AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY

FACULTY OF ELECTRICAL ENGINEERING, AUTOMATICS, COMPUTER SCIENCE AND BIOMEDICAL ENGINEERING

Modeling and Simulation of Systems

Modeling movement of people using a mobile application

Author: *Dariusz Czajkowski, Grzegorz Tłuszcz*
Degree programme: *Computer Science*
Lecturer: *dr hab. inż. Jarosław Wąs*

Kraków, 2018

Table of Contents

1. Introduction

1.1. Problem description

The goal of the following work is to model the movement of pedestrians using the GPS technology. Nowadays, tracking people is a very popular subject. It is used from tracking sport-related activities to tracking criminal suspects by the police or private detectives. Wanting to track anyone is one thing, but there are many challenges that developers face when it comes to the implementation. Many apps rely on accurate data where a few-meter difference is not acceptable.

The main problem with location-based apps is that they don't filter movement by other means of transportations. Imagine a situation where you have an app that tracks your rollerblade riding. First, you get in a car, get to a park with good quality asphalt paths, you turn on the tracker and start your activity. Whether by mistake or not, you get in your car afterward and forget to turn the tracker off. You get to your home and by the time you want to share your ride on social media you realize, the tracker had been on for the whole ride back home. Now it's too late, you can't remove this activity, and even if you could, it's too much of a hassle. This is obviously not ideal and software algorithms could quickly remove data that is irrelevant.

The subject of the following work was to create a mobile app that would allow to accurately track the movement of people. It is focused on gathering GPS-based data and filtering those, that are not connected to walking. Furthermore, all data considered legitimate is displayed on an interactive map in real time. Every user of the app is visible to everyone. If they start riding a car, their location disappears, as they are not walking anymore. Additionally, if you press on any person's location you will see their entire history of where and when they walked. The app is a proof of concept and it doesn't have any special meaning by itself, but its ideas and algorithms can be applied to any future work that requires gathering GPS-based locations.

1.2. Potential solution

We have looked at many apps on the market that are supposed to track walking, but none of them filtered riding a car. We looked at possible solutions to the problem and we realized, there isn't much on the web. We had to come up with an algorithm that would preserve data that involved walking and would discard data that didn't. Our first thought was to simply filter out points that were apart a distance, that wouldn't be possible for a human to be made. It appears,

that a maximum speed of a human is 44km/h[8], which is not ideal considering, this could be an average speed of a car in a city. The final algorithm and all considerations are mentioned in the work that follows.

2. Literature review

In this chapter, we would like to focus on existing approaches to GPS data collection and post-processing. Our goal is also to learn about possible ways of data filtering, travel modes detection and route approximation.

According to many of the reviewed publications, locations can be acquired using one of many technologies e.g. Geographic Information Systems (GIS), Radio Frequency Identification (RFID), Wireless Local Area Network (WLAN) and Global Positioning System (GPS). It is worth mentioning that the latter is by far the most popular and easiest to implement on a large scale. This is because such locations can be obtained using a module built into smartphones so there is no need for an additional device. Except for geolocalization data, engineers also collect acceleration, direction and height above the sea level. A popular solution is to use a mobile app for localization sampling and displaying results. Obtaining essential data can be performed in some interval (eg. 10 seconds) or after parameters' change. This allows them to achieve regular, accurate data in terms of time or location.

After collecting GPS samples many of the authors mention some way of data filtering to reduce further complications in computation. One of the problems with GPS is that the signal weakens inside buildings and dense livelihood. In other words, the points appearing to be falling inside a building may or may not be actually taken indoor, and vice versa. This is because signals are attenuated, reflected and scattered by roofs, walls and other objects. To solve this issue Yongyao Jiang in his work "Automated Human Mobility Mode Detection Based on GPS Tracking Data" used building footprints obtained from the state Massachusetts Office of Geographic Information (MassGIS) overlaid with given GPS points. Another interesting difficulty is to determine the travelling mode of the user. One of the approaches is to measure average speed and determine some threshold values to distinguish walking, riding a bike and travelling by car.

A man can perform data clustering based on travelling mode to remove undesired activities. Another interesting solution is to use the accelerometer combined with time measurements to decide if any steps were made during the tour. That way they can determine which travels were made with a vehicle. Some of the engineers behind the publications even reached for machine learning algorithms to recognize walking habits and to label them as 'walking activity'.

Last but not least, authors face a problem with displaying results. Algorithms often return discrete data (points) with various accuracy. It is important to somehow approximate walked path

Figure 2.1. Robust principal curve (green) fitted to GPS point cloud data.
source: Chris Brunsdon, Path Estimation from GPS Tracks

on the map. This task can be achieved using principal curves created from locations grouped into a singular path. Chris Brunsdon in his work[2] gives a good example of the algorithm for computing such a curve 2.1. Einbeck, Tutz and Evers in their work[5] dive deep into the process of constructing these curves and the benefits from choosing various methods.

3. Proposed model of the phenomenon

3.1. The model's purpose and description

Our goal is to build a mobile application that would allow users to review their past walking paths and the current position of their friends. The majority of the effort is focused on implementing a backend API that would filter out corrupted samples and create paths representing routes walked by the user in the past. Frontend mobile application retrieves data from the API and displays it on the map. It also handles all the user interactions and sends current user location.

3.2. Algorithm assumptions

Our algorithm assumes that:

1. mobile application is turned on forever,
2. mobile phones send requests with their location every 5 seconds,
3. user tracks only outdoor activities,
4. user cannot walk faster than 14 km/h.
5. movement with average speed higher than 14 km/h is considered as travel with a vehicle.

Due to the fact that we have limited deployment possibilities, we launched our app with Expo and it is only able to retrieve GPS location when the app is turned on.

3.3. Activity cycle diagrams

3.3.1. Front-end app cycle

The 3.1 shows the front-end app cycle diagram. The app is very simple in its flow. It simply gets the GPS signal and sends it to the API every 5 seconds. The app also has two modes. One of them allows to see current position of all the users. The second mode displays a history of a specific user.

3.3.2. Sending current location to the API

When the app sends the location along with user's id to the API, the API has to do some calculations depicted in 3.2 to validate provided data.

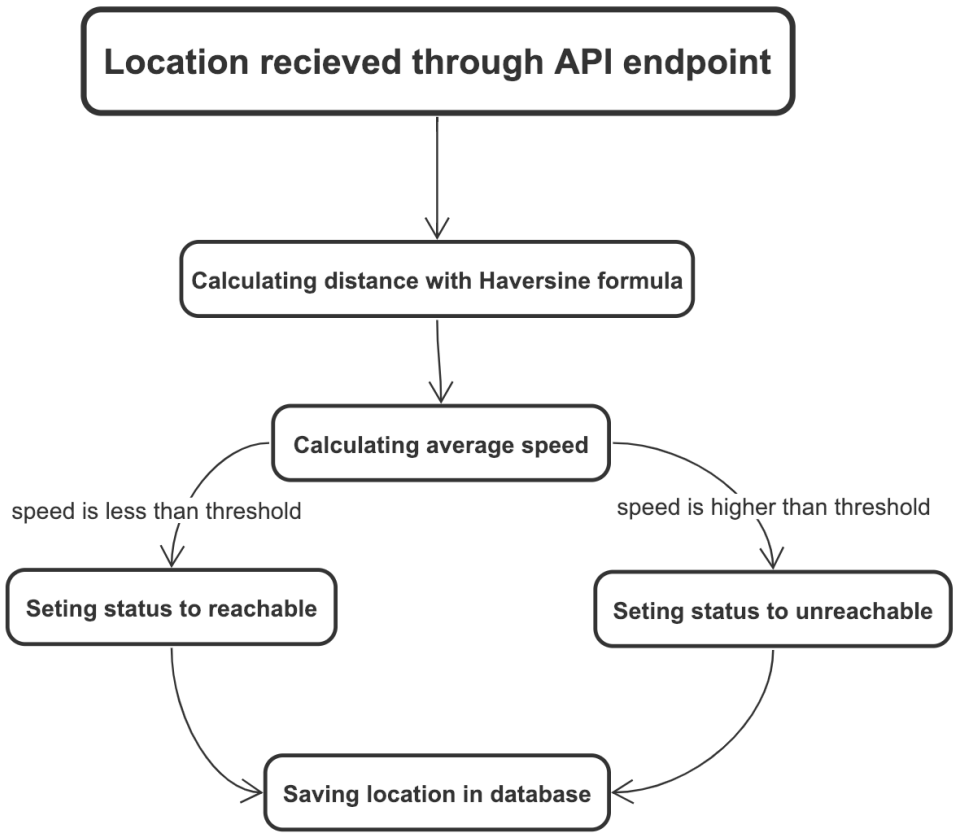

Figure 3.2. Sending current location to the API
source: own elaboration

3.3.3. Fetching historical locations

When the app asks for historical locations, the API calculates on-the-fly if points are reachable and removes all invalid data. This is done every time the app asks for locations to allow tweaking of the algorithm at any moment. If these calculations became expensive, everything could be cached and the cache would be invalidated whenever a new version of the algorithm arrives. The flow can be seen in 3.3.

3.4. Applied approach compared to other implementations

As well as many other engineers we collected locations using GPS technology. In our approach, we used three key components to reach our goal. First of them is calculating the average

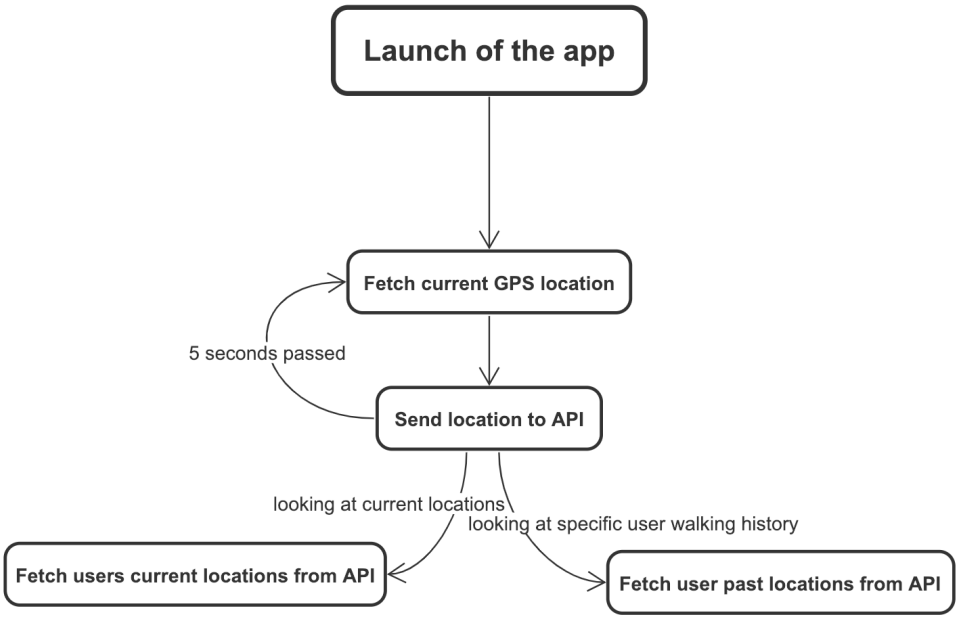

Figure 3.1. Front-end app cycle
source: own elaboration

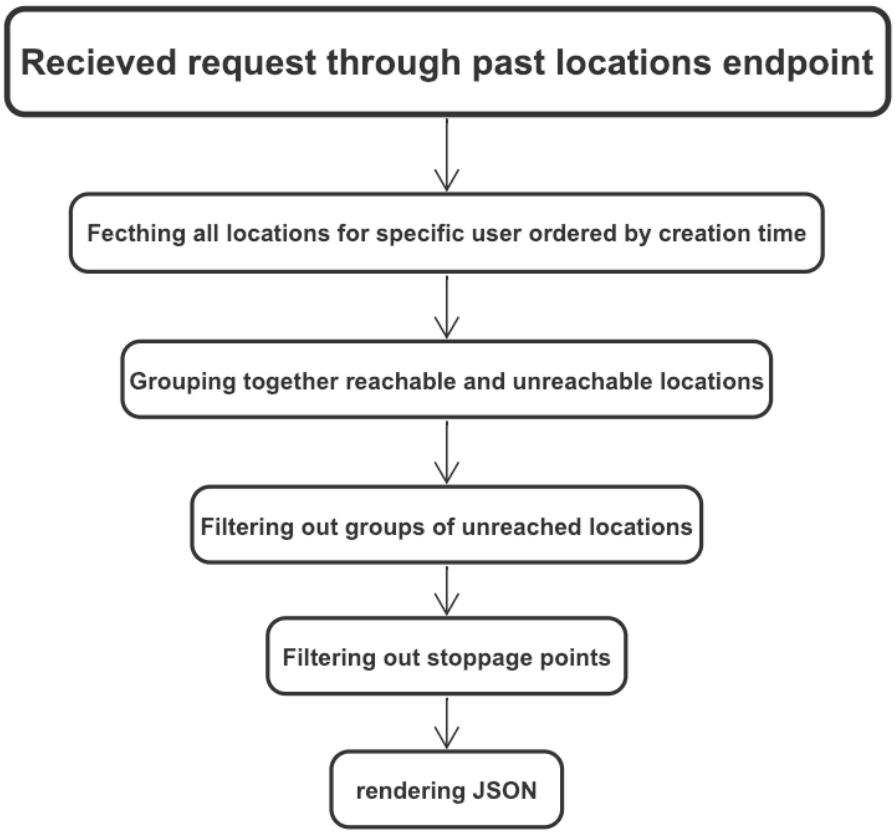

Figure 3.3. Fetching historical locations
source: own elaboration

speed of the user based on his last location and current position. To achieve that we used Haversine formula which is quite unique considering other implementations. Then we experimentally stated some threshold values which were taking into consideration GPS errors. Based on the final judgment we set the status of the location either as reachable or unreachable.

The second one is an algorithm that filters out unreachable locations. In other words, it filters out travelling modes other than walking. This idea is widely known and we thought that it fits our needs.

The third one is a piece of code which aims to remove stoppage points, mainly occurred by traffic light stops and staying in traffic jams. It is our own idea and differs from other projects.

3.5. The use case diagram

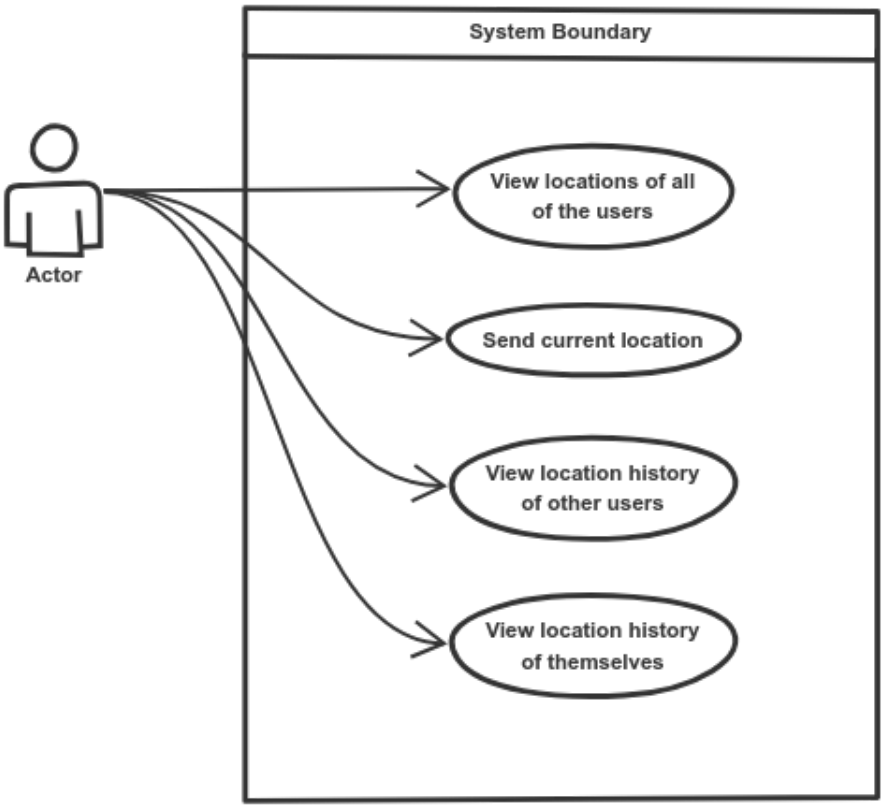

Figure 3.4. The use case diagram
source: own elaboration

4. Simulation of the phenomenon - implementation

4.1. Technology and tools

The project described in this work is written in two parts — backend and the app.

As we could pick any technology whatsoever we picked Ruby for the backend as this is the backend technology we are most familiar with. We have also picked a web framework called Ruby on Rails as it is the most popular web framework for Ruby. In the long run, it saved us a lot of time that would be required to set up a web server, handle requests, make database queries and many more.

The backend consists of a few REST[1] endpoints. One of them receives user's location and runs the algorithm against provided location to determine, whether the input is legitimate or not. After it is calculated, data is preserved in a database with an appropriate label. There is also an endpoint to list locations of all users at the time and an endpoint to return specific user's history.

Implementing a mobile application didn't seem like an easy task at first. Currently, if someone wants to make a mobile application they have to target a specific platform — Android or iOS. Developing for these platforms isn't easy at all. For the Android development, you have to know Java, Kotlin or C++, have Android Studio installed, and knowledge on what to do. Fortunately, there are many resources on how to start. Authors of this work do not have an Android phone, so there would be issues with running the app in a simulator, which would defeat the purpose of this work. The iOS side of things is not easier either, as it requires knowledge of Swift or Objective-C, having XCode installed on a macOS device, a paid developer account and many more. Even if we decided to go with iOS, it would require a lot of work, time and resources to develop a simple app. At that point, we started reading through the Internet and we learned there are many JavaScript compilers that compile your code into native-to-the-platform code. One of them is React Native — a technology that allows writing JavaScript code in a React Framework that is compiled to a native app. This made great news to us, as we didn't have to learn any other tools or languages, as we know JavaScript perfectly well.

The app consists of a view that presents a native-to-iOS map. On the map, there are markers that represent locations of all app users in real time. If a marker is held, more details about the location are shown — a name of the user and the time of the measurement. When a marker is

[1] Representational State Transfer

clicked, the map goes into the *history* mode where all historical locations of that particular user are displayed. You can then click anywhere on the map to go back to the *normal* mode. In the background, any time the app is open, current location of the user is sent to the server to update user's location.

4.2. Implementation details

In this section, we will describe two main algorithms used to store locations and retrieve past walking paths of the user.

In our database, we have two tables: one for representing users and one for locations. I will refer to the model of a user with *User* and to a location with *Location*.

The process of storing a new sample consists mainly of computing average speed needed to achieve new location. To compute the distance between to points we use Haversine formula which is described as below:

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \tag{4.1}$$

where φ_1, φ_2 are latitudes and λ_1, λ_2 are longitudes.

Here is some pseudo code of the storing algorithm:

```
my_user = User.find(user_id)
last_location = my_user.locations().last()
distance = Haversine(new_location, last_location)
time = new_location.creation_time() - last_location.creation_time()
speed = distance / time
if (speed > threshold)
    new_location.status = unreachable
else
    new_location.status = reachable
end
return new_location.save()
```

Another crucial part of our system is the algorithm rendering old paths. Here you can get a quick glimpse of it:

```
my_user = User.find(user_id)
my_user.locations()
routes = [] // group reachable locations into walking paths
counter = 0

for (location in locations)
    if (location.status == reachable)
        routes[counter].push(location)
    else
        if (routes[counter] != null)
```

```
            counter+=1
        end
    end
end

// remove stoppage points
routes.filter do (path)
    // calculating center of mass in path
    com = {lat: 0, lon: 0}

    for (location in path)
        com.lat += location.lat
        com.lon += location.lon
    end

    com.lat /= path.size()
    com.lon /= path.size()

    // check if there is a point in the path
    // further than a threshold from the centre of mass
    return path.any do (location)
        distance = Haversine(location, com)
        return distance > threshold
    end
end
```

As you can see we group reachable locations together and after grouping, we check if any of the paths in routes has all of its locations in a small area. This would mean that probably this path is actually a temporary stoppage on traffic lights and we do not want to render it as walking.

5. Results of the simulation

5.1. Statistics and gathered results

During the testing phase of the app we have gathered more than 8000 location records that had to be filtered and ran through our algorithm. We have made a total of 21 paths with walking, bus riding and car driving. During testing, we have tweaked the algorithm for it to show as accurate results, as possible. We think, the app performs alright, but the accuracy of the GPS signal could be improved.

5.2. Algorithm calibration and data validation

During the development phase we have picked 9 km/h as a maximum of human walking speed and set that as a limit. After making a few passes and trying the app out, we have found 9 km/h to be not enough, as it removes any kind of running and does not take into consideration GPS inaccuracy. What we have discovered, is that if someone moved 12 meters, which makes it a plausible distance to be walked in as depicted in the equation 5.1, GPS inaccuracy made that distance to be 14 meters, which removed the readout.

$$t = 5[s];\;\; x = 12[m];\;\; \bar{v} = \frac{\Delta x}{\Delta t} = \frac{12}{5}[\frac{m}{s}] = 2.4[\frac{m}{s}] = 7.2[\frac{km}{h}] < 9[\frac{km}{h}] \tag{5.1}$$

After some testing, we have found 20 km/h to be an optimal value. It does not exclude runners, but does correctly filter any car driving, as we have found a 30 km/h to be a minimum value a car moves in, even when going through speed bumps. The only exception is a standstill or a very slowly moving traffic jam. Our algorithm does filter out any standstill, but cannot handle a very slowly moving car. It would be invalid to assume someone is in a slowly moving car, as it is very possible they are walking at that moment. Perhaps a look at a larger picture, taking into consideration more context would help, but machine learning could be a valid solution as well.

5.3. Comparison of the results with real data

We have put a lot of time to validate data and polish the algorithm to give the best results possible. In the following subsections, there are a few examples presented. They should give the reader a whole picture on how the app works and what is happening behind the scenes.

Figure 5.1. Green path is the actual path, red pins depict non-filtered readouts.
source: own elaboration

5.3.1. Example 1: Accurate walking readouts

This is an example, when readouts during walking are correctly read and displayed on the map. As can be seen in the figure 5.1, presented location differs from actual position a little due to GPS inaccuracy. This inaccuracy is not substantial and is not a subject of this work.

5.3.2. Example 2: Accurate car-driving readouts

In this example you can see a path of a car being driven. In the figure 5.3 there are not many points as they were correctly filtered out by the algorithm.

As can be seen, especially near the very end of the journey, some points are not being filtered out correctly. This is due to very slow movement of the car, road layout etc. Actually, a human could interpret this data as if someone was searching for a free parking spot, which made a car to really slow down. Also, it is unclear if these points are there because someone is driving really slowly, or just got out and is walking fast. Authors consider these readouts as something that could be worked on by an AI[1] model or by taking the actual environment into consideration.

5.3.3. Example 3: Inaccurate walking readouts

Life is not always easy and things like to go badly. As can be seen in the figure 5.4, when approaching high buildings on the south side[2], readouts became very inaccurate. The readout with

[1] Artificial Intelligence
[2] All of the tests were conducted on the north side of the equator.

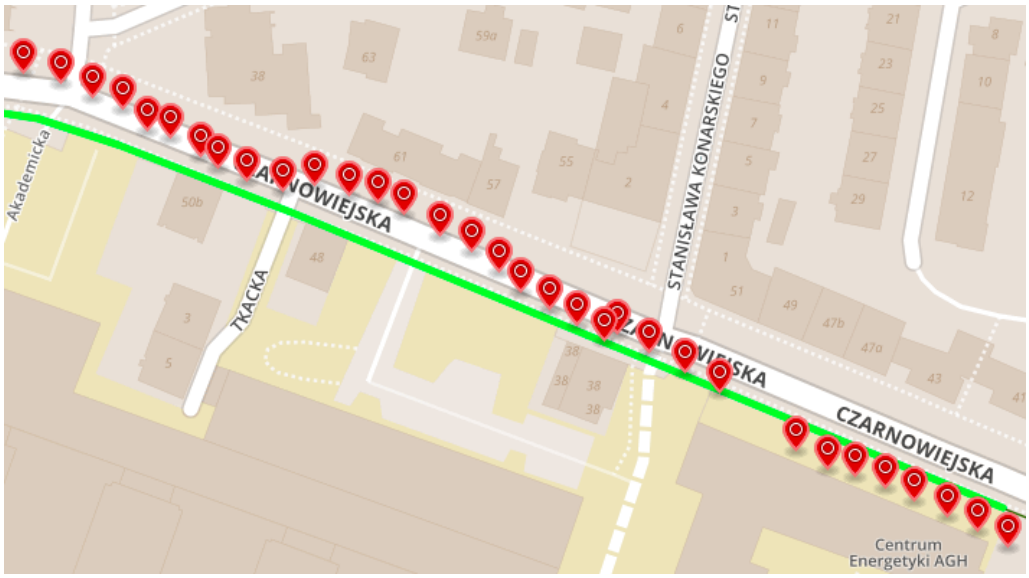

Figure 5.2. Green path is the actual path, red pins depict non-filtered readouts.
source: own elaboration

largest difference from the actual position was 19 meters, as shown in the figure 5.4. We think this is something that can be solved and we describe proposed solution in the section 6.4.

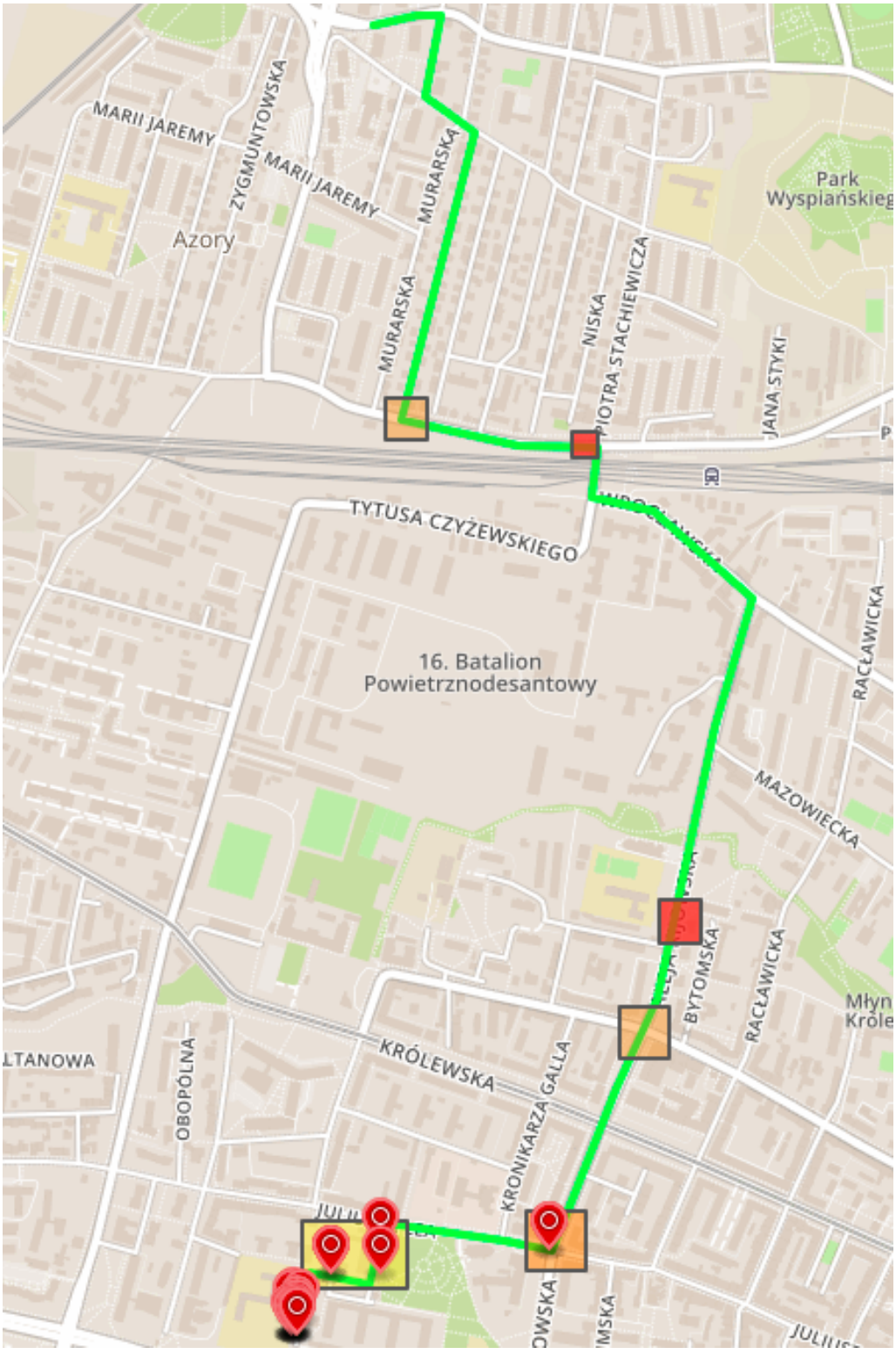

Figure 5.3. Green path is the actual path, red pins depict non-filtered readouts, red boxes mean stoppage points, orange boxes depict areas of real slowdowns, yellow area shows a very slow movement.
source: own elaboration

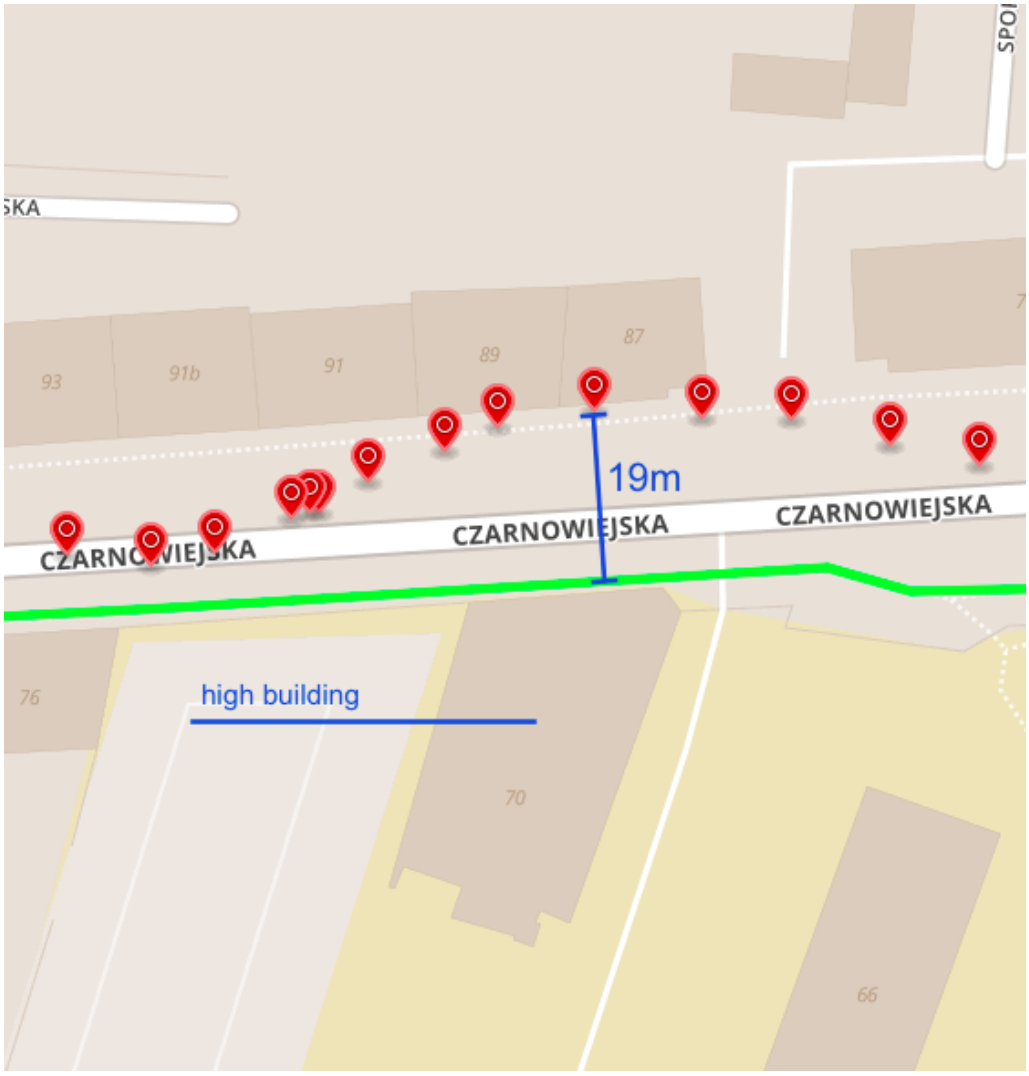

Figure 5.4. Green path is the actual path, red pins depict non-filtered readouts.
source: own elaboration

6. Summary

6.1. Conclusions

Although we had little problems with GPS accuracy, we managed to achieve our goal. This project proves that we are able to distinguish walking modes of the person using only GPS data. There is a lot of room for improvements and tweaks but it is working properly. We experienced how to work with inaccurate data and filter out possible anomalies.

6.2. Model in the context of existing solutions

Applications available today doesn't distinguish walking modes at all. Strava, Endomondo and Google maps can't tell whether you have been running, walking or driving a car. They only track your statistics. We think that our solution could revolutionize outdoor activity tracking apps. Being able to tell how long and where have you been walking in the past is a crucial element of data about your daily activities.

6.3. Biggest challenges and achievements

The biggest challenge we have encountered was creating a reliable algorithm that would accurately filter data that is not coming from a walking human. As can be seen in the chapter 3, proposed model is complicated and consists of many parts. Removing points that are apart from each other more than some distance, filtering out points that are in a near approximation that are a result of a *teleportation*, approximating stationary location by averaging coordinates etc. To come up with all of these solutions took quite some time. Another thing was implementing those solutions, which took even more time. We are very happy we came up with a working solution that is working quite reliably and, to us, is very impressive. Should we ever have to implement a location-based mobile app, we would definitely use a solution described in this article. It is not very hard to implement and improves a system heavily.

6.4. Future work and possible directions to be taken

There are many ways to improve on the work described in this article. Current algorithm does filter out data that seems to be generated by not walking and those that seem to be generated by a computer. The biggest challenge there is is the readout accuracy. Currently, when using simple GPS module embedded in an iPhone there are few-meter differences compared to the actual position. Of course, one way to get a more accurate position of a stationary object is to take multiple readouts and take an average of all points. The problem arises when the object is in motion. During testing, we gathered data that was up to 19 meters inaccurate because of tall buildings on the south side that were obtrusive to the GPS satellites located on the equator (more on that to be read in section 5.3.3). This is one thing to be worked on and to be improved. Authors of this article do not have a good way of determining a more accurate location in these cases. Probably an algorithm that would take the environment into consideration could help. Another thing would be to make the path more linear. Although not accurate, we have found that people move in close-to-straight lines. Such approximation could show more accurate results, but not necessarily. It could also lead to results that are off, because of making paths *too* straight. This is definitely something to consider.

References

[1] Patrick Bertagna. *How does a GPS tracking system work?* 2010. URL: *https://www.eetimes.com/document.asp?doc_id=1278363*.

[2] Chris Brunsdon. *Path Estimation from GPS Tracks*. 2007.

[3] Aaron W. Burgess. *Determining Trip and Travel Mode from GPS and Accelerometer Data*. 2016.

[4] D. Douglas and T. Peuker. *Algorithms for the reduction of the number of points required to represent a digitised line or its caricature*. 1973.

[5] J. Einbeck, G. Tutz, and Evers. *Local Principal Curves*. 2003.

[6] *Geolocation API*. 2018. URL: *https://facebook.github.io/react-native/docs/geolocation*.

[7] *Great-circle distance*. URL: *https://en.wikipedia.org/wiki/Great-circle_distance*.

[8] *Guiness World Records — Fastest Human (running)*. Guiness World Records, 2009. URL: *http://www.guinnessworldrecords.com/products/books/superlatives/fastest*.

[9] *Haversine formula*. URL: *https://en.wikipedia.org/wiki/Haversine_formula*.

[10] S. Reddy et al. *Determining transportation mode on mobile phones*. 2008.

[11] L. Stenneth et al. *Transportation Mode Detection using Mobile Phones and GIS Information*. 2011.

[12] Jie Tian et al. *Automated Human Mobility Mode Detection Based on GPS Tracking Data*. 2014.

[13] Fang Zong et al. *Identifying Travel Mode with GPS Data Using Support Vector Machines and Genetic Algorithm*. 2015.