Undergraduate's Thesis

Public Transit Map-Matching with GraphHopper

Michael Fleig

Examiner: Prof. Dr. Hanna Bast
Advisers: Patrick Brosi

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

May 18th, 2021

Writing Period

18. 02. 2021 – 18. 05. 2021

Examiner

Prof. Dr. Hanna Bast

Advisers

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 wholly or excerpts thereof.

_____ _____

Place, Date Signature

Abstract

TransitRouter is a tool for generating high quality shape files for GTFS feeds. We solve the following map-matching problem: given an ordered sequence of stations and the underlying network, find the most likely route of a public transit vehicle. *TransitRouter* provides two map-matching approaches that are based on a hidden Markov Model: the *GraphHopper Map-Matching library* (*GHMM*) and our own *TransitRouter Map-Matching* (*TRMM*) with support for (inter-hop) turn restrictions. We evaluate *TransitRouter* on real-world data and compare the results against *pfaedle*.

Zusammenfassung

TransitRouter ist eine Applikation zur Generierung akkurater shape Dateien von GTFS Feeds. Hierfür lösen wir das folgende map-matching Problem: Anhand der Stationspositionen eines ÖPNV-Fahrzeugs wollen wir die möglichst korrekte Route durch einen Graphen finden. *TransitRouter* bietet hierfür zwei verschiedene Ansätze: 1. die *GraphHopper Map-Matching library* (*GHMM*) und 2. unsere eigene *TransitRouter Map-Matching Implementierung* (*TRMM*). Beide Ansätze basieren auf einem hidden Markov Model. *TRMM* unterstützt zudem noch (inter-hop) Abbiegebeschränkungen. Wir evaluieren *TransitRouter* mit den GTFS Feeds von Stuttgart und Victoria-Gasteiz und vergleichen unsere Ergebnisse mit *pfaedle*.

Contents

1 Introduction

Consider a map service like Google where you want to find a way from A to B using public transport. Ideally the correct geographical course is shown on the map but for many areas one gets only a straight line connection. Map services get their public transit information from GTFS (General Transit Feed Specification) feeds. The geographical courses or the *shape* data as its called in GTFS is optional and therefor missing in some feeds. With the *TransitRouter* application our goal is to automatically generate the accurate shape data of a GTFS feed from a sequence of stations positions and a geographic network graph. Figure 1 shows an example of two generated shapes. Our problem is a specialization of the map-matching problem where we want to find the most likely path through a network from a sequence of positions.

Map-matching is a well researched problem but there exists only little work on map-matching of schedule data. The main challenge in our work is the notion of sparseness. Having only the station positions available which might be several kilometers apart we have to be more thoughtful when finding a path between stations. In typical map matching approaches we have a high sampling rate, meaning that the GPS positions are very close to one another.

For schedule data [7] introduces a shortest-path-search algorithm to map bus stations to a road network but does not care about the generation of a correct path. In [4] an iterative shortest-path search for generating shape data of GTFS feeds is presented.

A global approach based on a Hidden Markov Model is described in [3] but leaves (inter-hop) turn restrictions as an open problem. The paper [2] shows a global HMM approach with support for (inter-hop) turn restrictions and provide the CLI tool *pfaedle* for generating shape files for GTFS feeds.

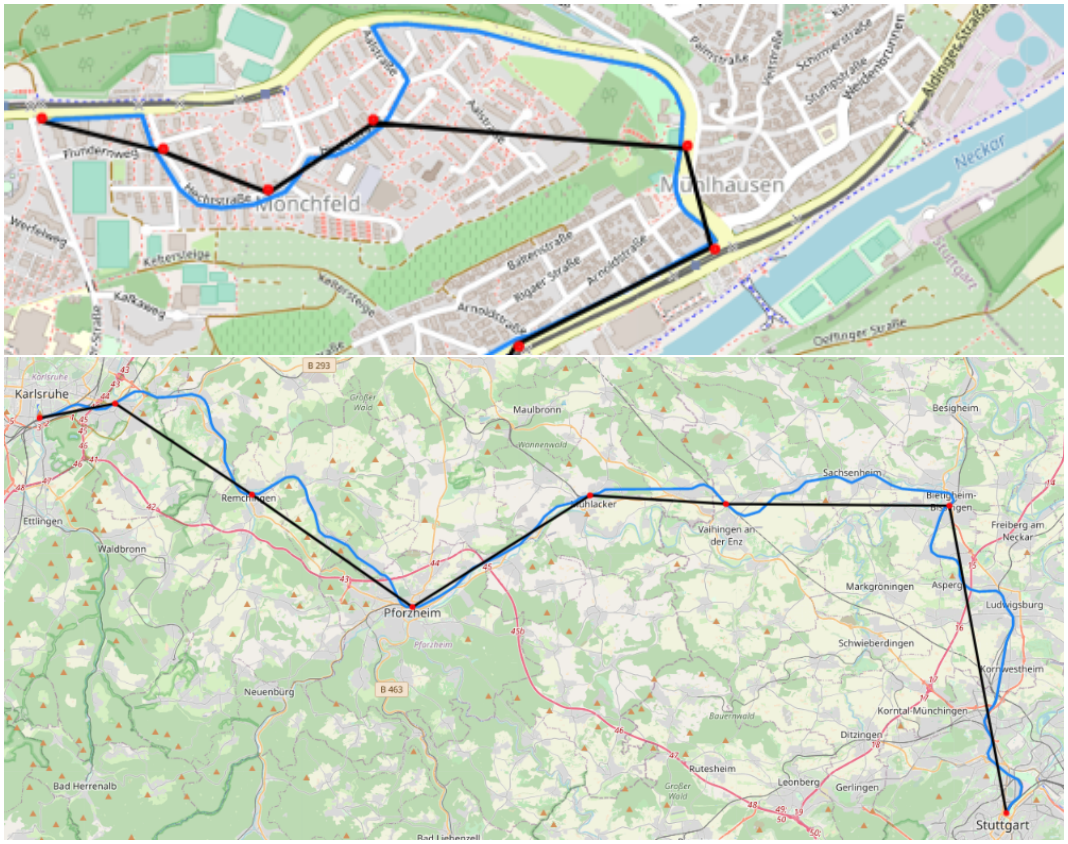

Figure 1: Example shapes of a bus trip (top) and a train trip (bottom). Black: route without shape data; Blue: shape generated by *TransitRouter*; Red: station positions

TransitRouter provides two implementations for generating shape data. First the *GraphHopper Map-Matching library* (*GHMM*) [5] and second, our own *TransitRouter Map-Matching* (*TRMM*) with support for (inter-hop) turn restrictions. Both implementations are based on the Hidden Markov Model map-matching approach described in [9]. We compare the quality our results with *pfaedle*[1] a similar tool for generating shape data for GTFS feeds.

For evaluation we use three different metrics described in the map-matching literature. The average Fréchet distance first mentioned in [3] measures the similarity between original and generated shapes. The overall accuracy is measured by a_N and a_L described in [8]. We compare the quality of the generated shapes with *pfaedle* on the GTFS feeds of Stuttgart(DE) and Victoria-Gasteiz(ES).

1.1 TransitRouter web application

With *TransitRouter* we provide a nice looking web application for generating and evaluation GTFS feeds. It allows you to upload a GTFS feed as a preset from which you can generate feeds with new shapes and compare their quality with each other. You can filter and search the trips of a feed and view the original and generated shapes on a map.

For basic feed generation *TransitRouter* provides a CLI tool with a limited set of configurations.

2 Theoretical Background

This chapter give a basic introduction to the data formats an algorithms used in our work.

2.1 General Transit Feed Specification - GTFS

GTFS was originally developed by Google in 2005 and is now the de facto standard format for public transit data. A GTFS feed is a zip file containing multiple comma separated text files. The specification [6] lists 17 files of which 7 are required. We will briefly cover the parts relevant for our problem.

routes.txt Contains the transit routes, e.g. "bus line 11 from A to B". Required attributes: `route_id`, `route_short_name`, `route_long_name`, `route_type`. The `route_type` indicates the transportation type e.g. tram(0), subway(1), rail(2) and bus(3). For each route we usually have hundreds of trips.

trips.txt Trips for each route. A trip is a sequence of two or more stops that occur during a specific time period. Relevant attributes: `route_id`, `trip_id`, `shape_id`.

stops.txt Stops or stations where vehicles pick up or drop off passengers. Required attributes: `stop_id`, `stop_name`, `stop_lat`, `stop_lon`.

stop_times.txt Times that a vehicle arrives at and departs from stops for each trip. Required attributes: `trip_id`, `arrival_time`, `departure_time`, `stop_id`, `stop_sequence`

shapes.txt A shape consists of a sequence of points and describes the path of a vehicle. A shape can be referenced by multiple trips. Attributes: `shape_id`, `shape_pt_lat`, `shape_pt_lon`, `shape_pt_sequence`, `shape_dist_traveled`.

2.2 Open Street Map - OSM

Open Street Map is an open-source project that provides geographic data of the world. The OSM specification defines the three element types *node*, *way* and *relation*. Each element has a unique id and can contain a list of key value pairs also known as *tags*.

Node A node is a geographical point. They can either represent single features such as a tree or a telephone box or they can be used in combination with ways to represent junctions or traffic lights on a road.

Way A way consists of a ordered list of nodes. Ways are used to model e.g. highways, rail roads or areas.

Relation A relation is a group of elements. It can contain ordered lists of nodes, ways and/or relations. They are used to define relations between other elements (e.g. turn restrictions). A member of a relation can have a role to describe its part in the relation.

Figure 2 shows an example of an XML osm file. For *TransitRouter* we use the binary format *pbf* for osm files.

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
<bounds minlat="54.0889580" minlon="12.2487570" maxlat="54.0913900" maxlon="
    12.2524800"/>
<node id="298884269" lat="54.0901746" lon="12.2482632" />
<node id="261728686" lat="54.0906309" lon="12.2441924" />
<node id="1831881213" lat="54.0900666" lon="12.2539381" >
  <tag k="name" v="Neu Broderstorf"/>
  <tag k="traffic_sign" v="city_limit"/>
</node>
...
<node id="298884272" lat="54.0901447" lon="12.2516513"/>
<way id="26659127" visible="true">
  <nd ref="292403538"/>
  <nd ref="298884289"/>
  ...
  <nd ref="261728686"/>
  <tag k="highway" v="unclassified"/>
  <tag k="name" v="Pastower Strasse"/>
</way>
<relation id="56688">
  <member type="node" ref="294942404" role=""/>
  ...
  <member type="node" ref="364933006" role=""/>
  <member type="way" ref="4579143" role=""/>
  ...
  <tag k="name" v="Kuestenbus Linie 123"/>
  <tag k="network" v="VVW"/>
  <tag k="operator" v="Regionalverkehr Kueste"/>
  <tag k="ref" v="123"/>
  <tag k="route" v="bus"/>
  <tag k="type" v="route"/>
</relation>
...
</osm>
```

Figure 2: OSM XML example

2.3 GraphHopper

GraphHopper[5] is an open-source high performance routing engine for *OpenStreetMap* written in Java. *GraphHopper* provides several routing algorithms such as *Dijkstra*, $A*$ and its bidirectional variants as well as modern routing optimization such as *Contraction Hierarchies* and *landmarks*. With *vehicle profiles GraphHopper* allows fine grained routing customization. In Chapter 4 we will take a deep dive into how we use vehicle profiles for realistic routing of buses, trams, subways and trains.

GraphHopper also provides an implementation of map-matching approach described [9] which we will use to solve our problem. In Chapter 3 we will explain how the *GraphHopper Map-Matching* (*GHMM*) implementation works.

Dijkstra's Algorithm

Given a graph $G = (V, E)$ and a weight function $w : E \to \mathbb{R}^+$ we want to find an optimal path $P = (e_1, e_2, ..., e_n \mid e_i \in E)$ between two nodes $s, t \in V$. Optimal meaning that the path weight $w(P) = \sum w(e_i)$ is minimal.

Two typical weighting strategies are *shortest path* (taking path with the smallest distance) and *fastest path* (taking the path with the fastest travel time).

Dijkstra is a greedy algorithm for finding the optimal path between two nodes in a graph. It was first designed by Edsger W. Dijkstra in 1956. Algorithm 1 shows an implementation in pseudo code. The running time of *Dijkstra* is $O(|E| \cdot \log |V|)$

Bidirectional Dijkstra We can speed up Dijkstra by searching the graph from s and t simultaneously. For this we maintain the structures from the normal algorithm for a forward search starting from s and a backward search starting from t. We have found

Algorithm 1 Dijksta's algorithm

```
dist[s] = 0
dist[v] = ∞   ∀v ∈ V \ {s}                ▷ tentative distances
prev[s] = NULL                            ▷ So we can backtrack the path
create priority queue Q based on dist
add s to Q
while Q is not empty do
    u = get_min(Q)                        ▷ settle u
    foreach neighbor v of u do            ▷ expand u
        d = dist[u] + w(u, v)             ▷ relax (u, v)
        if d < dist[v] then               ▷ we found a shorter path
            dist[v] = d
            prev[v] = u
        end if
    end for
end while
return path build from prev[t]
```

a optimal path from s to t if we have a node v that is settled in both the forward and backward search.

2.4 Markov Chain and Hidden Markov Model

A Markov Chain is a probabilistic model for describing the probability of a sequence of random variables which we denote as states. Consider the following example:

Given a set of States $S = \{\mathbf{h}appy, \mathbf{n}eutral, \mathbf{s}ad\}$ with initial probabilities

$$Pr(h) = 0.6, \quad Pr(n) = 0.3, \quad Pr(s) = 0.1$$

and a transition probabilities between states

$$Pr(h \to h) = 0.6, \quad Pr(h \to n) = 0.2 \quad Pr(h \to s) = 0.2$$

$$Pr(n \to h) = 0.4, \quad Pr(n \to n) = 0.4 \quad Pr(n \to s) = 0.2$$

$$Pr(s \to h) = 0.5, \quad Pr(h \to n) = 0.2 \quad Pr(s \to s) = 0.3$$

We can compute the probability of a sequence of states:

$$\begin{aligned} Pr(h \to n \to h \to s) &= Pr(h) \cdot Pr(h \to n) \cdot Pr(n \to s) \cdot Pr(s \to h) \\ &= 0.6 \cdot 0.2 \cdot 0.2 \cdot 0.5 \\ &= 0.012 \end{aligned}$$

$$\begin{aligned} Pr(s \to h \to h \to h) &= Pr(s) \cdot Pr(s \to h) \cdot Pr(h \to h) \cdot Pr(h \to h) \\ &= 0.1 \cdot 0.5 \cdot 0.6 \cdot 0.6 \\ &= 0.018 \end{aligned}$$

From the transition probabilities we can see, that the probability distribution for the current state only depends of previous state. This is called *Markov Property*.

Hidden Markov Model

Given a Markov Chain we can now easily compute the probability of sequence of observable states. But what if we cannot directly observe the states we are interested in. For example if we want to find out if a person ate candy or not based on their mood. We can do this by using a Hidden Markov Model (*HMM*).

Given a set of hidden states $H = \{c^+(\text{candy}),\ c^-(\text{no candy})\}$ and a set of observable states $O = \{\mathbf{h}appy, \mathbf{n}eutral, \mathbf{s}ad\}$.

We assume a Markov Chain for our hidden states:

$$Pr(c^+) = 0.5, \quad Pr(c^-) = 0.5$$

$$Pr(c^+ \to c^+) = 0.4, \quad Pr(c^+ \to c^-) = 0.6$$

$$Pr(c^- \to c^+) = 0.7, \quad Pr(c^- \to c^-) = 0.3$$

Our observed states a conditional on our hidden states. We call these the emission probabilities:

$$Pr(h \mid c^+) = 0.7, \quad Pr(n \mid c^+) = 0.2, \quad Pr(s \mid c^+) = 0.1$$

$$Pr(h \mid c^-) = 0.3, \quad Pr(n \mid c^-) = 0.3, \quad Pr(s \mid c^-) = 0.4$$

The probability of a given sequence of observations and hidden states is given by

$$\begin{aligned} Pr(o_1, ..., o_n, h1, ..., h_n) &= Pr(o_1, h_1) \cdot Pr(o_2, h_2 \mid o_1, h_1) \cdot \ldots \cdot Pr(o_n, h_n \mid o_{n-1}, h_{n-1}) \\ &= Pr(h_1) \cdot Pr(o_1 \mid h_1) \cdot Pr(h_2 \mid h_1) \cdot Pr(o_2 \mid h_2) \cdot \ldots \\ &\quad \cdot Pr(h_n \mid h_{n-1}) \cdot Pr(o_n \mid h_n) \\ &= \prod_{i=1}^{n} Pr(h_i \mid h_{i-1}) \cdot Pr(o_i \mid h_i) \end{aligned}$$

To find the most likely sequence of hidden states we need to solve the following optimization problem.

$$\max_{h_1,\ldots,h_n} \prod_{i=1}^{n} Pr(h_i \mid h_{i-1}) \cdot Pr(o_i \mid h_i)$$

A naive approach would be to try out all possible assignments of $h1, \ldots, h_n$ but with $|H|^n$ different assignments this is not a feasible solution.

Viterbi Algorithm

The Viterbi algorithm is a dynamic programming algorithm to compute the most likely sequence of a Hidden Markov Model in $O(n \cdot |H|^2)$ time. The recursive formula is constructed as follows:

$$\begin{aligned}
P_n(h_n) &:= \max_{h_1,\ldots,h_{n-1}} \prod_{i=1}^{n} Pr(h_i \mid h_{i-1}) \cdot Pr(o_i \mid h_i) \\
&= \max_{h_1,\ldots,h_{n-1}} \left\{ Pr(h_n \mid h_{n-1}) \cdot Pr(o_n \mid h_n) \cdot \prod_{i=1}^{n-1} Pr(h_i \mid h_{i-1}) \cdot Pr(o_i \mid h_i) \right\} \\
&= Pr(o_n \mid h_n) \cdot \max_{h_{n-1}} \{ Pr(h_n \mid h_{n-1}) \} \cdot \max_{h_1,\ldots,h_{n-2}} \left\{ \prod_{i=1}^{n} Pr(h_i \mid h_{i-1}) \cdot Pr(o_i \mid h_i) \right\} \\
&= Pr(o_n \mid h_n) \cdot \max_{h_{n-1}} \{ Pr(h_n \mid h_{n-1}),\ P_{n-1}(h_{n-1}) \}
\end{aligned}$$

3 Approach

In this section we describe the two different map-matching approaches that can be used with *TransitRouter*. The first one is the map-matching implementation of *GraphHopper*. We will refer to this as *GHMM*. The second one is a modified version of *GHMM* that supports (inter-hop) turn restrictions which we will refer to as *TRMM*. Both *GHMM* and *TRMM* are based on the Hidden Markov Model approach described in [9].

Given a trip T with a ordered sequence of station positions $S = (s_0, s_1, s_2, ..., s_n)$ and a street graph $G = (V, E)$ we want to find the most likely path P trough G.

In practice we do not generate a shape for each trip. Most trips of a route visit the same stations in the same sequence and only differ in their time. We combine all trips of a route with the same station sequence in one group and compute and store the shape only once per group.

3.1 Finding Candidates

Our station positions might not be accurate due to general GPS measurement error or mismatch with the OSM data. We have to map the station positions to the underlying road network. A naive approach would be to map the station to its nearest node in the road network. Consider the following example in Figure 3. We have two stations s_0, s_1 in green, the actual station positions in red and the nodes of the road network

in black. If we map a station directly to its closest node in G we get an incorrect solution.

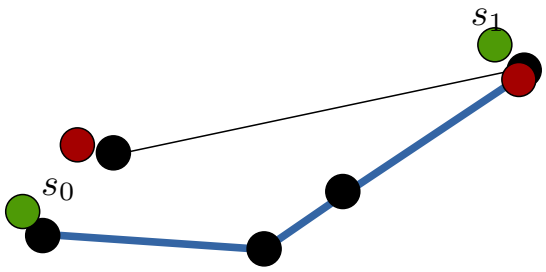

Figure 3: Imprecise station positions leads to wrong path

Generally we cannot map a station to a single node as we cannot guarantee the correctness. Instead we select a set of possible candidate nodes $C_i = \{c_i^0, c_i^1, ...\}$ for every stations s_i.

For every edge $e \in E$ within a radius r around s_i, we insert a candidate node at the projection of s_i on e. We use $r = 10m$ as a default value.

With multiple candidate nodes per stations we compute the path from each candidate node in C_i to each candidate node in C_{i+1}. Here we use either *shortest* or *fastest* routing. In Chapter 4 we will take a closer look at path finding with *Graphhopper* and in Chapter 5 we will see the differences between *shortest* and *fastest* routing.

As a last step we find the most likely candidate sequence as shown in the next section. The final path can be obtained by combining the path segments between the selected candidates. Figure 4 shows an example of a bus trip.

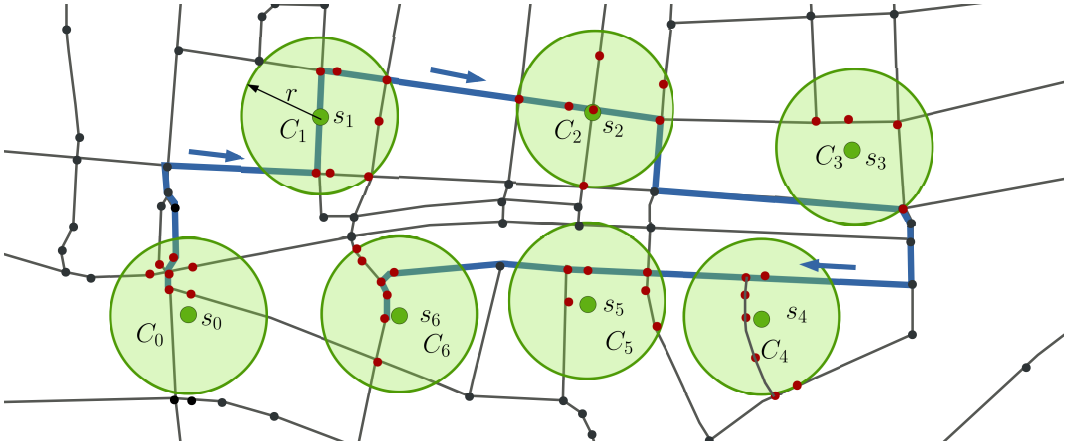

Figure 4: Example of a bus trip of a GTFS feed. Green: possible imprecise station positions $s_0, ..., s_6$. Red: candidate nodes within a radius r. Blue: path of the bus.

3.2 Finding the optimal candidate sequence

To find the most likely sequence of candidates we use the Hidden Markov Model (*HMM*) approach presented in [9]. We construct a *HMM* with our stations s_i as observations and the candidate nodes C_i as hidden states. We use the emission probabilities as initial probabilities for the hidden states.

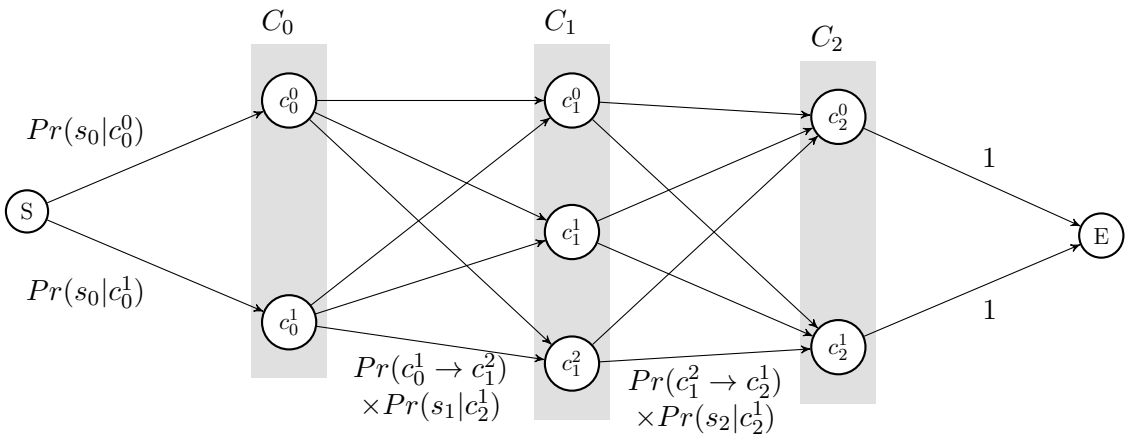

Figure 5: Hidden Markov Model for finding the most likely sequence.

Emission probability The emission probability $p(s_i|c_i^k)$ describes the likelihood that we observe s_i given that our vehicle drives through the candidate node c_i^k. We assume, that candidates closer to a station are more likely to be correct. We further assume that the measurement errors have a zero-mean normal distribution with σ as the standard deviation of the GPS measurements.

$$d = \|s_i - c_i^k\|_{\text{great circle}}$$

$$p(s_i|c_i^k) = \frac{1}{\sqrt{2\pi}\sigma} e^{-0.5(\frac{d}{\sigma})^2}$$

Transition probability The transition probability $p(c_i^k \to c_{i+1}^j)$ describes the likelihood that c_i^k and c_{i+1}^j are matching candidates. This probability should depend on the path between the two candidates. A transition with a complicated / long path is more unlikely than a transition with a direct path. Experiments in [9] have shown, that the distance between the stations $\|s_i - s_{i+1}\|_{\text{great circle}}$ and the length of the road path $\|c_i^k - c_{i-1}^j\|_{\text{route}}$ are very close to one another. For the transition probability we use their proposed exponential distribution with tuning parameter β:

$$d_t = |\|s_i - s_{i+1}\|_{\text{great circle}} - \|c_i^k - c_{i-1}^j\|_{\text{route}}|$$

$$p(c_i^k \to c_{i+1}^j) = \frac{1}{\beta} e^{-\frac{d_t}{\beta}}$$

Final solution Given an optimal sequence of candidate nodes we construct the final path P by combining the path segments found by *GraphHopper*.

3.3 GraphHopper Map-Matching Implementation - *GHMM*

The GraphHopper Map-Matching library is a one to one implementation of [9]. For our problem we need to make some minor adjustments.

Filtering of close observations Chapter 4.2 in [9] describes the filtering of observations. Observation that are within distance of 2σ of the previous observation are removed because the apparent movement might be due noise instead of actual vehicle movement. In our scenario every observation is a station that we need to visit so removing an observation will yield a wrong solution. Therefore we removed the filtering in *GHMM*

We further made the implementation thread safe to allow parallel computation of shapes made some cosmetic changes to the return types and interfaces to accommodate our needs.

3.4 TransitRouter Map-Matching Implementation - *TRMM*

Our TransitRouter Map-Matching Implementation (*TRMM*) is based on *GHMM* but takes a different path finding approach to enable turn restrictions and prevent inter hop turns.

Turn Restrictions

GraphHopper supports routing with OSM turn restriction relations. Enabling turn restrictions can improve the quality of the generated shapes. Figure 6 shows an

example of a route with and without turn restrictions enabled.

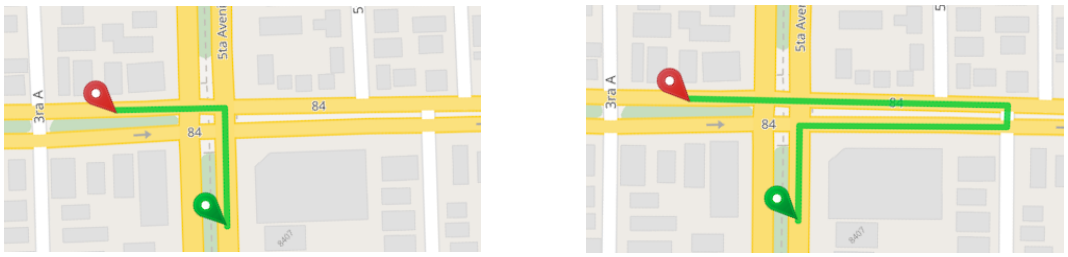

Figure 6: Left: without turn restrictions. Right: with turn restrictions

Inter Hop Turns

With turn restrictions enabled we prevent illegal turns for the path segments between candidates but in the final path we might still have illegal or highly unlikely turns at candidate nodes.

Let c be a candidate node and a, b two edges of c with a turn restriction meaning we are not allowed to travel along a, b through c. Let $P_1 = (e_1, e_2, ..., a)$ be the path segment from a previous candidate node to c and $P_2 = (b, e_1, e_2, ...)$ the path segment from c to a next candidate. On their own P_1 and P_2 are valid paths but when we connect them in our final solution we violate the turn restriction.

Even if no turn restriction is violated we still might have unlikely turns at candidate nodes. Consider the following example in Figure 7. On the left we have the most likely route calculated by our algorithm with a full U-turn at c_1^0 which is allowed but in reality highly unlikely (a bus usually does not make a full U-turn at a station). On the right we have the correct path of the trip.

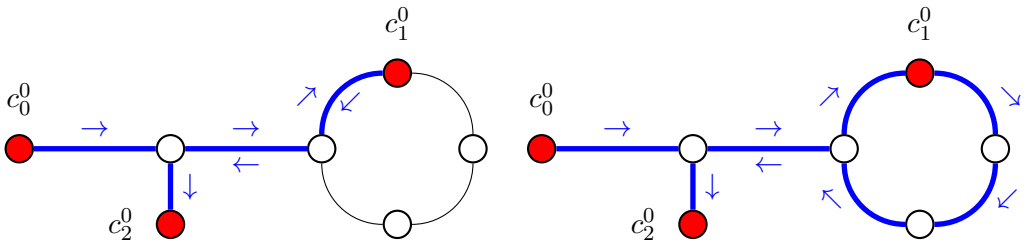

Figure 7: Example of inter hop turns

Directed Candidates To prevent inter hop turns we need to know on which edge we arrived at a candidate node. *TRMM* solves this by using directed candidates. We create a candidate for each node and connecting edge. $c_i^k = (u \in V, e \in E)$

Figure 8: Left: orientation less candidate node c. Right: directed candidates c_1, c_2

Path finding with directed candidates GraphHopper allows us to set an outgoing edge for the start node and an incoming edge for the end node.

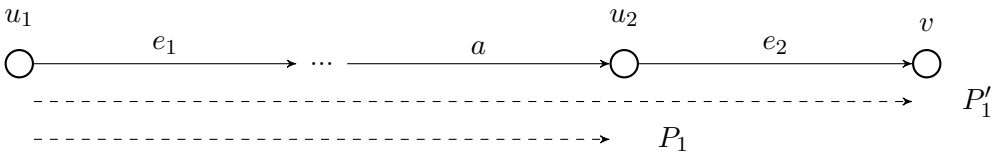

Figure 9: Path finding with directed candidates

Given two candidates $c_1 = (u_1, e_1), c_2 = (u_2, e_2)$. Let v be the neighbor of u_2 connected by e_2. We calculate the path P_1' from u_1 leaving on edge e_1 to v arriving through e_2. For the final solution we construct the path P_1 from P_1' by removing the last edge. To punish full U-turns (i.e. $a = e_2$) that are not forbidden by turn restrictions we increase the length of the path P by the length of the last edge.

Fallback With turn restrictions enabled we sometimes run in to the problem that we cannot find any path between two sets of candidates C_i, C_{i+1} resulting in a broken sequence. If this happens for a trip we restart the process with turn restrictions disabled. We tried to investigate the problem but could not find any reason as to why *GraphHopper* is not able to find a path.

4 GraphHopper Routing

GraphHopper allows fine grained routing customization through profiles, weightings and flag encoders.

Flag encoders define the basic rules for a vehicle. *GraphHopper* provides encoders for *car*, *bike*, *foot* routing. For *TransitRouter* we implemented specialized encoders for *bus* and *rail* vehicles. A flag encoder controls which nodes and ways are accessible, in which direction a way can be traversed and at what speed.

With a weighting we can set the weight of edges in the graph to define path finding strategy. *TransitRouter* uses the *fastest* and *shortest* weighting from *GraphHopper*.

At last we use profiles to combine vehicles and weighting and allow further configuration of the GraphHopper routing engine. TransitRouter provides the following profiles:

Key	Description	flag encoder	weighting	additional properties
$bus\_fastest$	bus	*bus*	$fastest$	turn costs
$bus\_shortest$	bus	*bus*	*shortest*	turn costs
rail	tram, subway and trains	*rail*	*shortest*	-

Table 1: Available profiles in *TransitRouter*

At startup *GraphHopper* creates an optimized graph for every profile from the OSM file.

4.1 Represent vehicles with flag encoders

A flag encoder has two processing steps. In the first step we filter out all ways that our vehicle is not allowed to access based on the OSM tags of a given way *(FlagEncoder.getAccess)*. In the second step we set the speed and direction of a way *(FlagEncoder.handleWayTags)* and check if we can access the nodes on a way *(FlagEncoder.handleNodeTags)*.

For our flag encoders we extend GraphHoppers *AbstractFlagEncoder* that handles some of the work for us. The *AbstractFlagEncoder* provides the following collections used for access rules:

Restrictions	Ordered list of OSM tags that might restrict access. Only the first existing tag is considered.
Restricted Values	List of values for the restriction tag that are not allowed.
Intended Values	List of values for the restriction tag that are allowed.
Potential Barriers	List of barrier values that might restrict access.
Absolute Barriers	List of barrier values that restrict access.

Table 2: Predefined collections for access rules

For the restrictions the order is important because we might have a way where access is generally allowed but forbidden for a certain vehicle type or generally forbidden except for a certain type. Consider a pedestrian zone where motor vehicles are not allowed (`motor_vehicle=no`) but a bus is (`bus=yes`). We use only the first tag existing tag.

Filter Nodes - AbstractFlagEncoder.handleNodeTags By default we allow access. We deny access if a node has a `barrier` tag that is a *absolute barrier*. If a node has a `barrier` tag that is a *potential barrier* we loop trough the *restrictions*. If no restriction is present deny access. If the restriction is a *restricted value* deny access. If the restrictions is a *intended value* and the node does not have a `locked=yes` tag allow access.

4.2 Bus Flag Encoder - (bus)

Public transit and bus vehicles often have special traffic rules. For example they are allowed to travel roads that forbidden for private motor vehicles or are allowed to travel in the opposite direction of a one-way street. So to get a realistic routing we need a specialized bus vehicle and cannot use the car vehicle from *GraphHopper*. Table 3 lists the access rules for our bus vehicle. In addition we keep a collection of allowed highway values with their default travel speed.

Restrictions	`bus`, `psv`, `motorcar`, `motor_vehicle`, `vehicle`, `access`
Restricted Values	`no`, `agricultural`, `forestry`, `restricted`, `delivery`, `military`, `emergency`, `private`, `customers`
Intended Values	`yes`, `permissive`, `designated`
Potential Barriers	`block`, `gate`, `lift_gate`, `swing_gate`
Absolute Barriers	`fence`, `bollard`, `stile`, `turnstile`, `cycle_barrier`, `motorcycle_barrier`, `sump_buster`
Allowed highways	`motorway`(100), `motorway_link`(70), `trunk`(70) `trunk_link`(65), `primary`(65), `primary_link`(60), `secondary`(60), `secondary_link`(50), `tertiary`(50), `tertiary_link`(40), `unclassified`(30), `residential`(30), `living_street`(5), `service`(20), `road`(20), `platform`(50), `bus_guideway`(50)

Table 3: access rules for the BusFlagEncoder

Filter ways - *BusFlagEncoder.getAccess*

First we check if the way has a highway tag and if we can access this highway type. If not, we skip this way. Otherwise we get the first restriction tag an check if it forbids access. Listing 4.1 shows the function in pseudo code.

```
1 public Access getAccess(ReaderWay way) {
2   highway = way.getTag("highway");
3   firstRestriciton = way.getFirstPriorityTag(restrictions);
4
5   if (highwayValue == null
6       || !allowedHighways.contains(highway)) {
```

```
7     return Access.CAN_SKIP;
8   }
9
10  if (restrictedValues.contains(firstRestriciton)) {
11    return Access.CAN_SKIP;
12  }
13
14  return Access.WAY;
15 }
```

Listing 4.1: BusFlagEncoder.getAccess

BusFlagEncoder.handleWayTags

Travel speed By default we use the speed predefined for the specific highway type. If the way has a valid value for the *maxspeed* tag we use max speed instead.

Travel direction and one ways A way is a one-way if it has one of the following tags: `oneway`, `vehicle:forward`, `vehicle:backward`, `motor_vehicle:forward`, `motor_vehicle:backward`, `junction=roundabout`, `junction=circular`. If the way has one of the following tags `onway=-1`, `vehicle:forward=no`, `motor_vehicle:forward=no` we can access the way backwards, otherwise forward.

Some one-ways have additional lanes for public transit vehicles that allow access in both directions. If a one-way contains one of the following tags `busway`, `oneway:psv=no`, `onway:bus=no` we allow access on both directions.

Another special case we have to look at are ways that are restricted to local access only (meaning "except for access" (UK) / "no thru traffic" / "local traffic only" (USA), "Anlieger frei (DE)"). We encounter those ways a lot in downtown areas and one-lane roads that are usually used for agriculture or forest vehicles. So in general we should not allow these ways and already filter them in the *getAccess*

step. Allowing access drastically reduces the quality for the generated shapes in Stuttgart(DE) while restricting access reduces the quality in Victoria-Gasteiz(ES). Many of these ways in downtown areas allow access for public transport vehicles but are missing this information in the OSM tags. To handle this problem we look at the relations of these ways. If a way has one of the tags `vehicle=destination` or `motor_vehicle=destination` we allow access only if it is part of a relation with the tag `route=bus`.

4.3 Rail Flag Encoder - $(rail)$

The *RailFlagEncoder* used for tram, subway and train routes is quite simple. We leave the collections of the *AbstractFlagEncoder* empty.

Filter ways - *RailFlagEncoder.getAccess*

We can access a way if it contains the `railway` tag with one of the following values: `tram`, `subway`, `rail`, `light_rail`.

RailFlagEncoder.handleWayTags

We set an arbitrary default speed and allow access in both directions.

Limitations for rail vehicles

For rail vehicles we only support *shortest* routing as we do not have enough information in the OSM data about speed restrictions. Turn restrictions are also not supported because there are no turn restrictions for railways in OSM.

5 Evaluation

In this chapter we will first introduce the different metrics used to measure the quality of our generated shape files. Then we will compare the results of *TransitRouter* with *pfaedle* on the GTFS feeds of Stuttgart(DE) and Victoria-Gasteiz(ES). Both feeds contain high a quality shape file that we can use as a base line.

5.1 Evaluation Method

Let P be the path of a generated shape and Q the path of the ground truth. We measure the quality of P in three metrics: average Fréchet distance δ_{a_F}, percentage of unmatched hop segments A_N and percentage of length of unmatched hop segments A_L.

Fréchet distance δ_F

The Fréchet distance δ_F measures the similarity between two curves. For an intuitive understanding one can imagine a person walking a dog on a leash. Both traverse their own curve at varying speed while never moving backwards. During the walk the leash is sometimes longer (e.g. dog walks ahead) and sometimes shorter (e.g. dog and person walk side by side). The Fréchet distance is the shortest length of the leash needed for the person and the dog to walking their separate paths from start to end.

Let $A, B : [0,1] \to \mathbb{R}^2$ to curves in the metric space $\mathbb{R}^2$. Let $\alpha, \beta : [0,1] \to [0,1]$ be two continuous, non decreasing mapping functions for A, B with $\alpha(0) = \beta(0) = 1$ and $\alpha(1) = \beta(1) = 1$. The continuous Fréchet distance is defined as:

$$\delta_F(A,B) := \inf_{\alpha,\beta} \max_{t\in[0,1]} \|A(\alpha(t)) - B(\beta(t))\|$$

In practice we compute an approximation of the continuous Fréchet distance. For this we only look at fixed sampling points of A and B.

Let $0 = i_1 < i_2 < \ldots < i_M = 1$ the sampling point for A and $0 = j_1 < j_2 < \ldots < i_N = 1$ the sampling points for B. A coupling P between A and B is a sequence $P = \langle (i_{n_0}, j_{m_0}), (i_{n_1}, j_{m_1}), \ldots, (i_{n_p}, j_{m_p}) \mid (i_{n_k}, j_{m_k}) \in \{i_1, \ldots, i_N\} \times \{j_1, \ldots j_M\} \rangle$ with $n_0 = m_0 = 0$ and $n_p = N$, $m_p = M$ as well as $n_{k+1} = n_k$ or $n_{k+1} = n_k + 1$ and $m_{k+1} = m_k$ or $m_{k+1} = m_k + 1$. The set of all possible couplings is denoted as $\mathcal{P}$.

A coupling P assigns every sampling point in A at least one sampling point in B. One can think of a coupling as a sequence of movements. To get back to our dog walking example with A as the dog owner and B as the dog. For every element of a coupling we have three cases:

1. $n_{k+1} = n_k$ and $m_{k+1} = m_k + 1$: Dog walks forward

2. $n_{k+1} = n_k + 1$ and $m_{k+1} = m_k$: Owner walks forward

3. $n_{k+1} = n_k + 1$ and $m_{k+1} = m_k + 1$: Both walk forward

For the approximation we define the Fréchet distance as:

$$\delta_F = \min_{P\in\mathcal{P}} \max_{k=1\ldots|P|} \|A(i_{n_k}) - B(j_{m_k})\|$$

Average Fréchet distance δ_{a_F}

The Fréchet distance is strongly affected by single outliers as it takes the maximum of a set of distances. This punishes long shapes with a single outlier harder than short shapes. To achieve a more fair comparison we use average the Fréchet distance that takes the sum of all distances and divides them by the length of the shape.

$$\begin{aligned}\delta_F^{int}(A,B) &:= \inf_{\alpha,\beta} \int_{\alpha,\beta} \|A(\alpha(t)) - B(\beta(t))\| dt \\ &= \inf_{\alpha,\beta} \int_0^1 \|A(\alpha(t)) - B(\beta(t))\| \cdot \left\| \begin{pmatrix} \dot{\alpha}(t) \\ \dot{\beta}(t) \end{pmatrix} \right\| dt\end{aligned}$$

$$\delta_F^{sum}(A,B) := \min_{P \in \mathcal{P}} \sum_{k=2}^{|P|} \|A(i_{n_k}) - B(j_{m_k})\| \cdot \left\| \begin{pmatrix} i_{n_k} - i_{n_{k-1}} \\ j_{m_k} - j_{m_{k-1}} \end{pmatrix} \right\|$$

To get the average Fréchet distance we divide the summed Fréchet distance by the length of the optimizing coupling P

$$\|P\| := \sum_{k=2}^{|P|} \left\| \begin{pmatrix} i_{n_k} - i_{n_{k-1}} \\ j_{m_k} - j_{m_{k-1}} \end{pmatrix} \right\|$$

$$\delta_{a_F}(A,B) := \frac{\delta_F^{sum}(A,B)}{\|P\|}$$

Accuracy by number a_N describes the percentage of unmatched hop segments of a trip. A hop segment is unmatched if its Fréchet distance $\delta_F > 20m$

$$a_N = \frac{\#\text{unmachted hop segmets}}{\#\text{hop segments}}$$

Summed accuracy a is the percentage of trips a_N below a given threshold. $a(0.2) = 0.6$ means that 60% of all trips have an $a_N \leq 0.2$

Accuracy by length a_L The accuracy by length a_L compares the length of the unmatched hop segments of a trip with its total length.

$$a_L = \frac{\text{lenght of umatched segments}}{\text{total length}}$$

5.2 Evaluation results

We evaluate quality for *TransitRouter* on the GTFS feeds of Stuttgart (DE) and Victoria-Gasteiz (ES). Both feeds already contain high quality shape files that we can use as ground truth. Additionally we compare our results with *pfaedle* [1].

For both feeds we generate shapes using *GHMM* and *TRMM* with both *shortest* and *fastest* routing. We use the default tuning parameters $\sigma = 10$ and $\beta = 1$.

The metrics for *TransitRouter* and *pfaedle* are computed by *shapevl* a CLI tool developed by Patrick Brosi at the chair of Algorithms and Data Structures at the University of Freiburg.

The evaluation results for Stuttgart are shown in Figure 12 with separate graphs for bus and rail trips in Figure 13 and Figure 14. Figure 15 shows the results for Victoria-Gasteiz. *TransitRouter* produces almost perfect shapes for both feeds with most trips having an $\delta_{a_F} \leq 20m$. Both *TRMM* and *GHMM* slightly outperform *pfaedle*. For Stuttgart the quality of *TRMM* and *GHMM* are almost identical while for Victoria-Gasteiz *GHMM* is slightly better. For Stuttgart *fastest* routing produces the best results while for Victoria-Gasteiz *shortest* routing is clearly more accurate. This might be due the average distance between the stations. In Stuttgart the bus

stops have an average distance of $715m$. For Victoria-Gasteiz the average distance is $380m$. For large distances the shortest path often includes minor roads that are usually not meant for drive through traffic.

For some trips we have a very large δ_{a_F}. This might happen when two stations with a large distance have several plausible routes and *TransitRouter* picks the wrong one (see Figure 10). We might solve this problem by using additional OSM meta data about transit routes. Another reason are errors in the ground truth were some shapes are wrong (see Figure 11) or where the shapes are not complete i.e. end in the middle of the route.

Figure 10: Two possible routes

Figure 11: Error in ground truth

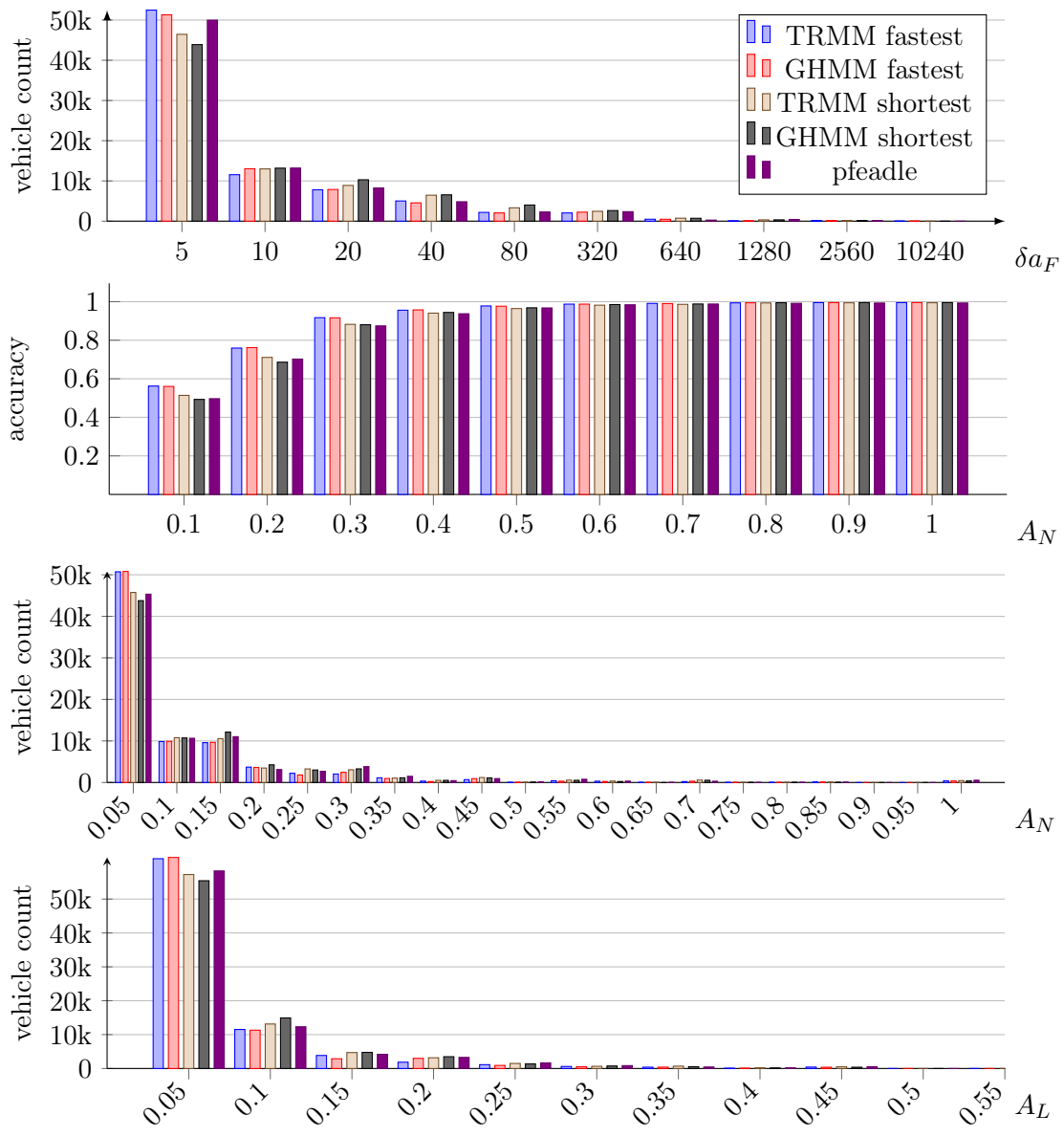

Figure 12: Evaluation results for Stuttgart

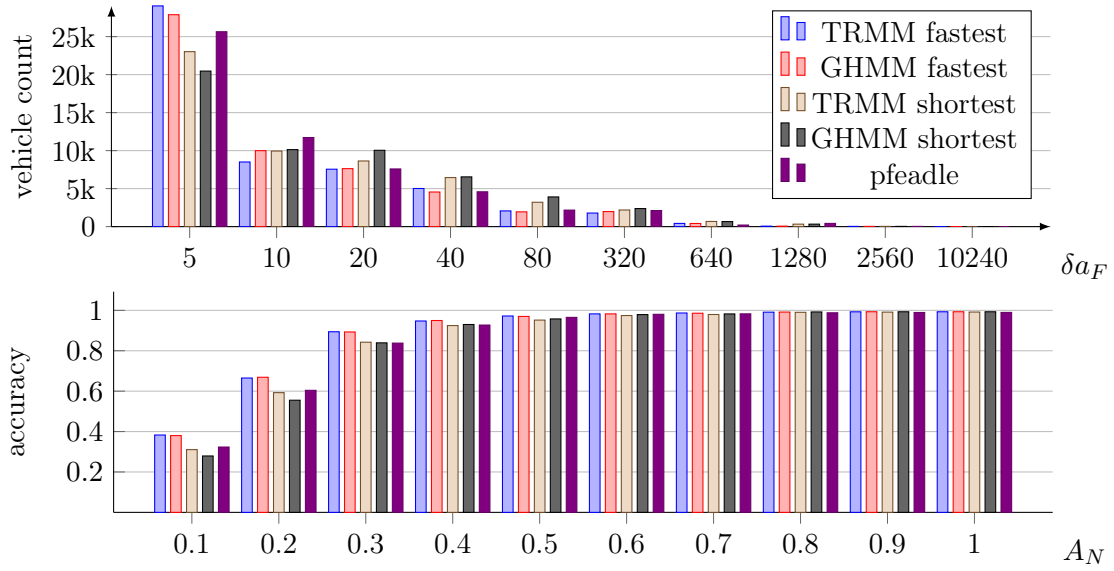

Figure 13: Evaluation results for bus routes of Stuttgart

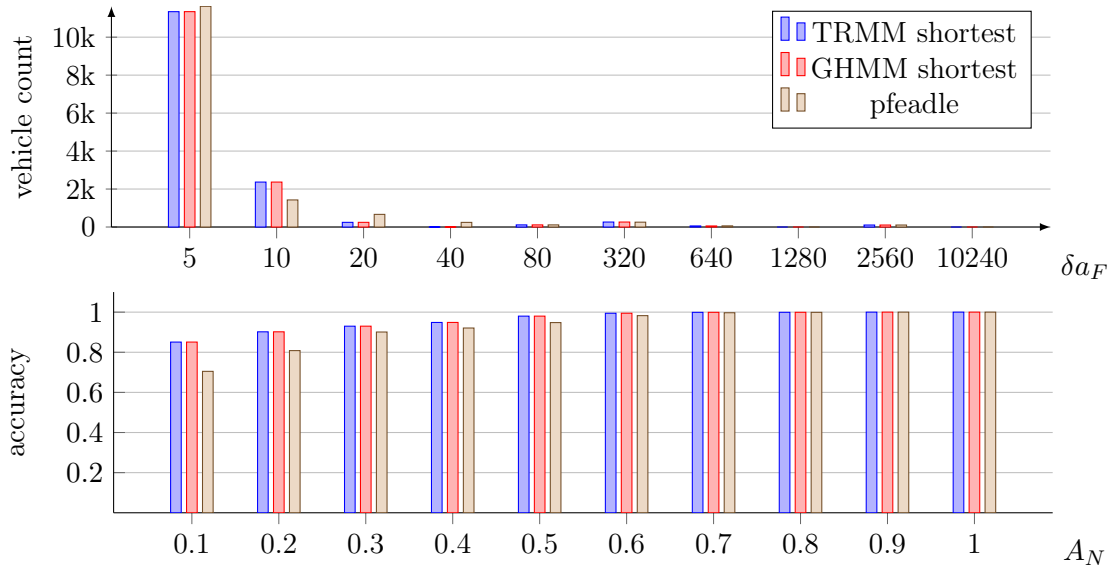

Figure 14: Evaluation results for rail routes of Stuttgart

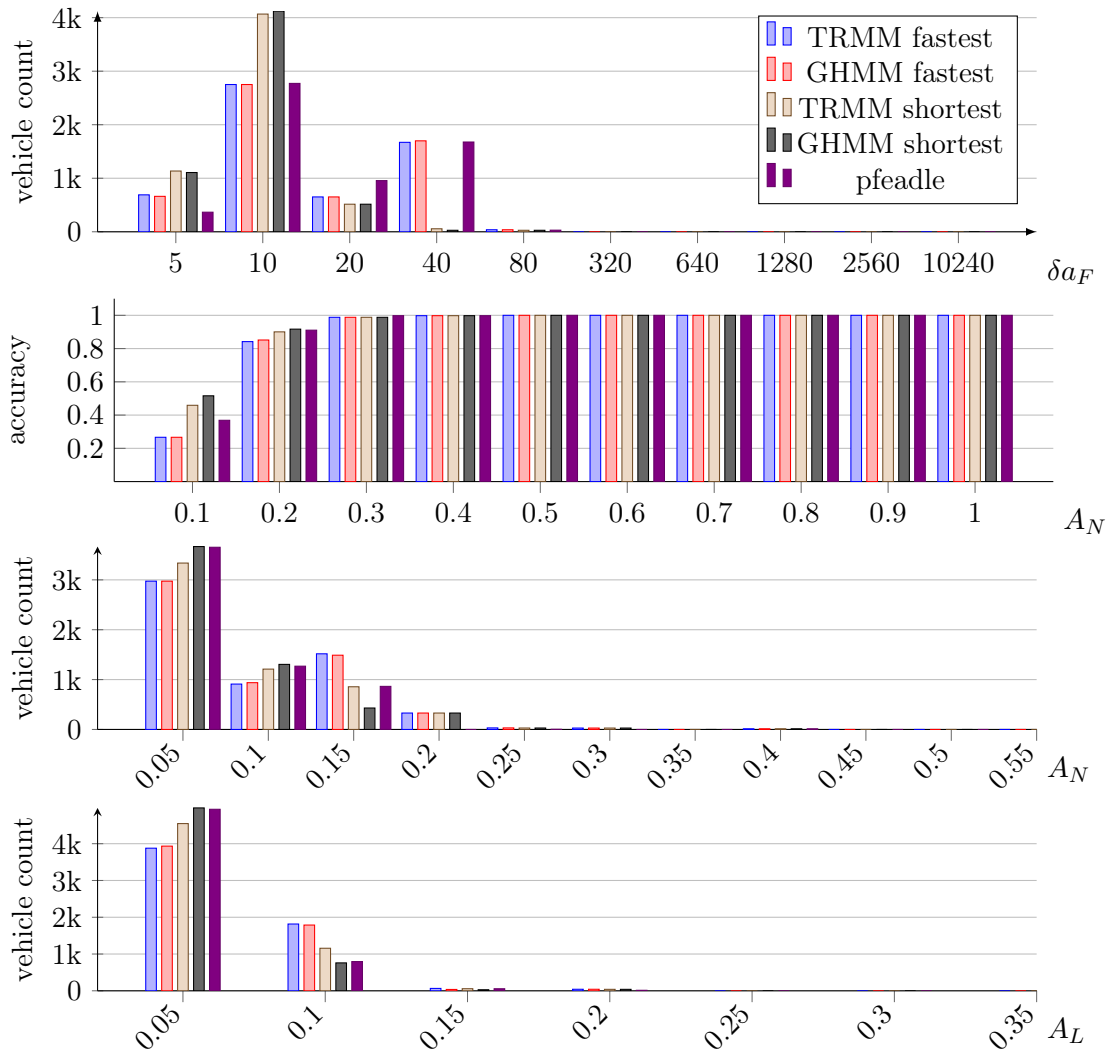

Figure 15: Victoria-Gasteiz: average Fréchet distance δ_{a_F}

6 Current Problems and Future Work

Use OSM relation meta data OSM contains valuable information about public transit routes. In our *BusFlagEncoder* we already use the route relation to determine if a highway with access restriction *destination* can be accessed. We could this information to prioritize ways during path finding that are part of a route relation. This might improve the overall quality of the generated shapes.

Turn restrictions based on turn angle For trains we might have turns that are not full U-turns or forbidden by turn restrictions but are impossible due to physical limitations. For example changing lanes / direction through a zick zack course (we usually cannot drive backwards). This could be prevented by punishing turns over 45°.

Non-optimal transition probability distribution We use the probability distribution described in [9]. In their experiments they showed, that the difference between the distance of the stations and the length of the road segment are close to 0. This is not the case for our data. Figure 16 shows the histogram of the differences for the bus trips of Stuttgart. As a result the transition probabilities for any two transitions might be almost identical. A different probability distribution might lead to better results.

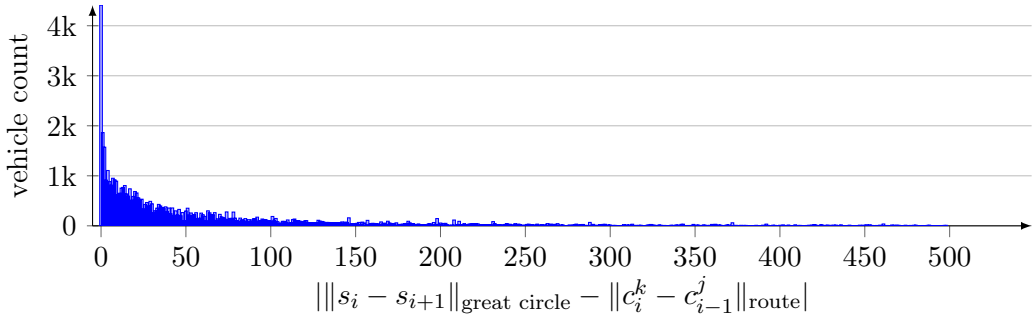

Figure 16: Histogram of distance difference between stations and road path for the bus trips of Stuttgart

Path finding difficulties with turn restrictions With turn restrictions enabled for some trips GraphHopper is not able to find any path between the candidates of two stations. If this happens we disable turn restrictions for that particular trip. For Stuttgart this affects around 10% of all trips. For Victoria-Gasteiz none of the trips are affected.

We were not able to find the reasons why GraphHopper is not able to find any path yet. This issue requires further investigation.

Bibliography

[1] Hannah Bast and Patrick Brosi. *pfaedle*. Chair for Algorithms and Data Structures, Department of Computer Science, University of Freiburg. URL: https://github.com/ad-freiburg/pfaedle.

[2] Hannah Bast and Patrick 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*. SIGSPATIAL '18. Seattle, Washington: Association for Computing Machinery, 2018, pp. 480–483. ISBN: 9781450358897. DOI: 10.1145/3274895.3274957. URL: https://doi.org/10.1145/3274895.3274957.

[3] Sotiris Brakatsoulas et al. "On Map-Matching Vehicle Tracking Data". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 853–864. ISBN: 1595931546.

[4] Patrick Brosi and Uli Müller. "Automatisierte Erstellung von fahrplanreferenzierten ÖV-Netzen". In: *AGIT* (), pp. 122–129.

[5] *GraphHopper - a fast and memory efficient Java routing engine for OpenStreetMap*. URL: https://github.com/graphhopper/graphhopper.

[6] *GTFS - General Transit Feed Specification*. URL: https://gtfs.org.

[7] Jing-Quan Li. "Match bus stops to a digital road network by the shortest path model". In: *Transportation Research Part C: Emerging Technologies* 22 (June 2012). DOI: 10.1016/j.trc.2012.01.002.

[8] Yin Lou et al. “Map-matching for low-sampling-rate GPS trajectories”. In: Jan. 2009, pp. 352–361. DOI: 10.1145/1653771.1653820.

[9] Paul Newson and John Krumm. *Hidden Markov Map Matching Through Noise and Sparseness*. Nov. 2009. URL: https://www.microsoft.com/en-us/research/publication/hidden-markov-map-matching-noise-sparseness/.

[10] Hong Wei et al. “Fast Viterbi Map Matching with Tunable Weight Functions”. In: *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*. SIGSPATIAL ’12. Redondo Beach, California: Association for Computing Machinery, 2012, pp. 613–616. ISBN: 9781450316910. DOI: 10.1145/2424321.2424430. URL: https://doi.org/10.1145/2424321.2424430.