trilateration. Typically, locations measured with GPS is accurate to about a few meters. In addition to the measurement noise, the signal of GPS satellites can be blocked or reflected from the environment of a mobile device. This can greatly reduce the accuracy of location measurements using GPS. For indoor environments, modern mobile phones are able to include further sources, such as Wi-Fi based positioning systems.

In the following, the tuple of latitude and longitude is referred to as location tuple $p = (lat, lon)$. With the location tuple we can describe any point on earth's surface.

If we want to measure the shortest distance between two location tuples, we cannot simply take the distance as in a Cartesian coordinate system. We need to account for the curvature of earth's surface. Instead, we can approximate the shortest distance which is called the great-circle distance with the haversine formula. The haversine formula assumes that the earth is a perfect sphere. Since earth is not a perfect sphere, we can expect errors for distances smaller than a few meters. For more accurate distance calculation, the computationally heavier vincenty formula can be used. For our application the precision of the haversine formula will suffice and we prefer the faster computation with the haversine formula. In the following, we denote the great-circle distance between two location tuples p_1 and p_2 as $||p_1, p_2||_{\text{gcd}}$.

4 Approach

4.1 Problem Definition

Our goal is to match a sequence of location time tuples collected from a mobile device to a PTV in D_{PTN}. A location time tuple pt contains the location as latitude, longitude, and a timestamp at which the measurement was taken $pt = (\text{lat}, \text{lon}, \text{time})$. Let g be a list of n location time tuples ordered by time $g = [pt_1, , \ldots, pt_n], \text{time}_1 < \ldots < \text{time}_n$. Every trip in D_{PTN} is identified by a unique `trip_id` and contained in the list $T = [\texttt{trip\_id}_1, \ldots]$. Let $P(\texttt{trip\_id}|g)$ be the probability that the trip with `trip_id` fits to g. We can formulate our dynamic map matching problem as the following maximization problem:

$$\texttt{trip\_id}^* = \underset{\texttt{trip\_id} \in T}{\arg\max}\ P(\texttt{trip\_id}|g)$$

After matching $\texttt{trip\_id}^*$ we can look up any additional information about the trip in D_{PTN}. To get an estimate for the current location of the user, we project the location of the last location tuple in g onto the shape of the trip.

4.2 Dynamic Map-Matching Idea

First, we briefly present the main steps of our approach. Similar to map-matching in [2], we use a HMM for calculating $P(\texttt{trip\_id}|g)$. In order to reduce the number of edges in the HMM we want to discard as many edges as possible. This is equivalent to assigning the probability 0. Thus, we can design the following algorithm.

Given a list of location time tuples g and the edges E in G_{PTN}:

1. For every $pt_i \in g$ find the following two sets:
   - $close(pt_i)$, find all edges $e \in E$ that are close to pt_i.
   - $active\_close(pt_i)$, for every close edge $e \in close(pt_i)$ check if there are any vehicles currently active according to the schedule at the time of the timestamp.

2. Build a HMM with all edges in $active\_close(pt_i)$, $pt_i \in g$. Then, calculate the sequence of edges that has the highest probability.

3. Find the trip that fits best to sequence of edges. Fetch and provide any relevant information about this trip.

With a general overview on the algorithm, we will go more into detail of how to solve every step.

4.3 Close Edges

For any location time tuple pt we need to find all edges in E that are within a distance d in meters to pt. We use a value of $d = 100$m. Since we do not need time information for this step $p = (\text{lat}, \text{lon})$ is the latitude and longitude from pt. All close edges are contained in the set $close(pt) := \{e \in E | \ ||e, p||_{\text{gcd}} <= d\}$. Figure 3 visualizes getting the close edges.

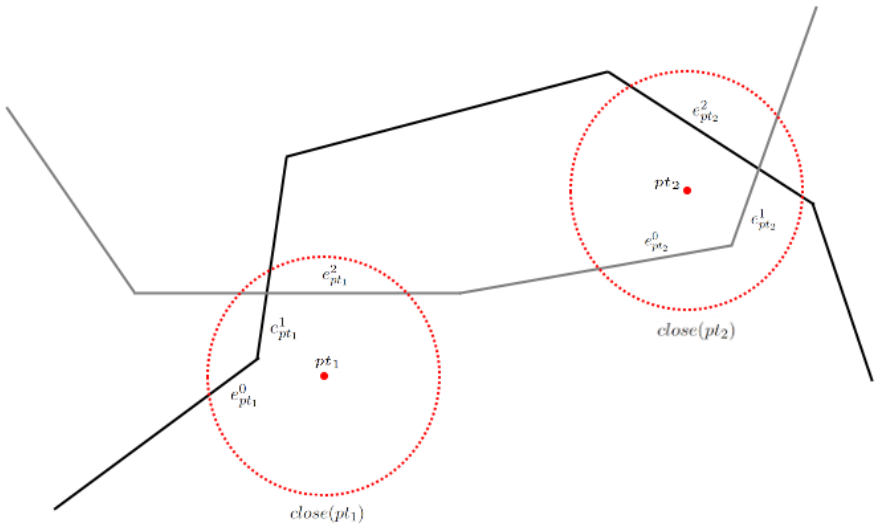


Figure 3: Illustration of *close(pt)*. All edges within distance d to a pt are fetched.

We have already build G_{PTN} that contains all edges from the shapes file. Unfortunately, a graph does not support spatial queries. Therefore, we cannot easily find any edges that are within a distance d to p, without checking the distance of every edge to p in the G_{PTN}.

In 1984, A. Guttman proposed R-Trees [12] as a data structure for efficient spatial queries. In R-Trees, the leaf nodes contain a list of tuples. The tuple contains the smallest rectangle that contains a stored geometric object and a reference to the stored geometric object. In our case, the geometric objects are the edges from E. The non-leaf nodes have a list of tuples. Each tuple contains a reference to a child node and the smallest rectangle that contains all the rectangles of the child node. The R-Tree can be used to find any edge, that intersects the point p. For this we need to traverse the R-Tree starting from the root. At every node we need to check all the rectangles stored in the node. If a rectangle contains p, then we need to check the child node as well. If a rectangle does not intersect p we can skip it. Once we arrive at a leaf node, we can retrieve the edge with the reference and check if the edge intersects with the point p.

Traversing the tree is on average proportional to the depth of the tree. Therefore, a lookup has an average time complexity of $\mathcal{O}(\log n)$ with n edges stored in the R-Tree. In the worst case we can expect $\mathcal{O}(n)$, because we might need to check every entry in the list at a node. Thus, if the tree is not split optimally, we might need to check every edge for intersection.

For our project we use the Python library Shapely[1] for handling any geometric objects, such as edges or points. Conveniently, shapely provides an interface for the STR packed R-tree (STRtree) from the GEOS library[2]. We can insert all edges $e \in E$ into an STRtree.

In our approach we want to find any edges that are within distance d to p. Instead of querying the STRtree with point p we can create a circle. This circle $C_{p,d}$ has its center in p and a diameter of d. The STRtree checks for any edges, that intersect the area of the circle $C_{p,d}$. Therefore, we get all edges within distance d to p.

4.4 Active Edges

After getting $close(pt)$ for a location time tuple pt, we need to determine which of the edges have vehicles on them at the timestamp t of the location time tuple. Only edges, that have a vehicle on them are considered active and are added to $active\_close(pt)$.

[1] `https://shapely.readthedocs.io/`
[2] `https://libgeos.org/doxygen/classgeos_1_1index_1_1strtree_1_1STRtree.html`

In G_{PTN} every edge is annotated with the shapes $shp_e = [s_1, s_2, ...]$ they belong to. For any given edge e, we only need to check trips that follow a shape in shp_e.

The time data for any trip is split into two categories in the GTFS dataset. The service contains the weekdays on which a trip is active. Moreover, it contains any dates with an exception. Secondly, there are arrival and departure times for every stop in a trip.

To check if a trip is active on a date, we can extract the date from t. Then, we need to check if the date is an exception date in the service. Exception dates can either be removed or additional dates. In case of a removed date there is no service. We discard the trip in this case. In case of an additional date there is a service on this date. We can skip the weekday check of the service. Otherwise, we continue with the weekday check.

Since the service contains the weekdays on which a trip is active, we only need to get the weekday of t and check if it is in the active weekdays.

Unfortunately, this approach does not work due to *overtime* trips. For instance, a trip only has service on Mondays and is running from 23:35:00 until 25:30:00 (translates to Tuesday 01:30:00). In the weekdays of the service only Monday is an active weekday. If the user is at a time of 00:30:00 on Tuesday, simply checking the weekday fails. Similarly, the exceptions dates check can fail, e.g., if a Monday is in the removed dates.

Instead, we introduce `active-weekday-hours` (`awh`). `Awh` is a set that contains `weekday-hour-tuples` $\texttt{wh} = (\mathit{weekday}, \mathit{hour}, \mathit{overtime}), \texttt{awh} = \{\texttt{wh}_1, \ldots, \texttt{wh}_n\}$. The first entry is an integer that represents a weekday (Monday $\rightarrow 0$, Tuesday $\rightarrow 1, \ldots$). The second entry is the hour information of a time. Hence, each weekday is split into 24 segments. The third entry is a flag that shows if the time is in overtime ($\geq$24:00:00). The sample trip that runs on Mondays from 23:25:00 until 25:30:00 has $\texttt{awh} = \{(0, 23, \texttt{False}), (1, 0, \texttt{True}), (1, 1, \texttt{True})\}$. We convert the user time t into both possible `weekday-hour-tuples` and check if any of them is in `awh` of a trip. If the `weekday-hour-tuple` without overtime is in `awh`, then the date of t can be used for the exception dates check. If the `weekday-hour-tuple` with overtime is in `awh`, then the date of the previous day of t can be used for the exception dates check. If none of the `weekday-hour-tuple`s are contained, the trip is not active. For the example above the user time on Tuesday 00:30:00 is converted into $(1, 0, \texttt{False})$ and $(1, 0, \texttt{True})$. Here $(1, 0, \texttt{True})$ is in the `awh` set of the sample trip. We need to check the exception dates with the date of the day prior to the user time.

With this data structure, we reduce the amount of trips that need to be checked. Instead of checking trips that are active on a whole weekday, we only need to check trips that are active in the given hour. After that, we need to determine if the edge is active for any trip with a more exact time.

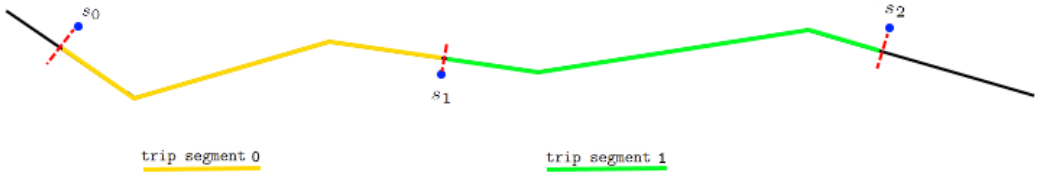


Figure 4: Illustration of trip segments. A trip segment is between two stops and begins/ends at a stop position projected onto the shape.

The time information is available for every stops of a trip as a tuple of arrival and departure time. The location of a stop does not have to be exactly on an edge or a vertex in G_{PTN}. Thus, we need to project the stop onto the closest edge. A stop can be projected onto an edge and there can be multiple edges between stops. Thus, it can be difficult to determine the time of an edge. We can solve this with `trip segments`, which are the segments between two consecutive stops of a trip. Figure 4 visualizes `trip segments`. Every trip has $len(\text{stops}) - 1$ `trip segments`. We identify each `trip segment` with an id ts. We create a dictionary `edge_to_ts` $= \{e : [ts_1, ts_2, \ldots]\}$ that maps every edge to `trip segments` that contain this edge. Due to the fact, that the stops do not have to be exactly on the shape of trip projecting a stop onto the shape, the projected point can be on an edge. Moreover, multiple stops can be projected onto the same edge. Thus, an edge can be in multiple `trip segments`.

In order to create `edge_to_ts`, we have to project the stops onto the edges of a shape. Projecting every stop to the closest edge does not work on all shapes. For instance some shapes have loops and some shapes traverse a street more than once. In these cases the wrong part of the shape can be projected onto. Instead of matching the stop to the closest edge, we also include information about how far a trip has progressed along the shape. If a street is used twice, during the first pass, we match to the edge that is closest to the beginning of the shape.

With the dictionary `edge_to_ts` we can easily find the trip segment numbers ts for a given edge. The arrival and departure times of stops in a trip is stored in a list `stop_times` $= [(arr_1, dep_1), \ldots, (arr_m, dep_m)]$. The `start_time` of a `trip segment` is the departure time of the ts^{th} entry in `stop_times`. Similarly, the `end_time` of a `trip segment` is the arrival time of $(ts + 1)^{\text{th}}$ entry in `stop_times`. Then, we check if the user timestamp t is within the start and end time of the trip segment `start_time` $-$ *earliness* $<=$ t $<=$ `end_time` $+$ *delay*. We can add some slack to the time with *earliness* and *delay*. This is important if the the trip has any delay. Without slack the trip would not be able to match properly.

To sum up, for every trip we store `stop_times`, the set `awh` and the dictionary `edges_to_ts` that maps edges to `trip segments`. To add an edge e to $active\_close(pt)$, we first check the service with the `weekday-hour-tuples` and then extra dates. Then, we find the `trip segments` e is on and check if there is vehicle at the user time. Only

then an edge e is active. All other non active edges are discarded for the HMM step.

4.5 Hidden Markov Model

With $active\_close(pt)$ for all $pt \in g$, we want to determine the sequence of edges with the highest probability of fitting to the sequence of location time tuples. To find the highest probability, we can use a HMM. If every edge of the close active edges is a state node, we can model the HMM as a directed acyclic graph G_{HMM}:

1. Add a start node to the vertices $V = \{\text{start}\}$. The edges are empty $E = \emptyset$. In each step, we remember the nodes created in the last step $L = \{\text{start}\}$.

2. For every $pt \in g$, add a state node for every edge in active close edges to the vertices $V \leftarrow V \cup active\_close(pt)$. For the edges, we connect newly created state nodes to the state nodes created in the last step $E \leftarrow E \cup \{(u, v) | u \in L, v \in active\_close(pt)\}$. For the next iteration set $L \leftarrow active\_close(pt)$.

3. Add an end node to the vertices $V \leftarrow V \cup \{\text{end}\}$. Connect edges to the previous active close edges $E \leftarrow E \cup \{(u, \text{end}) | u \in L\}$.

Figure 5 shows a Hidden Markov Build from the close edges in Figure 3.

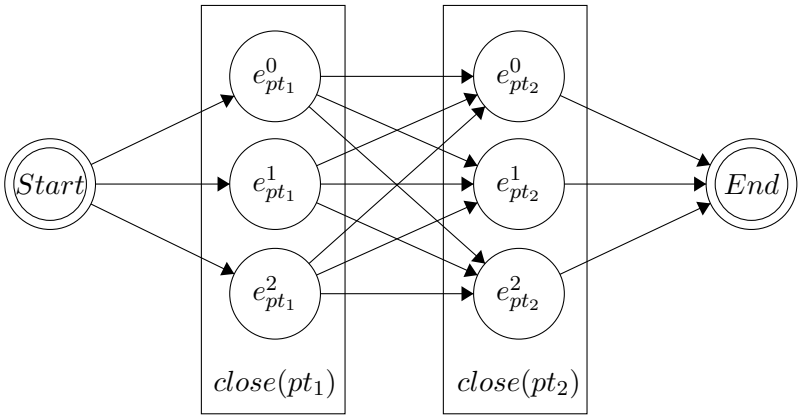


Figure 5: Example Hidden Markov Model. The HMM has two observed events pt_1 and pt_2. The hidden states use $close(pt_1)$ and $close(pt_2)$ from Figure 3.

Typically, the Viterbi algorithm [5] is used to determine the sequence of states in the HMM with the highest probability. Alternatively, we convert finding the path with highest probability into a shortest path problem as described in [8]. In order to find

the most probable path $path_{\text{match}}$, we must find a path in G_{HMM} from the start to the end node with the highest probability. The probability of path is the product of the probabilities of all edges on that path.

Shortest path algorithms find the path through a graph from a given start to an end node by minimizing the sum of weights on the edges in a path. In G_{HMM}, the weights are the probabilities. Therefore, we convert the transition probabilities to transition cost and perform all calculations in log-space. In this case, minimizing the sum of transition cost is the same as maximizing the product of transition probabilities.

We use Dijkstra's algorithm [7] to solve the shortest path problem. Dijkstra's algorithm is not guaranteed to be optimal in a graph with negative weights. Therefore, the transition cost should never be less than 0. We can speed up the calculation by using bidirectional Dijkstra.

4.5.1 HMM Probabilities

In a HMM, the probabilities are typically split into emission and transition probability. The emission probability is dependent on the current state and the observed event. The transition probability is only dependent on the previous state.

As mentioned before, we convert the emission and transition probability in to a emission and transition cost.

Emission Cost

The emission probability $P_{\text{emission}}(e|pt)$ is the likelihood that edge e is part of the shape of the public transit vehicle. We take the distance of the location tuple to the edge as a cost. Hence, edges that are closer to pt are considered more likely.

$$C_{\text{emission}}(e|pt) = ||e, pt||_{\text{gcd}}$$

Transition Cost

The transition probability $P_{\text{transition}}(e_1 \to e_2)$ is the likelihood that starting on edge e_1 a PTV transitions to e_2.

First, we want to measure whether two edge are on the same shape. We assume that it is unlikely that users changes the PTV during transit. Moreover, we want the matched edges to follow the overall direction of the locations tuples as best as possible. This can be done by including the distance in G_{PTN} from e_1 to e_2. We can try find a shortest path from the end of e^1 to the start of e^2 in G_{PTN}. If a path exists,

we can add the sum of the length of the edges to the transition cost. Otherwise, we can just add a penalty cost. In this way we can avoid matching to any detours.

$$C^1_{\text{transition}}(e_1 \to e_2) = ||e_1||_{\text{gcd}} + ||e_2||_{\text{gcd}} + ||shortest\_path((end(e_1), start(e_2))||_{\text{gcd}}$$

With this simple transition cost, it is possible to match a shape that follows the opposite direction. Due to the inaccuracy of location measurements, the shape of the opposite direction can be closer that the actual direction. Thus, we want to add a cost for travelling into the wrong direction.

$$C^2_{\text{transition}} = C^1_{\text{transition}} + direction\_penalty(e_1, e_2)$$

The direction can be calculated if we store the shape with a `shape_sequence` number for every edge. If we have e_1 and e_2, we can compare the `shape_sequence` and if it is ascending, we add no cost. In contrast, we can penalize a descending `shape_sequence`.

4.5.2 Matching to a Specific Trip

In the previous section, our matching only matched g to the edges of G_{PTN}. We still need to find the most likely trip $\texttt{trip\_id}^*$. Each edge in $path_{\text{match}}$ can be contained in multiple shapes, since we merged duplicated edges in different shapes. We can count the occurrence of every shape in the edges of $path_{\text{match}}$ and take the most frequent one as the matched shape. Unfortunately, different trips can have the same shape. In our application, the goal is to match to a specific and single PTV, or to be more specific, a single trip. Especially in crowded areas, such as city centers, there can be more than one trip active on a given edge. We need a way to choose a trip out of the possible ones for our matched path. We can use the following steps to determine $\texttt{trip\_id}^*$ for our matched path:

1. We get the most common shape in the matched edges in $path_{\text{match}}$.

2. Since there is a list of trips for every shape, we can count the occurrences for every trip that belongs to a most common shape and select the most common trip. If multiple trips have the same occurrence, we randomly pick one. This is $\texttt{trip\_id}^*$.

If multiple trips have the same occurrence, and thus same probability $P(\texttt{trip\_id}|g)$, randomly choosing one can yield the wrong trip.

4.5.3 Adding Time Data

We used a very broad time window for filtering close edges by the active time. This can be improved by calculating a more exact time for a PTV location. When matching

to a specific trip we can also try to match a trip that fits best in terms of time. We calculate the average time difference from the time in the location time tuples to a time a trip would presumably have at that location. Figure 6 shows the time difference for a single location time tuple pt to two possible trips. We pick the trip with the least amount of average time difference. Therefore, we want to solve the following problem efficiently: given a location time tuple and a trip, calculate a time at which the trip is at that given location according to the schedule.

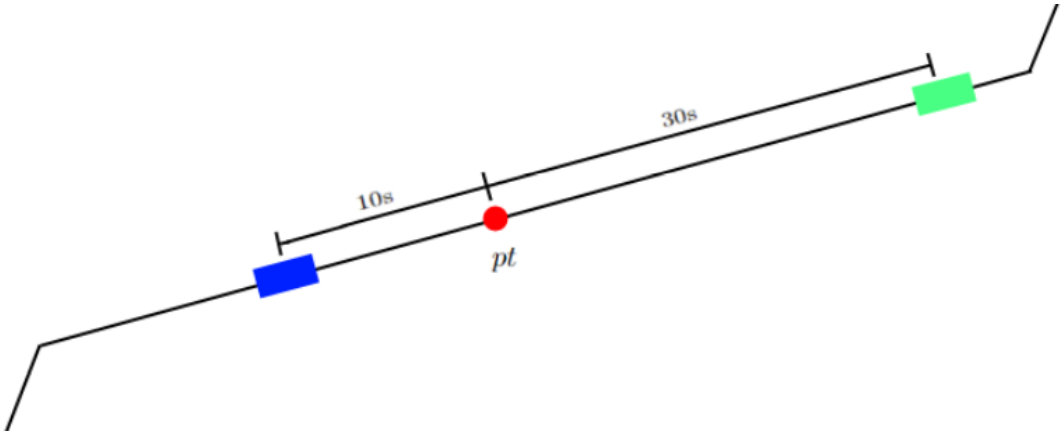


Figure 6: Visualization of minimal time difference for an edge. The blue and the green trip are both active on the same edge. We measure the time difference from pt to the blue and the green trip according to the schedule. Here, we would prefer the blue trip.

In order to calculate such information, we need to modify our data. To calculate which trip is closest on average to g, we measure the average time difference between the time from a location time tuple to the time the vehicle of a trip is at that location according to the schedule. In order to calculate such a time difference, we first need to calculate a time t for any given point p on the shape of a trip. The only time information we have is included in the stops of the trip. Therefore, we split the shape of a trip by the stops into `trip segments`. In contrast to the `trip segments` in section 4.4, we split the polyline of the shape into a list of `trip segments`. Each `trip segment` is between two consecutive stops of a trip. We split the polyline of the shape by first projecting the location of the stop onto the polyline, and then split at the projected point. Now onto calculating a time for the vehicle at point p on the shape:

1. Find the `trip segment` p is on and project p onto the `trip segments` which gives us the closest point ts_p on the `trip segment` to p.

2. Calculate the distance $d_p = ||ts_{start}, ts_p||_{\text{gcd}}$ along the `trip segment` from p to the start of the `trip segment`.

3. Take $tp = \frac{d_p}{||ts||_{\text{gcd}}}$ as the progression along the `trip segment`. Calculate a predicted time as the percentage of the total duration of the `trip segment` and add it to the start time of `trip segment`:

$$time(p) = time(start) + (time(end) - time(start)) \cdot tp$$

This approach assumes that a vehicle travels at a constant speed at a given trip segment. In reality this is unlikely. We expect a vehicle to accelerate or decelerate when starting or stopping at a stop. Moreover, we expect that vehicles to have a higher speed on straight lines, than in curves. Additionally, vehicles might need to stop during the journey at traffic lights or due to traffic.

With a time estimation of the point p we can measure the time difference of the location time tuple pt to $time(p)$ for every location time tuple in g. We can compare the average time difference for multiple trips. Then, we just select the trip with the smallest average time difference.

We can add the time difference during matching to a specific trip. If there are multiple possible most likely trips, instead of randomly choosing one, choose the trip with the least average variation from the schedule.

4.5.4 Including GTFS Realtime in Map-Matching

In our dynamic map-matching approach, we try to find trips that are as close as possible from an ideal schedule. If a user is in a PTV with delay, the user can be matched to another trip. We can mitigate this by adding real-time updates with GTFS Realtime. As described in section 3.3.2, we can get the delay of a trip at any stop. We need to apply this delay when working with the stop times of a trip and with `ahw`.

With `awh`, a delayed trip can be active in another `weekday-hour-tuple`. If this `weekday-hour-tuple` is not within the `awh` of a trip, this trip will not be considered active. To mitigate this, we can apply the negative delay as an offset to the user time and then generate a `weekday-hour-tuple`. For example, if a trip has one hour of delay, then we subtract one hour from the user `weekday-hour-tuple`. This way, the delayed trip is considered active.

The arrival and departure time of a stop also have to be adapted when using real-time data. Here, we must add the delay to any time during look-up.

4.6 Getting All Information About the Matched Trip

After matching a trip, our *frontend* should display further relevant information to the user. Therefore, we need to fetch any relevant information.

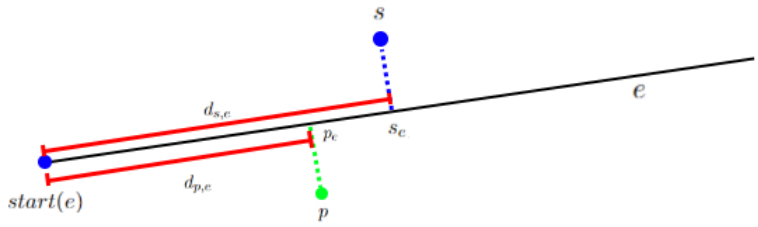
In the map-matching step we determined a most probable path $path_{\text{match}}$ of edges and `trip_id`*. We can predict the current location of a user by projecting the last location tuple in g onto the last edge in $path_{\text{match}}$. We get a predicted user location, that is on the shape of the matched trip `trip_id`*.

Further relevant data is included in D_{PTN}. Every `trip_id` has a `route_id` which contains the `route_short_name`, `route_color`, `route_text_color` and `route_type`. Additionally, every `trip_id` also has a list of stops. We can look up the destination of the trip by taking the name of the last stop in the list. If we take the last edge of the matched path, we can determine on which `trip segment` the user is currently on with `edge_to_ts` of the trip. The `trip segment` with id ts corresponds to the segment of the ts^{th} stop until the $ts + 1^{\text{th}}$ stop. If an edge only has a single trip segment id ts, the next stop is the $ts + 1^{\text{th}}$ stop in the list of stops. If an edge has multiple trip segment ids, then we need to determine, which of the stops is the next. We know that the stop must be on the last matched edge e. We can take p, the last location tuple of g, and project it onto the edge p_e. Then, we project the stop s onto the edge s_e. We measure both $d_{p,e} = ||p_e, start(e)||_{\text{gcd}}$ and $d_{s,e} = ||s_e, start(e)||_{\text{gcd}}$. If $d_{p,e}$ is greater than $d_{s,e}$ we have already passed the stop and check for the next stop. If not, then the stop s is the next stop. Figure 7 illustrates both cases. With the next stop we can also gather possible transfer options from D_{PTN}.

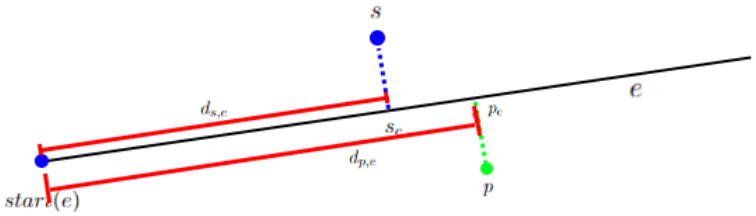
4.7 User Application

A *PublicTransitSnapper* user only sees our *frontend* on their mobile phone. Thus, all necessary information is displayed in our *frontend*. The *frontend* is written with the Flutter framework. The advantage of using this framework is, that we can build different mobile and web apps from a single code base. Currently, we only support Android 10+ with our native application. Additionally, we offer a web app that can be used through a browser. With Flutter it is possible to generate an iOS build. Due to the lack of an iOS device, we were not able to try it out.

The *frontend* measures the location of a mobile phone by accessing the location services. We store a list of the last locations with a timestamp. The *frontend* sends the list of location time tuples to an API on a server where the *backend* runs. The *backend* processes any incoming request. It performs a dynamic map-matching and calculates a most probable solution. It responds to the *frontend* by sending the matched `trip_id` with additional information about the trip.



(a) $d_{s,e} > d_{p,e}$



(b) $d_{s,e} < d_{p,e}$

Figure 7: Determining the next stop with two trip segments. In (a), s is the next stop. In (b), the stop after s is the next stop.

Our *frontend* does not store and information about the GTFS data set. It only displays information provided by the *backend*. A user can choose between three pages in the application. The first page displays information about the PTV the user has been matched to. Such as the vehicle type, the route number and terminal station. It also shows the name of the upcoming stop. The second page contains a map. An icon that fits to the vehicle type is displayed at the currently matched position. Moreover, the shape and stops of the currently matched trip are displayed in the `route_color` of the route. Thus, the user can easily see the previous and upcoming course of the road. On the third page, we display possible transfer options at the next stop for the user. It shows the type of vehicle, with their route number, destination and the departure time at the next stop. We use the `route_color` and the `route_text_color` for easier identification of the route by the user. Figure 8 shows the three pages.

Next to the three pages, the user can open a chat overlay in any of the three pages. We provide a group chat feature, where all users that are matched onto the same trip are able to communicate with each other. If a user enters a trip, all messages that have been sent since the beginning of the trip can be seen. The chat messages are only stored for the duration of the trip and are deleted after the trip has reached the destination stop.

For more details about the *frontend* application with user feedback see G. Freiwald's thesis about this project [10].

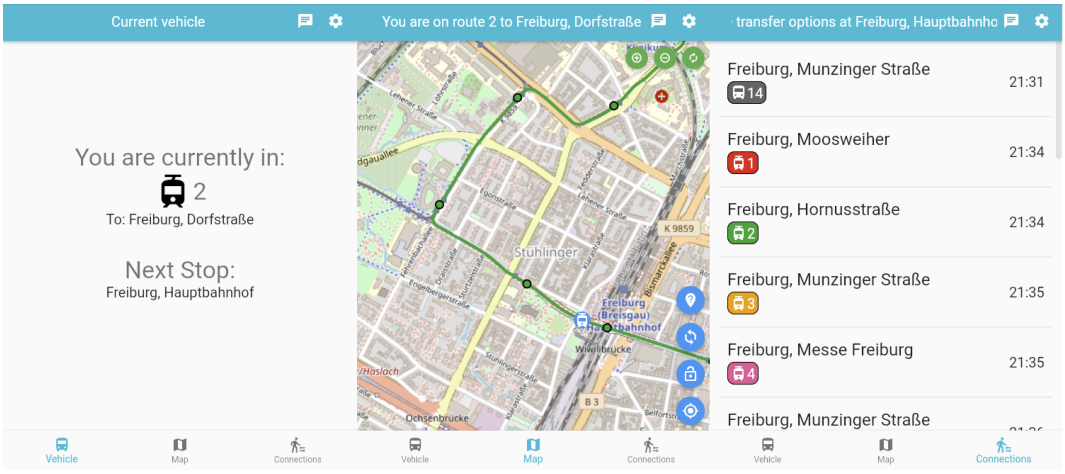


Figure 8: Screenshot of the different pages in the frontend.

4.8 Linking Backend and Frontend

In the previous chapters we have described how to solve the dynamic map-matching in the *backend* part of our project. Moreover, we have described the design and features of the *frontend* application. In this chapter, we describe the communication between *frontend* and *backend*. The *frontend* does not store any schedule data from the GTFS data set. Thus, all necessary information must be fetched from the *backend*.

In our *backend*, we handle any incoming requests with our API. The API is created with the Python package Flask[3]. We have created multiple endpoints for the different features of our *frontend*. The shape and connections contain a lot of data, but they only have to be fetched if the matched shape or the next stop changes. If we only request this information when needed, we can reduce the amount of data that needs to be sent over the network.

- `/map-match` This endpoint takes a list of location time tuples. After the map-matching is calculated information about the matched trip is returned for the *frontend* to display. Note that the shape and connections are not included.

- `/shape` Here the *frontend* can fetch all location tuples in a shape, so that the shape of a matched trip can be drawn on the map page. Moreover, the locations of each stop in the matched trip are returned as well.

- `/connections` Given a `stop_name` and a timestamp return the next transfer possibilities at that stop.

[3] `https://flask.palletsprojects.com/en/2.2.x/`

- `/chat` With a matched trip the *frontend* can use this to send and fetch any chat messages for a trip.

In every endpoint of our API, we need to exchange data between the *frontend* and the *backend*. We use the JavaScript Object Notation (JSON)[4] for interchanging data between the Dart *frontend* and Python *backend*. A sample request and reply for the `/map-match` endpoint can be found in Figure 9. For the other endpoints we use a similar JSON structure.

```
request = {
    "coordinates": [
        "47.984554414717,7.893493406577,1663594021401",
        "47.984396141097,7.894909945709,1663594032515",
        "47.985218134608,7.894933737771,1663594054057"
    ],
    "trip_id": ""
}

response = {
    "location":    (47.9852057074915, 7.8949523500852),
    "next_stop":   "Freiburg, Heinrich-Heine-Str.",
    "route_color": "646363",
    "route_dest":  "Freiburg, Langmatten",
    "route_name":  "18",
    "route_type":  "3",
    "shape_id":    "shp_3_233",
    "trip_id":     "352.T0.10-18-I-j22-1.6.R"
}
```

Figure 9: Example request by the frontend with the response by the backend.

4.9 Storage of Data

In the previous sections we assumed that all data about a specific public transit network is stored in D_{PTN}. We assumed, that we can retrieve any information, that we need from it.

In this chapter we introduce a Python class called *GTFSContainer* which contains all data necessary for the dynamic map-matching and API requests. Our main design goals are a quick lookup of information and a small memory consumption.

[4] `https://www.json.org/json-en.html`

As described before, we already have the graph G_{PTN} that contains all edges. Each edge has a list of tuples with `shape_id` and `shape_sequence` number. Additionally, we need a STRtree with edges from G_{PTN} for quick spatial queries.

For the other lookups we use Python lists, dictionaries and sets. The time complexity of all possible operations is described in here[5]. We only focus on lookup time complexity.

We use a list if we have elements that are ordered and we know the index for look-up. In this case, the average and amortized worst case is $\mathcal{O}(1)$. If we do not know the index we get $\mathcal{O}(n)$, because we need to possibly check all n elements in a list. We use a set, if we want to quickly check for membership, such as the `active-weekday-hours`. In the average case we get $\mathcal{O}(1)$, but in amortized worst case we get $\mathcal{O}(n)$. We use a dictionary if we have keys and want to associate values with the key. In the average case we get $\mathcal{O}(1)$, but in worst case we get $\mathcal{O}(n)$.

For the active edges we need to find edges that have trips active on them. Since edges are only annotated with `shape_id`s, we need to find candidate trips given a `shape_id`. Moreover, we need all location tuples in a shape for `/shape`. We do not have the shapes stored separately, which are already contained in G_{PTN}. Instead, we store the starting edge of every shape. Then, we can use G_{PTN} to find all edges of a given shape. Since G_{PTN} is a directed graph, we start at the starting edge. Then, we repeatedly examine all outgoing edges for the annotated list of `shape_id`s and `shape_sequence` and follow the edge containing the same `shape_id` and the next higher `shape_sequence`. This way, we can find all edges of a given shape, without introducing a new dictionary with look up time of $\mathcal{O}(n)$.

```
{
  shape_id: [
    ((lat, lon), (lat, lon)),
    [(trip_id, serivce_id, route_id), ...]
  ],
  ...
}
```

Given a trip, we need to determine the service information and arrival and departure time to determine if the trip is active at a given user time.

The service information is represented by the `service_id`. Multiple trips can have the same `service_id`. Thus, we use the `service_id` as a reference for the trip and only store the information for a `service_id` once. A service has a set of `removed_dates`, a set of `extra_dates`, a `start_date` and an `end_date`. With the set for `removed_dates` and `extra_dates` we can quickly determine if a given date is an exception. A service also has `active_weekdays`. We do not store this, since we created the `awh`.

[5] `https://wiki.python.org/moin/TimeComplexity`

```
{
  service_id: [
    start_date, end_date, {removed_dates}, {extra_dates}
  ],
  ...
}
```

Since the stop times vary from trip to trip, we store the stop times individually for every trip. This is just a list of the stop times. For the same reason, each trip has its own `awh`. If we know the trip segment id ts, we can quickly look up stop time information in a list with $\mathcal{O}(1)$. The ts^{th} stop is the stop at the start of a `trip segment`. The $ts + 1^{\text{th}}$ stop is the stop at the end of a `trip segment`. We determine the trip segment id for an edge with the dictionary `edge_to_ts`, that maps edges to trip segment ids. With `edge_to_ts` we can quickly determine trip segment ids and thus the start and end times.

The dictionary `edge_to_ts` does not need to be stored for every trip. For trips with the same `shape_id` and stops, the `edge_to_ts` is identical. Since multiple trips can have the same `shape_id` and stops, we can remove the duplicates by storing `edge_to_ts` in another dictionary. The trips then refer to the correct entry in this new dictionary.

For calculating a predicted time for a trip given a point, we need to determine which `trip segment` a point belongs to. Unfortunately, the `edge_to_ts` dictionary cannot provide this information. Therefore, we store a list of trip segment polylines. The trip segment polylines are again only depended on the shape and stops of the trip. Thus, we can store them together with the `edge_to_ts` for reduced storage.

```
{
  trip_id: [
    [[[arrival_time,depature_time], stop_id], ...],
    active_weekday_hours,
    route_id,
    service_id,
    hash_value
  ],
  ...
}
```

```
{
  hash_value: [
    edge_to_ts,
    [trip_segment_polyline, ...]
  ],
  ...
}
```

In the *frontend* application, we want to display information to the current user about the current vehicle. Moreover, we use the colors and type of a vehicle, to make the information more appealing. This information is mainly contained by the route information. Since the active edges also return the `route_id` we need to quickly look up any information related to the route.

```
{
  route_id: [
    route_short_name, route_type,
    route_color, route_text_color
  ],
  ...
}
```

For `/shape`, we also need the location of the stops to display. We can look at the information we saved for a `trip_id`. It contains all `stop_id`s in a trip. Thus, we need a new dictionary, that given a `stop_id` returns location of a stop. We also store the `stop_name`, so that this can be displayed to the user as the next stop.

```
{
  stop_id: [stop_name, stop_lat, stop_lon],
  ...
}
```

In the *frontend* we display possible connections for the next stop. Multiple stops can be grouped together. For example a central station can have a tram, a train and a bus station all with the same name. Thus, given a `stop_name` we need a list of `stop_id`s to determine which stops we need to examine.

```
{
  stop_name: [stop_id, ...],
  ...
}
```

Now with a number of possible `stop_id`s, we still need to find trips that stop at the next stop after a given timestamp. For every `stop_id` we store of list of `trip_id`s with

the `departure_time`. We can check if the `departure_time` is after the timestamp and only then check if the trip is active on the date of the timestamp. This gives us a list of possible transfer options at the next stop for a user.

```
{
  stop_id: [[trip_id, departure_time], ...],
  ...
}
```

Most of the information in *GTFSContainer* is stored similarly to the GTFS data set. We can precalculate most of the data structures described above. For the G_{PTN} and the STRtree we store all edges as a list in JSON format. On startup the server loads all precalculated data as dictionaries and generates the G_{PTN} and the STRtree from the list of edges. We have written the precalculation part in C++. A library for parsing CSV in C++ is available here[6]. All the precalculated data is stored in JSON format with the library from here[7].

[6] `https://github.com/ben-strasser/fast-cpp-csv-parser`
[7] `https://github.com/nlohmann/json`

5 Evaluation

In order to measure the performance of our dynamic map-matching, we need a test data set. Ideally, we want to capture data from the real world. Unfortunately, we are not able to take a high amount of public transit vehicles, in order to capture location time tuples for all possible public transit vehicles from a public transit network. Therefore, we decided to generate own test data based on D_{PTN}. We try to mimic data captured from the real world as closely as possible. Then, with the generated test data, we can evaluate the performance of our dynamic map-matching approach.

5.1 Evaluation Method

5.1.1 Generating Test Data

Our map matching algorithm works with a list of n location time tuples g. In our case we use $n = 10$. We can generate test data $td_{\texttt{trip\_id}}$ which contains location time tuples for a particular trip, that follows the shape of a trip. For any given trip, we need the stops and shape of a trip to generate test data $td_{\texttt{trip\_id}}$. First we generate m test location tuples $g_{\text{test}} = [p_1, \ldots, p_m]$ along the shape. Then, we try to mimic the inaccuracy of location measurements by applying some noise onto every generated test location tuple. We are using normally distributed noise in meters $\mathcal{N} \sim (0, 16)$. In the test data, a user always travels the same number of stops s. s does not include the stop the user gets onto a PTV. We assume that $s = 4$ is a realistic value. For each trip we generate a test instance for each possible four-stop journey. Thus, a trip with a total of 6 stops gets two test instances. One starting at stop 1 and ending at stop 5 and another one starting at stop 2 and ending at stop 6. For any trips that do not have enough stops for a test instance or if the shape is too short ($n > m$), we simply generate a single test instance, with the available stops or locations in g_{test}. We have already generated g_{test} for the whole shape. For a test instance, we randomly select n location tuples that lie between the start and end stops for the given test instance.

Next to a location tuple we need a fitting timestamp to complete a location time tuple for our map-matching algorithm. In order to calculate a fitting timestamp, we use the same approach as in section 4.5.3. For generating test data, this approach has a major drawback. We assumed that vehicles have a constant speed between stops. This is not true for most vehicles. Therefore, we also add noise to the timestamp to

simulate varying speeds of the vehicle. For every stop we assume, that public transit vehicles can have delay, but if they arrive to early, they will wait until the scheduled departure time. We add a normally distributed delay in seconds but map all negative values to 0 $\max(0, \mathcal{N} \sim (0, 60))$. Every timestamp between this stop and the next stop will have this delay added. Additionally, we add a delay to each timestamp in seconds independent of the stop $\mathcal{N} \sim (0, 30)$. This simulates vehicles having varying speed on each route.

The GTFS data sets describe the weekly schedule of PTVs. We test a single day from the schedule, due to high number of trips that need to be tested. Dynamic map-matching to a tight time schedule is more challenging because there are more trips active around the same time. On weekends there are typically less trips than on weekdays. Hence, we chose a Wednesday that is not a public holiday for our testing. We only generate a test data set for trips that are active on this day.

5.1.2 Accuracy Measure

Our main goal is to find the correct public transport vehicle a user is in. Moreover, we want to provide a responsive user experience. Therefore, we can measure the following three aspects in order to evaluate the performance of our dynamic map-matching.

First, we want to match the correct trip_id, given any test $test_{\texttt{trip\_id}} \in td_{\texttt{trip\_id}}$.

For each test we check if the trip matched by our map-matching algorithm is the same as the original trip the test was generated for with an indicator function:

$$\mathbb{1}(test_{\texttt{trip\_id}}) = \begin{cases} 1, & \text{if } \texttt{trip\_id} = match(test_{\texttt{trip\_id}}) \\ 0, & \text{otherwise} \end{cases}$$

Trips can have vastly different lengths of shapes and number of stops. Thus, the number of tests for each trip can vary. To avoid prioritizing longer trips over shorter trips in our accuracy measurement we take the average over all test instance in a test data set for trip. Then we take the average over all test data sets.

$$\text{accuracy} = \frac{1}{|\text{Test Data}|} \sum_{td \in \text{Test Data}} \left(\frac{1}{|td|} \sum_{test \in td} \mathbb{1}(test) \right)$$

With random elements included in each test data, we generate three test data sets and run the evaluation on each test data set. Then we take the average of the resulting average.

Moreover, we measure the memory and storage consumption of our *backend*. This is due to the large amount of precalculated data that need to be stored in memory for a fast map-matching during run-time.

The last measure is the run-time of our map matching algorithm. The run-time in this evaluation only measures the performance of the map-matching algorithm. The performance of the *frontend*, network latency and of our API are not included. The total latency can be higher than the measured value and might be highly dependent on network latency and hardware.

5.1.3 Evaluation Data Sets

We evaluate *PublicTransitSnapper* with different GTFS data sets Freiburg, Zürich and SWEG. The Freiburg GTFS data set is provided by Freiburger Verkehrs AG[1]. For Zürich, we filtered the GTFS data set Switzerland[2] for any trips by the Züricher Verkehrsverbund. For SWEG we filtered the GTFS data set SPNV Baden-Württemberg[3] for any trips by the Südwestdeutsche Landesverkehrs-AG.

In Table 1 we provide more detailed information about each data set. Freiburg contains an almost equal number of buses and trams. Zürich almost exclusively contains buses and contains more trips than Freiburg. The SWEG data set exclusively contains trains.

Dataset	total trips	tram	bus	funicular	train
Freiburg	19,153	9,063	10,090	—	—
SWEG	733	—	—	—	733
Zürich	33,178	—	31,971	1,206	—

Table 1: Route types in each evaluation data set.

5.1.4 Evaluation Algorithms

For the evaluation we use four different algorithms: Baseline, BaselineHMM, ActiveEdges, TimeAfter.

The Baseline algorithm takes the last location tuple from g and gets the closest edge in G_{PTV} with the STR-tree. Then it selects a trip, that fits to the edge. If there are multiple trips, that fit to this edge, randomly take one. This does not include any time information.

[1] `https://www.vag-freiburg.de/fileadmin/gtfs/VAGFR.zip`

[2] `https://opentransportdata.swiss/dataset/00811070-1b51-43da-87af-b1901e906323/resource/d97dedca-4e9e-454b-b7ad-5188b85f075f/download/gtfs_fp2022_2022-06-22_04-15.zip`

[3] `https://www.nvbw.de/fileadmin/user_upload/service/open_data/fahrplandaten_ohne_liniennetz/bwspnv.zip`

The BaselineHMM algorithm gets the close edges for every location tuple in g and performs a map-matching with HMM. With the most probable path calculated, it takes a trip that fits to the path. If there are multiple trips, that fit to this path, randomly take one. This does not include any time information.

The ActiveEdges algorithm is similar to the BaselineHMM algorithm. The only difference is that, after getting the close edges, ActiveEdges also filters for edges that have a trip on them at the given time.

The TimeAfter algorithm is similar to the ActiveEdges algorithm. After calculating the most probable path, we do not randomly select a trip. If there are multiple trips, that fit to this path, we take the trip, that has the least amount of average deviation from the schedule.

5.2 Evaluation Results

For the evaluation results all tests were run on the same test system with an AMD Ryzen 5 5600 6C/12T 3.50 GHz and 32 GB RAM. In order to decrease the total time needed for testing, we ran a backend instance on each of the 6 cores and split the test data equally among them.

In Table 2 we evaluate the performance of the two baseline algorithms Baseline and BaselineHMM. BaselineHMM outperforms Baseline with about double average accuracy. Both algorithms are not able to match a correct trip in most cases, due to many trips using the same shape at different times. The Baseline algorithm has a fast run-time since only the closest edge to the last location tuple is considered. With BaselineHMM, we get a high run-time. Without filtering edges that do not have trips on them at the user time, there are many edges that are inserted into the HMM. Thus, calculating the shortest path in the G_{HMM} takes much longer. With a high number of trips the likelihood of trips being close is higher thus the accuracy suffers. SWEG has a low run-time since the individual shapes of trains do not have a high overlap.

	Average Accuracy		Average Run-Time	
Dataset	Baseline	BaselineHMM	Baseline	BaselineHMM
Freiburg	0.4%	0.8%	0.06s	0.67s
SWEG	0.8%	1.6%	0.002s	0.037s
Zürich	0.2%	0.4%	0.06s	0.70s

Table 2: Average accuracy and run-time of baseline algorithms.

Table 3 shows the ActiveEdges algorithm run on the three data sets. Unfortunately, the average accuracy is at less than 22%. It is higher than the two baseline algorithms, but not enough to match a user to the correct trip in most cases. Interestingly, the added time data seems to benefit data sets with more dense trips.

Dataset	ActiveEdges
Freiburg	15.8%
SWEG	5.3%
Zürich	21.6%

Table 3: Average accuracy for ActiveEdges without time slack.

Examining our approach, we found that the accuracy is low due to no slack in the time checking for the active edges step. In Table 4, we are able to achieve much higher results if we introduce one minute of allowed earliness and five minutes of allowed delay. In the following we always allow this time slack.

	No time noise		Time noise	
Dataset	ActiveEdges	TimeAfter	ActiveEdges	TimeAfter
Freiburg	91.6%	92.4%	91.2%	91.3%
SWEG	34.9%	34.2%	32.5%	32.3%
Zürich	92.4%	95.7%	92.3%	94.8%

Table 4: Average accuracy of our dynamic map-matching algorithms.

In Table 4, we evaluate the average accuracy of our two dynamic map-matching approaches. In Freiburg and Zurich we are able achieve an average accuracy of over 90%. In both cases we notice an increase in average accuracy with the TimeAfter approach compared to ActiveEdges. In Freiburg the increase is much smaller. In the Freiburg data set, there are trams and buses that run on the same streets. Thus, there are multiple shapes very close together with different edges. With the noise added to the location measurements, we cannot easily determine which of the shapes is correct with our approach. Moreover, in the Freiburg GTFS data set we found trips, that that run at the same time for a number of stops. Figure 10 shows two example trips, that have this issue. In this case, we are not able to correctly determine the trip, since both have the same likelihood. We notice a lower average accuracy after adding time noise to the location time tuples. This is expected, since in our approach, we assume that trips perfectly follow the GTFS schedule. If a trip has delay, the probability that g gets matched to a later trip is higher. For the ActiveEdges approach, the accuracy drop is smaller compared to TimeAfter, since we allowed slack. In this case a small

delay has less impact due to still being in the time frame with slack.

```
Trip: 573.T0.11-3-I-j22-1.6.R

Trip active on: ['monday', 'tuesday', 'wednesday', 'thursday', 'friday']

Freiburg, Munzinger Straße       arrival time: 06:22:00  departure time: 06:22:00
Freiburg, VAG-Zentrum            arrival time: 06:23:00  departure time: 06:23:00
Freiburg, Am Lindenwäldle        arrival time: 06:25:00  departure time: 06:25:00
Freiburg, Bugginger Straße       arrival time: 06:26:00  departure time: 06:26:00
Freiburg, Rohrgraben             arrival time: 06:27:00  departure time: 06:27:00
Freiburg, Bissierstraße          arrival time: 06:29:00  departure time: 06:29:00
Freiburg, Runzmattenweg          arrival time: 06:31:00  departure time: 06:31:00
Freiburg, Rathaus im Stühlinger  arrival time: 06:32:00  departure time: 06:32:00
Freiburg, Eschholzstraße         arrival time: 06:34:00  departure time: 06:34:00
Freiburg, Hauptbahnhof           arrival time: 06:35:00  departure time: 06:35:00
                                        ⋮

Trip: 586.T0.11-3-I-j22-1.3.R

Trip active on: ['monday', 'tuesday', 'wednesday', 'thursday', 'friday']

Freiburg, Am Lindenwäldle        arrival time: 06:25:00  departure time: 06:25:00
Freiburg, Bugginger Straße       arrival time: 06:26:00  departure time: 06:26:00
Freiburg, Rohrgraben             arrival time: 06:27:00  departure time: 06:27:00
Freiburg, Bissierstraße          arrival time: 06:29:00  departure time: 06:29:00
Freiburg, Runzmattenweg          arrival time: 06:31:00  departure time: 06:31:00
Freiburg, Rathaus im Stühlinger  arrival time: 06:32:00  departure time: 06:32:00
```

Figure 10: Example for trips that cannot be distinguished. Both trips share a number of stops with the same arrival and departure times on the same weekdays.

Running our dynamic map-matching on the SWEG data set, we notice a much lower average accuracy compared to Freiburg and Zurich. The main difference in the data sets is that SWEG exclusively contains trains. The distance between stops in a train network is higher than for trams or buses. In Figure 11 we reduce the number of stops s a user takes for a test while keeping $n = 10$. With less stops, and thus smaller distances between each sample location tuple, we get a higher accuracy. We conclude that our map-matching approach does not work well with big distances between each sample location tuple. Moreover, using the TimeAfter approach does not seem to have an advantage compared to ActiveEdges.

The memory and storage requirements can be found in Table 5. For the precomputed files, we need about 1.3 times the storage compared to the GTFS size. Since we are able to reduce the amount of look-ups and calculations needed during run-time, we assume that this a reasonable amount of storage being used. For the memory usage, we take the memory usage of the docker container, where our whole `backend` runs. Comparing the precomputed files to the memory usage, for Freiburg and Zurich we

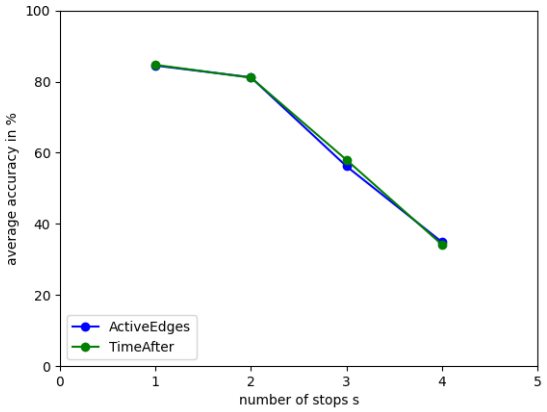


Figure 11: Accuracy in SWEG data set with different number of stops.

need about 11 times of memory. For the SWEG this is much higher. This is because the SWEG data set contains trains with long shapes and thus many edges. We store the edges of a shape in multiple data structures. We have G_{PTN}, a STRtree and also the `trip segments`. This leads to a high amount of duplicates. Hence, our memory usage does not scale well with the shapes.

Dataset	GTFS size	precomute size	precompute time	docker memory usage
Freiburg	28.3 MB	36 MB	1.65 s	384 MiB
SWEG	4.58 MB	5.6 MB	0.27 s	148.1 MiB
Zürich	45.5 MB	58 MB	2.47 s	626.6 MiB

Table 5: Memory and storage usage of our backend.

Table 6 shows the average run-time of our two dynamic map-matching approaches. Similar to the two baseline algorithms, SWEG has a low run-time due to low shape overlap. For Freiburg and Zürich we measure an average run-time of 210-250ms. For an interactive user experience, we aimed for a run-time of 500ms or lower. Therefore, our expectations were met.

	No time noise		Time noise	
Dataset	ActiveEdges	TimeAfter	ActiveEdges	TimeAfter
Freiburg	0.248s	0.230s	0.246s	0.241s
SWEG	0.018s	0.018s	0.018s	0.018s
Zürich	0.217s	0.219s	0.233s	0.238

Table 6: Average run-time of our dynamic map-matching algorithms.

6 Conclusion

PublicTransitSnapper is able to match a user to a public transit vehicle. It extends map-matching to a static road network with the dynamic element of public transit vehicles. We notice that our approach works well with buses and trams but has low accuracy when working with trains. In order for our map-matching to be accurate, we need a high sampling rate with low distances between each location measurement. Additionally, delays of PTVs decrease the performance of our map-matching. When matching to buses and trams, we are able to match to the correct PTV in most cases. Moreover, the application can provide information about the currently matched trip to the user. In the following, we suggest further improvements.

6.1 Future Works

In our approach we have used a fixed balance of parameters in the emission and transition cost functions. We are currently only using distance based emission and transition cost functions. It might be beneficial to add a time based cost to either emission or transition cost. Although, this might be more sensitive to delay. Moreover, we noticed that due to the time deviation based metric for selecting the final trip, delays can cause the map-matching to have a lower accuracy. Adding real-time data can mitigate some of the issue. Further research is needed in order to determine if adding parameters to emission and transition cost can make dynamic map-matching more robust to such delays. Due to different possible types of PTVs and varying size and density of PTNs it can be difficult to find a perfect balance. Therefore, we estimate that this could take half a year to a year.

For selecting the correct trip, we tried to predict the time at which a public transit vehicle is at any given time given any point on the shape. We assumed that vehicles travel at a constant speed between consecutive stops. This is highly unlikely in the real world. Due to factors such as traffic lights or corner public transit vehicles have varying speed. From the shapes, curves and straight can already be extracted. Further parameters such as traffic lights need additional sources. Adding of such information, e.g. from OpenStreetMap data might be possible. With a better estimate of positions, we expect an increase in accuracy. Solving this needs a lot of testing in the real world, thus we expect this to take up to half a year to complete.

Our map-matching approach requires much information stored in the memory. This can be become a limiting factor for increasing sizes of GTFS data sets. More efficient

data structures may be used to store the data to further reduce memory usage. Currently, we have duplicates when storing all edges from all shapes. Further research is needed to reduce the memory consumption of our stored look up data. Much restructuring of our stored data is needed, therefore we expect this to take at least half a year.

Moreover, run-time is a crucial for a smooth user experience with the *frontend*. We expect significant performance gains when switching the *backend* from Python to a faster programming language such as C++. As a first step the active edges and HMM part that take the most computation time could be build as a C++module for python. We estimate, that rewriting part of our approach C++takes a few weeks. Since our precalculation is already done in C++some the code base can be transferred.

Further improvements can also be made to the *frontend* of our project. We included real-time data in the *backend*, but no information is displayed to the user. Once matched to a public transit vehicle, we could display how much delay the vehicle has compared to the schedule. Next to delays further information can also be published with real-time data, such as warnings or network outages. This involves mostly adding features to the `frontend`. Designing a simple interface for real-time updates can be done in one to two weeks. Designing an intuitive and user friendly interface might take much longer.

Moreover, we could use our program in the future to create crowd-sourced real-time data. If there is a sufficient number of users matched onto a PTV, we could measure their delay to the schedule and generate our own real-time feed. This might be useful for cities that do not have any real-time feed available. The main difficulty is matching multiple people to an already delayed PTV. Therefore, we estimate that this could take about half a year to complete.

7 Acknowledgments

First and foremost, I would like to thank my adviser Patrick Brosi. His supervision and meetings filled with suggestions and discussions were essential during my work. I also want to thank Prof. Dr. Hannah Bast for enabling me to work on this project and examining this thesis. I thank Gerrit Freiwald for working together on this project and the many hours of debugging. I want to thank my friends and family for supporting me throughout my studies. Lastly, I want to thank everyone who took time for proofreading my thesis.

Bibliography

[1] B. Hummel, "Map matching for vehicle guidance," in *Dynamic and Mobile GIS*, pp. 211-222, CRC Press, 2006.

[2] 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*, pp. 336-343, 2009.

[3] H. Bast and P. Brosi, "Sparse map-matching in public transit networks with turn restrictions," in *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pp. 480-483, 2018.

[4] R. Geisberger and C. Vetter, "Efficient routing in road networks with turn costs," in *International Symposium on Experimental Algorithms*, pp. 100-111, Springer, 2011.

[5] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," *IEEE transactions on Information Theory*, vol. 13, no. 2, pp. 260-269, 1967.

[6] H. Wei, Y. Wang, G. Forman, Y. Zhu, and H. Guan, "Fast viterbi map matching with tunable weight functions," in *Proceedings of the 20th international conference on advances in geographic information systems*, pp. 613-616, 2012.

[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[8] 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, IEEE, 2015.

[9] T. A. J. Nicholson, "Finding the shortest route between two points in a network," *The computer journal*, vol. 9, no. 3, pp. 275-280, 1966.

[10] G. Freiwald, "PublicTransitSnapper: Working with GTFS in Germany to match Mobile Phones to Public Transit Vehicles," *Bachelor's Thesis*, 2022.

[11] "General Transit Feed Specification." `https://gtfs.org/`.

[12] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pp. 47-57, 1984.