CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

An algorithm for determining in which public transport vehicle a passenger is traveling

Master's thesis in Computer Science and Engineering

MIKI SWAHN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

Master's thesis 2019

An algorithm for determining in which public transport vehicle a passenger is traveling

MIKI SWAHN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2019

An algorithm for determining in which public transport vehicle a passenger is traveling
MIKI SWAHN

Supervisor: Carl-Johan Seger, Department of Computer Science and Engineering
Advisor: Jonas Williamsson, Consat AB
Examiner: Graham Kemp, Department of Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

An algorithm for determining in which public transport vehicle a passenger is traveling

MIKI SWAHN
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This thesis presents an algorithm, that can determine in which public transport vehicle a passenger is traveling. The algorithm makes use of data such as Global Positioning System (GPS) location, vehicle length, bearing, speed, acceleration, Wi-Fi Positioning System (WPS) and planned itinerary. The algorithm collects the passenger location every few seconds, and queries all the nearby vehicles. Every vehicle is given a vote based how well the data match the data of the passenger. After a series of iterations the vehicle consistently appearing as a likely candidate will be returned. Tests show that the algorithm successfully can determine in which vehicle a passenger is traveling, in less than 1 minute. These tests were performed on an implementation of the algorithm that only uses GPS data. It can thus be concluded that the idea for the algorithm is sound.

Keywords: telematics, public transport, moving objects, GPS, matching passenger and vehicle, trip planner, public transport payment system, privacy.

Acknowledgements

I would like to thank my academic supervisor Carl-Johan Seger for his commitment to helping with my thesis. He has offered his time generously and been a great support in prioritizing for both short- and long term goals. I would also like to thank my company supervisor Jonas Williamsson for the chance to elaborate on ideas and the ability to reach out to the right people. Finally, I want to thank everyone at Consat for the time with the company, and Anna Edin for making the thesis happen and introducing me to the company. Thank you.

Miki Swahn, Gothenburg, June 2019

Contents

List of Figures **xi**

List of Tables **xiii**

1 Introduction **1**

2 Background **3**
2.1 Defining the Algorithmic Problem 3
2.2 Applications of the Algorithm 4
2.3 Difficult Cases 6
2.4 Limitations 7

3 Main Idea **9**
3.1 The Algorithm 10
3.2 The Voting System 11

4 Detail **15**
4.1 Implemented Algorithm 16
4.2 Passenger Data 17
4.3 Vehicle Data API 17
4.4 Fetch Nearby Vehicles 18
4.5 Architecture 19

5 Test Results **21**
5.1 Successful Tests 23
5.2 Failed Tests 26
5.3 Key Performance Indicators 28

6 Discussion **29**
6.1 Sources of Error 30
6.1.1 Passenger Data Errors 30
6.1.2 Vehicle Data API Errors 33
6.2 Kinds of Data 36
6.2.1 GPS Location 36
6.2.2 Vehicle Length 36
6.2.3 Bearing 37
6.2.4 Speed 38

6.2.5 Acceleration 39
6.2.6 Planned Itinerary 39
6.2.7 Wi-Fi Positioning System 40
6.3 Feasibility and Ethical Discussion 40
6.3.1 Ethical Discussion 41
6.3.2 Market Demand 42
6.4 Future Work 43

7 Related Work **45**
7.1 Voronoi Diagrams 45
7.2 Interpolation 46
7.3 Map Matching 46
7.4 Dead Reckoning 47

8 Conclusions **49**

A Source Code - Java Implementation **I**

List of Figures

2.1 Vehicles in a junction with different directions and velocities. 6
2.2 Vehicles in a queue, driving at the same speed in the same direction close to each other. 7

3.1 The main idea for the algorithm. 9

4.1 The App in an initial phase, displaying some recorded data about the passenger while traveling by bus. 16
4.2 An API bounding box in blue, surrounding a passenger 18

5.1 Picture of the final implementation of the app, taken from just before journey 10. 22
5.2 Journey 10, Friday May 17, 17:06. The trip went through the most central junction, Brunnsparken, during evening rush hour. 24
5.3 Given that GPS errors average around 5-8 meters, there must be another explanation why the nearby vehicles so frequently are missing in the API response. 27

6.1 GPS locations from journey 3, displayed on a map. 31
6.2 How the API verificaton test was performed. 34
6.3 From the location of the vehicle, the direction of travel and the length of the vehicle, a bounding box surrounding the vehicle can be obtained. 37

7.1 An example of a Voronoi diagram. 46

List of Tables

5.1 Journey 10, Friday May 17, 17:06. Successful output from algorithm in 40 seconds when traveling with bus 55. 23
5.2 Journey 8, Friday May 17, 14:06 24
5.3 Journey 9, Friday May 17, 14:11 25
5.4 Journey 11, Monday May 20, 09:42 26
5.5 Journey 7, Friday May 17, 10:59 26
5.6 Journey 6 Thursday May 16, 16:46 27

6.1 The data from journey 3. 32
6.2 API verification result summary. 35
6.3 onlyRealtime=yes 35
6.4 onlyRealtime=no 35

1

Introduction

The thesis is about designing an algorithm, that can determine in which vehicle in public transport a given passenger is traveling. The solution utilizes the passenger's smartphone to collect data such as Global Positioning System (GPS) location, and match it with data collected from the vehicles in a traffic management system. To understand why the problem is relevant, let us explore potential applications for the algorithm. One application is a payment system, that as automatically as desired will handle the payments based on when, how and where a passenger travels. Another application is a trip planner, which suggest new routes in real time in case of traffic delays, since it knows which vehicle the passenger is on board. As opposed to common positioning problems of putting entities on maps, this problem is characterized by inferring which vehicle a passenger is traveling with. There is not much research in the area, and the novelty of the study lies in the ability to detect that two objects are moving together.

Envision the following scenario: *A person looks up an itinerary in the trip planner app. He or she hops on board a vehicle, and within a minute the app pops up with the information "You are on board bus 11, but in the wrong direction. Get off at the next stop, within 3 minutes you can take bus 11 headed towards South Bank instead."* Or perhaps: *"There is a traffic jam on your route. If you choose to get off at the next station, and catch train 431 at 17:42, you will get to Preston Road faster."*

The difficulty of the study lies in the inaccuracy of GPS locations. Thus, a key part is to evaluate what additional kinds of data can be incorporated. The other key part is how the computation itself can be performed such that the output is reliable. An automatic payment system requires high accuracy such that the correct payments can be made, even in corner cases where several vehicles are close to the passenger. In addition, for a trip planing application to be useful, the algorithm has to have a short response time to detect the vehicle, or else the passenger can reschedule themselves faster.

2

Background

The aim is to devise an algorithm that takes a passenger as input and outputs the vehicle they are on board. The interest arise from the public transport domain where passengers demand hassle-free payment systems and trip planners. Likewise, transport providers demand information about when and where passengers travel. This chapter will define the problem and motivate why the study is carried out. To begin, let us introduce the research questions of the project:

Research questions:

- Is it possible to create an algorithm that solves the problem, and what Key Performance Indicator (KPI) values (Section 2.1) will it have?
- What kinds of data are suitable?
- What are the passenger privacy concerns versus the passenger value of such an application, and could there be a demand on the market?

2.1 Defining the Algorithmic Problem

The algorithmic problem can be defined as follows.

Given:

- A target passenger p and for that passenger:
  - A set of snapshots, where each snapshot is a pairing of a timestamp t and a GPS location with longitude x and latitude y in WGS 84 [1]
- A set of vehicles: $V : \{v_0, ..., v_n\}$, and for each v_i
  - A set of snapshots, where each snapshot is a pairing of a timestamp t and a GPS location with longitude x and latitude y in WGS 84 [1]

It is possible to collect the snapshots in real time for both the passenger and vehicles. Other than GPS, it could be possible to have other kinds of data, such as the momentary speed.

Find:

The vehicle that the passenger is on board.

Key Performance Indicators:

The scope of the thesis is the feasibility of solving the problem itself. The goal is thus an algorithm that can match a passenger with their vehicle. To verify whether the algorithm has been successfully developed the following KPI:s will be used as measurements.

- Average response time
- Time complexity
- Accuracy of the output, as a percentage of the performed tests

2.2 Applications of the Algorithm

The study is motivated by the applications the algorithm can be used for. The development of applications is not a part of the project, but they are important as they determine some specifications for the algorithm and make it relevant.

A trip planner is the most important application. Passengers already use such applications to get an itinerary from point A to B. However, if the app knows when a passenger is on board a vehicle, it can combine that with the information of traffic delays. If a passenger is stuck in traffic, and will miss their connection, the app can suggest a new itinerary as it might be faster to take another route. Likewise, if the connecting vehicle is delayed, the passenger can be informed and offered another connection. The trip planner would also be able to help a passenger who boarded the wrong line or in the wrong direction. Likewise reschedule someone who missed their departure, or managed to catch an earlier one.

Another application is **a payment system**, which if it knows the passenger's itinerary, could be as automatic as desired. It is however unlikely that a fully automated system would be a success on the market, since people like to be in control of payments they make and since the algorithm would not be 100% accurate in its output. Instead, let us imagine a payment system that serves the passenger the right options, but leaves the execution of any payment in the hands of the passenger. Imagine the following scenario: *The passenger activates the app when traveling with public transport. When the trip has ended the app can prompt them "You have travelled from Central Station in Zone 1 to Alexandra Park in Zone 2. Do you want to pay for this trip ($4)?", with options "Yes, pay $4", "No" or "Details are incorrect. Edit trip".*

To get information on how passengers travel is a third application, which is incredibly useful for the public transport providers. If a passenger-vehicle pairing

application reaches a critical mass of users, it would open up for optimization on a city scale. Public transport companies generally only have a limited amount of information on where journeys begin and end. Today, it is possible to know how many passengers are on board at a given time via sensors in the vehicles. However, it cannot determine where each passenger travels to or from. It can merely confirm that more people travel in the city center than at the end of the lines. One might think that existing trip planers can tell where people travel, but they are mainly used by those making unusual trips, and not commuters, thus not being representative for all passengers. By instead gathering data when passengers board a vehicle and when they get off, it is possible to optimize which lines operate at what times, as well as what stops should be included on a line. This is possible if such a trip planner or payment system is used by a critical amount of passengers.

Yet another application is **pre-hospital solutions** similar to eCall services in cars, but for public transport. If a vehicle is in an accident it is useful to know what passengers are involved. The public transport company could have access to a set of anonymous passenger ID:s that were on board the crashed vehicle. It could then send an alert to their apps, and if the passenger has activated the functionality, a message could be sent to their In Case of Emergency contact. Such a system protects passenger integrity by storing their information locally on their phones, at the same time as it enables the rescue operation to be effectively coordinated. A research project at Chalmers University of Technology has resulted in the motorcycle app *Detecht* which analyze driving behaviour and identify if a serious accident has occurred [2]. That algorithm collects information from the smartphone sensors for velocity, g-force and rotation and connect to the Swedish emergency service SOS Alarm [3]. From bus data alone it is hard to tell if a passenger is seriously injured, but using a smartphone carried on their person it might be clear if they remained seated with their seat belt, or if they (or their phone) flew out the window because they were standing up. With the algorithm of this thesis project, in combination with the *Detecht* algorithm, passengers involved in an accident, can provide their medical information as well as how serious the accident was to their family, the public transport company or emergency services. This will allow for the right amount of ambulances to be sent to the site of the accident, or perhaps just a taxi.

Lastly, **other applications** can instead pair passengers with each other if they travel with the same vehicle. Friends who happen to take the same bus can enjoy each others company. Beyond that, there are other applications outside the public transportation sector. Since the algorithm collects standard positional data it could also match autonomous vehicles traveling in a similar pattern, which could be used to increase the capacity of a highway by creating convoys, known as platooning. It could even be used for traffic planning, for instance, by creating “green waves” through a series of traffic lights if a cluster of vehicles are detected to travel together.

These suggested applications showcase the variety of possibilities of an algorithm that matches moving objects travelling together. For these applications to be feasible one must first design an algorithm that determines which vehicle a passenger travels with. The scope of this project are not the applications, but simply the matching

algorithm itself.

2.3 Difficult Cases

One can assume that the problem is trivial if there is only one vehicle in the vicinity of the passenger. What makes the problem interesting are the difficult cases where more than one vehicle is a likely candidate. Some such cases are described below to illustrate the complexity of the problem. In Västra Götaland region (which is the basis of the thesis project, see Section 2.4), a tram is 30 meters long and a bus is about 20 meters long [4]. Moreover, the GPS transmitter is situated in the front of the vehicles, behind the driver's seat. This means that when a passenger it seated in the back of a vehicle, they might be closer to the GPS transmitter of a vehicle behind them.

Junction. In a large junction, or stop, a lot of vehicles can pile up close to each other. The passenger might be seated such that the transmitters of other vehicles are closer, than the transmitter of the vehicle they travel with. The hypothesis is that bearing and speed will help determine which vehicle the passenger is on board. Especially when vehicles are traveling in opposite directions, or stopping on opposite sides of the street, the bearing should quickly be able rule out which is the wrong vehicle and which is the right vehicle.

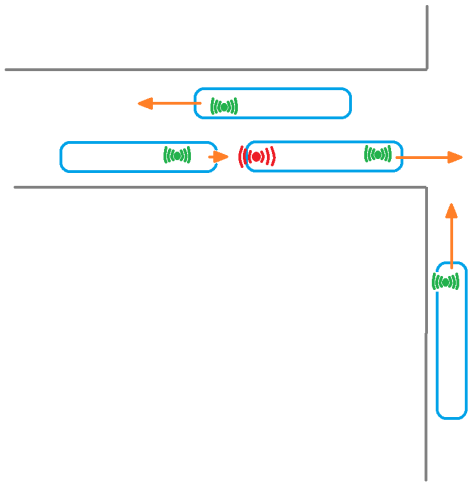

Figure 2.1: Vehicles in a junction with different directions and velocities. Passenger is in red and vehicle transmitters in green. Bearing and speed illustrated with orange vectors.

Queue. If vehicles are in a queue, their bearing will be the same and their speed will be similar. This is likely to hold at any given time while they are in a queue. At the same time, a passenger seated in the back will be closer to the transmitter of the vehicle behind them. This case is illustrated in Figure 2.2.

Figure 2.2: Vehicles in a queue, driving at the same speed in the same direction close to each other. Passenger is in red and vehicle transmitters in green. Bearing and speed illustrated with orange vectors.

Walking inside the vehicle Another case is if the passenger walks towards the back of the car, as the vehicle slowly drives forward. The passenger moves in the opposite direction of the vehicle, and might even stay in the same geographical location. This case is supposedly solved by querying for vehicles for a time period longer than the time it takes to walk across the vehicle.

2.4 Limitations

The public transport system acting as reference for the project is the Västra Götaland region in Sweden. Assumptions about the public transport are based on this region. However, as discussed in Section 2.2 about the applications, the general idea of matching moving objects can be applied to any situation.

There are assumptions about the availability of input data. Only over ground transportation, such as buses, trams, cars, boats, are in scope, unlike underground transportation. Furthermore, the study is also only applicable in urban areas where there is cellular reception. And finally, it is assumed passengers have agreed to share their data as input to the algorithm, and that they are connected to the internet. What it means is only that the algorithm will only work if the data is available.

The validity of input data is however not assumed, but instead covered for in the algorithm. Sensors are not perfectly reliable and will produce spurious data points which the algorithm should tolerate.

An important limitation is that the problem definition is only concerned with the initial pairing of a passenger to a vehicle. For most applications it is also of interest to re-evaluate if the passenger is still on the same vehicle after some time has past, as well as detecting if the passenger makes a change. This task is deemed trivial once the initial pairing is done, since the problem is the same except there is additional information including which vehicle the passenger was previously on board.

3

Main Idea

The algorithm gets the location of the passenger every few seconds. It will then query all the nearby vehicles. For every such vehicle, a number of points will be awarded. The points can be awarded based on distance to passenger, or based on other kinds of data collected. The algorithm will do a series of iterations with new passenger locations, and for every iteration make a list of the top vehicle candidates. Finally, it will return the vehicle that consistently has appeared as a likely candidate.

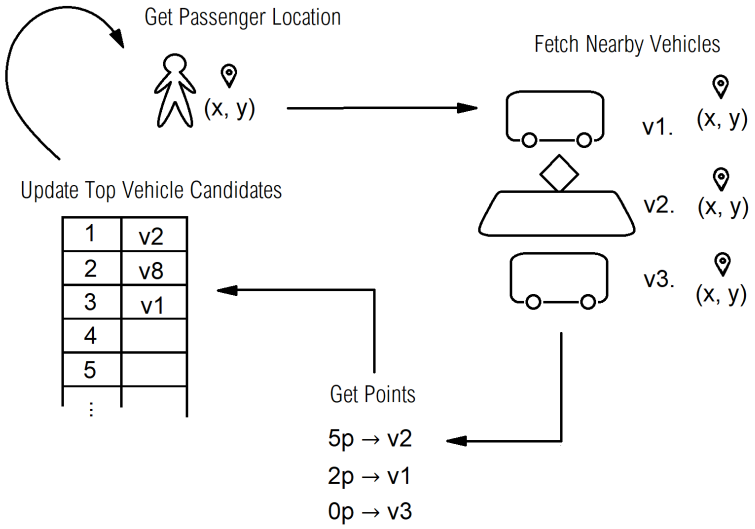

Figure 3.1: The main idea for the algorithm.

This solution is flexible in two ways. First, what kinds of data to incorporate can be chosen based on the implementation and what data is available. Second, how the points are awarded can be done in any way. For instance, one kind of data can be weighted much lighter in the voting system than another depending on how reliable it is.

3.1 The Algorithm

With only GPS location as input data, quite a lot can be done. In most cases a passenger vehicle pairing can be found in less than a minute, as tests will show in Chapter 5. In order to shorten the response time and increase accuracy, other kinds of data than GPS must be included. Suggestions include vehicle length, bearing, speed, acceleration, Wi-Fi position and planned itinerary. See Section 6.2 for more on this. Below is a pseudo code implementation of the algorithm, that uses only GPS location data, and awards points solely based on distance to passenger.

Algorithm 1 Matchmaking algorithm

```
1:  list topVehicleCandidates
2:  list result
3:  iterations ← 0
4:
5:  loop forever:
6:
7:      iterations ++
8:      x, y ← getPassengerLocation ()
9:      boundingBox ← getBox (x, y)
10:     vehicles ← fetchNearbyVehicles (boundingBox)
11:     for every vehicle v ∈ vehicles do
12:         distance ← distanceBetween (v, x, y)
13:         points ← getPoints(distance)
14:         if v ∈ topVehicleCandidates then
15:             v.points ← v.points + points
16:         else
17:             topVehicleCandidates .add (v)
18:             v.points ← points
19:
20:     bestCandidate ← getBest(topVehicleCandidates)
21:     for every vehicle v ∈ result do
22:         if v 50% worse than bestCandidate, or more then
23:             result .remove (v)
24:     for every vehicle v ∈ topVehicleCandidates do
25:         if v 50% worse than bestCandidate, or more then do nothing
26:         else result .add (v)
27:     if ( result .length () = 1 && iterations > 10 ) then
28:         Break
29:
30: return result
```

Remembering the problem definition, for the passenger p there is a set of snapshots, where each snapshot is a pairing of a timestamp t_p, a longitude x_p and a latitude

y_p. There is also a set of vehicles: $V : \{v_0, ..., v_n\}$, and for each v_i a set of snapshots with t_{v_i}, x_{v_i} and y_{v_i}. When the algorithm has retrieved the nearby vehicles, for each vehicle it should also check that the timestamps for the snapshots reflect the same time. The following time condition should be checked, where t in UNIX Epoch time and with a suitable value for the delay L in milliseconds:

$$C_{time}(t_{v_i}, t_p) = |t_p - t_{v_i}| \leq L \tag{3.1}$$

As for the GPS locations, the below points function can be used for awarding points to the vehicles based on distance. D is the maximal Euclidean distance between the passenger GPS location and their vehicle GPS location. It should be defined based on the typical error in GPS locations. According to a 2011 empirical study the median error is 5 to 8.5 meters for smartphones [5], and very similar for the vehicles (E. Lundin, Consat Telematics, Personal interview, February 4 2019). D should also be defined based on the length of a vehicle, such that a passenger sitting far away from the GPS transmitter still can be considered to be on board that vehicle.

$$distance(t_{v_i}, t_p) = \sqrt{(x_p - x_{v_i})^2 + (y_p - y_{v_i})^2} \tag{3.2}$$

$$\begin{aligned} points(t_{v_i}, t_p) = (distance(t_{v_i}, t_p) \ &\leq \ \frac{1}{3} D \ \rightarrow \ 10) \\ \wedge \ (distance(t_{v_i}, t_p) \ &\leq \ \frac{2}{3} D \ \rightarrow \ 7) \\ \wedge \ (distance(t_{v_i}, t_p) \ &\leq \ D \ \rightarrow \ 4) \end{aligned} \tag{3.3}$$

What is considered to be a consistently likely candidate can also be decided rather flexibly depending on the implementation. Here, the condition is that the resulting candidate should have at least 50 % more points than any other candidate. Also, there must be 10 iterations before any candidate can be deemed to be superior. A similar condition is that a vehicle should have accumulated at least a certain number of points to be considered the correct one. In case two vehicles are likely candidates for a long period of time, such as if the vehicles are in a queue, yet another condition is needed to ensure the algorithm terminates. Such a condition is that if a vehicle has been the top candidate for 15 iterations, it will be returned as the result.

3.2 The Voting System

The benefit of the voting system is that no data filtering is needed. Unless an error is consistent, minor noise will have no effect on the total vote for any vehicle. If during one iteration the correct vehicle GPS location jumps to an erroneous location, such as a river 200 meters from the road, the only effect is that the vehicle will not get

any points from that particular iteration. Nevertheless, it can still accumulate a high vote from all other iterations and become the resulting candidate. Likewise, if the passenger location is wrong during an iteration, the effect is simply that some points will be awarded incorrectly. Some incorrect points will not matter since the total vote is based on several iterations. The resulting candidate will have to be consistently close to the passenger.

Another benefit is that the various kinds of data can be given different importance, or weight, in the voting system. The weight of each kind of data should be based on how reliable that data is, both in terms of data validity and in terms of how suitable it is for matching two moving objects. For example, GPS data can weigh heavier than speed, since if a vehicle is 5 meters from the passenger it probably says more than if they both move at 20 km/hour. And so, if passenger and vehicle GPS locations are close, the vehicle will get a high vote, say 10 points. If they also move at similar speed the vehicle will get an additional small vote, say 2 points. By using the voting system, the result can be made more accurate by incorporating several sources of data and weighing them appropriately.

In Algorithm 3 is an implementation of a voting system. Points are awarded to every vehicle in the bounding box. The implementation gives many points to vehicles that are closely nearby, and bonus points if the bearing and speed are similar. Bearing is an angle in relation to true north, a compass direction. The vehicle will not be awarded any points if the snapshots have timestamps that differ too much.

Algorithm 2 Get Points

```
1: points ← 0
2: distance ← input
3: speedVehicle ← input
4: speedPassenger ← input
5: bearingVehicle ← input
6: bearingPassenger ← input
7: timeVehicle ← input
8: timePassenger ← input
9:
10: if |timeVehicle − timePassenger| ≤ someTimeDelay then
11:
12:     if (distance ≤ 15 meters) then
13:         points = 10
14:     else if (distance ≤ 30 meters) then
15:         points = 7
16:     else if (distance ≤ 45 meters) then
17:         points = 4
18:
19:     if |speedVehicle − speedPassenger| ≤ someThreshold then
20:         points = points + 2
21:
22:
23:     if |bearingVehicle − bearingPassenger| ≤ someOtherThreshold then
24:         points = points + 1
25:
26: return points
```

Comment: All numbers, distances in meters as well as points, are just examples. They will need to be configured for the implementation, i.e. by learning from performed tests in the relevant public transport system.

4

Detail

To verify the functionality of the algorithm, an app for matching passengers with vehicles has been developed. It collects passenger data and queries nearby vehicles through an API. It is an android app written in Java in the Android Studio development environment[1]. The choice was made based on the low threshold for starting implementation and to get an app up and running on a mobile device for free. Moreover, within the project there is prior experience in such development.

An early version of the app can be seen in Figure 4.1. The data captured by the app, such as passenger longitude and latitude and the nearby vehicles, is saved to a txt-file such that tests can be re-run by inputting the same data to the algorithm, and such that the results can be analyzed. The hardware where the app is running, is a Huawei Nexus 6P smartphone manufactured in 2015, where the CPU is a 2.0 GHz octa core 64-bit ARMv8-A. This is the hardware where all tests have been performed.

[1] Android Studio: https://developer.android.com/studio/

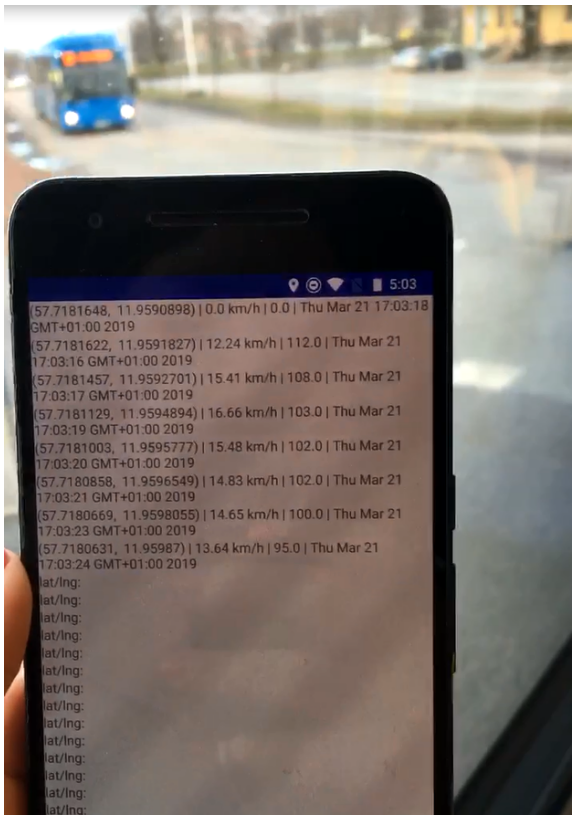

Figure 4.1: The App in an initial phase, displaying some recorded data about the passenger while traveling by bus.

4.1 Implemented Algorithm

The app follows the code in Algorithm 1, Section 3.1. The passenger GPS location is collected every third second, using the Android Location library [6]. The GPS system use the WGS84 standard for the coordinates around the globe [1]. All vehicles inside a bounding box, of 120×120 meters centered around passenger, are queried via the *Reseplaneraren v2 Livemap* API, provided by the Västra Götaland public transport company; Västtrafik [7]. The box was made slightly larger than necessary such that all nearby vehicles would be included in the response. Unfortunately, the API cannot supply timestamps for when a GPS location was recorded, and thus no validation is made to ensure that no time delay affects the output.

Points are awarded solely on GPS distance, 5 points for vehicles within 30 meters, 4 points for within 40 meters, and 1 point for within 50 meters. The lowest threshold, 30 meters is based on the length of the longest vehicle[4]. It was chosen such that when a passenger is seated in the back of a vehicle and there is another vehicle behind, the vehicle behind should not get a higher point than the passenger vehicle, even if GPS transmitters are in the front of vehicles (E. Lundin, Consat Telematics, Personal interview, February 4 2019). It was also chosen such that a passenger in the back of a vehicle shall have equal chances of getting a correct output as a passenger

in the front. The drawback of this threshold is that the distance is measured in any direction, meaning effectively that a vehicle is considered to have their potential passengers anywhere within 30 meters from the GPS transmitter. A solution to this is suggested in section 6.2.2, namely to include the length of vehicles in the input data, along with the direction of travel. Furthermore, the 40 meter threshold is based on a GPS error of 5 meters for both passenger and vehicle in opposite directions. Finally, the 50 meter threshold is based on the maximal Euclidian distance between a passenger and a vehicle, i.e. if both GPS errors are 9 meters and the vehicle length 30 meters.

4.2 Passenger Data

The passenger GPS location and time is obtained using Android's Location library[6]. The passenger location is mainly obtained from GPS. Additionally, the Android Location library might fuse GPS data with cellular triangulation and Wi-Fi positioning system to provide more accurate locations, yet the exact sources cannot be declared. The validity of the passenger location data is discussed in Section 6.1.1. The error was measured between 1.4 and 10.8 and found to average between 1.9 and 7.7 meters (mean error ± standard deviation).

4.3 Vehicle Data API

The *Västtrafik Reseplaneraren v2 Livemap* API takes two longitudes x_{min} and x_{max} and two latitudes y_{min} and y_{max}, and returns all vehicles enclosed in that bounding box [7]. To get the bounding box corners, a passenger GPS location x_p, y_p must be offset a certain amount of meters r. It can be done with Equation 4.1 [8] [9], where R is the earth radius (6363000 meters in the Västra Götaland region [8]). An example of a bounding box is visualized in Figure 4.2. The API response contains the current server time, and for each vehicle the real time GPS coordinates. It also includes the name of the line and a trip id used for differentiating between multiple vehicles of the same line. The drawback with the response is that all vehicles have the same timestamp, such that it is impossible to tell when the GPS location was registered in the vehicle.

$$\begin{aligned} y_{min\ or\ max} &= y_p \pm \frac{180}{\pi} * \frac{r}{R} \\ x_{min\ or\ max} &= x_p \pm \frac{\frac{180}{\pi} * \frac{r}{R}}{cos(y_p)} \end{aligned} \tag{4.1}$$

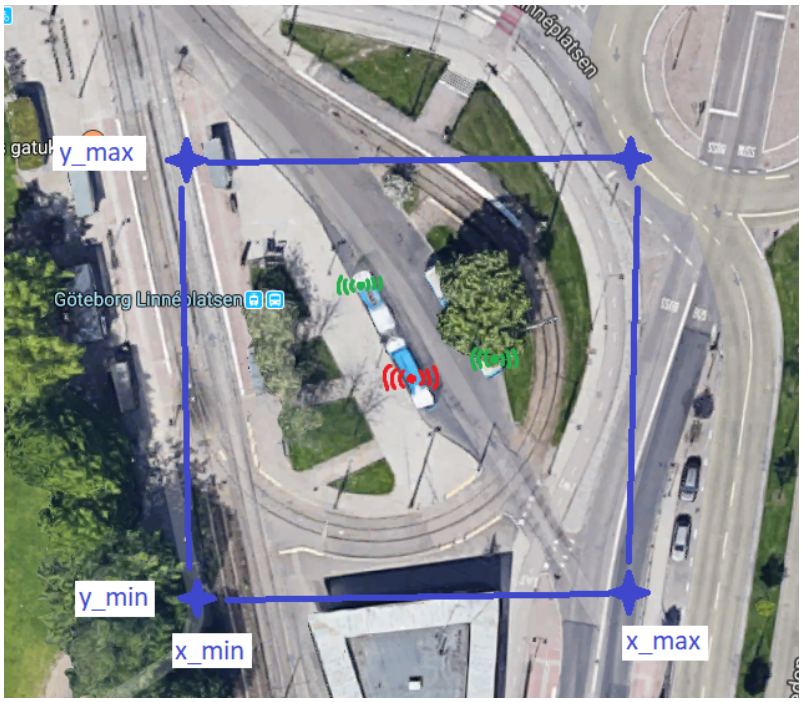

Figure 4.2: An API bounding box in blue, surrounding a passenger in red. Two green marks indicate vehicle GPS transmitters. *Source: Google Developers, Google Maps, https://www.google.com/maps/about/* 2019.

Vehicle locations are updated on server at every turn the vehicle makes, at every stop the vehicle makes, and within a time interval of about a minute (E. Lundin, Consat Telematics, Personal interview, February 4 2019). The location data is then filtered from noise and is map matched to the road network. It ensures every location where the vehicle has been is recorded, consequently, the vehicle locations are presumed to be accurate. This is further discussed in Section 6.1.2.

4.4 Fetch Nearby Vehicles

The implementation makes use of an API for querying nearby vehicles, as mentioned above in Section 4.3. The API already returns the vehicles in a certain area, and thus this part of the implementation was already completed. However, as part of the project it is discussed here how this part can be implemented.

One could make use of a naive, linear time, algorithm for retrieving the vehicles within a radius of a person. Such a brute force algorithm is described below in Algorithm 3. The issue is that, despite being of linear time complexity, it will execute too slowly due to the sheer amount of vehicles in a city (2000 in the Västra Götaland region (E. Lundin, Consat Telematics, Personal interview, February 4 2019)). Taking into account that such a system should be deployed and serve thousands of users making queries every second, it's just not feasible.

Algorithm 3 Naive Fetch Nearby Vehicles

```
1: list vehicles ← all vehicles in town
2: list result
3: x, y ← Passenger location
4: radius ← e.g. 40 meters
5: for every vehicle v ∈ vehicles do
6:     distance ← distanceBetween (v, x, y)
7:     if distance ≤ radius then
8:         result .add (v)
9: return result
```

Instead, there are state-of-the art solutions for databases of moving objects which makes it quicker to query the nearest objects [10][11]. What makes it difficult to sort moving objects is that their locations constantly has to be updated which requires new sorting. Otherwise Voronoi diagrams (described in 7.1) could be used. Instead, Moving Object Databases frequently make use of dead reckoning (a method explained in 7.4) and spatio-temporal databases to store locations such that they can be efficiently queried [12].

4.5 Architecture

The system architecture hugely influence the integrity of both passengers and the public transport system. The main question is where data is stored and where computations are performed; on the passenger phone or the public transport servers. Beyond integrity, the architecture affects the power consumption and data usage on passenger phone, as well as the computational power required on the servers. Consequently, the architecture is very important for feasibility evaluation.

The chosen architecture collects and stores the passenger data only on their phone, performs the major filtering of vehicle data on the public transport servers, then transmits it to the phone, where the algorithm is run. This architecture is chosen because it is in the interest of the passengers to have their data privately stored on their phone only, and for the major filtering of vehicles to be done before transmitting the data such that it consumes less cellular data. It is also chosen because it is in the interest of the public transport company to not run the algorithm for every passenger travelling with them, as it requires a lot of computational power.

5

Test Results

Tests are carried out to test the performance of the algorithm itself. The test are performed using the implemented app, but some tests are disregarded due to the input data to the algorithm being inaccurate. Invalid input data is is a flaw in the implementation of the system, not the algorithm being tested. Since it is only of interest to evaluate the algorithm itself, only test cases with valid input data are included.

A test is performed by a passenger who travels by public transport and simultaneously use the app with the implemented vehicle matching algorithm. The algorithm runs and predicts a matching vehicle for the passenger. The passenger documents the name of the actual vehicle they traveled with. The algorithm output is then compared to the correct vehicle, which determines if the test failed or passed.

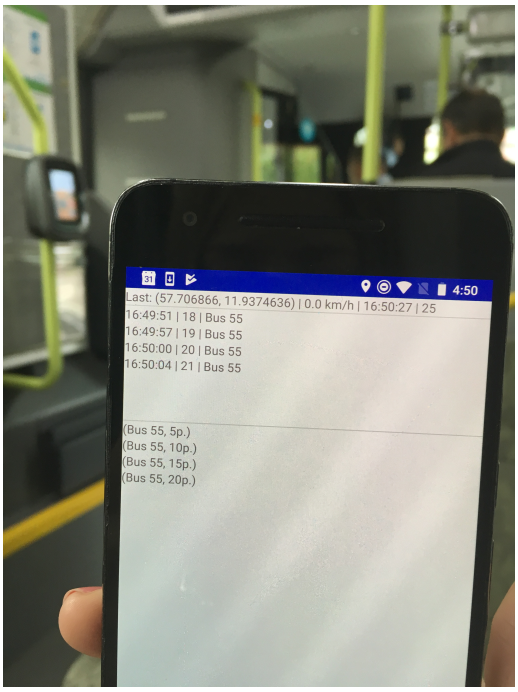

Figure 5.1: Picture of the final implementation of the app, taken from just before journey 10. On the top line on the screen the person performing the test can see their latest location and time. In the second section of lines, the latest API responses are listed, i.e. the nearby vehicles. In the third section, the updated top list is printed out after each iteration. The top list contains the top vehicle candidates and their points. When the top list has a single vehicle and at least ten iterations have past, the vehicle in the top list is returned as the algorithm output.

5.1 Successful Tests

y_p	x_p	time	Nearby vehicles	Top candidates / Result	pass/ fail
57.7081508	11.9671028	**17:06:54**	No vehicles nearby	()	
57.7076852	11.9672392	17:07:00	Tram 5	(Tram 5, 4p.)	
57.707437	11.9675534	17:07:03	Tram 5	(Tram 5, 9p.)	
57.7073652	11.9675868	17:07:06	No vehicles nearby	(Tram 5, 9p.)	
57.7071724	11.9677209	17:07:12	**Bus 55**	(Tram 5, 9p.)(Bus 55, 5p.)	
57.7071672	11.9677762	17:07:17	**Bus 55**	(Bus 55, 10p.)(Tram 5, 9p.)	
57.7071371	11.9678119	17:07:21	Gron express, **Bus 55**	(Bus 55, 15p.)(Tram 5, 9p.)(Gron express, 5p.)	
57.7071382	11.9678457	17:07:27	**Bus 55**	(Bus 55, 20p.)(Tram 5, 9p.)(Gron express, 5p.)	
57.7070988	11.9679405	17:07:30	**Bus 55**	(Bus 55, 25p.)(Tram 5, 9p.)(Gron express, 5p.)	
57.7071138	11.9678592	**17:07:33**	**Bus 55**	(Bus 55, 30p.)	40 sek
57.7071219	11.9678575	17:07:38	**Bus 55**	(Bus 55, 35p.)	
57.7070068	11.9679112	17:07:41	**Bus 55**	(Bus 55, 40p.)	
57.7063067	11.9682715	17:07:44	No vehicles nearby	(Bus 55, 40p.)	
57.7066531	11.9680792	17:07:47	No vehicles nearby	(Bus 55, 40p.)	
57.7061243	11.9682389	17:07:50	No vehicles nearby	(Bus 55, 40p.)	
57.7055586	11.9685332	17:07:56	Tram 4	(Bus 55, 40p.)	
57.7053507	11.9686025	17:08:00	**Bus 55**	(Bus 55, 40p.)	
57.7053272	11.9686218	17:08:03	**Bus 55**	(Bus 55, 40p.)	
57.7053	11.9686443	17:08:08	**Bus 55**	(Bus 55, 40p.)	
57.7051081	11.968898	17:08:14	Tram 10	(Bus 55, 40p.)	
57.7050953	11.9688863	17:08:17	Tram 10	(Bus 55, 40p.)	
57.7050418	11.9687218	17:08:23	Bus 18, Bus 52	(Bus 55, 40p.)	
57.7050424	11.9687521	17:08:26	Bus 18, Bus 52	(Bus 55, 40p.)	
57.7049409	11.9688091	17:08:32	Bus 18, Bus 18, Bus 52, **Bus 55**	(Bus 55, 45p.)	
57.7048797	11.9688505	17:08:38	Bus 18	(Bus 55, 45p.)	

Table 5.1: Journey 10, Friday May 17, 17:06. Successful output from algorithm in 40 seconds when traveling with bus 55.

Journey 10, Friday May 17, 17:06 The passenger was traveling with bus 55, and seated in the front of the 10 meter long vehicle. The API returned a very accurate representation of the nearby vehicles. The vehicles returned were always nearby, and only a few were missing. The algorithm returned a successful output after 40 seconds. See the data in Table 5.2. The journey passed a busy, central area, as can be seen in Figure 5.2.

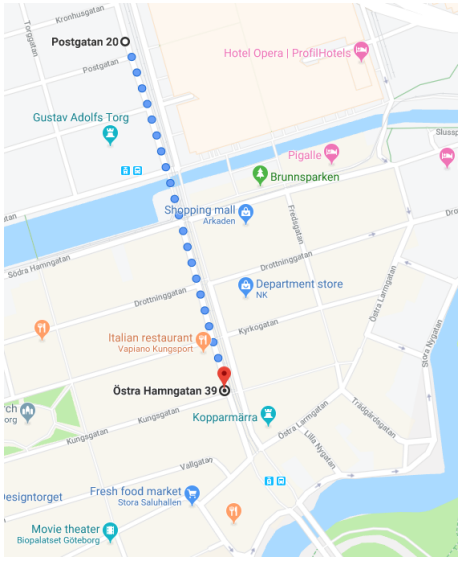

Figure 5.2: Journey 10, Friday May 17, 17:06. The trip went through the most central junction, Brunnsparken, during evening rush hour. *Source: Google Developers, Google Maps, https://www.google.com/maps/about/, 2019.*

y_p	x_p	time	Nearby vehicles	Top candidates / Result	pass/ fail
57.7109215	11.9429073	**14:06:53**	**Bus 16**	(Bus 16, 5p.)	
57.7107404	11.9428031	14:06:56	**Bus 16**	(Bus 16, 10p.)	
57.7107243	11.9427379	14:07:00	**Bus 16**	(Bus 16, 15p.)	
57.7107249	11.942703	14:07:03	No vehicles nearby	(Bus 16, 15p.)	
57.7107168	11.9426779	14:07:07	No vehicles nearby	(Bus 16, 15p.)	
57.7107207	11.9426573	14:07:10	No vehicles nearby	(Bus 16, 15p.)	
57.7107226	11.9426598	14:07:13	No vehicles nearby	(Bus 16, 15p.)	
57.7107226	11.9426599	14:07:16	No vehicles nearby	(Bus 16, 15p.)	
57.7107226	11.94266	14:07:20	No vehicles nearby	(Bus 16, 15p.)	
57.7111484	11.9432158	14:07:25	No vehicles nearby	(Bus 16, 15p.)	
57.7113326	11.9435133	**14:07:30**	No vehicles nearby	(Bus 16, 15p.)	40 sec
57.7117763	11.9443282	14:07:34	No vehicles nearby	(Bus 16, 15p.)	
57.7120173	11.9448192	14:07:37	No vehicles nearby	(Bus 16, 15p.)	
57.7126978	11.9460466	14:07:56	**Bus 16**, Bus 55, Bus 16	(Bus 16, 20p.)	
57.7126927	11.9460464	14:08:00	**Bus 16**, Bus 55, Bus 16	(Bus 16, 25p.)	
57.7128991	11.9463321	14:08:20	No vehicles nearby	(Bus 16, 25p.)	
57.7129543	11.9465136	14:08:23	No vehicles nearby	(Bus 16, 25p.)	
57.7130591	11.946572	14:08:26	No vehicles nearby	(Bus 16, 25p.)	
57.7134492	11.9467715	14:08:32	No vehicles nearby	(Bus 16, 25p.)	
57.7138952	11.9470399	14:08:38	No vehicles nearby	(Bus 16, 25p.)	
57.7141665	11.9471901	14:08:41	No vehicles nearby	(Bus 16, 25p.)	
57.7144724	11.9473175	14:08:44	No vehicles nearby	(Bus 16, 25p.)	
57.7146968	11.9475348	14:08:47	No vehicles nearby	(Bus 16, 25p.)	
57.7151556	11.9482799	14:08:53	No vehicles nearby	(Bus 16, 25p.)	
57.7155448	11.9490881	14:08:59	No vehicles nearby	(Bus 16, 25p.)	
57.7157484	11.949349	14:09:02	No vehicles nearby	(Bus 16, 25p.)	
57.7160031	11.9497727	14:09:05	Bus 58	(Bus 16, 25p.)	
57.7164721	11.9506893	14:09:11	No vehicles nearby	(Bus 16, 25p.)	

Table 5.2: Journey 8, Friday May 17, 14:06

Journey 8, Friday May 17, 14:06 The passenger was traveling with bus 16, and seated in the back of the 20 meter long vehicle. The API returned the correct

passing vehicles and it also returned the passenger vehicle several times. Despite the input data frequently claimed no vehicles were nearby (trivially the passenger vehicle is always nearby), the algorithm still managed to output the correct vehicle.

y_p	x_p	time	Nearby vehicles	Top candidates / Result	pass/ fail
57.7200008	11.9599687	**14:11:39**	No vehicles nearby	()	
57.7200936	11.9601611	14:11:43	No vehicles nearby	()	
57.7200742	11.9608308	14:11:47	No vehicles nearby	()	
57.7198982	11.9617668	14:11:53	Bus 52	(Bus 52, 4p.)	
57.7197382	11.9622314	14:11:56	No vehicles nearby	(Bus 52, 4p.)	
57.7195307	11.9626818	14:11:59	No vehicles nearby	(Bus 52, 4p.)	
57.7189942	11.9632677	14:12:05	No vehicles nearby	(Bus 52, 4p.)	
57.7184005	11.9635757	14:12:11	No vehicles nearby	(Bus 52, 4p.)	
57.7177879	11.9638794	14:12:17	No vehicles nearby	(Bus 52, 4p.)	
57.7172294	11.9641075	14:12:23	No vehicles nearby	(Bus 52, 4p.)	
57.7166952	11.964402	14:12:29	Bus 17, Tram 10	(Bus 52, 4p.)(Bus 17, 4p.)(Tram 10, 1p.)	
57.7164676	11.9645736	14:12:32	No vehicles nearby	(Bus 52, 4p.)(Bus 17, 4p.)(Tram 10, 1p.)	
57.716286	11.9646873	14:12:35	No vehicles nearby	(Bus 52, 4p.)(Bus 17, 4p.)(Tram 10, 1p.)	
57.7158608	11.9650271	14:12:41	No vehicles nearby	(Bus 52, 4p.)(Bus 17, 4p.)(Tram 10, 1p.)	
57.7154697	11.9655288	14:12:47	Bus 810	(Bus 52, 4p.)(Bus 17, 4p.)(Bus 810, 4p)(Tram 10, 1p.)	
57.7151344	11.9660292	14:12:53	No vehicles nearby	(Bus 52, 4p.)(Bus 17, 4p.)(Bus 810, 4p)(Tram 10, 1p.)	
57.7149919	11.966214	14:12:56	No vehicles nearby	(Bus 52, 4p.)(Bus 17, 4p.)(Bus 810, 4p)(Tram 10, 1p.)	
57.7148602	11.9663991	14:12:59	No vehicles nearby	(Bus 52, 4p.)(Bus 17, 4p.)(Bus 810, 4p)(Tram 10, 1p.)	
57.7146267	11.9667253	14:13:05	No vehicles nearby	(Bus 52, 4p.)(Bus 17, 4p.)(Bus 810, 4p)(Tram 10, 1p.)	
57.714603	11.9667542	14:13:09	No vehicles nearby	(Bus 52, 4p.)(Bus 17, 4p.)(Bus 810, 4p)(Tram 10, 1p.)	
57.714569	11.9668029	14:13:12	Bus 58, Tram 6	(Bus 52, 4p.)(Bus 17, 4p.)(Bus 810, 4p)(Tram 10, 1p.)(Bus 58, 1p.)(Tram 6, 1p)	
57.7145338	11.9668582	14:13:16	Tram 6	(Bus 52, 4p.)(Bus 17, 4p.)(Bus 810, 4p)(Tram 10, 1p.)(Bus 58, 1p.)(Tram 6, 2p)	
57.7144905	11.9669309	14:13:20	No vehicles nearby	(Bus 52, 4p.)(Bus 17, 4p.)(Bus 810, 4p)(Tram 10, 1p.)(Bus 58, 1p.)(Tram 6, 2p)	
57.713797	11.9679601	14:13:23	No vehicles nearby	(Bus 52, 4p.)(Bus 17, 4p.)(Bus 810, 4p)(Tram 10, 1p.)(Bus 58, 1p.)(Tram 6, 2p)	
57.7134118	11.9684534	**14:13:29**	**Bus 16**	(Bus 16, 5p.)	1 min 50 sec
57.7132108	11.9687159	14:13:32	**Bus 16**	(Bus 16, 10p.)	
57.7127905	11.9692461	14:13:38	**Bus 16**	(Bus 16, 14p.)	
57.7123633	11.9695107	14:13:44	Bus 58, Bus 16, Rosa express, **Bus 16**	(Bus 16, 18p.)	
57.7121719	11.9695824	14:13:47	**Bus 16**, Rosa express	(Bus 16, 23p.)	
57.711774	11.9696317	14:13:53	Gron express, Gron express, **Bus 16**	(Bus 16, 28p.)	
57.7113449	11.9695672	14:13:59	**Bus 16**	(Bus 16, 33p.)	
57.711143	11.9695076	14:14:02	**Bus 16**	(Bus 16, 38p.)	
57.7108752	11.9694442	14:14:08	**Bus 16**	(Bus 16, 43p.)	
57.710755	11.9694529	14:14:11	**Bus 16**	(Bus 16, 48p.)	
57.7105182	11.9695258	14:14:14	Gul express, Bus 16, **Bus 16**	(Bus 16, 53p.)	
57.7101607	11.9698753	14:14:20	**Bus 16**	(Bus 16, 57p.)	
57.7099842	11.9701103	14:14:23	Gron express, **Bus 16**	(Bus 16, 61p.)	
57.7100995	11.9700005	14:14:26	Gron express, **Bus 16**	(Bus 16, 66p.)	
57.7097093	11.9703639	14:14:30	**Bus 16**	(Bus 16, 70p.)	

Table 5.3: Journey 9, Friday May 17, 14:11

Journey 9, Friday May 17, 14:11 The passenger was traveling with bus 16, and seated in the back of the 20 meter long vehicle. In Table 5.3 it is visible that during the first two minutes the API was not representative and returned almost no vehicles. In reality, there was both oncoming traffic and vehicles in front and behind at times. Fortunately, after the first two minutes the API returned a much better representation of the nearby vehicles. Some of the other vehicles such as *Rosa express* and *Gron express* were noted to be present during the test, and the passenger vehicle was consistently in the response. It it therefore reasonable to disregard of the initial 1 minute and 30 seconds of the test. The reason is that the algorithm itself is tested, not the implemented system as a whole. During this initial period the API falsely claimed there were no nearby vehicles, and there is no point in evaluating the algorithm with inaccurate input data.

When the input data from time 14:13:12 and forward is fed as input to the algorithm, the correct response would be returned in 40 seconds (after 10 iterations, i.e. at time 14:13:53). The test is thus deemed successful in 40 seconds instead of 1 minute 50 seconds. The output is successful in any case, but it is safe to not include the

waiting time in the response time since it reflects the quality of the API and not the algorithm. When the algorithm is given accurate input, it also gives an accurate output.

y_p	x_p	time	Nearby vehicles	Top candidates / Result	pass/ fail
57.6961052	11.9713773	**09:42:53**	**Bus 16**	(Bus 16, 5p.)	
57.6967334	11.9708679	09:42:59	**Bus 16**	(Bus 16, 10p.)	
57.6972608	11.9705108	09:43:05	No vehicles nearby	(Bus 16, 10p.)	
57.6978581	11.9701041	09:43:11	No vehicles nearby	(Bus 16, 10p.)	
57.6982994	11.9698417	09:43:17	Bus 19, Bus 16	(Bus 16, 10p.)(Bus 19, 5p.)(Bus 16, 5p.)	
57.6982298	11.969886	09:43:23	Bus 19, **Bus 16**	(Bus 16, 15p.)(Bus 19, 10p.)(Bus 16, 5p.)	
57.6984285	11.9697637	09:43:29	**Bus 16**	(Bus 16, 20p.)(Bus 19, 10p.)(Bus 16, 5p.)	
57.6984688	11.9697436	09:43:33	Tram 3, **Bus 16**	(Bus 16, 25p.)	
57.6984971	11.9697236	09:43:35	Tram 3, **Bus 16**	(Bus 16, 30p.)	
57.6985429	11.9697003	09:43:38	Tram 3	(Bus 16, 30p.)	
57.6985471	11.9696992	09:43:41	Tram 3	(Bus 16, 30p.)(Tram 3, 20p.)	
57.69921	11.9692851	09:43:44	Bus 16, **Bus 16**	(Bus 16, 35p.)(Tram 3, 20p.)	
57.6992975	11.969252	09:43:47	Bus 16, **Bus 16**	(Bus 16, 40p.)(Tram 3, 20p.)	
57.6993143	11.9692336	**09:43:53**	**Bus 16**	(Bus 16, 44p.)	1 min
57.6993372	11.9692956	09:44:00	**Bus 16**	(Bus 16, 49p.)	
57.6993442	11.9693086	09:44:02	No vehicles nearby	(Bus 16, 49p.)	

Table 5.4: Journey 11, Monday May 20, 09:42

Journey 11, Monday May 20, 09:42 The passenger was traveling with bus 16, and seated in the middle of the 20 meter long vehicle. The API returned fairly representative nearby vehicles, such as an oncoming bus 16 and the behind tram 3. A successful output was returned after 1 minute.

5.2 Failed Tests

y_p	x_p	time	Nearby vehicles	Top candidates / Result	pass/ fail
57.7103493	11.94192	**10:59:38**	Bus 99	(Bus 99, 4p.)	
57.7104582	11.9421576	10:59:41	Bus 58, Bus 99	(Bus 99, 9p.)(Bus 58, 1p.)	
57.7106038	11.9425084	10:59:47	Bus 58, Bus 99, **Bus 16**	(Bus 99, 14p.)(Bus 58, 6p.)(Bus 16, 5p.)	
57.7105956	11.9425038	10:59:52	Bus 58, Bus 99, **Bus 16**	(Bus 99, 19p.)(Bus 58, 11p.)(Bus 16, 10p.)	
57.7106042	11.9425099	10:59:56	Bus 58, Bus 99, **Bus 16**	(Bus 99, 24p.)(Bus 58, 16p.)(Bus 16, 15p.)	
57.7106057	11.9425053	11:00:02	Bus 58, Bus 99, **Bus 16**	(Bus 99, 29p.)(Bus 58, 20p.)(Bus 16, 20p.)	
57.7106047	11.9425039	11:00:08	Bus 99	(Bus 99, 34p.)(Bus 58, 20p.)(Bus 16, 20p.)	
57.7106037	11.9425049	11:00:12	Bus 99	(Bus 99, 39p.)(Bus 58, 20p.)(Bus 16, 20p.)	
57.7106044	11.942505	11:00:17	Bus 99	(Bus 99, 44p.)(Bus 58, 20p.)(Bus 16, 20p.)	

Table 5.5: Journey 7, Friday May 17, 10:59

Journey 7, Friday May 17, 10:59 The passenger was traveling with bus 58, and seated in the back of the 12 meter long vehicle. The API included all present vehicles (bus 58, bus 16 and bus 99). The issue is that the behind bus 99 was returned in every API response, but the passenger vehicle bus 58 was only returned half of the responses. Because of this, the algorithm returned the wrong output.

y_p	x_p	time	Nearby vehicles	Top candidates / Result	pass/ fail
57.7039431	11.9697107	**16:46:29**	**Tram 7**, Tram 5, Tram 4, Tram 3	(Tram 7, 4p.)(Tram 5, 4p.)(Tram 4, 4p.)(Tram 3, 4p.)	
57.7039687	11.9696878	16:46:32	**Tram 7**, Tram 5, Tram 4, Tram 3	(Tram 7, 8p.)(Tram 5, 8p.)(Tram 4, 8p.)(Tram 3, 8p.)	
57.703933	11.9697159	16:46:38	Bus 52, **Tram 7**, Tram 5, Tram 5, Tram 4, Tram 3	(Tram 7, 13p.)(Tram 5, 13p.)(Tram 4, 13p.)(Tram 3, 13p.)(Bus 52, 4p.)(Tram 5, 4p.)	
57.703962	11.9698647	16:46:44	Bus 52, **Tram 7**, Tram 5, Tram 5, Tram 4, Tram 3	(Tram 7, 18p.)(Tram 5, 18p.)(Tram 4, 18p.)(Tram 3, 18p.)(Bus 52, 9p.)(Tram 5, 9p.)	
57.7037543	11.9702381	16:46:47	Tram 5	(Tram 7, 18p.)(Tram 5, 18p.)(Tram 4, 18p.)(Tram 3, 18p.)(Tram 5, 13p.)(Bus 52, 9p.)	
57.7037673	11.9700631	16:46:50	Tram 5	(Tram 7, 18p.)(Tram 5, 18p.)(Tram 4, 18p.)(Tram 3, 18p.)(Tram 5, 18p.)(Bus 52, 9p.)	
57.7036624	11.9703668	16:46:53	Tram 5	(Tram 5, 22p.)(Tram 7, 18p.)(Tram 5, 18p.)(Tram 4, 18p.)(Tram 3, 18p.)(Bus 52, 9p.)	
57.7034146	11.9705598	16:46:59	No vehicles nearby	(Tram 5, 22p.)(Tram 7, 18p.)(Tram 5, 18p.)(Tram 4, 18p.)(Tram 3, 18p.)(Bus 52, 9p.)	
57.7031341	11.9710049	16:47:02	Bus 50	(Tram 5, 22p.)(Tram 7, 18p.)(Tram 5, 18p.)(Tram 4, 18p.)(Tram 3, 18p.)(Bus 52, 9p.)(Bus 50, 4p.)	

Table 5.6: Journey 6 Thursday May 16, 16:46

Journey 6 Thursday May 16, 16:46 The passenger was traveling with tram 7, and seated in the back of the 30 meter long vehicle. Tram 7. API returned reasonable vehicles, such as Tram 7, 5, 4 and bus 52 which were all nearby. The algorithm failed to return the correct vehicle as a result of the correct vehicle not being in the input. This could be an effect of the passenger being 30 meters away from the vehicle GPS transmitter. Although, the bounding box was configured to have sides of 2×60 meters, meaning that both passenger and vehicle would have to have 15 meters of GPS error in opposite directions in order for the vehicle to be missing in the response. See Figure 5.3 illustrating this. Given that GPS errors average around 5-8 meters[5], there must be another explanation why the nearby vehicles so frequently are missing in the response.

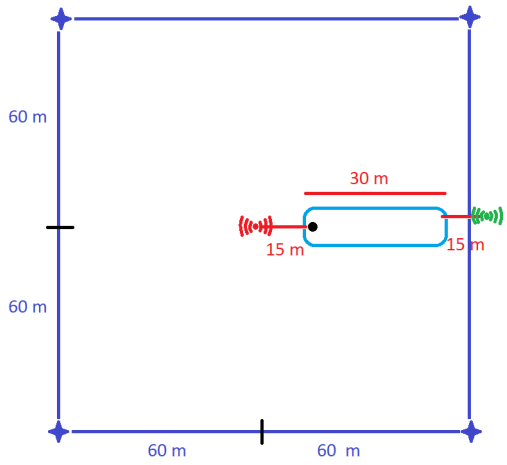

Figure 5.3: Both the passenger (black) and the vehicle (blue) would have to have an error of 15 meters in opposite directions to explain why the vehicle was not included in the API response. Given that GPS errors average around 5-8 meters, there must be another explanation why the nearby vehicles so frequently are missing in the API response.

There are many more failed tests that are not included in the report. They are about 5-10 times as many as the successful tests. The reason why all of them have been omitted is that the nearby vehicles returned from the API were constantly not representative of the reality. There is no point in evaluating a test of the algorithm if the input data in invalid. The errors in the API are discussed further in Section 6.1.2

5.3 Key Performance Indicators

Average response time The successful tests returned the correct vehicle in less than a minute. It took 40 seconds in journey 10, journey 8 and journey 9. It took 60 seconds in journey 11. Although, due to the unreliable input data, tests could not be performed to such an extent that an average response time can be calculated.

Time complexity The algorithm has linear time complexity in regards to the number of vehicles. This holds both for a naive and a sophisticated solution to fetch nearby vehicles as discussed in Section 4.4. The naive solution has one loop, and the algorithm itself has three consecutive loops meaning that the time complexity is linear in respect to the data iterated over.

Accuracy of the output, as a percentage of the performed tests Since not enough tests could be performed, this KPI cannot be evaluated. This is a consequence of the implementation using an unreliable API, which will be discussed in Section 6.1.2.

6

Discussion

The tests show that the algorithm works, if the input data is valid. The issue in the implementation is that the API used for retrieving nearby vehicles often returns the wrong vehicles. A prerequisite for the algorithm to work, is that the passenger vehicle is one of the nearby vehicles. The errors in the API are analyzed in Section 6.1.2. Another prerequisite is likewise that the passenger data is satisfactory valid, which is confirmed below in Section 6.1.1.

The tests that were successful returned the correct vehicle in less than a minute. This is an acceptable response time for the applications, even for the trip planner which is the application with the most strict requirement for response time. The response time in a trip planner should not be longer than 30-60 seconds in order to be better than if the passenger manually makes a new query in a regular trip planer. It also has to be fast enough to make correct predictions even if the passenger just travels one stop. 60 seconds is deemed an acceptable time (counting from when the passenger data begins to be collected, to when the name of the vehicle is returned). However it is not excellent. To improve the response time other kinds of data probably need to be utilized as well as GPS location. The other kinds of data are discussed in Section 6.2

The difficult cases described in the Background Section 2.3, can either definitely or presumably be solved with the proposed algorithm. The case when a passenger is walking inside the vehicle, is easily solved by the algorithm by performing a number of iterations which allows some time to pass such that the passenger will move together with their vehicle. The case of a junction has proven to be solved even for an implementation that only uses GPS location. This is clear from examining journey 10 in Chapter 5. Bearing is not needed, because a vehicle traveling in opposite direction will not be close to the passenger for long enough such that it risks becoming the top candidate. Similarly, speed is not needed for the case of a junction. Since the algorithm will run for a few iterations, the passenger will be consistently closer to vehicle that travels with the same speed (and in the same direction) and any vehicle driving slower or faster will drag behind or drive away from the passenger.

As for the case of a queue, particularly where the passenger is seated in the back and another vehicle is behind, the hypothesis is that if speed and acceleration are included as input it is possible to solve. Both vehicle GPS transmitters are within

the length of a vehicle and will get equal points based on distance (disregarding any noise). However, only the vehicle where the passenger is on board will get a high vote from keeping the same speed and acceleration as the passenger at each snapshot. Consequently, after some iterations the passenger vehicle will be consistently more likely than the one behind. Still, it is not certain that this case of a queue can be solved this way, since no tests have been performed to confirm the hypothesis. If the speed or acceleration data is too unreliable or suffers from time delays between vehicle and passenger it will be difficult to get any match at all. Furthermore, if the threshold for a matching speed or acceleration is too generous, both queued up vehicles will match the passenger due to the nature of a queue. That is, both will be within the passenger bearing ± some degrees, and the same for speed. Most certainly other kinds of input data should be inputted in the algorithm and tests should be performed on queues to verify this hypothesis.

6.1 Sources of Error

The sources of error are deemed to be invalid input data to the algorithm. The input data is: the passenger GPS location and time, the name of the nearby vehicles, and for each of the vehicles; the GPS location, time, name of the line and a trip id. The errors on the passenger side and vehicle side respectively are discussed below.

6.1.1 Passenger Data Errors

The data was validated through tests and proven to be satisfactory accurate. During a bus journey, the running app was video recorded as it displayed GPS locations on screen. The on-screen locations are referred to below as *the measured GPS locations*. These can be compared to what is referred to as the *the correct GPS locations*, which is the best estimate of the passenger's actual location at the time when the measured location was displayed. *The correct GPS locations* could be estimated by comparing the road surface markings, signs etc. outside the bus window in the video recording with aerial and street images in the Google Maps mapping service which also provides GPS locations (Source: Google Developers, Google Maps, https://www.google.com/maps/about/, 2019). The comparisons between *measured GPS locations* and *correct GPS locations* can be seen in Table 6.1.

It should be mentioned that the Google Maps mapping service is not completely accurate [13] [14]. It has been shown that the mapped location might be about 10 meters off from the real GPS position. Hence, the "correct" locations in Table might actually have errors of several meters. This means that the measured errors from journey 3 might be even larger. On the contrary, the errors could in fact be smaller instead, since the errors in Table 6.1 might be entirely a result of the errors in the Google Maps mapping service. Thus, the GPS error measured from journey 3 is just an indication of the *size* of potential errors. In either case, the impact

errors of this size has on the vehicle matching algorithm is very small. The reason is that, as mentioned before, the problem is not about determining the whereabouts of the passenger or the vehicles, instead it is about matching the objects that travel together. Furthermore, the algorithm is designed to handle errors in the input data. The test results in section 5 show that the algorithm successfully is able to do so.

Another weakness in this test for passenger data errors, is the small data set. The 14 data points are collected from one street alone, where the GPS errors are very similar. It does therefore not represent the error in other areas very well. However, during the course of the project, many more data points in other areas were cross checked and proven to be equally accurate. Thus, this test can safely be interpreted as a indication of the size of passenger GPS error.

The test concluded that the error on the passenger GPS locations varied between 1.4 and 10.8 meters from the correct location. The mean error is 4.8 meters, with a standard deviation of 2.9 meters. This is a manageable size of the errors for the algorithm to be able to output the right vehicle. The median error of the measured data point from the journey is 3.9 meters. This is even better than the median smartphone GPS error of 5 to 8.5 meters as concluded in the 2011 empirical study [5], and hence the voting system could be configured accordingly. The GPS locations are visualized on a map in Figure 6.1. The data is shown in Table 6.1 below.

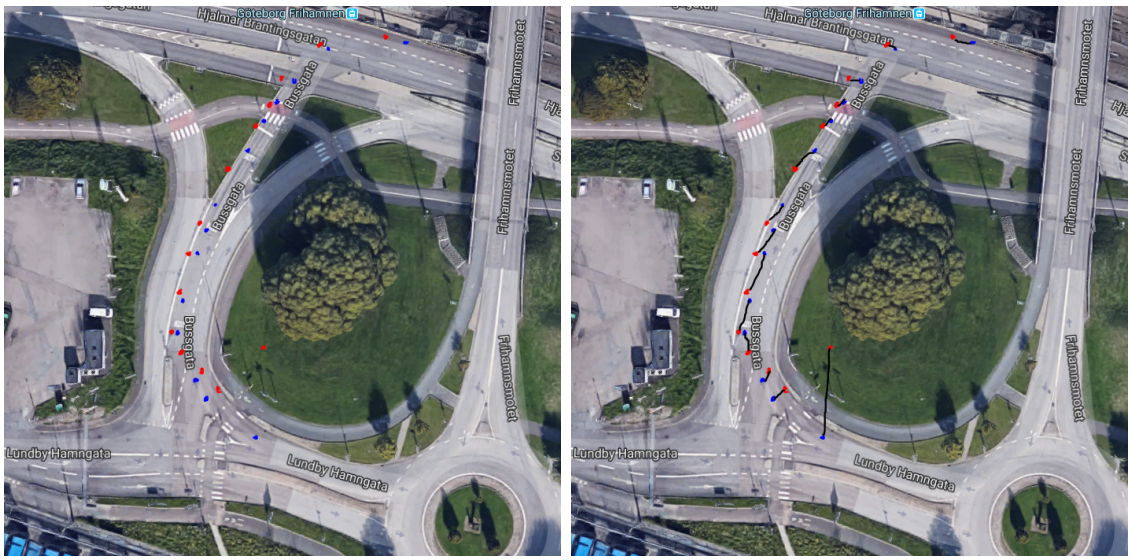

Figure 6.1: GPS locations from journey 3, displayed on a map. Blue dots mark the correct location and red dots the measured location from the app. Black lines mark the error between correct- and measured locations from the same time. *Source: Google Developers, Google Maps, https://www.google.com/maps/about/, 2019*

Table 6.1: The data from journey 3. Collected Thursday Mar 21 from 17:04:08 to 17:04:26. The Error is given in meters between the measured GPS coordinate and the real GPS coordinate of the passenger at the time the data was displayed in the app.

Time	17:04:08 (*1:st*)	17:04:10	17:04:11
Measured	(57.71962, 11.95997)	(57.71956, 11.95981)	(57.71958, 11.95975)
Correct	(57.71946, 11.95993)	(57.71953, 11.95980)	(57.71955, 11.95975)
Error	17.8 m	3.1 m	3.6 m
Time	17:04:12	17:04:13	17:04:14
Measured	(57.71962, 11.95970)	(57.71966, 11.95968)	(57.71971, 11.95968)
Correct	(57.71965, 11.95968)	(57.71970, 11.95969)	(57.71981, 11.95973)
Error	3.9 m	5.0 m	10.8 m
Time	17:04:15	17:04:16	17:04:18
Measured	(57.71977, 11.95970)	(57.71984, 11.95975)	(57.71994, 11.95983)
Correct	(57.71984, 11.95975)	(57.71989, 11.95979)	(57.71999, 11.95990)
Error	8.3 m	5.9 m	6.2 m
Time	17:04:20	17:04:21	17:04:22
Measured	(57.72003, 11.95994)	(57.72007, 11.95998)	(57.72011, 11.96002)
Correct	(57.72005, 11.95997)	(57.72008, 11.96000)	(57.72012, 11.96006)
Error	2.4 m	1.4 m	2.2 m
Time	17:04:24	21 17:04:26	
Measured	(57.72018, 11.96016)	(57.72020, 11.960355)	
Correct	(57.72017, 11.96019)	(57.72018, 11.96046)	
Error	2.5 m	7.1 m	
Mean error	5.7 m	*Mean, excl.1:st*	**4.8 m**
Median error	4.5 m	*Median, excl.1:st*	**3.9 m**
Std.Dev.	4.3 m	*Std.Dev., excl.1:st*	**2.9 m**
$Mean \pm Std.Dev.$	[1.4, 10.0] m	$Mean \pm Std.Dev.$	**[1.9, 7.7] m**

An interesting side note is that the first data point *always* has a large error, and it is therefore discarded in the implementation. The first location appears instantly when starting the app, and it seems to not be using GPS, but instead only Wi-Fi and cellular sources. The assumption is based on the fact that the size of the errors are as large as when configuring the app to not use GPS but only other sources. The hypothesis is that the Android Location library quickly gives a rough initial approximation, before computing the location with GPS which is more time consuming. The first data point is therefore ignored in the implementation.

The measured GPS location, marked red in Figure 6.1, is consistently behind *the correct GPS location*, marked blue. This might be a result of a time delay from when satellite signals are retrieved on the phone, to when the coordinates have been computed and through the Android Location library and sent and displayed

in the app, as *the measured GPS location*. During this time, the passenger can have traveled away from the location where the satellite signals were retrieved, to another location which is then considered *the correct GPS location*. It means that the mean error of 4.8 meters might be in part due to this time delay. However, what matters to the algorithm is the relation between a passenger timestamp and a vehicle timestamp from the API, not the actual whereabouts of the two. As with the passenger side, there is no validation on the vehicle side of the time delay for the locations from the API. What can be concluded from the test is that the error of a passenger location is small and in worst case a few meters.

6.1.2 Vehicle Data API Errors

The vehicle locations were initially presumed to be very accurate, and indeed "real time positions" as the API documentation claimed [7]. However, after having performed the tests it was evident that the API does not accurately return the nearby vehicles, and presumably does not know the real time positions of the vehicles. Therefore a verification test of the API was executed. It confirmed that the API is very unreliable and has large errors of up to hundreds of meters.

The test was performed by a person situated on a hill and overlooking a bus stop. The bounding box for the API surrounds the bus stop and is indicated with a blue box with stars marking the coordinates in Figure 6.2. The person's location is indicated with a blue arrow in the figure. At a given time the vehicles inside the bounding box were recorded, and at the same time a query was sent the API requesting the vehicles inside the box. The vehicles returned from the API were then compared to the actual vehicles inside the bounding box.

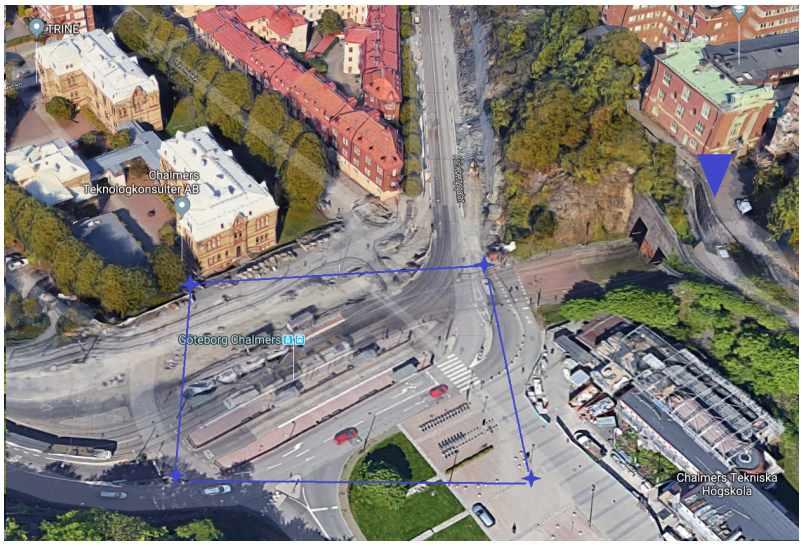

Figure 6.2: How the API verificaton test was performed. A person on a hill, marked by the arrow, monitored a bounding box, marked by the square with star corners for bounding coordinates, lower left: (11.972594, 57.689718) and upper right: (11.973667, 57.690267). The API response could then be compared with the actual vehicles in the box. *Source: Google Developers, Google Maps, https://www.google.com/maps/about/, 2019*

The results from the test are summarized in Table 6.2 in three categories; accurate, missing and extra, described in the table text. For half the tests the *onlyRealtime* parameter was set to 'yes' (Table 6.3), and for the other half to 'no' (Table 6.4). The *onlyRealtime* parameter is defined in the API documentation as "*Can be used to define whether all vehicles should be returned or only those vehicles which have realtime information. If it is set to "yes", only vehicles with realtime information are returned, if it's set to "no", all vehicles in the bounding box are returned.*" [7]. It was mistakenly interpreted as if set to yes, only real time positions of vehicles should be returned, and if set to 'no', estimated positions based on time table could also be included.

The test concludes that the API used in the implementation is too unreliable. The main issue are all the nearby vehicles that are missing in the API response. It is about three times likelier that a vehicle in the bounding box will not be returned as that it will. The probability for *missing* is $\frac{23}{30}$ compared to $\frac{7}{30}$ for *accurate*, when *onlyRealtime*=yes ($\frac{22}{29}$ and $\frac{7}{29}$ for 'no'). The consequence for the implementation is that the vehicle the passenger travels with is rarely returned as a candidate. This input data makes it impossible for an algorithm to reliably match a passenger with their vehicle. In addition, the extra vehicles returned from the API that are not present within the bounding box, are instead likely to be returned, which can result in the algorithm outputting the wrong vehicle. The poor data validity of this API is the reason why many tests failed.

Table 6.2: API verification result summary. Every row represents a query to the API and is compared to the vehicles actually present in the bounding box.
Accurate - The number of vehicles inside the box also returned from the API.
Missing - The number of vehicles inside the box not returned from the API.
Extra - The number of vehicles returned from the API but that weren't in the box.

Table 6.3: onlyRealtime=yes

Test no.	Accurate	Missing	Extra
1	I	I	I
2		I	
3		II	I
4	I	I	I
5	I	I	II
6		III	
7		I	
8		II	
9		II	
10		III	I
11	I	I	I
12	II		I
13		III	
14		I	I
15	I	I	

Table 6.4: onlyRealtime=no

Test no.	Accurate	Missing	Extra
16	I	I	
17		III	
18		II	
19	I		
20	I		I
21		I	
22		I	
23		I	I
24		III	I
25	II	I	I
26		III	
27		III	
28	I	II	
29	I	I	

The reason why the API returned such invalid data was realized after having contacted the support on the developer forum for the API, i.e. the public transport provider Västtrafik [15]. They stated that the API has no information of the exact coordinates of the vehicles. What it instead knows is if a vehicle is delayed from the time table. From this information is computes where the vehicle should be, e.g. 2 minutes away from a certain stop, and translates this into a GPS location. This is what is meant by "real time positions" in the documentation. As a result, if there is a long distance between two stops, if there are red lights or if there is traffic, the vehicle GPS location is very unlikely to be within the 60×60 bounding box of the API call. What the *onlyRealtime* parameter does is limit the display of vehicles to those that correctly reports their position to the back-end system and gets an updated time table. Otherwise the static time table estimates are displayed for each vehicle. [15]

There is a back-end system that keeps track of the actual GPS locations of all vehicles in real time (E. Lundin, Consat Telematics, Personal interview, February 4 2019). It logs a position at every turn a vehicle makes and matches the positions to the road map as previously mentioned. This system is maintained by *Consat Telematics AB* but has no open API that can be used for this implementation. The *Västtrafik Reseplaneraren v2 Livemap* API does not use these locations directly.

Instead the Livemap API only use the static time table and updated time table with any deviations.

6.2 Kinds of Data

The implementation only use GPS data and tests show that the algorithm matches a passenger with the vehicle they travel with in less than a minute. In order to shorten the response time and increase the accuracy of the output, other kinds of data must also be included. Suggestions are discussed below.

6.2.1 GPS Location

The GPS location is considered to be somewhat mandatory for the algorithm to work, since no other kind of data is as effective for sorting out a small set of candidates from the entire collection of vehicles in the city. The only exception is Wi-Fi Positioning System which might work alone as discussed in Section 6.2.7. The general accuracy of GPS data is a few meters which has proven to be accurate enough for the algorithm to work. Moreover, location data is generally available in any smartphone development and any telematics system, which makes it suitable for implementation. Only latitude and longitude are needed, and not altitude, unless the system is implemented in an area with a considerable amount of road bridges.

An interesting fact about GPS errors, is that the error is likely to be the same for objects that are at the same location. The reason being that GPS errors are usually caused by factors related to atmospheric conditions and the location of the receiver, such as satellite signal blockage or signal reflection both caused by nearby buildings, trees or similar structures [16]. It means that even if GPS errors are considerable, it might not be an issue as long as both passenger and vehicle have the same error, and thus are in the vicinity of each other. Although, this is perhaps not likely, partly because the body of the vehicle itself blocks out satellites for the passenger and might not do so for the vehicle, but in the Västra Götaland region the vehicle GPS transmitter is also fitted inside the body. Furthermore, it is also partly because the GPS hardware can be superior on either the passenger or vehicle side, such that they will reach different number of satellites and have different errors. And finally, partly since there can be a distance the size of a vehicle length between the passenger and vehicle transmitter, such that the signal blockage can still be different for passenger and vehicle.

6.2.2 Vehicle Length

A weakness in the implementation, is that nearby vehicles get points based on their distance to the passenger, regardless of the direction of travel or the length of the

vehicle. In reality, if a passenger is 30 meters away from a 20 meter long vehicle, there is no way the passenger can be on board that vehicle. A simple condition that the distance cannot be longer than the vehicle itself plus some error will improve the point system. The length can be implemented based on the particular vehicle in service, the line, or simply the type of vehicle such as bus or tram.

It ought to be possible to define an interval where passengers of a vehicle can have their locations, based on bearing and vehicle length. This can be done by letting the coordinates of the front and rear of the vehicle serve as min and max values of a bounding box around the vehicle, as depicted in Figure 6.3. The coordinate of the front is simply the location from the GPS transmitter. The coordinate of the rear of the vehicle is obtained by offsetting the front location, with the length of the vehicle, in the opposite direction of travel. This method limits the box orientation parallel to the longitude and latitudes, which in worst case captures extra space when the vehicle is traveling diagonally in respect to longitude and latitude, as seen in the middle image of Figure 6.3. Even in this case, the bounding box will still be more accurate than only checking the distance to the passenger as illustrated to the right in the same figure. In summary, some consideration to various vehicle lengths will definitely improve the point system since fewer nearby vehicles will get points they do not deserve.

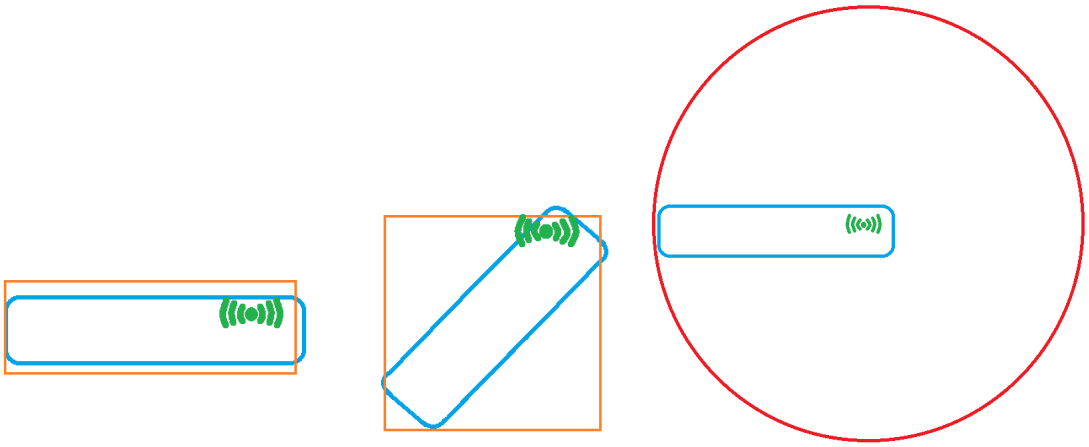

Figure 6.3: From the location of the vehicle, the direction of travel and the length of the vehicle, a bounding box surrounding the vehicle can be obtained. Such a bounding box is marked orange in the left figure, with blue vehicles with green GPS transmitters. Such a bounding box will always be better than simple comparison of distance alone, effectively a radius, as seen in the right figure.

6.2.3 Bearing

Bearing can be used together with vehicle length as mentioned above. Apart from that, bearing is not considered to improve the algorithm much in the general case. When a vehicle moves in a different direction than the passenger, it will not be close to the passenger for long. Hence, after a few iterations, it will not have enough

points to become the top candidate.

Although, in the case of a queue the bearing can possibly differentiate between the passenger vehicle and the vehicle behind/in front. One vehicle will turn before the other, and the passenger will shift their bearing in sync with the vehicle they travel with (that is, unless the vehicle is very long and possibly jointed). Another case where bearing is useful is if the passenger and nearby vehicles are stationary, such as in a terminal or a stop. It is not ideal to match a stationary passenger simply with the nearest vehicle, due GPS error and the potential distance between passenger and vehicle GPS transmitter. Instead it could be useful to compare the compass direction that a vehicle is oriented, i.e. the last known bearing. If the passenger was traveling with a vehicle in a certain direction and then stopped, the vehicle the passenger is on board must also be parked in that direction.

The reason why bearing is not implemented in this project is that for the passenger, only the phone orientation could be obtained, and not the bearing. It can naturally be calculated from the deltas of two consecutive GPS positions, but it was considered unnecessary since vehicles traveling in the wrong directions would not accumulate a high vote anyway. It would nevertheless be interesting to investigate the effect bearing can have in the case of a queue or a stationary passenger.

6.2.4 Speed

Speed has great potential to improve the algorithm. A passenger will generally have the same speed as the vehicle they travel in (unless e.g. a long vehicle makes a turn). In the case of a queue or a congested area such as a junction, speed is thought to filter out vehicles that are nearby but not likely to carry the passenger.

It is particularly interesting to test the performance when using speed in a case where the passenger vehicle is followed by another vehicle, not in a queue, simply on the same route. This is a common scenario, and because of driver independence and traffic lights, crossroads and similar, vehicles are often a few meters apart as well as driving at different velocities for given times. A way to handle this case is to check the following motion condition in the voting system. It also use bearing, which is optional and might not improve the performance. The condition is satisfied when the passenger p and vehicle v_i are considered to travel with the same motion vector. Here, s is the speed in km/h, b is the bearing in degrees from true north, and constants S and B define the interval where two speeds or bearings are considered the same.

$$C_{motion}(s_{v_i}, s_p) = (s_{v_i} - s_p \leq S) \wedge (b_{v_i} - b_p \leq B) \tag{6.1}$$

Through the Android Location library[6], used for passenger location, the speed of the passenger can be obtained. Initial tests showed that it match the speed on the speedometer of the passenger vehicle. Any time delay between the registered speed

and the speedometer did not seem larger than tree seconds (the sample rate). The reason why speed is not used in the implemented algorithm is solely because it is not available for the vehicle.

Vehicle speed can not be collected through the API, and neither computed from the non-real-time locations. Before realizing the flaws of the API, it was still considered if the vehicle speed should be computed from GPS location deltas and timestamps. It was decided not to, due to the quickly shifting nature of vehicle speed. Even if locations are sampled every half second, the calculated speed will not represent the actual momentary speed unless it is very consistent (i.e. not accelerating or decelerating much). More importantly, the GPS error is far too large for an accurate momentary speed to be calculated from it. Most certainly, a phone relies on the accelerometer not the GPS unit to determine speed. Lastly, the server collecting vehicle locations is not updated frequently enough to provide speed. It will have to be logged in the vehicle and transmitted with a timestamp in order to be comparable with the passenger speed.

6.2.5 Acceleration

Acceleration is considered difficult to implement reliably, since it can vary a lot only seconds apart. It puts constraints on the data collection for both passenger and vehicles to be synchronized, such that two accelerations can be compared. If there is even a minor time delay the momentary acceleration is likely to have changed. With that said, if it can be implemented reliably it has perhaps even greater potential to solve even difficult cases such as queues. To implement a point system that use acceleration, traffic data should beneficially be analyzed such that the right thresholds and conditions are put in place.

6.2.6 Planned Itinerary

If a passenger has searched in the trip planner to go from Central station to the Botanical Garden, and perhaps even chosen the departure with tram 1 at eight o' clock, it is likely the passenger is on board that vehicle. If the algorithm is used for a trip planner this information can surely improve the performance of the algorithm. A simple way to include it in the voting system is to give an additional bonus point if the vehicle is part of the itinerary. It should be configured such that is doesn't take the upper hand. If a passenger missed their departure but caught another vehicle right behind, it would be useful if the trip planner could announce it to the passenger such that the arrival at the right time and place can be ensured.

In contrast to what was just said, if the trip planner does mainly rely on planned itinerary, it will consume far less battery and cellular data. It can use GPS data merely to cross-check that the passenger has departed and sticks to the route by examining some sparse locations. Furthermore, it might be easier to launch the app

on the market too, since it is only a slight upgrade to existing applications, meaning the chances might be higher that a user permits the app to use their data. What is more, if the app only makes rough estimates the users might not perceive it as though they are being monitored, which could increase the chances they will use the app.

6.2.7 Wi-Fi Positioning System

In the Västra Götaland region there is still no Wi-Fi on the vehicles (only 2 routes are exceptions). Thus it is not feasible to implement an algorithm that takes so called Wi-Fi positions as input data. However, it has great potential of solving the problem altogether, even without additional input data. The Wi-Fi Positioning Systems (WPS) use nearby access points of a user to determine their location. The fingerprinting method relates the mac address of each access point to a geographical location, thus, if the user is near the access point they are also near the geographical location [17][18]. Additionally, the signal strength can be used to estimate the closeness.

The difference in this problem is that the vehicles are moving. The intended solution is thus to relate the mac addresses to the name of a line (or a vehicle id which in turn is related to a trip or a line). For the solution to work the vehicle access points can not have too far reach, such that it is impossible to filter out the right vehicle from all nearby access points. In that case additional input data is still needed. The reason why WPS might be superior to other kinds of data is that the sources of errors are low due to the short chain of information: from a piece of hardware in the vehicle directly to the passenger phone. No satellites, complex computations, time delays, momentary states or other sensitive relations, just a simple signal between two devices. The only drawback is that is requires the passenger to have Wi-Fi activated on their phone and that vehicles need to have Wi-Fi.

6.3 Feasibility and Ethical Discussion

The ethical aspects of this algorithm as well as the market demand and how they are related will be discussed in this section. Societal beneficially applications, such as the pre-hospital application that can notify relatives if a passenger in involved in a traffic accident 2.2, have a very clear value. Other applications, such as an automatic payment system or a system that feeds large amounts of data to public transport providers does not necessarily provide enough value for them to be successful. The cost of sharing one's data is evaluated very differently for different people.

6.3.1 Ethical Discussion

There are many privacy aspects to be considered for any application that monitors its users. Many would be uneasy if a mobile app told them it knew where there were and what they were doing. However, not as many feel spied upon when using a mobile app for navigation. The reason as to why some mobile apps successfully utilize user data, and some raise headlines in newspapers is an important part to evaluate in this project. One hypothesis is that the customer value decides. If the user finds the app useful they will be more inclined to share their data as the price for the service. Some users value their data a lot more than others and will not share their data for any functionality.

A key strategy is transparency towards the users. Transparency in regards to what information is collected, as well as what can be inferred from it, and perhaps most importantly, what the data is used for. Several newspapers have investigated popular apps and the companies behind them [19][20]. Apps often claim that the user data is stored anonymously, but when analyzing the data is is rather easy to identify individuals. A typical example is, by logging GPS locations under a static id for an anonymous user, it is still rather easy to infer what is a home address and a work address. From this information a person can be identified. Subsequently their visits to hospitals, graveyards, schools, family members, stores, military facilities and other sensitive locations can be monitored, as well as a person's routines. Most users are not aware of to what extent they are being surveilled, which makes it even more uncomfortable. Because of this, the EU General Data Protection Regulation (GDPR) and similar data privacy regulations are enforced such that users can decide what data is collected and stored.

Apart from the Corporate Social Responsibility (CSR) of the company, if users feel they are in control of what they share, they will feel more safe and be more inclined to share their data. For them to actually share anything the must also either be unaware of the risks, or feel they can benefit from sharing their data. Many would agree that "personalized marketing" has a low value, or even a negative value when perceived as unwelcome, unpleasant and manipulative. On the contrary, a state-funded application that takes care of one's public transport payments and is distributed by the non-profit municipality, feels much more beneficial and welcome. In particular if they do not capitalize on one's data by selling it to third parties. If such decisions are made to protect the users, they should favourably be made aware of the consideration taken to establish trust.

In the case of a system that supplies information to the public transport provider on how people travel, the societal benefits and the integrity of the people weighs against each other. The passengers would benefit from the stops and lines being re-designed with their needs in focus. The local pollution levels and climate footprint could also potentially be reduced if the transport system improved. It can improve both in terms of planning the lines more fuel efficiently, and in terms of meeting the demand of those who do not travel with public transport an instead drive their car. However, these improvements imply great investments from the municipality and has

to be founded upon reliable facts. Thus, if not enough people provide information of their trips, these improvements will never take place. A solution for securely and privately storing the data, is to collect it under unique, random ID:s for each trip. Then it is not possible to link trips made by the same person and thus no personal information can be inferred. The gathered data will only be a collection of individual trips, such as Central Station - Botanical Garden. Furthermore, stops that are only used by a handful of passengers should not be included, to further guarantee that no one can be identified from the collected data. How the data is analyzed must also be regulated by the public transport company, and the confidentiality must be guaranteed in the same way as other public services (such as health care).

Section 4.5 elaborated on the architecture of the implemented algorithm. The main focus of the architecture is perhaps confidentiality of data. In the proposed architecture, the only information escaping the phone is unavoidably the coordinates for fetching the nearby vehicles. Through these queries the passenger is leaking information of their whereabouts. An initial idea for anonymizing the queries to generate random, anonymous ID:s for every query. Yet, the device IP address will still be accessible. A better solution is to also query for random vehicles and follow them on bogus trips. This will make if far more difficult to filter out the real trips and from them be able to identify passengers. It will have to be implemented with server capacity in mind as well as passenger integrity.

6.3.2 Market Demand

A trip planner will be useful if it feeds the passenger with information they would otherwise have missed and if it saved them time and effort. If it is useful enough, passengers are willing to give access to their data. The information system is mainly in demand of the companies that provide public transport services. They have strong incentives to reduce their fuel and labour costs. There is however probably no demand from the passenger side for such data to be collected. Thus the incentive for the passengers to share their data must come from other products or services such as a trip planner or payment system. If the vehicle matching algorithm is implemented and distributed in such another app, it might as well come as a package deal with the information being shared with the public transport provider. It can be compared to trips being logged in regular ticket machines to predict usage, where also merely destination A - B data is collected.

The demand for pre-hospital solutions is unlikely to be big enough that users would download an app to use when traveling with public transport, in the rare event that they would get in an accident. Just like the information sharing application, this would probably have to be made a package deal with some more appealing application.

As for the automatic payment system, a reasonable way to introduce it is as a complement to already existing payment methods. Is means that if the automatic system fails in monitoring the passenger, or if the passenger does not share their

data correctly, they will have to buy their ticket in another way. The benefit of the automatic payment system over traditional ones, is only that it will be simpler for the passenger. Simpler in the sense that they would not be bothered with zones and fares, registering connections, or the hassle of checking in and out at ticket machines. By making the payment system easier, is likely to yield more payments, and less problems with passengers not paying the fares. That being said, the need for ticket inspectors would still exist, but their work could be improved by comparing data on how many passengers are on board a vehicle, and how many that are paying.

6.4 Future Work

Using a better API is the most important future work. It should be an API of the kind described in Section 4.3, where nearby vehicles are queried based on a passenger location. The main requirement is that the API supplies real time locations for the vehicles.

More tests can be performed if a reliable API is in place. By performing tests the algorithm can be evaluated more reliably and KPI:s established. Especially test cases such as junctions, and in particular queues, can be performed to a larger extent. Additionally, tests can be made in both urban and suburban areas. If more tests are performed the algorithm and voting system can also be configured or fine tuned such that the output is even more accurate and faster.

Using other kinds of input data is the second most important future work, since it is interesting to see how it improve the result. In particular vehicle length, speed and acceleration would be interesting to implement, test and evaluate. It could possibly be interesting to examine what fault tolerance can be achieved in case of GPS outage, or lost reception and how the algorithm could handle these cases, at least before launching an application on the market.

7

Related Work

There is no prior research found on paring a passenger with the vehicle in which they are traveling. There is not even any identified research on detecting that moving objects are traveling together. Since this has not been done before, the study is very much needed. Needed because of the knowledge gap and because of the many applications that such a system can be used for. There is however similar research in this area. Especially, there are many computational geometry algorithms that appear to be useful since the data is of a geometric nature [21]. Some such techniques are covered below.

7.1 Voronoi Diagrams

Voronoi diagrams will for a set of fixed points in the plane, yield a map of the areas where each point is the closest [11], as illustrated in Figure 7.1.These diagrams are useful for the type of algorithms that tell you where the nearest gas station is. The difference in this thesis problem is that the points of interest, are moving objects. So, one might be temped to use Voronoi diagrams, but they are only useful for retrieving the nearest stationary object, and vehicles are not stationary. This makes the Voronoi diagrams less useful.

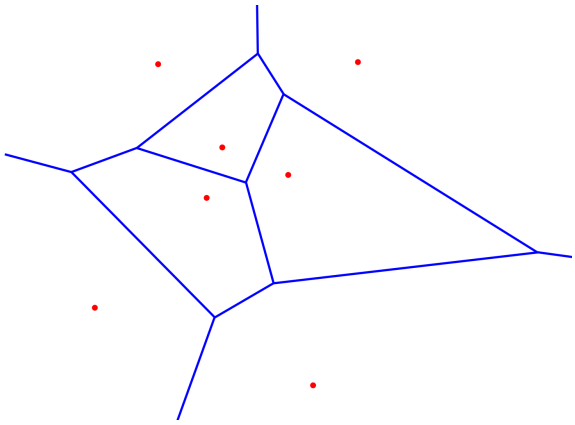

Figure 7.1: An example of a Voronoi diagram. Every position in an enclosed area will have the dot of that area as its closest dot. Thus, to find the closest dot, one need only figure out which area their position lies within. *source: Wikipedia, Wikimedia Commons Image Gallery, https://en.wikipedia.org/wiki/File:Voronoi_diagram.svg*

7.2 Interpolation

A method considered for the location data is linear interpolation, where lines are drawn between each data point to create a continuous function of data points. A more sophisticated version of this is spline interpolation, where a curve is fitted between each data point, which will produce a smooth curve from the set of data points [22]. A feature in the traffic management back-end system is that data is sampled at every turn the vehicle makes, as mentioned in Section 6.1.2. It suggests that linear interpolation would be more correct that spline interpolation.

These techniques could have been useful since vehicle and passenger data are sampled at various times. By creating continuous curves for each vehicle and passenger location, there are more possibilities for matching them. This seems all good, if it was not for the voting system being better. The voting system is simple and efficient, as described in Section 3.2, unlike interpolation. Interpolation is deemed unnecessarily complex since the suggested solution (see 3.1) solves the problem in a time and computationally efficient manner.

7.3 Map Matching

Another method considered is map matching where GPS locations are snapped to the nearest roads [23][24]. This technique is already used in the back-end for the vehicle data as mentioned in Section 6.1.2. The passenger data on the other hand is not map matched, meaning a passenger location can be in a river or building. The fact that vehicle data is filtered but not passenger data an be a source of error. If GPS errors would have been larger than they are, it might have been worth map

matching them, but that is not the case. Furthermore, the idea of map matching assumes that there is an interest in knowing the geographical locations of objects. In this project, the interest is to determine if two objects are at the same location, but no real interest in what that location is. Finally, the thesis solution provides an accurate output in a more efficient way than would have been possible if data should be map matched.

7.4 Dead Reckoning

Yet another method is dead reckoning. It is a mathematical model to infer from a known location, speed and bearing at a point in time, the locations at a next point in time [25]. This could be useful in in accounting for missing data points, such as in case of GPS outage or lost cellular reception [26]. However as explained in the Limitations section 2.4, the availability of data is assumed in this study, such that there is no need to calculate missing data points.

8

Conclusions

It can be concluded that the idea for the algorithm successfully solves the problem. Given a passenger and the right input data about them, as well as the right input data about the public transport vehicles, the algorithm will output which vehicle the passenger is traveling in. Additionally, the algorithm has the advantage that is can be implemented with various kinds of data. It can make use of GPS locations as well as speed or nearby Wi-Fi access points. Furthermore, how this data is compared to produce the output can be configured with weights in a points system. The algorithm can be summarized:

Algorithm 4 Matchmaking algorithm, summary

```
1:  list topVehicleCandidates
2:
3:  loop forever:
4:
5:      x, y ← getPassengerLocation ()
6:      vehicles ← fetchNearbyVehicles (x, y)
7:
8:      for every vehicle v ∈ vehicles do
9:          v.points ← v.points + getPoints(distanceToPassenger(x,y,v))
10:         topVehicleCandidates .add (v)
11:
12: Finally:
13: return getBest(topVehicleCandidates)
```

The algorithm gets the location of the passenger every few seconds. It will then query all the nearby vehicles. For every such vehicle, a number of points will be awarded. The points can be awarded based on distance to passenger, or based on other kinds of data collected. The algorithm will do a series of iterations with new passenger locations, and for every iteration update the list of top vehicle candidates. Finally, it will return the vehicle that consistently has appeared as a likely candidate.

Test results show that in less than a minute the algorithm will determine what vehicle a passenger is traveling in. A prerequisite is that the input data is representative for the reality. This was unfortunately not the case in many tests, due to an erroneous

API used for the implementation. These API errors will have to be mitigated in a commercial application. The implementation does notably only make use of GPS locations, and no other data. Thus, the response time could get even faster, and the accuracy even better, if other kinds of data are also used as input in the algorithm. Examples of useful kinds of data are:

- GPS location
- Vehicle length
- Bearing
- Speed
- Acceleration
- Wi-Fi Positioning System (WPS)
- Planned itinerary

This algorithm can be used in several applications. The key applications are an automatic payment system logging the trips of a passenger, and a trip planner which gives itineraries that are updated in real time based on the traffic situation and what the passenger does. When these applications reach a critical mass, the information of what trips people make can be used for the public transport providers to improve their service. It can even be used for pre-hospital services in case of traffic accidents. For the latter two applications, the passenger integrity is at stake. Any such application must be developed with consideration to privacy and both societal and personal benefit.

Bibliography

[1] the National Coordination Office for Space-Based Positioning, Navigation, and Timing, the U.S. Air Force, "Gps geodetic reference system wgs 84." https://www.gps.gov/multimedia/presentations/2009/09/ICG/wiley.pdf. Accessed: 2019-05-24.

[2] Y. Jonsson, S. Candefjord, and B. A. Sjöqvist, "Appen som själv larmar vid en mc-olycka." https://www.chalmers.se/sv/institutioner/e2/nyheter/Sidor/Appen-som-sjalv-larmar-vid-en-mc-olycka.aspx. Accessed: 2019-05-28.

[3] SOS Alarm, "Pilotprojekt för sensorlarm vid mc-olyckor." https://www.sosalarm.se/pilotprojekt-sensorlarm-mc/. Accessed: 2019-05-28.

[4] Göteborgs Spårvägar AB, "Vår flotta." http://goteborgssparvagar.se/om-oss/var-flotta/. Accessed: 2019-05-26.

[5] P. A. Zandbergen and S. J. Barbeau, "Positional accuracy of assisted GPS data from high-sensitivity GPS-enabled mobile phones," The Journal of Navigation, vol. 64, no. 3, pp. 381–399, 2011.

[6] Android Developers and Google Developers, "Android location." https://developer.android.com/reference/android/location/Location. Accessed: 2019-05-24.

[7] Västtrafik, "Västtrafik - utvecklarportalen. reseplaneraren v2, livemap." https://developer.vasttrafik.se/portal/{#}/api/Reseplaneraren/v2/landerss. Accessed: 2019-05-24.

[8] J. Kummer, "Earth radius by latitude calculator." https://rechneronline.de/earth-radius. Accessed: 2019-05-24.

[9] C. Veness, "Calculate distance and bearing between two latitude/longitude points using haversine formula." https://www.movable-type.co.uk/scripts/latlong.html. Accessed: 2019-05-24.

[10] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos, "Fast Nearest-Neighbor Query Processing in Moving-Object Databases," GeoInformatica, vol. 7, no. 2, pp. 113–137, 2003.

[11] Z. Song and N. Roussopoulos, "K-Nearest Neighbor Search for Moving Query Point," pp. 79–96, Springer, Berlin, Heidelberg, 2001.

[12] H. Mokhtar, J. Su, and O. Ibarra, "On moving object queries," in Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '02, (New York, New York, USA), p. 188, ACM Press, 2002.

[13] L. Pawlowicz, "Why are my gps positions in the wrong place in google earth?." https://freegeographytools.com/2007/why-are-my-gps-positions-in-the-wrong-place-in-google-earth. Accessed: 2019-06-13.

[14] W. Holder, "How accurate is google maps - wayne's tinkering page." https://sites.google.com/site/wayneholder/self-driving-car---part/how-accurate-is-google-maps. Accessed: 2019-06-13.

[15] Västtrafik Support, "Västtrafik utvecklarportalen - community topic 93." https://developer.vasttrafik.se/portal/{#}/community/topic/93. Accessed: 2019-05-27.

[16] U.S. Air Force, "Gps.gov: Gps accuracy." https://www.gps.gov/systems/gps/performance/accuracy/. Accessed: 2019-02-23.

[17] M. Lubbad, M. Z. Alkurdi, and A. AbuSamra, "Robust indoor wi-fi positioning system for android-based smartphone," International Journal of Research in Business and Technology, vol. 3, no. 2, pp. 159–162, 2013.

[18] N. Brachet, F. Alizadeh-Shabdiz, J. N. Nelson, and R. K. Jones, "Method and system for selecting and providing a relevant subset of wi-fi location information to a mobile client device so the client device may estimate its position with efficient utilization of resources," Feb 2013. US Patent 8,369,264.

[19] K. Örstadius and L. Larsson, "Appar spårar var du är – uppgifterna säljs öppet." https://www.dn.se/din-plats-till-salu/. Dagens Nyheter, Apr 2019. Accessed: 2019-05-07.

[20] J. alentino DeVries, N. Singer, M. H. Keller, and A. Krolik, "Your apps know where you were last night, and they're not keeping it secret." https://www.nytimes.com/interactive/2018/12/10/business/location-data-privacy-apps.html. New York Times, Dec 2018. Accessed: 2019-05-12.

[21] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, "Computational Geometry," in Computational Geometry, pp. 1–17, Berlin, Heidelberg: Springer Berlin Heidelberg, 1997.

[22] M. Christie and K. J. Moriarty, "A bicubic spline interpolation of unequally spaced data," Computer physics communications., vol. 17, no. 4, pp. 357–364, 1979.

[23] Y. Huabei and O. Wolfson, "A weight-based map matching method in moving objects databases," in Proceedings. 16th International Conference on Scientific and Statistical Database Management, 2004., pp. 437–438, IEEE.

[24] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang, "Map-matching for low-sampling-rate GPS trajectories," in Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems - GIS '09, (New York, New York, USA), p. 352, ACM Press, 2009.

[25] G. J. Geier, A. Heshmati, K. G. Johnson, and P. W. McLain, "Position and velocity estimation system for adaptive weighting of gps and dead-reckoning information," May 1995. US Patent 5, 416,712.

[26] D. M. Bevly and B. Parkinson, "Cascaded Kalman Filters for Accurate Estimation of Multiple Biases, Dead-Reckoning Navigation, and Full State Feedback Control of Ground Vehicles," IEEE Transactions on Control Systems Technology, vol. 15, pp. 199–208, mar 2007.

A

Source Code - Java Implementation

This appendix contains some code snippets from how the vehicle matching algorithm was implemented to be able to test the functionality. The code has been edited slightly for this report.

The Location activity class is the main class. It uses the Android Location library to retrieve the passenger location every third second. It will also call the method for fetching the nearby vehicles in another class. It will create a PassengerVehiclePairing object, to pair the passenger snapshot that initiated the query, with right the API vehicle response. This pairing is done since both the passenger location and the nearby vehicles are collected concurrently.

```
import android.support.v4.app.FragmentActivity;
import android.location.Location;
import com.google.android.gms.location.FusedLocationProviderClient;
import com.google.android.gms.location.LocationCallback;
import com.google.android.gms.location.LocationRequest;
import com.google.android.gms.location.LocationResult;
import com.google.android.gms.location.LocationServices;
import com.google.android.gms.location.LocationSettingsRequest;

public class LocationActivity extends FragmentActivity implements
  AsyncResponse{

  private LocationCallback locationCallback;
  private LocationRequest locationRequest;
  private FusedLocationProviderClient fusedLocationClient;
  private LocationSaver locationSaver;

  //unique for a passenger snapshot, to pair* API vehicle response
  private Integer passengerSnapshotIdCount = 1;
  //*passenger snapshot & list of nearby vehicles at that time
  private ArrayList<PassengerVehiclePairing> pairings =
    new ArrayList<>();
```

```
  ArrayList<Vehicle> topCandidates = new ArrayList<>();

@Override
  protected void onCreate (Bundle savedInstanceState){
    super.onCreate(savedInstanceState);
    fusedLocationClient =
      LocationServices.getFusedLocationProviderClient(this);
    locationCallback = new LocationCallback() {
      @Override
      public void onLocationResult(LocationResult
        locationResult) {
        if (locationResult == null) {
          return;
        }
        for (Location location : locationResult.getLocations()) {
          final Integer passengerSnapshotId =
            passengerSnapshotIdCount++;
          locationSaver.savePassengerLocation(location,
            passengerSnapshotIdCount);
          if (discardedCounter > DISCARD_INITIAL_LOCATIONS){
            final double lat = location.getLatitude();
            final double lng = location.getLongitude();
            Date time = new Date(location.getTime());
            pairings.add(new PassengerVehiclePairing
                (passengerSnapshotId, time, lat, lng));
            Integer[] boundingBox = getBoundingBox(lat, lng);
            fetchVehicles(passengerSnapshotId, boundingBox);
          }
          discardedCounter++;
        }
      }
    };
    createLocationRequest();
  }

  protected void createLocationRequest() {
    locationRequest = LocationRequest.create();
    locationRequest.setInterval(THREESECONDS);
    locationRequest.setFastestInterval(THREESECONDS);
    locationRequest.setPriority(
        LocationRequest.PRIORITY_HIGH_ACCURACY); //for GPS
    LocationSettingsRequest.Builder builder =
      new LocationSettingsRequest.Builder()
      .addLocationRequest(locationRequest);
  }
```

```
  private void startLocationUpdates() {
    try {
      fusedLocationClient.requestLocationUpdates(locationRequest,
          locationCallback, null);
    }
    catch (SecurityException e){}
  }

}
```

The following permissions are required in the AndroidManifest file and the following dependency is needed in the build.gradle file.

```
<uses-permission android:name=
  "android.permission.ACCESS_FINE_LOCATION" />
<uses-feature android:name=
  "android.hardware.location.gps" />
<uses-permission android:name=
  "android.permission.INTERNET" />
<uses-permission android:name=
  "android.permission.ACCESS_NETWORK_STATE" />
.
.
.
dependencies {
    implementation 'com.google.android.gms:play-services:11.0.4'
}
```

The GetNearbyVehicles class performs the HTTPS request for fetching nearby vehicles through the *Västtrafik Reseplaneraren v2 Livemap* API. It uses the OAuth 2.0 authorization protocol as can be seen in the generateToken method (OAuth 2.0: https://oauth.net/2/).

```
import android.os.AsyncTask;
import android.util.JsonReader;
import java.util.Base64;
import java.io.DataOutputStream;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.URL;
import javax.net.ssl.HttpsURLConnection;

public class GetNearbyVehicles extends AsyncTask<
  Integer, Void, ArrayList<Vehicle> > {

  private String apiKey = "21b461ac1fb....";
```

```
private String apiSecret = "ac1fb21b461...";
private String tokenType = "Bearer";
private String accessToken;

@Override
protected ArrayList<Vehicle> doInBackground(Integer... params){
  int passengerSnapshotId = params[0];
  int minx = params[1];
  int maxx = params[2];
  int miny = params[3];
  int maxy = params[4];
  ArrayList<Vehicle> vehicles = null;
  HttpsURLConnection connection;
  try {
    accessToken = generateToken(); //see below
    String endpoint =
      "https://api.vasttrafik.se/bin/rest.exe/v2/livemap";
    String query = "?minx=" + minx + "&maxx=" + maxx +
      "&miny=" + miny + "&maxy=" + maxy + "&onlyRealtime=yes";
    URL url = new URL(endpoint + query);
    connection = (HttpsURLConnection) url.openConnection();
    connection.setRequestProperty("Authorization", tokenType +
      " " + accessToken);
    connection.setRequestProperty("Accept-Charset", "UTF-8");

    if (connection.getResponseCode() == 200 ){
      InputStream responseBody = connection.getInputStream();
      InputStreamReader responseBodyReader =
        new InputStreamReader(responseBody, StandardCharsets.UTF_8);
      JsonReader jsonReader = new JsonReader(responseBodyReader);
      //Another method for parsing the JSON vehicle response
      vehicles = parseVehicles(passengerSnapshotId, jsonReader);
      jsonReader.close();
    } else{
      //Error handling
    }
    connection.disconnect();
  }
  catch (Exception e) {
    //Error handling
  }
  return vehicles;
}

public String generateToken (){
```

```
HttpsURLConnection connection = null;
try {
  URL url = new URL("https://api.vasttrafik.se/token");
  connection = (HttpsURLConnection) url.openConnection();
  connection.setRequestMethod("POST");
  byte[] myData = (apiKey + ":" + apiSecret).getBytes("utf-8");
  String keySecret64 = "Basic " + Base64.getEncoder()
    .encodeToString(myData);
  connection.setRequestProperty ("Authorization", keySecret64);
  connection.setRequestProperty("Content-Type",
    "application/x-www-form-urlencoded");
  connection.setRequestProperty("grant_type",
    "client_credentials");
  connection.setRequestProperty("scope", id);
  connection.setDoOutput(true);

  DataOutputStream outputStream =
    new DataOutputStream(connection.getOutputStream());
  outputStream.writeBytes(
    "grant_type=client_credentials&scope=device_1");
  outputStream.flush();
  outputStream.close();
  if (connection.getResponseCode() == 200 ){
    InputStream responseBody = connection.getInputStream();
    InputStreamReader responseBodyReader =
      new InputStreamReader(responseBody, "UTF-8");
    JsonReader jsonReader = new JsonReader(responseBodyReader);
    jsonReader.beginObject();

    while (jsonReader.hasNext()) {
      String key = jsonReader.nextName();
      if (key.equals("token_type")) {
        String value = jsonReader.nextString();
        tokenType = value;
      } else if (key.equals("access_token")) {
        String value = jsonReader.nextString();
        accessToken = value;
      } else {
        String value = jsonReader.nextString();
      }
    }
    jsonReader.close();
  } else{
    //Error handling
  }
  connection.disconnect();
```

```
    }catch (Exception e) {
        //Error handling
    }
    return accessToken;
  }
}
```

The voting system for determining which vehicles are most likely candidates.

```
public class LocationActivity extends FragmentActivity implements
  AsyncResponse{

  //unique for a passenger snapshot, to pair* API vehicle response
  private Integer passengerSnapshotIdCount = 1;
  //*passenger snapshot & list of nearby vehicles at that time
  private ArrayList<PassengerVehiclePairing> pairings =
    new ArrayList<>();
  ArrayList<Vehicle> topCandidates = new ArrayList<>();

  @Override
  //Receives the result from GetNearbyVehicles class
  public void onAPIResponse(ArrayList<Vehicle> vehicles){
    int lastIndex = pairings.size()-1;
    for (int i = lastIndex; i >= 0; i--){
      PassengerVehiclePairing pairing = pairings.get(i);
      if (pairing.passengerSnapshotId.equals(
          vehicles.get(0).getPassengerSnapshotId())){
        pairing.setVehicles(vehicles); //see below
        if (!pairing.candidates.isEmpty()){
          for (Vehicle newVehicle : pairing.candidates){
            boolean isAdded = false;
            for (Vehicle recordedVehicle : topCandidates){
              if (newVehicle.getGid() == recordedVehicle.getGid()){
                recordedVehicle.setPoints(recordedVehicle.getPoints() +
                  newVehicle.getPoints());
                isAdded = true;
                break;
              }
            }
            if (!isAdded){
              topCandidates.add(new Vehicle(newVehicle));
            }
          }
        }
        else{
          //no vehicle candidates returned, nothing to add
```

```
        }
        break;
      }
    }
    locationSaver.saveTopCandidates (topCandidates);
    locationSaver.saveVehicles (vehicles);
  }
}
```

```
public class PassengerVehiclePairing {

public Integer passengerSnapshotId;
public String passengerTime;
public double passengerLat;
public double passengerLng;
public ArrayList<Vehicle> vehicles;
public ArrayList<Vehicle> candidates = new ArrayList<>();

//Called on main tread after having recieved an API response
//with the same 'passengerSnapshotId' as this instance
public void setVehicles(ArrayList<Vehicle> vehicles){
  this.vehicles = vehicles;
  computeTopCandidate();
}

public void computeTopCandidate(){
  double HIGHEST_POINT_THRESHOLD = 30;
  double NEXT_HIGHEST_POINT_THRESHOLD = 40;
  double LOWEST_POINT_THRESHOLD = 46;
  if (vehicles.get(0).getName().equals("No vehicles nearby")){
    //nothing to add
  }
  else {
    String vehicleTime = vehicles.get(0).snapshot.getTime();
    timeDelayOk(vehicleTime);
    for (Vehicle v : vehicles){
      double distance = computeDistance(v.getSnapshot().
        getLat(), v.getSnapshot().getLng(), v);
      if (distance <= HIGHEST_POINT_THRESHOLD){
        v.setPoints(5);
        candidates.add(v);
      }
      else if (distance <= NEXT_HIGHEST_POINT_THRESHOLD){
        v.setPoints(4);
        candidates.add(v);
      }
```

```
        else if (distance <= LOWEST_POINT_THRESHOLD){
          v.setPoints(1);
          candidates.add(v);
        }
        //else v gets no points and is not a candidtae
      }
    }
  }
}
```

```
public class LocationSaver {

  public void saveTopCandidates (ArrayList<Vehicle> topCandidates){
    String outprint = TopCandidatesPrettyPrint(topCandidates) + "\n";
    writeToFile(fileCandidates, outprint);
  }

  public String TopCandidatesPrettyPrint(ArrayList<Vehicle>
      topCandidates){
    String topList = "";
    if (!topCandidates.isEmpty()){
      Collections.sort(topCandidates, reverseOrder);
      Integer highestScore = topCandidates.get(0).getPoints();
      for (Vehicle v : topCandidates){
        if ( highestScore < 25 ||
             (0.5*highestScore) < v.getPoints() ){
          //Add all candidates to the toplist, unless:
          //If top candidate has at least 25 points,
          //no candidates 50% worse will be added
          topList = topList + "(" + v.getName() + ", "
           + v.getPoints() +"p.)";
        }
      }
    }
    return topList;
  }
}
```