UNIVERSITÉ
LIBRE
DE BRUXELLES

Map Matching

Map Matching as service

Mohammadreza Amini

Preparatory work for master thesis submitted under the supervision of
Prof. Mahmoud Sakr

Academic year
2021-2022

Abstract

The problem of map matching consists of matching GPS measurements which can be abstracted as triplets of (longitude, latitude, time) to road network in order to identify the correct path on which the vehicle is traveling. Using only GPS data to know the location of a vehicle and its path is not enough as GPS points can be inaccurate and in some areas there are GPS blockage. This is the reason why the map matching problem is important as it can improve the performance of navigation systems and matching GPS points with the underlying road network.

There are a long range of tools and algorithms that solve the problem of map matching. These tools differ on the way that they solve the problem. Some use geometric approaches while others use probabilistic model approaches. The aim of this paper is to first give a state of the art of the problem and existing solutions and second, to test some well known existing tools in order to see the results and the performance of these tools.

All of the data used in our tool testing and experiments are ground truth data that were gathered by the author. This gave us the needed domain knowledge on the road network to better analyze and observe the results. The aim of the experiments is to see how these tools differentiate from each other.

Introduction

The problem of map matching might seem relatively easy at first glance. One could simply match each GPS point to the nearest road segment. This, of course, could produce inaccurate result as GPS points are not accurate and sometimes in dense city areas, there are a lot of road segments that are close to each other. Another solution is to consider a sequence of GPS points and create an arc with them and match them to the nearest road segment. This also might yield inaccurate results when GPS samples are not accurate enough. Hence, we need a more complex solution that takes into account the inaccuracy of GPS samples.

Some topological and advanced approaches are proposed to solve the problem of geometric approaches mentioned above. Nevertheless, these approaches are not that much accurate in city areas where there are a lot of road segments near to each other.

One of the best approaches proposed so far is the use of Hidden Markov Model (HMM) to solve the map matching problem. This approach exploits the knowledge of past matched point to match the current point. This approach shows good and accurate results.

In the first place, this paper gives a state of the art of the map matching problem by describing some of the proposed approaches. It focuses on HMM map matching and provides detailed explanation on how HMM is used to solve the problem. In the second place, this paper uses some existing map matching tools with ground truth data collected by the author to see how these tools perform. The experiments intend to demonstrate the difference between tested tools by giving an overview of their performance and correctness.

The paper is structured as follow. Part I gives a state of the art for map matching problem while Part II focuses on the experiments on existing tools. In chapter 1, the problem of map matching is described by giving some definitions about the problem and the format of the data. Afterward, some of the existing approaches are described briefly. In chapter 2, we initially talk about Markov model and Hidden Markov model by giving some definitions and then we focus on how HMM can be used to solve the map matching problem. We also talk about some optimizations that can be performed on the algorithm. We complete this chapter by doing a small introduction to some tools that use HMM to do map matching. These are the tools that will be used in our experiments described in Part II. In chapter 3, we first explain the ground truth data format and modalities of our experiments and than we focus on the performance benchmarks of the tools that were tested. Finally, chapter 4 focuses on the evaluation of the obtained results from tool testings by comparing them.

Contents

Part I

State of the art

Chapter 1

The Map Matching Problem

In this chapter we will define notions about map matching and definition of the problem. We will talk about the format of the processed data, and we talk loosely about different types of approaches taken to solve the problem. For this we will be using some of the articles in the bibliography, and we will give a general image of different used algorithms.

1.1 Problem description

In [1], the problem of map matching (MM for short) is defined as the procedure that determines which road a vehicle is on utilizing data from sensors. Similarly, [2] defines map matching as the process of aligning a sequence of observed user positions with the road network on a digital map. Meanwhile [3] presents a more formal definition: "Map matching integrate positioning data with spatial road network data (roadway centrelines) to identify the correct link on which a vehicle is traveling and to determine the location of a vehicle on a link".

Map matching has been and is important because of its broad usage in navigation systems and transportation. In-vehicle navigation systems use different map matching algorithms to align the GPS data with underlying road network. Transportation industries use map matching to monitor and analyze the movement of their shipments. It is also employed in traffic flow managements. Map matching is becoming more important every day as vehicles navigation systems are used to build traffic models that can help to pick the fastest road that has less traffic jam in navigation assistant.

There are two types of map matching, online map matching and offline map matching. In online map matching, the location of an object's current position needs to be determined in real life while in offline map matching a sequence of positions are given which means all future positions are available when map matching algorithm is run [4]. In online map matching, the map matching is done with live GPS data coming as a stream and there is no knowledge of future data. The algorithm must be fast enough to handle the incoming data and map match every location. In offline map matching there is a sequence of locations and the algorithm can for example take advantage of the fact that there is a knowledge about previous locations while matching the current location.

1.2 Data format

The sensor data used in map matching is mostly GPS data because of its wide availability. Map matching algorithms use GPS data to match GPS coordinates with underlying road network. It is important to understand the GPS format and how the map is represented to be able to develop a reliable map matching algorithm.

1.2.1 GPS data format

A well-known GPS data format is GPX (GPS Exchange Format). GPX is an XML schema designed for applications and services that utilize GPS data. GPX could contain waypoints (intermediate point or place on a route or line of travel), tracks (GPS coordinates) and routes. The location data contains a longitude and latitude. It could in addition contain elevation, time, speed, heading and other kind of relative information.

In Listing 1.1 you can see a GPX sample file. We can see that we have a list of track segments and each track point contains a latitude and longitude. There is also additional information such as elevation and time. In addition of `<trk>` tag we can also have tags such as:

- `<wpt>` which contains an individual waypoint.
- `<rte>` which is a route.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

<gpx xmlns="http://www.topografix.com/GPX/1/1" xmlns:gpxx="http://
   www.garmin.com/xmlschemas/GpxExtensions/v3" xmlns:gpxtpx="http
   ://www.garmin.com/xmlschemas/TrackPointExtension/v1" creator="
   Oregon␣400t" version="1.1" xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance" xsi:schemaLocation="http://www.topografix.
   com/GPX/1/1␣http://www.topografix.com/GPX/1/1/gpx.xsd␣http://
   www.garmin.com/xmlschemas/GpxExtensions/v3␣http://www.garmin.
   com/xmlschemas/GpxExtensionsv3.xsd␣http://www.garmin.com/
   xmlschemas/TrackPointExtension/v1␣http://www.garmin.com/
   xmlschemas/TrackPointExtensionv1.xsd">
  <metadata>
    <link href="http://www.garmin.com">
      <text>Garmin International</text>
    </link>
    <time>2009-10-17T22:58:43Z</time>
  </metadata>
  <trk>
    <name>Example GPX Document</name>
    <trkseg>
      <trkpt lat="47.644548" lon="-122.326897">
        <ele>4.46</ele>
        <time>2009-10-17T18:37:26Z</time>
      </trkpt>
      <trkpt lat="47.644548" lon="-122.326897">
        <ele>4.94</ele>
```

```
        <time>2009-10-17T18:37:31Z</time>
      </trkpt>
      <trkpt lat="47.644548" lon="-122.326897">
        <ele>6.87</ele>
        <time>2009-10-17T18:37:34Z</time>
      </trkpt>
    </trkseg>
  </trk>
</gpx>
```

Listing 1.1: GPX sample file

1.2.2 Map representation

In map matching we need to know the underlying road network to be able to match GPS data to roads. This is why map servers are used in map matching algorithms. One of the leading and well known projects related to map representation is OpenStreetMap (OSM). OSM is a free and open source project that allows users to create a free editable geographic database of the world. OSM provides map representations of the world, countries and cities that contain road network and much more.

There are files that can be downloaded for each city and country in the world. Almost all map matching services use OSM for their map representation because of its availability and versatility.

1.3 State of the art approaches

Because of its important, a wide range of map matching algorithms have been developed by industries and researchers around the world. These algorithms use different techniques, and they have different performances. Some of them are more accurate than others while some of them are faster. The performance and correctness of the algorithm depend heavily on the implementation and the input given to it. It is essential that the map matching algorithm used in a service meet the specified requirement set for that particular service [3].

There are different ways of approaching the problem of matching GPS points to roads. Each of these approaches has advantages and disadvantages of their own. In [3] several ways and approaches of solving the map matching problem is detailed that we are going to briefly talk about in incoming sub sections.

1.3.1 Geometric approaches

Algorithms based on this approach use geometric information of spatial road network data by considering the shape of the links. This means that this approach does not consider how links are connected together. Three types of geometric approaches are listed in [5].

Point-to-Point matching

This type consists of matching each point P_i to the closest node or shape point of a road segment. Defining "close" is important in this approach. One of the most common and natural ways to see closest points is to use the Euclidean distance equation. For each point, we do not have to

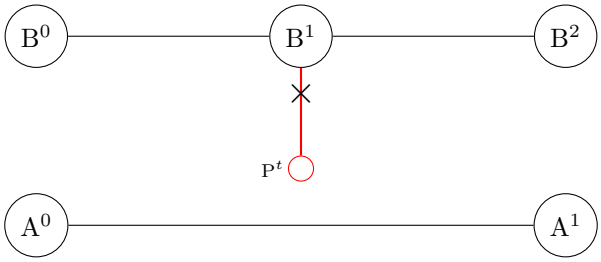


Figure 1.1: Point-to-Point map matching illustration

calculate this distance with all shape points in the network. We ca define a reasonably large radius and only calculate the distance between the point and shape points in this radius before choosing the closest one. In Figure 1.1, you can see an illustration of Point-to-Point matching. Here P^t is much closer to B^1 than either A^0 or A^1, hence it will be matched to arc B even though it is intuitively clear that it should be matched to arc A.

Although this approach is fast and easy to implement, it is not the most practical one. Its accuracy and correctness depend heavily on the way that the spatial road network was created. Arcs that have more shape points in the network are more probable to be matched with given points. Hence, this does not always produce the most accurate solution.

Point-to-Curve matching

In this approach the point P_i is matched to the closest curve in the network. The problem of defining the quantity "close" arises here again. To find the minimum distance from a point x to a curve A, one must find the minimum distance from x to each of the line segments that comprises A and select the smallest. This smallest line segment is than chosen as the match to the point. An illustration of this can be found in Figure 1.2. The green route represents the actual route taken by the vehicle and as we can see, point P^3 has been matched incorrectly with wrong road.

Clearly, this approach produces a better result than the Point-to-Point approach but it has some of its own flaws that makes it not so much usable in practice. This approach does not return a good result in areas where the road density is high (e.g. urban area). The reason is that in these areas, the closest link is not consistently the right choice and noises in data could even cause more erroneous results. Another problem is that this approach does not make use of historical information. Meaning that it does not use any knowledge about last matched points and because of this, it can make the wrong choices when matching. This is, of course, a more general problem in geometric approaches.

Curve-to-Curve matching

Lastly, a better way is to consider m points together and match them to the arc that is closest to the piecewise-linear curve built by these m points. This means that we compare vehicle's trajectory against roads.

This requires the ability to measure the distance between two curves. To achieve this, one first

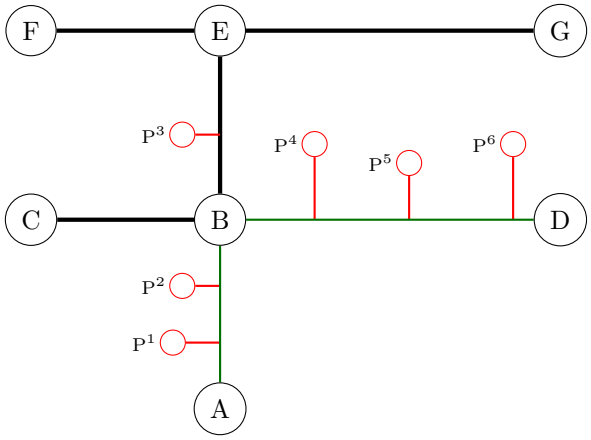


Figure 1.2: Point-to-Curve map matching illustration

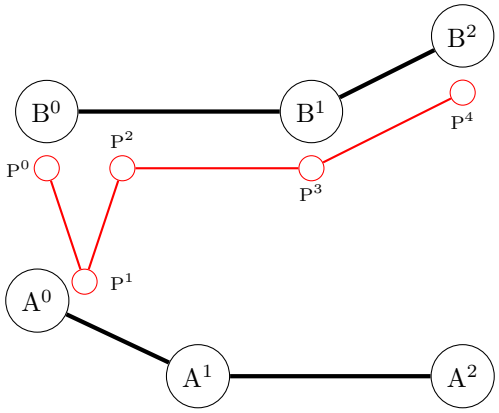


Figure 1.3: Curve-to-Curve map matching illustration

creates a curve from m points and afterward determines the distance between the created arc with other road arcs in the network. Lastly, we chose the closest arc as the matched trajectory. An illustration of this can be found in Figure 1.3. This approach depends heavily on the way that the ordering of the points is done in the curves as it uses Point-to-Point approach to calculate the distance between these points. Consequently, this approach produces sometimes unexpected results.

1.3.2 Topological approaches

In this approach, we try to exploit the relationship between entities such as points, lines and polygons. A map matching algorithm that makes use of the geometry of the links as well as the connectivity and contiguity of the links is known as a topological map matching approach.

Weighted topological algorithms can be implemented to solve the map matching problem [6]. This is based on the topological analysis of the road network and using only longitude and latitude information of data position points. This algorithm does not consider any heading, speed or elevation information. This algorithm is incredibly sensitive to the outliers. One must be cautious about the outliers while implementing this algorithm.

Another type of algorithm makes use of the correlation between the trajectory of the vehicle and the topological features of the road such as road turn, road curvature and road connection [7]. In this algorithm, one must first define a number of thresholds by analyzing the statistical information from field data. These thresholds then are used to eliminate some candidate road segments when matching points. This algorithm experiences some difficulties at junctions where the bearing of the roads is not similar. To correct this problem, the algorithm uses post-processing, but this makes it unsuitable for online map matching.

Lastly, [8] proposes an enhanced topological algorithm which is based in similarity criteria between the road network geometry and derived navigation data. To improve the performance, other criteria and parameters such as speed of the vehicle, the position of the vehicle relative to candidate links, and heading information are taken into consideration. The best matching procedure is than applied with regard to the various weighting factors defined in the algorithm.

1.3.3 Probabilistic approaches

Probabilistic approaches are defined in [3] as "The algorithm that requires the definition of an elliptical or rectangular confidence region around a position fix obtained from a navigation sensor". In the enhanced probabilistic algorithm developed in [9], the elliptical error region is constructed when the vehicle travels through a junction. This is a reliable method since the construction of an error region in each epoch could lead to erroneous results as there might be incorrect detected link that are close enough to the link on which the vehicle is traveling. This algorithm additionally contains some criteria that helps to improve the matching process. For example in this algorithm, the inaccuracy of the heading information of the data in low speeds is taken into consideration. This algorithm takes into account various errors from GPS data and underlying road network.

1.3.4 Advanced approaches

These approaches consist of types of approaches that use more refined concepts such as Markov Models and Hidden Markov Models (HMM) (used in [1]) that we are going to discuss in details

in chapter 2. For now let us briefly discuss some other advanced approaches. As there are many advanced approaches, we are going to list only some of them that seem more interesting to us. For interested readers, more advanced approaches could be found in [10, 11, 12, 3, 13, 14, 15].

Some approaches combine multiple approaches together in order to get better results. For example, [16] uses first a Point-to-Curve approach to identify correct links. Afterward, an orthogonal projection of the position fix onto the link is exploited to match the given points to the road network. This reduces what is called a cross-track error which represent the error across the width of the road. Meanwhile, to reduce the along-track error, an Extended Kalman Filter (EKF) is used. The performance of this filter depends heavily on the representation of curves in the road network and spatial information. This method suffers from the same problem as Point-to-Curve approach which is the high dense urban areas with a lot of close curves. If the data calculated at first step is incorrect, other steps will also result in incorrect results.

To solve the problem of urban canyons (where GPS signals are blocked by tall buildings and trees), [17] proposes a solution that consists of approximately modeling the path of the vehicle as pieces of the curves such as straight lines, arcs and polynomials. This method constraint the path of a vehicle to a known segment of road when the vehicle enters a known hard to match area. Basing on this constraint (which is a big one, nonetheless), only two GPS satellites are required to obtain the positioning information. Their method implements a probabilistic algorithm integrated with EKF to estimate the location of the vehicle at junctions and identify the correct match. Further testing of this algorithm is needed to evaluate its performance and correctness. The algorithm fails where one or more than two stellites are available (i.e. there should be only two satellites).

To overcome the present problems of map matching mentioned above, [8] develops a fuzzy login algorithm that uses some new number of inputs: 1. speed of the vehicle, 2. connectivity among road links, 3. quality of position solution, and 4. position of a fix relative to candidate link. These inputs are incorporated into the algorithm and there are rules that are applied based on the incoming data.

Chapter 2

Hidden Markov Model (HMM) Map Matching

In this chapter, we will talk about Hidden Markov Model and how it is employed to solve map matching problem. Initially, we will see some definitions and then we will see how map matching is done with HMM. Before seeing some comparisons between different HMM map matching algorithms, we will see how HMM process could be optimized.

2.1 Definitions

In this section we will define some notions because it is important to know them before going any further. In order to define HMM, one has to first know what is a Markov model. Therefore, in the first place we will define Markov model and then we will define Hidden Markov Model.

2.1.1 Markov Model

A Markov model (chain) is a mathematical system. This system has a set of states and a set of transitions between these states. There is a transition from a state to another based on some probability. In Figure 2.1, an example of a Markov model can be found that represents some kind of a system that has two critical sections. We can see for example that the probability of going to state `CS1` from state `wait` is 0.2 while the probability of going to state `wait` from state `CS1` is 0.4 .

A Markov model is specified by following components [18]:

- $Q = q_1, q_2, \ldots, q_N$: a set of N states;
- $A = a_{11}, a_{12}, \ldots, a_{n1}, \ldots, a_{nn}$: a transition probability matrix A where each a_{ij} represents the probability of moving from state i to state j such that $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$. This indicates that each row of the matrix is a probability vector and thus, the sum of each row must be equal to 1;
- $\pi = \pi_1, \pi_2, \ldots, \pi_n$: an initial probability distribution over states. π_i is the probability that the Markov model will start in state i. Some states j may have $\pi_j = 0$, meaning that they can not be initial states. Also, $\sum_{i=1}^{N} \pi_i = 1$.

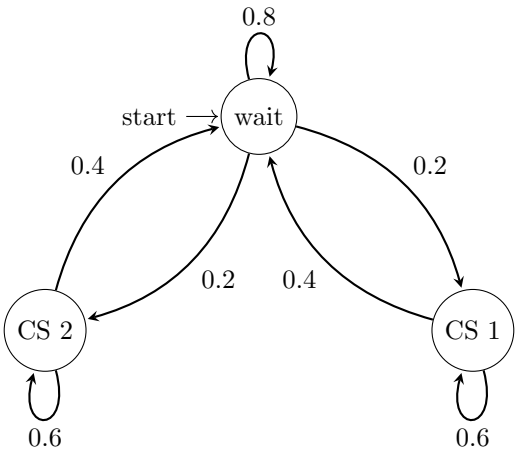


Figure 2.1: Example of Markov model

Let us now define the notion of Markov property which is fundamentally important in Markov model and defines the strong fundamentals of this model. A stochastic process satisfies the Markov property if the conditional probability distribution of future states of the process (conditional on both past and present states) depends only upon the present state, not on the sequence of events that preceded it [19]. Meaning that the probability of going to any particular state in the future depends only on current state and not on the sequence of states that leads to it. In other words, the probability of a particular state depends only on the previous state. In a more formal way, we have the Markov assumption as

$$\mathbb{P}(q_i = a \mid q_1 \dots q_{i-1}) = \mathbb{P}(q_i = a \mid q_{i-1}) \tag{2.1}$$

This implies that Markov model is memoryless and this is why Markov model differs from a general stochastic process.

2.1.2 Hidden Markov Model

We use Markov chains when we want to compute the probability for a sequence of observable events. This is, of course, not useful, when the events that we require to study are hidden in a way that we can not observe them directly. This is where Hidden Markov Model comes into the play. Hidden Markov Model allows us to model both observed events and hidden events into our probabilistic model. An HMM is specified by the following components [18]:

- $Q = q_1, q_2, \dots, q_N$: a set of N states;
- $A = a_{11}, a_{12}, \dots, a_{n1}, \dots, a_{nn}$: a transition probability matrix A where each a_{ij} represents the probability of moving from state i to state j such that $\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i$. This indicates that each row of the matrix is a probability vector and thus, the sum of each row must be equal to 1;
- $O = o_1, o_2, \dots, o_T$: a sequence of T observations, each one drawn from a vocabulary $V = v_1, v_2, \dots, v_V$;

- $B = b_i(o_t)$: a sequence of observation likelihoods, also called emission probabilities, each expressing the probability of an observation o_t being generated from a state i;
- $\pi = \pi_1, \pi_2, \ldots, \pi_n$: an initial probability distribution over states. π_i is the probability that the Markov model will start in state i. Some states j may have $\pi_j = 0$, meaning that they can not be initial states. Also, $\sum_{i=1}^{N} \pi_i = 1$.

In addition to the Markov assumption (Equation 2.1), we additionally have the output independence which is

$$\mathbb{P}(o_i \mid q_1 \ldots q_i, q_T, o_1, \ldots, o_i, \ldots, o_T) = \mathbb{P}(o_i, q_i) \tag{2.2}$$

Equation 2.2 states that the probability of an output observation o_i depends only on the state that produced the observation q_i and not on any other state or any other observation.

To illustrate how HMM works by an example, we reiterate the example used in [18]. We are tasked to study the history of global warming in a region, but the problem is that we are missing some necessary weather data about the region in a specific period. However, we have access to the diary of a person that contains how many ice creams this person has consumed every day in the period that our weather data is missing. We can use these observations in order to estimate the daily temperature in that period. To simplify, we only assume that there are two types of days: cold (C) and hot (H). Concretely, the task (which is also called Eisner task) is to find the hidden sequence Q of weather states which caused the person to eat ice cream given the sequence of observations O which is the number of ice creams that the person had in a given day. In Figure 2.2 you can see a simple HMM that represents this problem. Two hidden states are `HOT` and `COLD` and observations (which is on alphabet $O = \{1, 2, 3\}$) correspond to the number of ice creams eaten in a given day. For example the probability of eating one ice cream in a cold day is 0.5 while the probability of eating three ice creams in a hot day is 0.4.

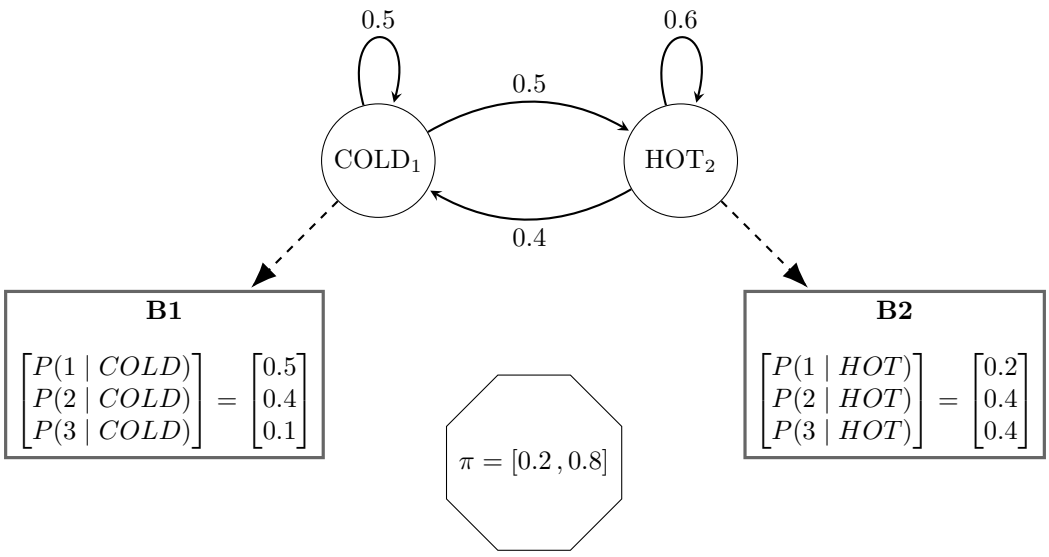


Figure 2.2: Example for Hidden Markov Model

Hidden Markov Model is fundamentally characterized by three problems [20]:

- **Likelihood:** given an HMM $\lambda = (A, B)$ and an observation sequence O, determine the likelihood $P(O \mid \lambda)$.
- **Decoding:** given an observation sequence O and an HMM $\lambda = (A, B)$, discover the best hidden state sequence Q.
- **Learning:** given an observation sequence O and the set of states in the HMM, learn the HMM parameters A and B.

2.2 Map Matching with HMM

In this section, we will see how Hidden Markov Model is used to solve the problem of map matching. We will use [1] as our main reference since this is the paper that is most used in real world implementations and delivers a really clear and good explanation of the solution. This paper emphasizes on maintaining a principled approach to the problem while simultaneously making the algorithm robust to the location data that is both geometrically noisy and temporally sparse. It is worth mentioning that this paper, solves the map matching problem as a batch problem. Meaning that it works for offline map matching. However, authors speculate that a sliding window version of their algorithm would work properly for online map matching.

In chapter 1 and in section 1.3, we talked about some approaches that have been taken to solve the map matching problem. We also saw that each of them had some problems when the data were noisy or not that much accurate. One of the problems that most of these approaches had in common was the absence of knowledge about past matched measurements. HMM map matching solves this problem by considering the previous matched measurement in the process of matching the current measurement. As said in chapter 1, the key problem in map matching is the tradeoff between the roads suggested by the location data and the feasibility of the path. Simply matching every point to the nearest road could result on strange paths and bizarre driving behavior. This is why we introduce the knowledge of the connectivity of the road network to avoid having bizarre behaviors. The HMM offers an effective solution to integrate noisy data and path constraints in a principled and elegant way.

In the algorithm proposed by [1], the states of the HMM are the individual road segments and the state measurements are the noisy vehicle location measurements. The goal is to match each location measurement with the proper road segment. More formally, the discrete states of the HMM are the N_r road segments, r_i, $i = 1 \ \ldots \ N_r$. In the representation, distinct road segments run between intersections. For each location measurement (latitude and longitude) z_t, the goal is to find the road segment that the vehicle was actually on. In Figure 2.3 an illustration of the HMM for map matching can be found. Each vertical slice represents a point in time corresponding to a location measurement z_t for three times $t = 1, 2, 3$. At $t = 1$ there are three roads near z_1 (shown as three black dots in the first column). There are three possible paths from each of the nearest points of these three roads to the two roads near z_2 at t_2 and similarly for t_3. The goal of the algorithm is to find the most probable path through the lattice by picking one road segment for each t. Let us now dive in to see how different probabilities are measured in this algorithm and how the optimal path is chosen with regard to these measurements.

Emission probabilities

In subsection 2.1.2, we defined emission probabilities as the likelihood that a measurement resulted from a given state, based on that measurement alone. Given a location measurement z_t,

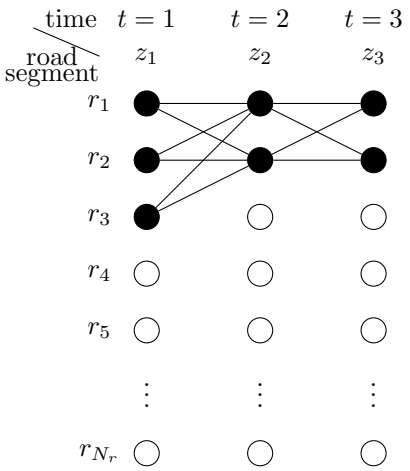


Figure 2.3: HMM illustration for map matching

there is an emission probability for each road segment r_i which is $p(z_t \mid r_i)$. This is the likelihood of observing measurement z_t if it was actually on road segment r_i. For a given z_t and r_i we denote the closest point on the road segment by $x_{t,i}$. The intuition is that road segments farther from the measurement are less likely to have produced the measurement.

In Figure 2.4, an example of this notation can be found. There are three road segments (r_1, r_2 and r_3) and two measured points (z_t and z_{t+1}). The first measured point (z_t) has $X_{t,1}$ and $X_{t,3}$ as candidate road segments. Each match candidate results in a route to $x_{t+1,2}$ which is a match candidate for the second measured point (z_{t+1}). These two roads have their own length as does the great circle path between the two measured points. Based on results from [1], they conclude that the route distance and great circle distance are closer together for correct matches than for incorrect matches. In Equation 2.3, you find the formula to calculate the emission probability. σ_z is the standard deviation of GPS measurements which is estimated as $\sigma_z = 1.4826 \text{ median}_t \left(\|z_t - x_{t,i^*}\|_{\text{great circle}}\right)$ in [1].

$$p(z_t \mid r_i) = \frac{1}{\sqrt{2\pi}\sigma_z} e^{0.5(\frac{\|z_t - x_{t,i}\|_{\text{great circle}}}{\sigma_z})^2} \tag{2.3}$$

To start the algorithm, we also need to know the initial state probabilities π_i, $i = 1 \ldots N_r$. In map matching, this indicates the probability of the vehicle's first road over all the roads at the beginning of the drive. Some algorithms assign a uniform distribution to π_i but in [1], the algorithm uses the first measurement z_1, *i.e.* $\pi_i = p(z_1 \mid r_i)$.

Transition probabilities

Transition probabilities indicate the probability of a vehicle moving between the candidate road matches of z_t and z_{t+1}. Intuitively, we favor transitions whose driving distance is close to the great circle distance between the measurements. We know that for measurement z_t and candidate road segment r_i, the coordinate (longitude and latitude) point on the road segment nearest the measurement is $x_{t,i}$. For he next measurement z_{t+1} and candidate road segment r_j, the corresponding point is $x_{t+1,j}$. We compute the driving distance between these two points

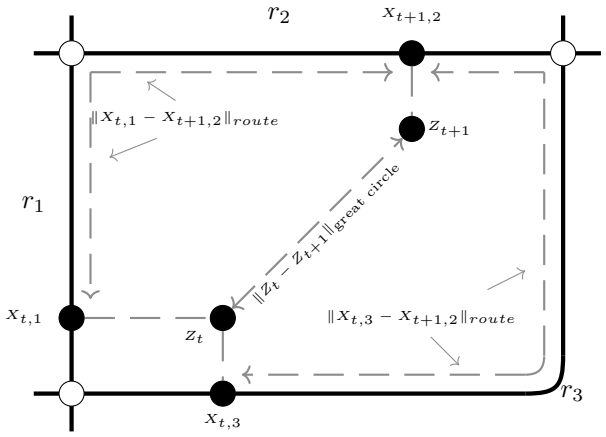


Figure 2.4: Emission probabilities notation illustration

using a route planner which is configured to give the shortest route distance possible. This driving distance is referred as *"route distance"* which is noted as $\|x_{t,i} - x_{t+1,j}\|_{\text{route}}$. In [1], the comparison between route distance and measured points (great circle) shows that the histogram of $|\ \|z_t - z_{t+1}\|_{\text{great circle}} - \|x_{t,i} - x_{t+1,j}\|_{\text{route}}\ |$ follows an exponential probability distribution. Authors suspect (after conducting experiments) that these two distances will be about the same for correct matches. The reason is that the relatively short distance traveled on the roads between a pair of correct matches will be about the same as the distance between the measured GPS points. This result was obtained by computing these distances on some ground truth data that were collected by the authors. The exponential probability distribution is given in Equation 2.4. The term d_t is detailed in Equation 2.5. The value of β is estimated as $\beta = \frac{1}{\ln(2)}\text{median}_t(|\|z_t - z_{t+1}\|_{\text{great circle}} - \|x_{t,i} - x_{t+1,j}\|_{\text{route}}|)$ in [1].

$$p(d_t) = \frac{1}{\beta} e^{-d_t/\beta} \tag{2.4}$$

$$d_t = |\|z_t - z_{t+1}\|_{\text{great circle}} - \|x_{t,i} - x_{t+1,j}\|_{\text{route}}| \tag{2.5}$$

Optimal path

Viterbi algorithm [21] is used to compute the best path through the HMM lattice with emission probabilities obtained from Equation 2.3 and transition probabilities obtained from Equation 2.4. The Viterbi algorithm uses dynamic programming to quickly discover the path through the lattice that maximizes the product of the emission and transition probabilities. The Viterbi algorithm takes as input

- the observation space $O = \{o_1, o_2, \ldots, o_N\}$.
- the state space $S = \{s_1, s_2, \ldots, s_K\}$.
- initial probabilities π_i with $i = 1, 2, \ldots, N$.
- a sequence of observations $Y = (y_1, y_2, \ldots, y_T)$ such that $y_t = o_i$ if the observation at time t is o_i.

- transition probabilities matrix A of size $K \times K$ such that A_{ij} stores the transition probability of transitioning from state s_i to the state s_j.
- emission probabilities matrix B of size $K \times N$ such that B_{ij} stores the probability of observing o_j from state s_i.

and produces as output the most likely hidden state sequence $X = (x_1, x_2, \ldots, x_T)$ which in map matching is matched points to road segments. In algorithm 2.1 you can find the pseudocode of the Viterbi algorithm.

Algorithm 2.1 Viterbi algorithm

```
function VITERBI(O, S, Π, Y, A, B)
    for each state i = 1, 2, ..., K do
     T_1[i, 1] ← π_i.B_{iy_1}
     T_2[i, 1] ← 0
    end for
    for each observation j = 2, 3, ..., T do
        for each state i = 1, 2, ..., K do
          T_1[i, j] ← max_k(T_1[k, j − 1].A_{ki}.B_{iy_j})
          T_2[i, j] ← arg max_k(T_1[k, j − 1].A_{ki}.B_{iy_j})
        end for
    end for
     z_T ← arg max_k(T_1[k, T])
     x_T ← s_{z_T}

    for j = T, T − 1, ..., 2 do
     z_{j−1} ← T_2[z_j, j]
     x_{j−1} ← s_{z_{j−1}}
    end for
     return X
end function
```

2.3 HMM Map Matching optimizations

In section 2.2, we saw how HMM could be used to solve the map matching problem. The described formulation represents a principled approach to balance the effects of measurement noise and route behavior. Nonetheless, the algorithm could be improved to be better and faster in practice. In this section we will talk about some aspects that we can take advantage of, in order to improve the HMM map matching algorithm.

2.3.1 Preprocessing

Before the GPS points are used to construct the HMM, we parse all points through the time sequence and we remove points that are within $2\sigma_z$ of the previous included point. The reasoning behind this preprocessing is that until we see a point that is at least $2\sigma_z$ away from its temporal predecessor, our confidence is low that the apparent movement is due to actual vehicle movement and not noise. This reduces the number of steps in the HMM for high sample rate data. For

example, in [1], they have estimated that this preprocessing speeds up the processing phase up to 38.9% on their truth ground data compare to the result without this preprocessing phase.

2.3.2 HMM breaks and proposed solutions

A break happens in an HMM lattice when all transition probabilities from one time step to the next one are zero. Let us now detail the conditions that are causing breaks in HMM map matching:

- **Route localization:** in a pure implementation of the algorithm, every road in the road network would be considered as a potential match candidate and a complete route would be calculated between every one of these match candidates. It is not practical to match measurement to road segments that are too far from the measurement location. This is why, HMM map matching algorithms set to zero any measurement probability from a road segment that is more than a constant meter away from z_t. This constant should be defined in the algorithm. This helps to reduce the number of candidate matches that the algorithm needs to consider for each z_t. This is represented in Figure 2.3 with empty circles representing road segments that are too far away to consider. With this, it is possible to have GPS points in the data that have no match candidates within the defined radius which will cause the implementation to have no match candidate for a particular time step in the HMM. This situation can arise, for example, when the vehicle enters tunnels or city canyons.

- **Low probability routes:** on a connected road network, it is possible to discover a route between any two road segments. However, once the route distance ($\|x_{t,i} - x_{t+1,j}\|_{\text{route}}$) becomes considerably larger than the great circle distance ($\|z_t - z_{t+1}\|_{\text{great circle}}$), the transition probability ($p(d_t)$) corresponding to that route becomes extremely small. This could happen when the route becomes circuitous and strange. Instead of continuing to search for a strange road that may not exist, we can terminate the search for a route when $\|x_{t,i} - x_{t+1,j}\|_{\text{route}}$ becomes greater than $\|z_t - z_{t+1}\|_{\text{great circle}}$ by a given constant and assign a probability of zero. This constant should be defined in the algorithm and depends on the environment and ground truth data.

- **GPS outliers:** if we know the origin of the GPS data and the underlying road network, we can exclude GPS data that do not respect any constraints and consider the route to be unreasonable and set its probability to zero. For example, if a calculated route would require the vehicle to exceed a speed of 180 km/h in a city area, we can consider this route to be unreasonable and put its probability to zero.

In a pure implementation of the HMM, the Viterbi algorithm will always be able to discover a complete optimal path through the HMM lattice. However, the above simplifications can limit the number of match candidates and can eventually cause an incomplete path through the HMM. This causes to have long stretches of easy to match locations in the HMM and interceptions of short stretches of locations that break the HMM. One way to solve this is to manually remove points that are causing the breaks. Nevertheless, this solution is not that practical. Another solution is to automate the process of removing the points that break the HMM. In [1], the following way is used to automate this step. When a break is detected between time step t and time step $t+1$, the measured points z_t and z_{t+1} are removed from the model and a check is done to see if the break has been healed. The break is considered healed if the measured points at $t-1$ and $t+2$ lead to a reconnection in the HMM after rechecking the mentioned points above. If the breaks are still present, we continue to remove points on either sides until the breaks are healed

or they have last more than a defined constant second (in [1], this constant is 180 seconds). If the break exceeds this threshold, we split the data into separate trips and do map matching on each one of them separately.

2.4 State of the art HMM Map Matching tools

Now that we have seen how map matching is done using HMM, it is time to talk about existing implementations and tools. There are a lot of tools and services that do map matching using HMM. In this section we focus on most well-known tools, and we introduce them briefly. In Part II we use these tools in order to assess them and match some ground truth data.

Barefoot

Barefoot is an open source `Java` library for both offline and online map matching. It uses OpenStreetMap (OSM) as its map. Barefoot consists of two core parts:

- **Map server:** this is a Docker-base map server that provides access to street map data from OpenStreetMap. This server is flexible to be used anywhere (e.g., a distributed cloud infrastructure) as map data server or used side-by-side with Barefoot's matcher servers.
- **Software library (and matcher servers):** Barefoot library can be used to do map matching (online and offline) and spatial data analysis on the map. If one does not wish to use Barefoot as a `Java` library, they can use matcher servers. There are two types of stand-alone matcher servers in Barefoot:
  - **Offline matcher server:** this is a stand-alone server for offline map matching. The server receives a JSON array of JSON objects containing GPS coordinates and time and after matching the points, it sends back a response containing matched points. The response can be either in GeoJSON format or in SlimJSON format. There is, moreover, the possibility of a debug format that outputs useful information such as timestamps of the measurements and geometries of the routes in WKT format.
  - **Online tracker server:** this is a stand-alone server for online map matching. The server receives position updates periodically and matches them to the underlying road network. The server matches each position right away and maintains track of objects' movement on the map in real-time.

GraphHopper

GraphHopper is a fast and memory-efficient routing engine that also supports HMM map matching. It is written in `Java` and it is licensed under Apache License 2.0. It can be used as a `Java` library or stand-alone web server. This routing engine could calculate the distance, time, turn-by-turn instructions and many road attributes for a route between two or more points. It also supports map matching (or snap to road), isochrone calculation (calculate and visualize the reachable area for a certain travel mode), mobile navigation and more. Like Barefoot, GraphHopper also uses OpenStreetMap as default road network representation. It is possible to use other maps but a custom import procedure is necessary in that case. In contrary to Barefoot, GraphHopper does not use any map server to store road data. It works directly with OpenStreetMap data file (`osm.pbf`) and does not need any kind of database or Docker container for its map component.

Valhalla

Valhalla is an open source routing engine and accompanying libraries to use with OpenStreetMap data. Valhalla additionally includes tools like time+distance matrix computation, isochrones, elevation sampling, map matching and tour optimization. It is written in `C++` and it has an HTTP server-client implementation. The tool is partitioned into smaller APIs that are designed to do specific tasks such as map matching, routing graph, tile generation, routing algorithms, etc. Each API can be used via an endpoint by sending HTTP requests. In Valhalla, the map is represented as a graph. Graph tiles can be downloaded for use by client-side routing applications or by hosted services that don't want to undergo the pain of data creation. A structured graph hierarchy (e.g., highways, arterials, local, transit) along with shortcut edges will ensure high performance. There is also an online demo server that can be used to test out the tool. This does not require any installation.

FMM

Fmm (Fast Map Matching) is an open source map matching framework written in `C++` and `Python`. It considers maximizing performance, scalability and functionality. Like other tools, FMM uses OpenStreetMap as its map. It is equally possible to use ESRI shapefile as map. The tool takes configuration in `XML` format and GPS data in comma separated `CSV` format where each row stores a trajectory with geometry in WKT linestring format.

Part II

Evaluation, Observation and Benchmarking of Map Matching Tools

Chapter 3

Tool testing and benchmark

At the end of chapter 2, we listed some existing tools that use HMM in order to solve the problem of map matching. In this chapter we will talk about how we tested these tools and how they perform with regard to execution time in different phases such as road network construction and map matching process. We will also talk about the data with which we tested these tools.

3.1 Ground truth data

In order to test map matching tools, we need some data. The data must consist of GPS measurements (containing at least, longitude and latitude) that are gathered from trips taken by vehicles. The ground truth data used to test out map matching tools were gathered by a vehicle driving in Brussels. The GPS measurements were conducted by an application on a smartphone that employs the GPS tracker of the phone to gather GPS measurements. A total of 14 trips were gathered with various numbers of GPS points to see how algorithms perform with dense and sparse data. Trips contain between 20 to 350 GPS points based on how parameters were define to gather GPS points. Most of these trips last from 25 to 40 minutes for a distance of 8 to 15 kilometers in a city area with speed limits of 30 and 50 km/h.

The advantage of gathering the ground truth data instead of utilizing some available data from some other source is that we can clearly observe anomalies and problems when matching points because of having the domain knowledge of data, trips and underlying network. This helped us to better observe the result of matched points.

3.2 Tool utilization and modality of the benchmarks

From tools mentioned in section 2.4, all were tested except FMM. In order to have more flexibility when using each tool, we decided to use them as stand-alone libraries. To do so, some code needed to be written to

1. read the GPS data from `GPX` file and if needed, transform it into the representation that is needed for the library
2. create the underlying road network from OSM file if needed
3. call the map matching process of the tool on the data

4. output the matched points in either GeoJson format or in GPX format (so we can observe them using sites such as geojson.io)

By using these tools as stand-alone libraries, we have the advantage of having more flexibility on inputs and outputs. We could also time the execution time of the necessary parts of the algorithm and avoid timing unnecessary parts such as reading and storing GPS data.

Being written in Java, it was very easy to use Barefoot and GraphHopper as stand-alone libraries. We only needed to use Java JAR executables and import them into an Java project. It should be noted that Valhalla was not tested as stand-alone library. The HTTP client was used to test the tool. The reason is that Valhalla is not only a map matching tool. It is a routing engine with a lot of interconnected components, and it is written in C++. Because of time restrictions, we did not manage to prepare the necessary work to use this engine as a library. A Python script was written that sends an HTTP request to Valhalla server with GPS measurements and returns the response in GeoJson format.

These tools were benchmarked and tested on a system consisting of an Intel Core i7 - 8th Gen @ 4.6 GHz CPU with 16 Gb of RAM memory running on Arch Linux with kernel version 5.17.6.

3.3 Benchmarks

In this section, we will talk about how each tool performs with regard to execution time. We will first focus on how each tool generates the map from OSM file and then we will talk about the execution time performance of each tool while matching points. It should be noted that this section does not intend to show which tool is better than the other by comparing their speed. The idea is to demonstrate the difference between these tools in various aspects so that the reader could better understand each tool.

3.3.1 Road network generation

As we discussed earlier, each of these tools has their own way of using OpenStreetMap to create the needed road network for the map matching process. Let us now discuss how each tool does this process of road generation.

Barefoot

Barefoot uses an interesting approach to create the road network. It has a Docker-based map server which in addition can be used as a stand-alone server in a distributed cloud infrastructure as a map data server. This server uses OpenStreetMap and PostgreSQL in order to store necessary information representing the road network. This approach makes it possible to use this map server in other projects and it is not only limited to the Barefoot environment.

The process of road network creation from OSM file is done only once and then the map can be used by consulting the propitiate database and database tables. For map matching, Barefoot needs a property file in which necessary information (such as the name of the database containing road network) is provided in order to connect to map server. The process consists of giving the OSM file to Docker container and launching the necessary script. The process of road network creation takes considerably more time with regard to the approach taken by other tools such

as Valhalla and GraphHopper. It is worth mentioning that if the user does not wish to use the Docker container, a `PostgreSQL` database could be used as map server database directly. In that case, it is up to the user to create the road network and insert it into the database. Barefoot provides necessary instructions to put in place such a database. This approach, makes Barefoot one of the more flexible tools when it comes to map representation. If one wishes to do map matching and has already an infrastructure with road network, Barefoot might remain a good choice because there is no need to change the existing infrastructure.

Valhalla

In Valhalla, the route network is generated from OSM files with the provided executable. There is no database use or map server. The necessary information for road network is simply placed in some files and the tool uses these files. The road network generation uses multithreading to produce necessary information and is much more faster than Barefoot. The sole problem being the fact that this approach does not offer the same level of flexibility as Barefoot. Like Barefoot, this process is done only once and then the generated files are used for routing engine algorithms in the tool. A property file should be provided to routing engine server that provides necessary information about the location of the route network files.

GraphHopper

With GraphHopper, the road network generation is really fast compare to those of Valhalla and Barefoot. The tool does not use any database to insert the road network. It creates routable files for GraphHopper which are placed in a folder, and it also creates data for a special routing algorithm to dramatically improve query speed. You can see these files as a kind of cache system. If these files already exist, this step will be skipped. The OSM file is consulted directly when needed and there is no need to create the whole road network. This is done via `OSMReader`. Hence, the server should always be launched by giving it the path to the OSM file. Like Valhalla, this does not offer the same level of flexibility as Barefoot but, this makes GraphHopper a more portable tool. With Barefoot, every time that we want to migrate from one machine to another, the map server (or database instance itself) and matching server need to be migrated and all necessary property files should be placed in right places while with GraphHopper, all you need is the `JAR` executable and OSM file.

3.3.2 Map matching execution time

As mentioned previously, Barefoot and GraphHopper were tested as libraries in this paper. This gave us the opportunity to test the execution time of only map matching process for each tool without considering all the preprocessing (e.g. reading the input) and postprocessing (e.g. producing the GeoJson output). Regarding Valhalla, although this tool was not tested as a stand-alone library, it is however interesting to benchmark the `HTTP` request and response time of the server. Therefore basically, for Valhalla we measured the time between sending the request and receiving the response. In Table 3.1 and Figure 3.1 the result of this benchmark can be found. The map matching algorithm in each tool has been run 1000 times and then the average of the time spent on each round has been taken into consideration as average execution time. Times are in milliseconds. Benchmarks were done with two sets of GPS points. One with 302 GPS points and the other with 22 points. We wanted to see how these tools differ with regard to different sizes of GPS points.

# GPS points \ Tool	Barefoot	GraphHopper	Valhalla
302 points	1825 ms	556 ms	81 ms
22 points	796 ms	94 ms	72 ms

Table 3.1: Execution time of HMM map matching tools

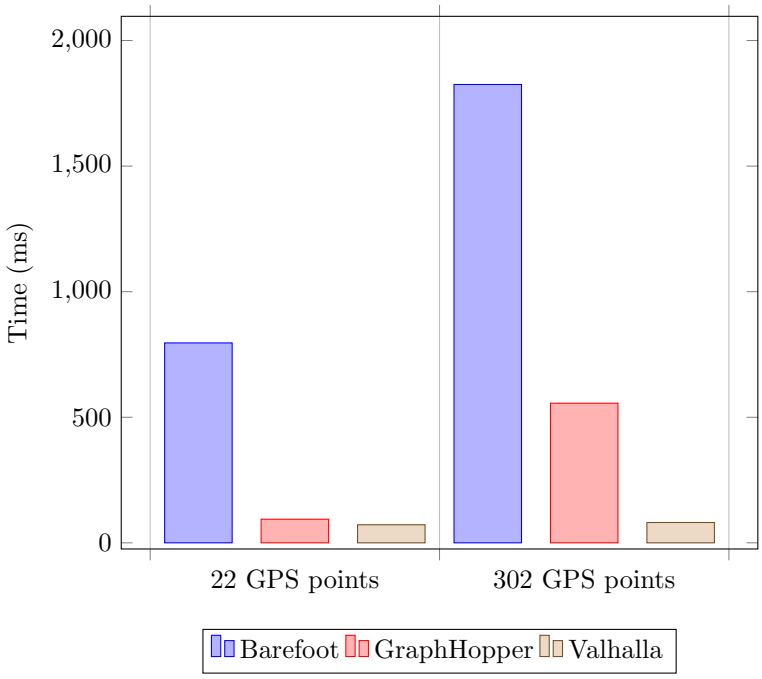


Figure 3.1: Bar chart of execution time of HMM map matching tools

As we can see, Valhalla (even though it was not tested as a library) outperforms GraphHopper and Barefoot on execution time. It should be noted that the output of Valhalla is not a linestring (it is just the matched point to the road of each GPS measurement) while the outputs of Barefoot and GraphHopper are both linestring. The process of creating the linestring does also take some time. Barefoot is 106.6 % slower than GraphHopper and 183 % slower than Valhalla when it comes to matching big number of GPS measurements. For small number of GPS measurements, Barefoot is 157.7 % slower than GraphHopper and 166.8 % slower than Valhalla. GraphHopper is 166.4 % slower than Valhalla for big number of GPS measurements and 26.5 % slower when matching small number of GPS measurements. It is interesting to see that the time difference between matching small number of measurements and big number of measurements in Barefoot is 78.5 % while in GraphHopper, this difference is 142.1 %. For Valhalla, this difference is not that much noticeable (near 11.7 %). It is worth mentioning that the formula used to calculate these percentages is "percentage difference formula" presented in Equation 3.1.

$$\% \text{ difference} = 100 \times \frac{|A - B|}{\frac{A+B}{2}} \tag{3.1}$$

Chapter 4

Results Evaluation

In this chapter we discuss about results obtained from matching GPS measurements using map matching tools. We will be focusing on observations that we have performed in our experiments utilizing these tools. To be specific, we discuss about accuracy of the results with regard to dense ans sparse data. We will also mention some phenomena that we observed while using map matching tools.

4.1 Modality of tests and experiments

We have used our ground truth data with these tools, and we have run them on multiple sets of GPS measurement. Using ground truth data made it possible for us to better observe the results. The domain knowledge of the road network gave us the opportunity to better observe inaccuracy in the results. Each of the tools was tested with the same set of GPS measurements and the same OSM map. By doing so, we ensured that the data and map stayed a consistent factor across our testings and experiments.

It is also worth mentioning that some of these tools are no longer maintained and supported, and this could influence the results. For example, Barefoot discontinued the development and maintenance nearly 5 years ago while GraphHopper nad Valhalla are still maintained and continue to improve their algorithms and tools.

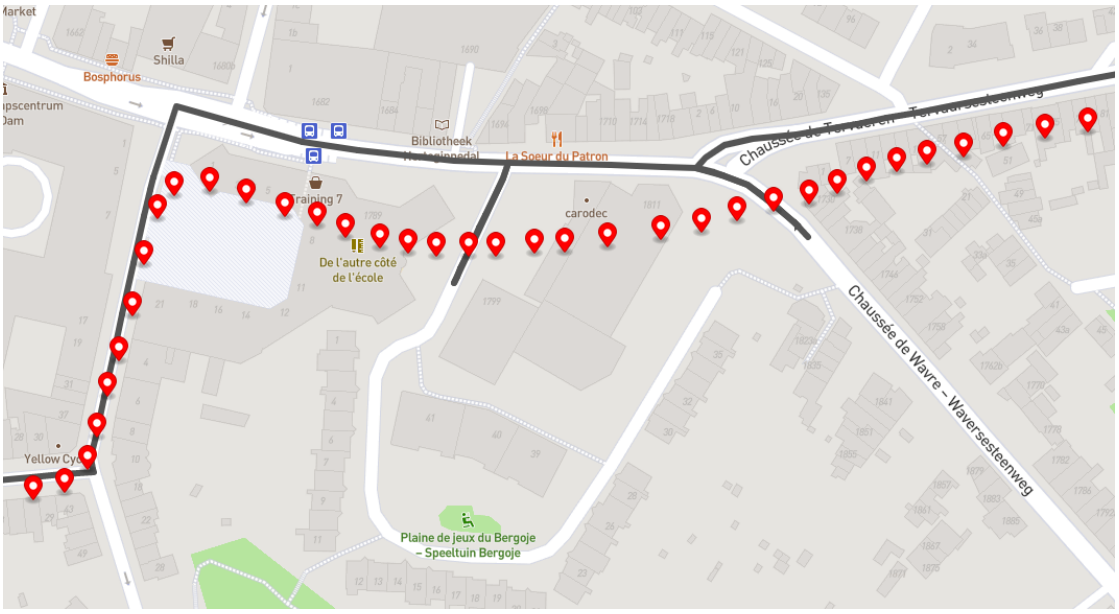
4.2 Accuracy and correctness comparison

In this section, we will show some inaccuracies that we have found in some of the tools while matching GPS measurements. We will also compare the result of these three tools for some of our trips. Some trips contain dense GPS measurements, meaning that the time delay between collecting each GPS point is really small (1 or 2 seconds), and others contain sparse GPS measurements, meaning that the time delay between collecting GPS point is 1 or 2 minutes. Like so, we could observe how each tool perform while working with less or more GPS data.

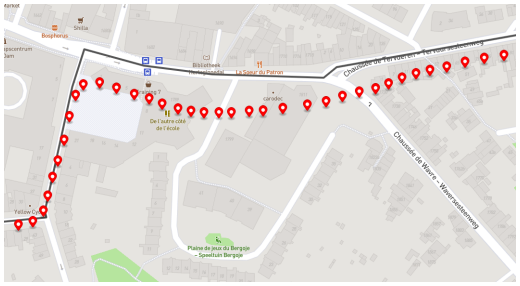
4.2.1 Dense GPS measurements

In Figure 4.1 an excerpt of the results of matching dense GPS points with three tools can be found. Red markers are collected GPS measurements and black path is the matched path. As

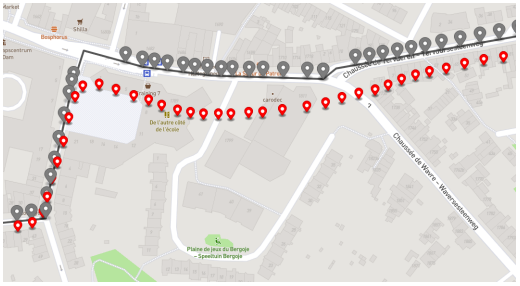
you can see, Barefoot has two inaccuracies in the given figure while GraphHopper and Valhalla have none. These inaccuracies appear in Barefoot when a sequence of measurements diverges a little too far from the real path. We can see that Barefoot provides some unusual path with some strange U-turns. This is a consisting error source while map matching with Barefoot. Some algorithm updates might be useful to remove these types of errors.



(a) Barefoot



(b) GraphHopper

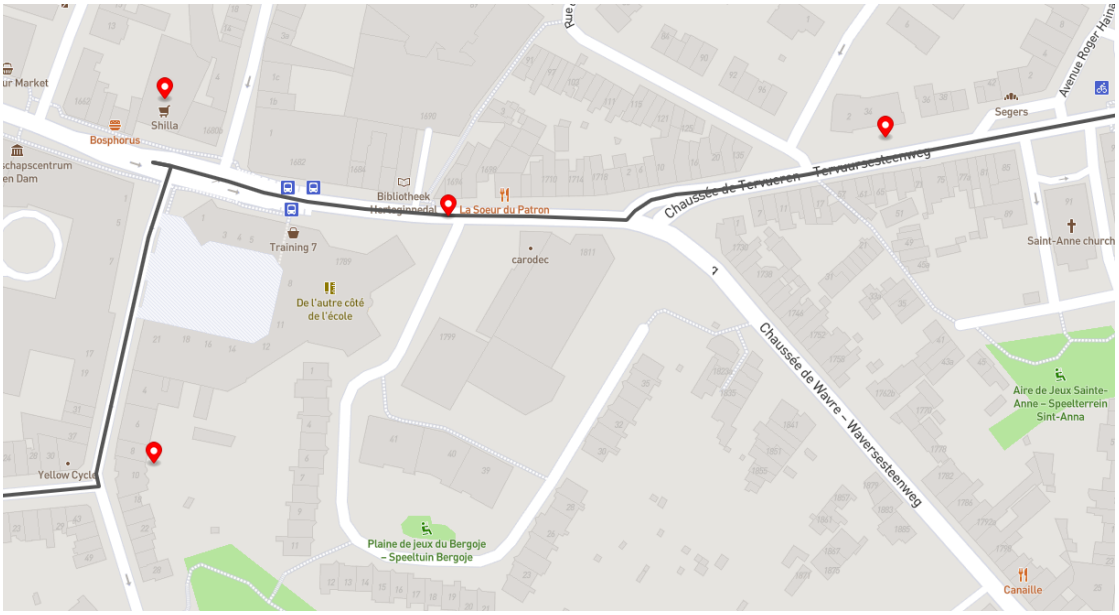


(c) Valhalla

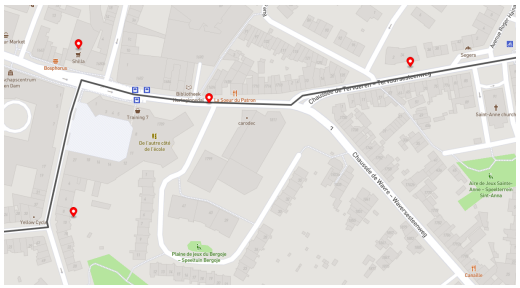
Figure 4.1: Map matching results with inaccuracies for dense GPS data with red points as GPS measurements and black path as matched path

4.2.2 Sparse GPS measurements

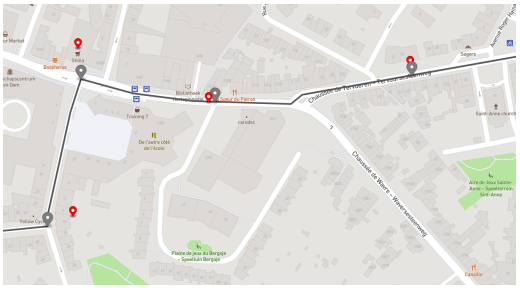
In Figure 4.2, an excerpt of the results while the GPS measurements are sparse can be found. As you can see, Barefoot performs better on the dataset when GPS measurement is not that much close together. We can see that the inaccuracies are reduced considerably. Nonetheless, there are still some minor inaccuracies with Barefoot. These small inaccuracies consist of some small and strange U-turns while a given GPS point is too far from a road or a turn. These inaccuracies are not found neither in GraphHopper nor in Valhalla.



(a) Barefoot



(b) GraphHopper



(c) Valhalla

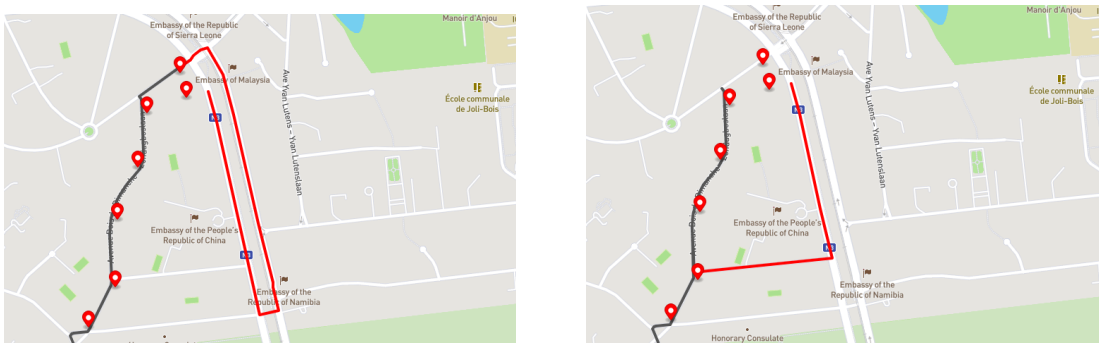
Figure 4.2: Map matching results with inaccuracies for sparse GPS data with red points as GPS measurements and black path as matched path

4.2.3 Matching the first measurement

In section 2.2, we talked about how HMM map matching algorithms chose the initial state probabilities to start the algorithm. This is the probability of the vehicle's first road over all the roads at the beginning of the drive. Some algorithms assign a uniform distribution to π_i while others simply chose it to be the first measurement ($\pi_i = p(z_1 \mid r_i)$).

This probability is important because sometimes, it can cause anomalies. In our experiments with Barefoot and GraphHopper, we have observed that whenever the first GPS measurement is far from the actual path, the map matching algorithm matches this point to another path and hence, there is some U-turns or really strange driving behaviors at the start of the trip. This observation is no where to be found in Valhalla with our experiments. In Figure 4.3, an example of these inaccuracies in Barefoot and GraphHopper can be found. The red path is the inaccurate matched path. In Barefoot, the inaccurate matched path is much more reasonable than GraphHopper.

We can see that in GraphHopper, this behavior completely changes the actual path and misses the start of the trip and introduces a really strange path with an unreasonable U-turn.



(a) Barefoot (b) GraphHopper

Figure 4.3: Map matching results with inaccurate paths for starting GPS point with red points as GPS measurements, black path as accurate matched path and red path as inaccurate matched path

Conclusion

This paper focused on the problem of map matching. A state of the art of the problem was first discussed and than we discussed about hidden Markov models and how they can solve the problem of map matching. To see how our ground truth data would work on map matching algorithms, some HMM tools were tested. These tools were tested to see how they differ with regard to performance and accuracy of matched points.

This paper secured two main objectives. To do an introduction and state of the art on the map matching problem and to test some of the existing tools with our ground truth data and compare their performance and accuracy. For the second objective, the idea was not to show which tool is better than the other. The idea was to show how they differentiate in some aspects, so the reader can choose the best tool that suits their ecosystem and their environment.

As we saw, some tools offer more flexibility (like Barefoot with a dedicated map server) while others offer a better execution time (like Valhalla). We saw that some tools suffer significantly from small inaccuracies in GPS data and show strange driving behaviors in the results. For example, we saw that Barefoot suffers from strange U-turns when GPS points are far from the actual road. We also saw that the starting GPS point could sometimes introduce inaccurate paths if it is not close enough to the actual path. This was shown in some results with strange driving behavior in GraphHopper and Barefoot.

Although in our experiments some tools were shown to be better than the others, one must also consider in which ecosystem and environment these tools are going to be integrated. For example if we already have a map server, we might be better off using Barefoot because by doing so we do not need to setup a map server from scratch. Or if we want to have something portable and easy to use without any overhead setup, we could use GraphHopper. The choice of the tool depends on the environment and requirements. It should be noted that all of the tested tools in this paper are open source and some of them are no longer maintained (like Barefoot) and this can play an important role on the results of our experiments and observations.

Bibliography

[1] Paul Newson and John Krumm. Hidden markov map matching through noise and sparseness. In *17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, November 4-6, Seattle, WA*, pages 336–343, November 2009.

[2] Yin Lou, Chengyang Zhang, Yu Zheng, Xing Xie, Wei Wang, and Yan Huang. Map-matching for low-sampling-rate gps trajectories. pages 352–361, 01 2009.

[3] Mohammed A. Quddus, Washington Y. Ochieng, and Robert B. Noland. Current map-matching algorithms for transport applications: State-of-the art and future research directions. *Transportation Research Part C: Emerging Technologies*, 15(5):312–328, 2007.

[4] Christian S. Jensen and Nerius Tradišauskas. Map matching. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1692–1696, Boston, MA, 2009. Springer US.

[5] Dave Bernstein and Alain L. Kornhauser. An introduction to map matching for personal navigation assistants. 1998.

[6] Josh Greenfeld. Matching gps observations to locations on a digital map. *Transportation Research Board 81st Annual Meeting*, 01 2002.

[7] Meng Yu. Improved positioning of land vehicle in its using digital map and other accessory information. *Hong Kong Polytechnic University – Dissertations*, 2006.

[8] Mohammed Quddus, Washington Ochieng, Lin Zhao, and Robert Noland. A general map matching algorithm for transport telematics applications. *GPS Solutions*, 73, 12 2003.

[9] Mohammed A. Quddus, Robert B. Noland, and Washington Y. Ochieng. Validation of map matching algorithms using high precision positioning with gps. *Journal of Navigation*, 58(2):257–271, 2005.

[10] Sotiris Brakatsoulas, Dieter Pfoser, Randall Salas, and Carola Wenk. On map-matching vehicle tracking data. volume 2, pages 853–864, 01 2005.

[11] Sotiris Brakatsoulas, Dieter Pfoser, Randall Salas, and Carola Wenk. On map-matching vehicle tracking data. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, page 853–864. VLDB Endowment, 2005.

[12] Chalermchon Satirapod, Chris Rizos, and Jinling Wang. Gps single point positioning with sa off: How accurate can we get? *Survey Review*, 36:255–262, 10 2001.

[13] Lianxia Xi, Quan Liu, Minghua Li, and Zhong Liu. Map matching algorithm and its application. *International Journal of Computational Intelligence Systems*, 10 2007.

[14] Helmut Alt, Alon Efrat, Günter Rote, and Carola Wenk. Matching planar maps. *Journal of Algorithms*, 49(2):262–283, 2003.

[15] D. Yang, B. Cai, and Y. Yuan. An improved map-matching algorithm used in vehicle navigation system. *IEEE Proceedings on Intelligent Transportation Systems*, 2:1246–1250, 01 2003.

[16] Wuk Kim, Gyu-In Jee, and JangGyu Lee. Efficient use of digital road map in various positioning for its. In *IEEE 2000. Position Location and Navigation Symposium*, pages 170–176, 2000.

[17] Youjing Cui and Shuzhi Sam Ge. Autonomous vehicle positioning with gps in urban canyon environments. *IEEE Transactions on Robotics and Automation*, 19(1):15–25, 2003.

[18] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall PTR, USA, 1st edition, 2000.

[19] Christoph Haase and Stefan Kiefer. The odds of staying on budget. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming*, pages 234–246, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[20] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[21] A.J. Viterbi. A personal history of the viterbi algorithm. *IEEE Signal Processing Magazine*, 23(4):120–142, 2006.