POLITECNICO
MILANO 1863

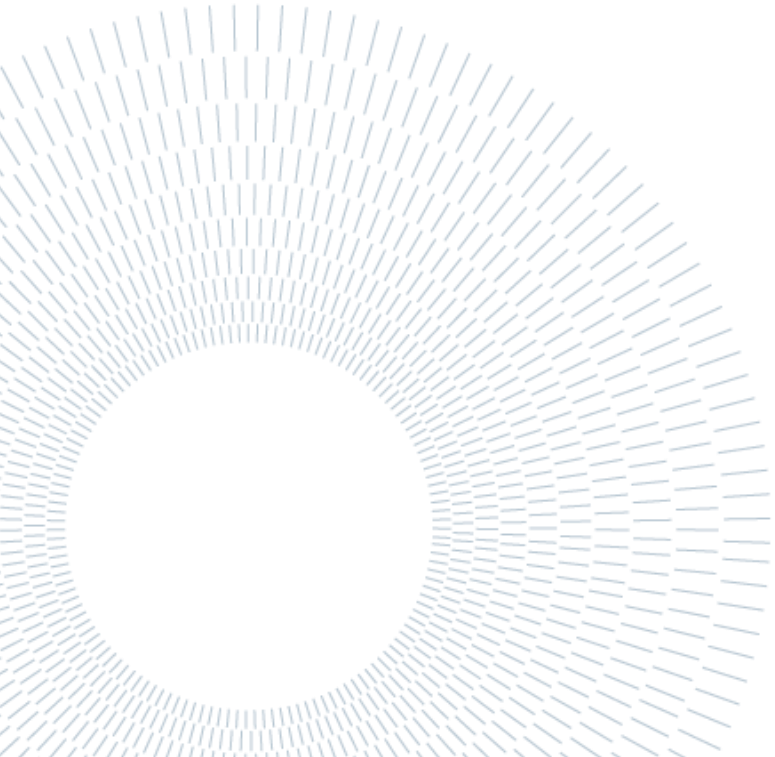
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

# Implementation and evaluation of a Map Matching algorithm

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE ENGINEERING
INGEGNERIA INFORMATICA

Author: **Matias Giangualano**

Student ID: 6801380
Advisor: Matteo Giovanni Rossi
Academic Year: 2022-23



# Abstract

Map matching is still one of the most active research topics in science despite having been on the scene for many years. The main reason for this longevity is the emergence of commercial sectors in which the localization of moving objects is fundamental. Both in open and closed environments. Very common examples are autonomous driving and orientation in closed spaces with the presence of obstacles.

In this thesis, we will see the implementation of a Map Matching algorithm and some of its possible variations. We will begin by introducing the problem of Map Matching and its relevance in today's world. Indeed, the proliferation of devices capable of communicating with GPS to obtain the coordinates of own position and the development of entire commercial sectors, such as, for example, the automotive, make the problem of localization one of the most studied. All this leads Map Matching to be one of the most important active research branches.

Since this problem has a wide scope of application, the article provides a classification of various methods invented in recent years. Furthermore, some possible application scenarios are illustrated below to give an idea of the multiple uses of Map Matching. Among them, the most important are real-time and off-line scenarios; that is when the localization problem is faced and solved, with a certain margin of error, during the movement of the object to be localized or, instead, when the problem is faced later by memorizing the path of the moving object by collecting its positions over time.

In our opinion, the real-time scenario is the most interesting and, consequently, we have focused our analysis on one of the most promising algorithms produced by scientific research, which is called Spatial-Temporal Proximity and Improved Weighted Circle (STP-IWC). This is an improvement of an already published process, which bases its operation on the Spatial-Temporal Proximity method, that is the identification of the road the object is traveling among those that fall within a specific area around its position in a given moment.

This technique, unfortunately, has a weak point, which occurs during the calculation phase of the candidate road. The algorithm is unable to determine the candidate when two consecutive points of the trajectory are located in the bisector of a road intersection. Furthermore, the calculation of the candidate road requires parameters, called weights, which moderate the impact of the distance between the road and the trajectory point, and the divergence between the directions of the candidate and the trajectory during the calculation of the result. Moreover, these parameters have a high impact on the complexity of the algorithm.

For these reasons, the algorithm we are going to see has been proposed and it is more advanced and robust than the one presented so far. Its name is Spatial-Temporary Proximity and Improved Weighted Circle (STP-IWC) and adds the concept of directional similarity between the trajectory and the candidate road. This new idea is implemented in the procedure with another parameter called $b$.. In addition, the candidate road calculation mechanism has been modified so that the weights of the previous version are calculated dynamically thus improving performance.

We will therefore see the implementation of different versions of this algorithm, moving from the less complex one, in which the weights are fixed and decided a priori, to the actual STP-IWC version of which we will show a possible implementation. We will then compare the results obtained and see how they differ in terms of performance. Finally, we will propose any improvements both to the code and to the implemented versions that could be a starting point for subsequent papers.

**Keywords:** Map Matching, angular similarity, spatial temporary proximity, improved weighted circles.

# Abstract in italiano

Il Map Matching è ancora uno degli argomenti di ricerca più attività in ambito scientifico nonostante sia presente sulla scena da molti anni. La principale ragione di questa longevità è l'affermarsi di settori commerciali in cui la localizzazione degli oggetti in movimento è fondamentale. Sia in ambienti aperti quanto in locali chiusi. Esempi molto comuni sono la guida autonoma e l'orientamento in spazi chiusi con la presenza di ostacoli.

In questo articolo vedremo l'implementazione di un algoritmo di Map Matching e alcune delle sue possibili variazioni. Inizieremo introducendo il problema del Map Matching e la sua rilevanza nel mondo di oggi. Infatti, il proliferare di dispositivi in grado di comunicare con il GPS per ottenere le coordinate della propria posizione e lo sviluppo di interi settori commerciali, come, ad esempio, quello automobilistico, rendono il problema della localizzazione uno dei più studiati. Ciò porta il Map Matching ad essere uno dei rami di ricerca più importanti ad attivi.

Poiché questo problema ha un vasto campo di applicazione, in questo articolo viene fornita una classificazione dei vari metodi inventati negli ultimi anni. Inoltre, di seguito, sono illustrati alcuni possibili scenari applicativi per dare un'idea dei molteplici utilizzi del Map Matching. Tra i più importanti ci sono gli scenari real-time e off-line; quando cioè il problema di localizzazione viene affrontato e risolto, con un certo margine di errore, durante lo spostamento dell'oggetto da localizzare o, invece, quando il problema viene affrontato in un momento successivo memorizzando il percorso dell'oggetto in movimento raccogliendo le sue posizioni nel tempo.

A nostro avviso lo scenario in tempo reale è quello più interessante e, di conseguenza, abbiamo focalizzato la nostra analisi su uno degli algoritmi più promettenti prodotti dalla ricerca scientifica, che si chiama Spatial-Temporal Proximity and Improved Weighted Circle (STP-IWC). Si tratta di un miglioramento di un processo già pubblicato, che basa il suo funzionamento sul metodo Spatial-Temporal Proximity, ovvero dell'identificazione della strada che l'oggetto sta percorrendo tra quelle che ricadono in una determinata area attorno alla sua posizione in un dato momento.

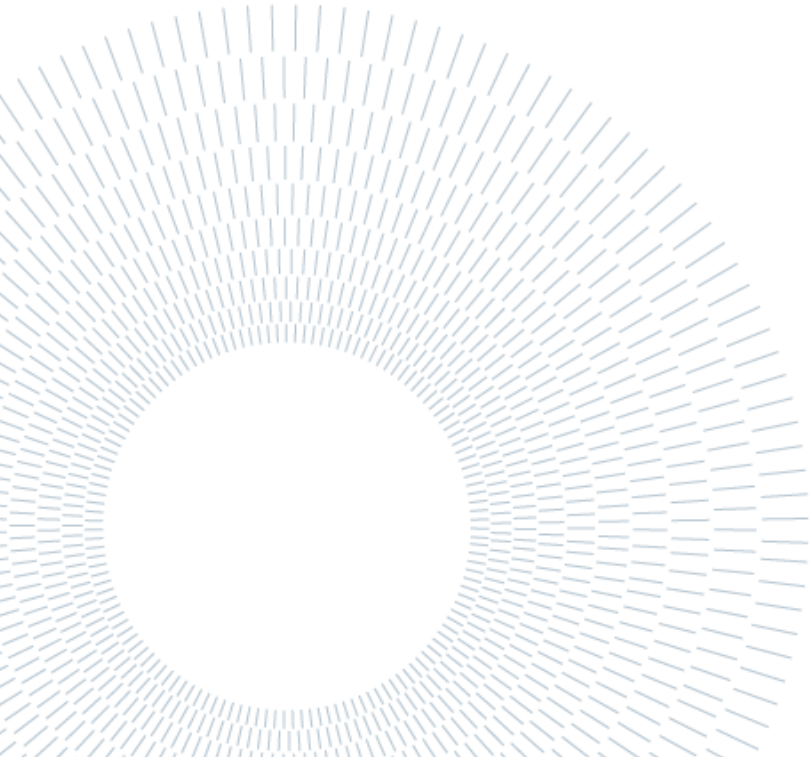
Questa tecnica, purtroppo, ha un punto debole, che si presenta in fase di calcolo della strada candidata. L'algoritmo non è in grado di determinare la candidata quando due punti consecutivi della traiettoria si trovano nella bisettrice di un incrocio stradale. Inoltre, il calcolo della strada candidata richiede dei parametri, detti pesi, i quali moderano l'impatto della distanza tra la strada e il punto della traiettoria, e la divergenza tra le direzioni della candidata e della traiettoria durante il calcolo del

risultato. Inoltre, questi parametri hanno un elevato impatto sulla complessità dell'algoritmo.

Per questi motivi è stato proposto l'algoritmo che andremo a vedere, il quale è più avanzato e robusto rispetto a quello presentato finora. Il suo nome è Spatial-Temporary Proximity and Improved Weighted Circle (STP-IWC) e aggiunge il concetto di somiglianza direzionale tra la traiettoria e la strada candidata. Questa nuova idea viene implementata nella procedura con un altro parametro chiamato $b$. Inoltre, il meccanismo di calcolo della strada candidata è stato modificato in modo che i pesi della versione precedente vengano calcolati dinamicamente migliorando così le prestazioni.

Vedremo quindi l'implementazione di diverse versioni di questo algoritmo, passando da quella meno complessa in cui i pesi sono fissati e decisi a priori, alla versione STP-IWC vera e propria di cui mostreremo, appunto, una possibile implementazione. Successivamente confronteremo i risultati ottenuti e vedremo come differiscono in termini di prestazioni. Infine, proporremo eventuali miglioramenti sia al codice che alle versioni implementate che potrebbero essere spunto per elaborati successivi.

**Parole chiave:** Map Matching, angular similarity, spatial temporary proximity, improved weighted circles.



# Contents

# Introduction

One of the most active topics in the last two decades and, nowadays, still one of the most important due to new emerging localization technologies, is the **Map Matching problem**. Indeed, the ubiquity of positioning devices allows the tracking of entities on the earth and, their intrinsic inaccuracy makes pre-processing steps, like the Map Matching, essential to correct errors in positioning or trajectory calculating.

The main important process in this scenario is, as said, the Map Matching problem, which finds the object's travel route aligning position data to the underlying road network. The Map Matching problem involves some key concepts like large datasets, trajectories, road networks and routes. In this document we are going to go in deep with these concepts, presenting the problem definition and then we will show how they can be implemented concretely. Then we will show also how the rest of the document is organized.

First, it is worth pointing out that the application field of this problem is very large. Just to make an example, it can be used to localize a robot inside an indoor building or to calculate the vehicle's route on a road map in open spaces. In the latter case, which is the context where we will see the problem application, usually, the calculated route is expected to be continuous since it represents the vehicle's travel, but it is quite often that it contains disconnected edges.

All the above navigation functions, including the *real-time tracking the vehicle's location*, rely on the implementation of the Map Matching problem that uses vector map information integrated with other data from positioning sensors like GPS, GPS/INS[1]. The aim is to produce the best estimate of the vehicle's real position, overcoming the influence of GPS errors or vector map errors and making the GPS trajectory match the corresponding position on the road map. Moreover, the purpose is to provide the positioning basis for more advanced path planning and guidance (e.g., autonomous vehicles, vehicle dispatch, etc.).

Indeed, most of the previous algorithms have shortcomings of lag or low precision, so improving the Map Matching problem by shorting the time of the real-time positioning and positioning accuracy is a current and fervently active challenge.

The autonomous drive is another application field in which the Map Matching problem plays an important role. Unmanned ground vehicles must judge and drive to

[1] **GPS/INS** is the use of GPS satellite signals to correct or calibrate a solution from an inertial navigation system (INS) [40]

the designed destination and four representative technologies are considered to perform complete autonomous driving.

The first is the position estimation which must obtain the current vehicle's position; the second is the environment perception which must recognize the surrounding environmental information. The third is the obstacle-avoiding technology that must help the vehicle to move without any collision with other objects on the path, and the fourth is path planning which must generate the optimal way to the destination within the space in which the vehicle is moving. It is quite clear that *position estimation, which determines the current position of a vehicle, is the basis for an autonomous driving system*.

Most of the present position estimation technologies are based on the global positioning systems identified as the Global Navigation Satellite System[2] (GNSS), such as the Global Positioning System (GPS). Many studies and advances in technology have improved the accuracy of GNSS (with a maximum error of ten meters without the presence of interference), but in signal-degraded environments, the GNSS cannot guarantee the same efficiency and continuous positioning.

In conclusion, for autonomous driving, is essential to solve the Map Matching problem to enhance the position accuracy of low-cost GPS receivers based on map data without using any additional sensor, which would produce an overload of calculations due to the huge amount of data to process.

This paper aims to implement one of the proposed algorithms by analyzing various aspects of the elaboration. To do this, we will examine all the steps needed to pass from a theoretical idea to a concrete program. We have decided to consider an algorithm in the *real-time* scenario in which the Map Matching problem is solved while a vehicle is moving onto a road. We made this decision because we think that the challenge is there.

Indeed, we believe that technical and technological development will lead to the spread of GPS devices that are more powerful in calculation and less expensive in consumption. We think that in this constantly evolving scenario where new roads are created and new alternative road routes increase, the main tool to exploit these new possibilities is real-time algorithms. This doesn't mean that *offline* algorithms aren't a viable solution. On the contrary. Nowadays they represent an important support for *real-time* technologies and allow them to overcome their limits by enriching the data with the information determined by them once the vehicle has finished moving.

We will start with the data collection and their representation in memory. Then we start to implement the algorithm facing the data elaboration and we will do it going from a first and simple version to a more complex one, in which all the advanced techniques and calculations are applied. We will do this through two other versions of

[2] A Global Navigation Satellite System is a land, sea, or air geo-radiolocation and navigation system that uses a network of artificial satellites in orbit and pseudo-satellites on the earth [37] [38](Wang et al. [39]).

the same process which will gradually be more complex by adding a technical aspect compared to the previous one.

The rest of the paper is organized as follows: in the next chapter (chapter one) we will show the state of the art of the Map Matching problem illustrating some of the possible approaches to resolve it. Then, in chapter two, we examine a particular solution that uses a weighted graph technique in which the route is calculated starting from the candidate graph, and the best matching result is obtained by a weights distribution where, usually, the closest candidate has the maximum weight.

In chapter three we will see a possible implementation of the solution discussed in chapter two and some of its possible variations. In chapter four, we will analyze the results obtained and we will make some considerations on them. Finally, in chapter five, we will discuss some possible improvements to the algorithm and what is possible to add to complete and enrich this work.

# 1 State of the Art

In this chapter, we talk about the state of the art on the Map Matching problem. We begin by giving some useful definitions and then continue by describing some possible approaches to the Map Matching problem and some possible solutions proposed in these long years of research, explaining how they work.

## 1.1. Problem definition

**Definition 1.1.** *A **trajectory Tr** is a sequence of time ordered spatial points* $Tr : p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_n$ *sampled from a moving object.*

Usually, a point $p_i$ consists of a couple of cartesian coordinates $\langle x_i, y_i \rangle$. But it is possible add more information like timestamp $t_i$, speed $v_i$, and heading $\theta_i$.

Therefore, we can find $p_i$ as $\langle x_i, y_i, t_1, v_i, \theta_i \rangle$.

**Definition 1.2.** *A **road network** or **map** is a graph* $G = (V.E)$ *in which every vertex* $v = \langle x, y \rangle \in V$ *represents a crossroads or a road end; and an edge* $e = \langle o, d, l \rangle \in E$ *is a road linking vertices* $e.o$ *and* $e.d$ $(e.o, e.d) \in V$ *with a spatial points sequence* $l$.

**Definition 1.3.** *A **route R** represents a sequence of connected edges:* $R : e_1 \rightarrow e_2 \rightarrow \cdots \rightarrow e_n$ *where* $e_i \in G.E$, $(1 \leq i \leq n)$ *and* $e_k.d = e_{k+1}.o$.

**Definition 1.4.** *Given a road network* $G = (V.E)$ *and a trajectory Tr, the **Map Matching problem** find a route **R** that represents the sequence of roads travelled over by Tr.*

## 1.2. Classification

As mentioned above, the Map Matching problem has been present for a long time on the scientific and computer science research scene. Today even more, thanks to the diffusion of localization tools in the most disparate disciplines and daily activities. So, it is easy to imagine that, in all this time, a classification of the proposed solutions and the resulting algorithms has been made.

Here we contextualize the Map Matching problem on positioning a vehicle on a road present on a street map, but this is not only the main application of this problem. We will see also an example of application that does not involve vehicle positioning on a map but localize people moving in a city.

This application shows that Map Matching can be used to assist delay estimation on people's trip and to use this information for suggesting them to change their itinerary and minimize the delay. Let us start by saying that, in the scenario of vehicle localization, a correct trajectory should lie on the road map. Therefore, barring errors on the map, which are increasingly less frequent and corrected with map updating procedures [1], the difficulty of the Map Matching problem depends entirely on the *quality* of the aforementioned trajectory.

Quality that is mainly affected by two aspects: inaccurate trajectory measurements (*measurement error*) and low sampling frequency (*sampling error*). Consequently, a first classification of the problem can be based on the aspect just mentioned. That is between **on-line mode** and **off-line mode**.

In the **online Map Matching problem**, the vehicle position is sampled continuously, and the resolution is based on the current sample plus some antecedents or successes ones. On the contrary, in the **off-line mode the Map Matching problem** is solved after obtaining the entire trajectory and calculating the optimal route limiting errors and computational constraints.

From methodology perspective, Quddus et al. [2] classify solving methods into four categories: *geometric*, *topology*, *probabilistic*, and *advanced* methods. The *geometric methods* focus on the distance between trajectory elements and the road network. The *topology methods* take into consideration the connectivity and shape similarity. The *probabilistic methods* try to model the trajectory's uncertainty including error measurement and the unknow travel path between two samples. Their aim is to find the route that has got the highest probability to generate the given trajectory. Finally, the *advanced methods* are those based on some advanced models such as the Kalman filter, particle filters and fuzzy logic.

It is worth considering that the classification reported so far, also shows how the research on the Map Matching problem has evolved. It started from simple, fast but inaccurate methods based on geometry; going as far as more complicated and precise methods based on probability or more advanced theories. However, after more than ten years' development, most of the methods mentioned have been outperformed by their new successors and the previous categorization also requires a review.

Several surveys proposed afterwards categorized again the resolution method in different perspectives. For example, Hashemi et al. [3] focused on *on-line Map Matching* scenario and proposed an updated reclassification of them. Kubička et al. [4] distinguished the Map Matching problem according to the application field. It proposed three other categories named: *navigation*, *tracking* and *mapping*.

Wei et al. [5] discriminated the Map Matching problem on how the solutions determine the most probable match for a sample by calculating a weight for each candidate road location. The output route is the candidate sequence with the **highest** score. They call these categories of methods *max-weight methods*, which can be further divided into

*incremental max-weight* and *global max-weight* methods. The other major category consists of *global geometric methods*, which determine the best matched path solely by geometric measures such as *Fréchet distance*.

*Incremental max-weight methods* determine the output for each point **before** advancing to the next one. *Global max-weight methods* consider **the whole trace** in determining the output. In the most recent work on Map Matching problem produced by Pingfu et al. [6], another and more comprehensive classification has been proposed. It is based on the model used to resolve the Map Matching problem; in other words, on the structure, the tools and methods used to determine the candidate route.

As model is meant the whole framework or the theoretical principle used to calculate the result. It is usually a set of modules each of which performs a specific activity. Furthermore, the way they are related to each other also defines the model itself.

This type of approach overcomes some limitations encountered in previous classifications in which for example: the geometric category has lost its relevance due to the progress of resolution techniques; the classification by application field is not able to really distinguish the methods because the same tool can be used in different ones (e.g. Markov chains) and finally, the classification by scoring methods used suffers of the same problem since, as the previous one, the same mathematical tool can be used for different purposes vanishing the distinguish. For example, the Kalman filter can be used both to estimate the error committed by the GPS signal and to relate measurements from multiple sources.

So, according to this last classification, we identify four classes: *similarity models*, *state transition models*, *candidates evolving model*s and *scoring model*s. **Similarity models** use a general approach that returns the candidate route *closest* to the trajectory based on the geometry and the topology of the road map. Since, as said, the context in which we manage the Map Matching problem is the vehicle's localization on a road, its trajectory must follow the geometry and the topology of the map because, by hypothesis, the vehicle's movement must be on roads.

In this model, the main aspect is *how define the closeness*. **State-transition models** build a weighed topological graph which contains all possible routes the vehicle might travel. In this graph the vertices are the possible *states* the vehicle may be located at a particular moment, while the edges represent the *transitions* between states at different times. The weight related to an element of the graph represents the possibility of a state or a transition, and the best matching result comes from the *optimal path in the graph*.

**Candidate-evolving models** refer to models which hold a set of candidates (also called particles or hypothesis) during the Map Matching process. The candidates set is initiated based on the first trajectory sample and keeps evolving by adding new candidates propagated from old ones close to the latest measurements while pruning irrelevant one.

Interpreting a candidate as a vote, these algorithms can find a segment with the most votes, thereby determining the matching path. Compared to the *state-transition models*, the *candidate-evolving models* are more robust to the off-track matching issue since the current matching is influenced not only by previously defined solution, but also by other candidates. **Scoring models** are a group of algorithms which assign a group of candidates to each trajectory segment, and find a road edge, from each group, that maximize the predefined scoring function.

The found segment in every timestamp is either returned if applied to the on-line scenario or waited to be joined with other matched segments if applied in off-line scenario. This kind of model identifies the candidate using the information into the given road map and splitting them accordingly. At each timestep, the partitions are evaluated with a score relative to the observed position of the vehicle. The partition with the highest score, that will contain the best candidate, is returned.

The scoring function is a *linear combination of four features*: the proximity between the grid and the trajectory samples, the estimated vehicle's location in the next timestep, the reachability of the actual location from the considered partition and the probability that the vehicle turns.

About the *offline* scenario, it seems quite strange to determine the optimal route *after* that the entire trajectory is obtained. But the classification proposed by Pingfu et al [6] includes also the aspect of sampling frequency insisting that GPS information is recorded at low sampling rates in order to limit energy consumption and costs by simplifying the required hardware.

Moreover, a big data collection has made in Beijing on more than ten thousand cabs, and it stated that the 66% of the data were record from low-rate sampling GPS [7]. So, Map Matching algorithms with low-frequency GPS data collection has gained attention in recent year and Yuan et al [7] developed an interactive voting-base Map Matching algorithm in which spatial information, temporal information and the mutual correlation between matched points and GPS ones are used to stand up a voting method in order to obtain the matching candidate road.

By the way, also in *offline* scenario we got shortcomings to face. The three main problems are that few models achieve both high speed and high accuracy. The needed of some additional information as the vehicle's speed and angle or direction. Indeed, in *offline* algorithms the analysis is made after the journey has taken, so this additional information requires to be stored in some way. Finally, some of this kind of algorithm needs to *hyperparameter* for tuning the results and this must be calculated using a lot of time and resources.

## 1.3. An Enhanced Map Matching Algorithm for Real Time Position Accuracy Improvement with a Low-Cost GPS Receiver

This algorithm (Kang et al. [8]) improves real-time position accuracy for a low-cost GPS. It uses geographic data for creating a digital road database from digital map information.

The Map Matching links the vehicle's position to the digital map, improving accuracy. Thereafter, an iterative closest point method is used to calculate the rotation matrix and the transaction vector to match the GPS position with the one obtained on the digital map.

The last step is to increase the accuracy by correcting the error introduced by the matrix and vector calculation using the least square method.

This algorithm works into localization scenario, especially in that related to unmanned industry. The main topic in this research branch is unmanned ground vehicles for autonomous driving.

Here four representative technologies are considered when performing a complete unmanned autonomous drive to a fixed destination. They are: the position estimation that can obtain the current vehicle's position; the environment perception that can recognize the surrounding environmental information by various sensors; the obstacle avoiding that can move the vehicle without hitting objects to its destination and the path planning that generates an optimal path to a destination in a given environment. Of course, position estimation is the basis of this method.

Most of the present position estimation technologies are based on the global positioning systems identified as Global Navigation Satellite System[3] (GNSS), such as the Global Positioning System (GPS). Many studies and advances in technology have improved the accuracy of GNSS (with a maximum error of 10 meters without the presence of interference), but in signal degraded environments the GNSS cannot guarantee the same efficiency and continuous positioning.

To compensate for this shortcoming, GNSS uses satellite constellations. However, an easier approach is the *differential GPS*, which inverts the error contained in each satellite signal at the reference station where the exact position is known and improves the accuracy using a separate communication network. Its disadvantage is a possible reliability problem on the separate network.

[3] A Global Navigation Satellite System is a land, sea or air geo-radiolocation and navigation system that uses a network of artificial satellites in orbit and pseudo-satellites on the earth [37] [38](Wang et al. [39]).

Other approaches rely on adding sensor like Inertial Measurement Unit[4] (IMU) onto the vehicle. But these solutions require additional infrastructure, hence drawbacks of demanding a large amount of calculation and costs.

Other strategies for enhancing position accuracy use Map Matching on precise digital maps.

Most of them are tailored toward mapping the current position onto a vector representation of a road to the closest node. It is also known as *point-to-point* Map Matching. A *point-to-curve* Map Matching exists also, which matches a location point to the closest curve on the map [9].

Beside these techniques, a weighting method of the links has been proposed to further improve the accuracy of the Map Matching algorithm. It is easy to perform, but the amount of computation and the position error increase as the number of edges.

Other geometric approaches have been developed to achieve better accuracy. They consist of comparing the trajectory of a vehicle with the road of the digital map.

Unfortunately, they have got a high computational cost. So, they are insufficient for real time application with low sampling process.

In the early 2000s, probabilistic methods and methods based on fuzzy logic were also proposed [10] [11]. They can quickly recover from a location-map match; but unfortunately, they suffer from a limited sampling frequency of GPS positions.

Additional sensors have also been applied to Map Matching, but they have the disadvantage of increasing the probability of breakages and the computing power required to process the additional information. Furthermore, they have found no use except in locating the object within a narrow radius of action, such as placement in a street or aisle.

To manage low sampling, an *iterative closest point* (ICP) method has been presented. Indeed, it uses only the GPS signal and the information deduced from the digital map.

It matches the difference between the GPS trajectory and the information extrapolated from the digital map considering only a single direction using the center of the road as a reference.

## 1.4. Off-line Map Matching algorithm using time expanded graph and low frequency data

In *offline* scenario an algorithm using previous research on time expanded graph technology (TEG) [12] has been developed. It leverages this concept and represents the

[4] An inertial measurement unit (IMU) is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the orientation of the body, using a combination of accelerometers, gyroscopes, and sometimes magnetometers.

Map Matching problem as the shortest path finding. This allows to achieve high speed and accuracy with only position data. Moreover, the weight elaboration based on TEG technology makes the algorithm parameter free [13].

The time extended graph is a technique that, for each measured position by the GPS, calculates the *potential path area* instead the candidate points. So, it is not necessary to determine the shortest path between two candidate points, but instead, the algorithm produces a network-time roads using the expected travel time and the expected dwell time at the possible arc of the graph.

Although this approach succeeds both in offline and real-time Map Matching, it has been used with a high sampling rate of the GPS, but few studies showed that it is valid also with a low sampling rate of the global position system. Therefore, this algorithm relies on this feature combining time expected graph with low sampling rate GPS.

The offline Map Matching algorithm using the time expected graph and low frequency data abstract the problem discarding the elevation. Thus, all considerations, practical and theoretical, will be based on a two-dimensional model of the road map. The graph on which the algorithm works has got two kinds of nodes: *junction nodes* and *shape nodes*. A road is represented as a polyline composed by an arc of the graph, but it is oriented. This means that if a road is a two-way street, in the graph there will be two edges: one that connects two nodes in one direction and one that connects them in the other. Moreover, the first and the last point of the road are called *junctions* while those included in the polyline are defined *shape nodes*.

So, let $V$ be the set of *junction* and *shape nodes*, $A$ be the set of the arcs $G$ be a road network with $A$ as edge set. An arc $a$ is represented as a node sequence $a := (v_1, v_2, \dots, v_m)$ with $\{v_j\}_{j=1}^{m} \subset V$ and every two-way road is resented such as $(v_1, v_2, \dots, v_m) \in A$ and $(v_m, v_{m-1}, \dots, v_1) \in A$.

A pair of two consecutive points defines a *shape arc,* and it is denoted by $(v_i, v_{i+1})$ where $v_i, v_{i+1} \in V$. A vehicle trajectory is a chronologically ordered positions $\boldsymbol{P} = (P_1, P_2, \dots, P_{n+1})$ generated by a GPS device. For each time step $i$, $P_i$ includes latitude and longitude coordinates. The algorithm aims to find the most likely path $(a_1, a_2, \dots, a_m)$ (with $a_j \in A$ and $j = 1, \dots, m$) of the vehicle given a trajectory $\boldsymbol{P} = (P_i)_{i=1}^{n+1}$. This path is referred as the *matched path* and the arcs included into it are called *matched arcs*. The *correct path* denotes a finite sequence in $A$ on which the vehicle actually travels. Of course, the arcs into the *correct path* are defined as *correct arcs*.

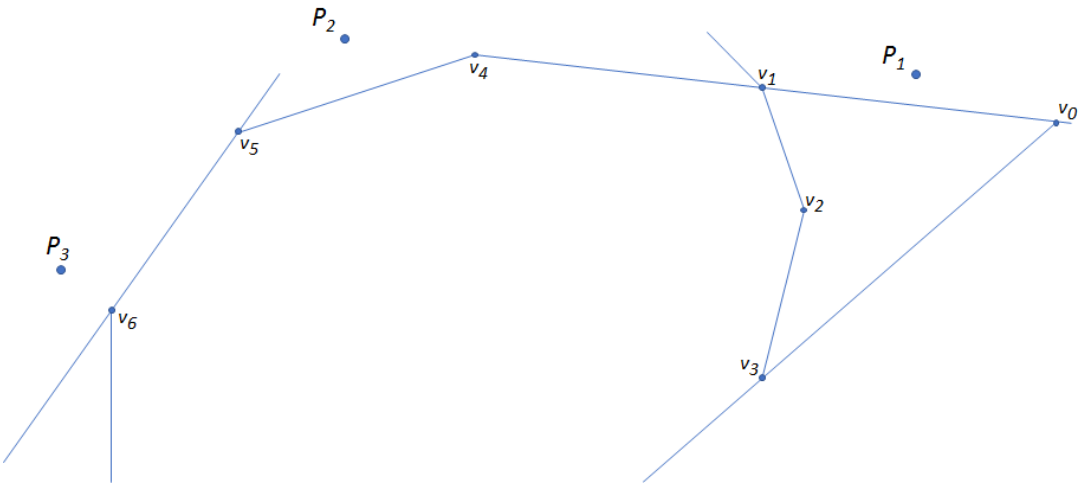



Figure 1-1: Example of a time expanded graph

Junction: $v_0, v_1, v_3, v_5, v_6$

Shape node: $v_2, v_4$

Shape arc: $(v_0, v_1), (v_1, v_0), (v_1, v_2), \ldots$

Arc: $(v_0, v_1), (v_1, v_0),\ (v_1, v_2, v_3), (v_3, v_2, v_1\ ), \ldots$

Matched path: $\big((v_0, v_1), (v_1, v_4, v_5), (v_5, v_6), \ldots\big)$

The main hypothesis is if the area between the vehicle trajectory and the *matched path* is sufficiently small, the *matched path* coincides with the *correct path*. So, the algorithm finds the *matched path* that reduces this area. To do this, a potential area is defined over which the vehicle could transit between time instants $i$ and$i + 1$. Then, each pair of two consecutive positions $(P_i, P_{i+1})$ is simultaneously to a path (named *partial path*) on graph $L(G, P_i)$.

At the same time, the connectivity of the two consecutive *partial paths* is maintained and each one tends to have few abrupt direction changes, creating, in this way, a small area with the positions $(P_i, P_{i+1})$. The matching is given by identifying the smallest area between $(P_i, P_{i+1})$ and a selected *partial path*. This approach results in an algorithm based on the TEG-matching and free from line parameters that requires only the vehicle's ordered positions. Moreover, applying fractional cascading (FC) to candidate *shape nodes* a boost is obtained and applying the bottom-up segmentation to the road network another boost is gained, but in this case, the reduction of *shape nodes* requires a small cost in terms of accuracy of the result.

The proposed algorithm starts with a preprocess step in which a distinction between *junction nodes* and *shape nodes* is made. So, every node of the road network is classified into these two categories according to its topology. Here, long *shape arcs* are split such that their maximum length is less or equal to a predefined parameter $\ell_{max}$. The reason

is that we may overlook *arcs* close to a certain point if the *arc* contains some long *shape arcs*. An example is when both end points of a long *shape arc* are far from the trajectory point because the algorithm reports if at least one *shape node* of the *arc* is in a square centered on the trajectory point. The road network resulting from this process is called *processed road network*.

After that it starts to obtain the *arcs* close to the vehicle's trajectory and it uses classic geometry. This allows the algorithm to report those within *r* meters from a trajectory point, and, by definition they are the arcs that fall inside an imaginary circle of radius *r*. However, this process is computationally expensive especially if the road network has got a huge number of *arcs*. So, to speed up a fractional cascading (FC) technique is used to report every *arc* that has got at least one *shape node* belonging to a square centered on a trajectory point.

It is possible to prove that if we set the side length of the square to $c = \max\left(r, \frac{\ell_{max}+2r}{2\sqrt{2}}\right)$ if $\ell_{max} \leq 2\left(1+\sqrt{2}\right)r$ (otherwise $c = \frac{\ell_{max}}{2}$), we obtain either of the endpoints of the *shape arc* within a radius $r$ meter from a certain trajectory point. This implies the algorithm acquires all *arcs* within $r$ meters from the considered trajectory point. So, before performing the Map-Matching, a Fractional Cascading (FC) data structure with all the *shape nodes* of *processed road network* has been constructed. Then it will be used in the next step TEG-matching obtaining neighborhood *arcs*. At the end of this step, the bottom-segmentation is used to reduce the redundant *shape nodes* of the *processed road network* contributing to memory reduction and boosting the Map Matching process.

At this point the Time Expanded Graph (TEG) can be built, and trajectory neighborhood can be found. So, given the vehicle's trajectory $\boldsymbol{P} = (P_i)_{i=1}^{n+1}$, the algorithm determines the *arcs* where the vehicle may travel from time stamp $i$ to $i+1$ ($i \in \{1, \dots, n\}$). The hypothesis is that these *arcs* lie within $r' = d(P_i, P_{i+1})/2 + r_{GPS}$ from the midpoint of $P_i$ and $P_{i+1}$ (also called $P_{i,i+1}$) where $r_{GPS}$ is the upper bound of the spatial measurement error and $d(x, y)$ is the Euclidean distance between $x$ and $y$. Now, recalling what was said previously, if we set $\ell_{MAX} = 2\left(1+\sqrt{2}\right)r_{GPS}\left(\leq 2\left(1+\sqrt{2}\right)r'\right)$ and $c_i = \max\left(r', \frac{\ell_{MAX}+2r'}{2\sqrt{2}}\right)$ we can obtain either endpoints of shape arcs within $r'$ meters from $P_{i,i+1}$, and get all *arcs* that lie within $r'$ and $P_{i,i+1}$. The FC technique can be used to speed up this process.

Once the TEG has been built, it represents the space-time movements of the vehicle. To attain the most plausible *matched path*, the algorithm finds the shortest path on the TEG and infers the *matched path* from the shortest one.

## 1.5. Delay Estimation for Shared Rides from GPS Data

This algorithm is an example of how broad the effective possibility of using the Map Matching problem is and how infinite its fields of application are.

In Samavati et al. [14], Map Matching is employed in the localization of people who need to move around the city in the shortest possible time or at least in order not to miss their expected arrival time at their destination.

The main concept is the *ridesharing* which is a way for more riders to go where they must go sharing a single vehicle, like a car or a van, that's going in their direction. This vehicle makes stops along the route, picking up or dropping off people.

An example of this topic is the European project called **RIDE2RAIL** and it aims to combine ride-sharing services with other forms of transportation. The goal is to provide users with a variety of options for their trips.

Since people must wait for the ride to arrive at their destination, the timing is crucial. And very precise timing facilitates the integration of ride sharing into a wider multi-modal ecosystem.

In this scenario planning and monitoring the timelines of the ride is critical, since the ride operates as a minibus connected to a wider transportation network. So, it must meet its appointed target time.

The mechanism detects delays affecting a shared ride using GPS data of vehicle in urban context. The tool on which it relies is the Map Matching that helps to localize people on van and estimating the possible delay.

The latter is used to inform people of disruptions that impact on their trips and, eventually, and possibly propose new journey planning if the delays are such as to impact the possibility of missing the connections of the means of transport.

It is worth noting that all location calculations are meant to be performed locally on the user's device.

# 2 Integrating Spatio-temporal Proximity and Improved Weighted Circle algorithm

In this chapter we illustrate a new algorithm proposed to compensate for the limitations of real-time Map Matching, which include low accuracy. It places itself within the scope of real-time map matching. The decision to study it in depth and implement it is because we believe it is a good compromise between the computing power required to run it and the mobile resources on which it will be run.

In the first paragraph we will see the general lines of the algorithm. Later we will illustrate its details showing the theory on which it is based on. *Finally, we will conclude the chapter by showing the implementation of some possible variants of the process.*

## 2.1. STP-IWC main lines

To respond to the time lag and low accuracy in the real time Map Matching problem; especially in that related to the in-vehicle navigation systems, a new algorithm has been proposed [15]. This new method integrates the *spatio-temporal proximity* (STP) and an advanced weight distribution called *improved weight circles* (IWC). The first refines the candidate matching roads and the latter adaptively identifies the optimal one.

In this algorithm, the *spatio-temporal proximity* part (STP) is designed to create a dynamic three-dimensional cone by introducing three spatio-temporal proximity variables: GPS coordinates, location interval and vehicle's speed. In this way, it ensures that the candidate matching roads can be dynamically selected from the projected two-dimensional plane of the stereoscopic cones, which also causes the candidate region to dynamically adapt itself to the real-time speed of the vehicle.

The *improved weight circles* (IWC) part integrates three parameters (vehicle's driving angle, projection distance from vehicle to candidate road and similarity between road trajectory and real road) to adaptively determine the best matching road. By this integration between STP and IWC, the real time Map Matching problem can be solved efficiently and accurately.

It is important to remember that the assumption on which the method is based is that the vehicle is moving on roads. So, it is not possible that there is no candidate road at some point of the execution.

The overall process of the method is the following:

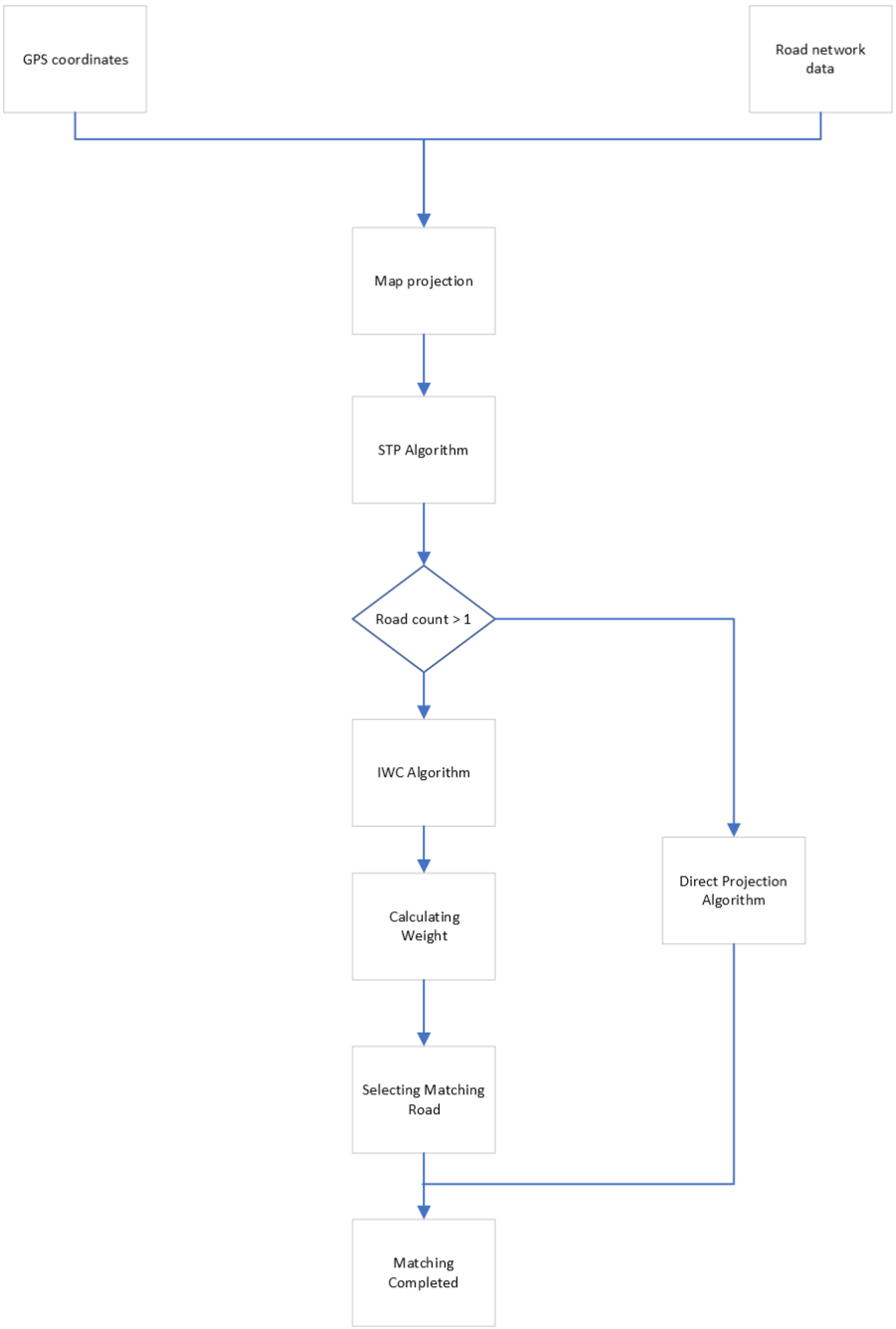



Figure 2-1: STP-IWC entire process

## 2.2. STP part and candidate roads search range

Considering that the real position of the vehicle is not given only by the GPS coordinates, but, also, by its cruising speed, the STP part (introduced in Kuijpers et al. [16]) of the algorithm integrates both together. It adds also GPS sampling time. So, to adaptively construct a dynamic search region, in which narrow the candidate roads with great matching possibilities, it uses the vehicle's speed ($v_i$), the GPS coordinates ($x_i$, $y_i$) and the time elapsed between two GPS samples ($t$).

In Figure 2-2, we can see that, as time goes by, the GPS produces a sequence of sampling points $p_i(t_i, x_i, y_i)$, where $t_i$ represents the instant in which the GPS sample is generated, while $x_i$ and $y_i$ are the latitude and longitude of the vehicle at that moment. Since it is a sampling, we can define $i \in \mathbb{N}^+$. Then, taken $p_i$ and $p_{i+1}$ as GPS points received at time $t_i$ and $t_{i+1}$, the STP constructs two three-dimensional cones using the GPS points as cone vertex and the values $t_i * v_i$ and $t_{i+1} * v_{i+1}$ as radius.

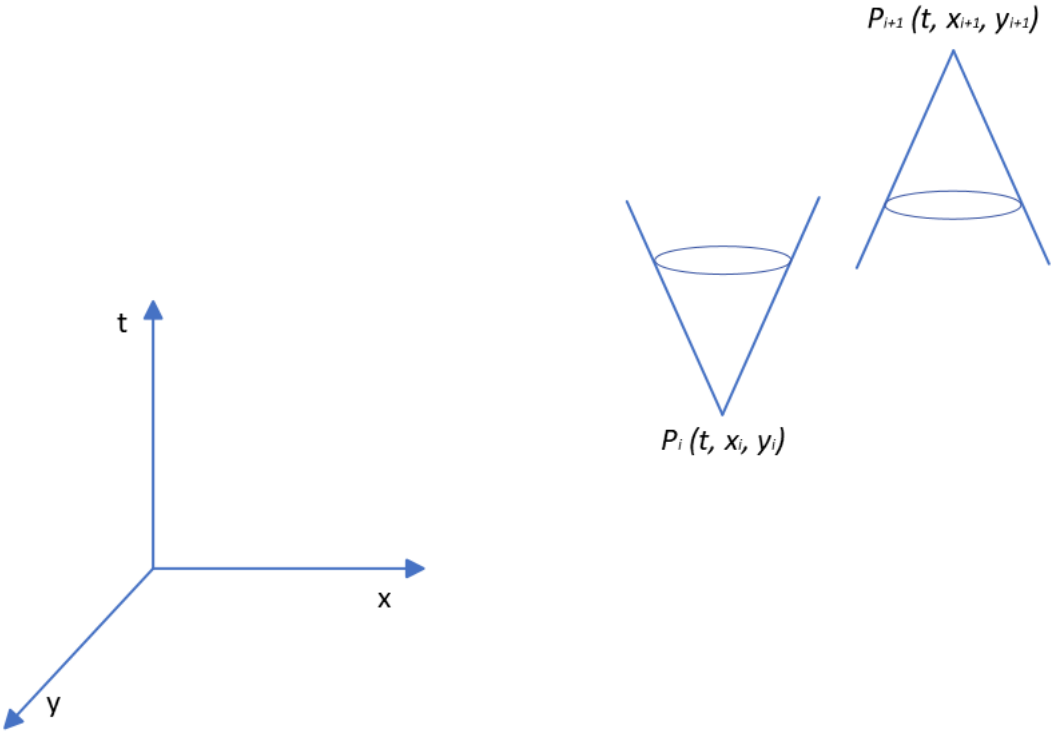



Figure 2-2: Spatial Cones in STP

Subsequently, the intersection between these two solids is calculated and it is projected onto a two-dimensional plane obtaining an ellipse. So, the vehicle's real-time location $t \in [t_i, t_{i+1}]$ will be contained in the ellipse (Figure 2-3) and it will exist if the following three conditions will be satisfied:

$$t_i \leq t \leq t_{i+1} \qquad (2.1)$$

$$\left(x - x_i^2\right) + \left(y - y_i^2\right) \leq v_i^2 (t - t_i)^2 \quad (2.2)$$

$$(x - x_{i+1})^2 + (y - y_{i+1})^2 \leq v_i^2 (t - t_{i+1})^2 \quad (2.3)$$

We can explain these conditions with an example (Figure 2-4 and Figure 2-5). Given two GPS points, one after the other, $p_1(t_1, x_1, y_1)$ and $p_2(t_2, x_2, y_2)$ and their related speed $v_1$, $v_2$, the STP algorithm builds two three-dimensional cones in the space $(x, y, t)$ using $p_1$ as the vertex of the first cone and $p_2$ as the vertex of the second one. Moreover, it uses the products $(t - t_1) * v_1$ and $(t_2 - t) * v_2$ as the radius of the first and the second cone, respectively.

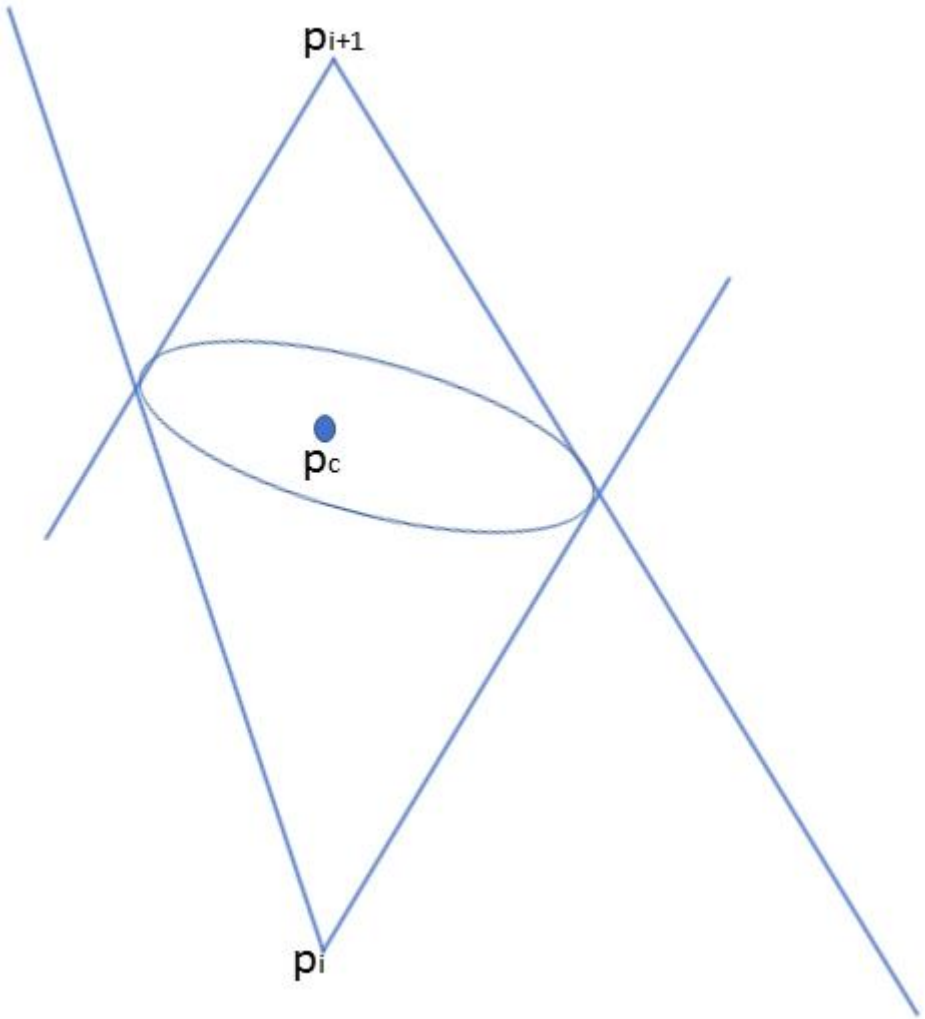



Figure 2-3: Cones intesection

Then, an ellipse is formed when the projections of these two cones intersect on a two-dimensional plane. The two focus points of the resultant geometric shape are the two vertices $p_1(x_1, y_1)$ and $p_2(x_2, y_2)$, while the semi-axis is $\frac{v_1 * (t_2 - t_1)}{2}$. Therefore, the real-

time location of the vehicle at time $t$ will exist in the ellipse and this one will be the search range of the candidate roads.

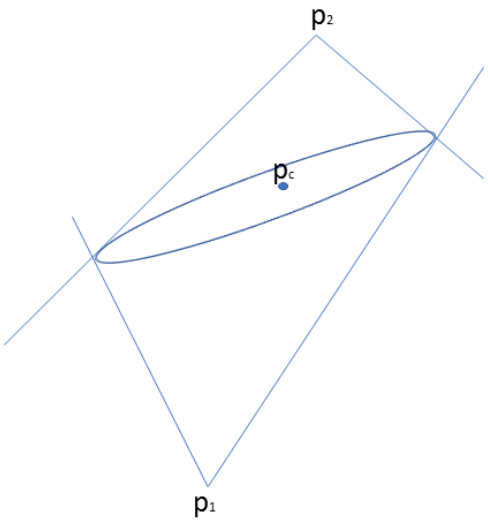



Figure 2-4: Cones intersection example

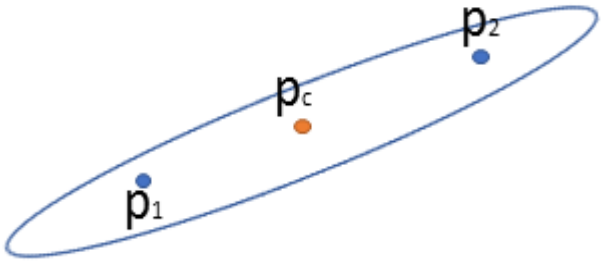



Figure 2-5: Intersection projection

To improve operational efficiency and simplify algorithm code, we can consider the *minimum* rectangle that contains it as search scope for selecting the candidate roads. Starting from the example in Figure 2-4 we know that $(x_1, y_1)$ and $(x_2, y_2)$ are the focal points of the ellipse created from the projection of cones intersection. So, we can define $d_f$ the distance between them.

Let $p_c(x_c, y_c)$ be the mid-point of the two focal points and $l$ and $L$ be the semi-minor axis and the major semi-axis, respectively, then the border rectangle is $[X_1, X_2] \times [Y_1, Y_2]$ on the condition that $x_1 \neq x_2$. If $x_1 = x_2$ then $X_1 = x_c - l$, $X_2 = x_c + l$, $Y_1 = y_c + L$, $Y_2 = y_c + L$. If $y_1 = y_2$ then $L = l$.

$$x_c = \frac{x_1 + x_2}{2} \quad (2\text{-}1), \qquad y_c = \frac{y_1 + y_2}{2} \quad (2\text{-}2),$$

$$L = v_1(t_2 - t_1) \quad (2\text{-}3), \qquad d_f = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (2\text{-}4),$$

$$l = \frac{1}{2}\sqrt{4L^2 - d_f} \quad (2\text{-}5), \qquad s = \frac{y_1 - y_2}{x_1 - x_2} \quad (2\text{-}6),$$

$$X_1 = x_c - \sqrt{\frac{s^2 l^2 + L^2}{1 + s^2}} \quad (2\text{-}7), \qquad X_2 = x_c + \sqrt{\frac{s^2 l^2 + L^2}{1 + s^2}} \quad (2\text{-}8),$$

$$Y_1 = y_c - \sqrt{\frac{s^2L^2 + l^2}{1 + s^2}} \quad (2\text{-}9), \qquad Y_2 = y_c + \sqrt{\frac{s^2L^2 + l^2}{1 + s^2}} \quad (2\text{-}10)$$

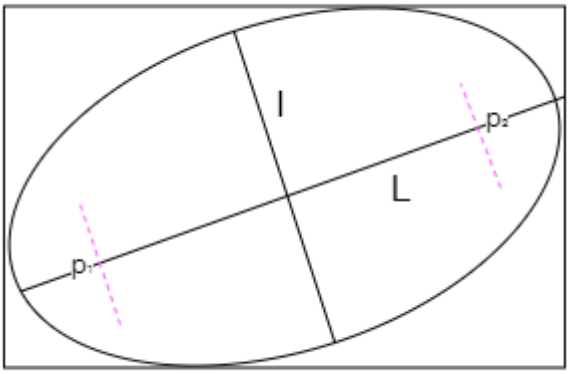



Figure 2-6: Ellipse intersection projection

## 2.3. IWC part and candidate road determination

Once the area, in which the road we are looking for lies (the one on which our vehicle is located), has been identified, the algorithm that assigns weights to the candidates begins its execution and then finally obtains the best candidate; that is, *the road that will get the greatest weight*. The basic version of this procedure involves the use of two main parameters: the projection of the distance from the point of the trajectory to the candidate road considered ($d$), and the angle between the direction of the trajectory and the direction of the candidate road ($a$)

In this version of the algorithm part, the weight $u = W_d d + W_a a$ is selected. Where $W_d$ and $W_a$ denotes the weight of $d$ and $a$ [17]. These ones are calculated using the topological information on the whole search range and required a lot of calculations to get the best. Found the maximum $u$, the corresponding candidate road is selected as result. Moreover, when this weighting map matching process encounters two consecutive GPS points that are at the corner of a road intersection, it calculates more candidate roads with the same weight. So, it cannot find the optimal candidate road.

To overcome this limit, the IWC part of the algorithm introduces another parameter called *angle similarity* ($b$) which indicates the angle between the vehicle's travel trajectory and the candidates chosen in the previous step. The reason is that there will

be an angle difference among the candidate roads direction at a crossroad. Finally, the IWC part assigns adaptively weights to these three parameters: $d$, $a$ and $b$.

Indeed, traditional weighting map algorithms [17] use static ways to assign weight to candidates without adjusting dynamically them on roads condition. Therefore, these versions lack precision and have not got enough strong theoretical foundation. So, according to the principle that the nearer the location point is to the road intersection, the greater the weight for $a$ $b$ parameters and the smaller the weight for $d$, the improved weight circles (IWC) sub process adaptively sets them.

To do this, it is used the weight circle techniques [18] improved by adaptation. The weighted circles are established with the crossroad center (or intersection point) $O$ as circles center and a radius $R$ as long as the turning points of the candidate roads. This means that they are different and reflect the adaptivity to the candidate roads.

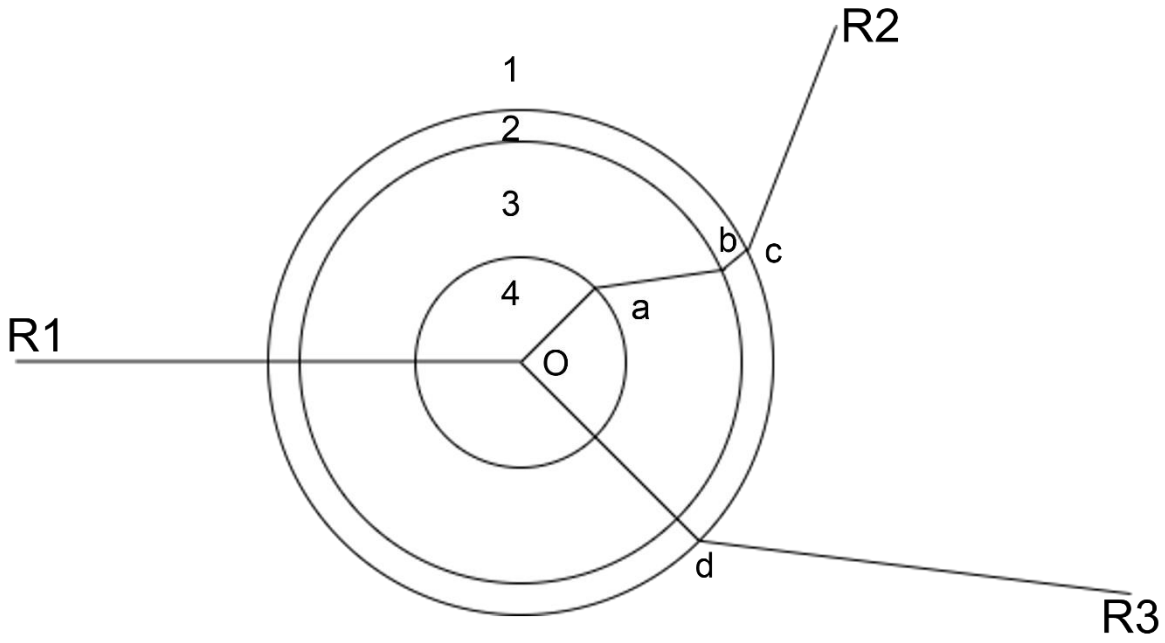



Figure 2-7: Weighted Circles

So, around a roads intersection there exists a bunch of selected candidate roads by STP part $R1$, $R2$ and $R3$ (Figure 2-7). The IWC process establishes the circles' centers in that point (crossroad center) and considers a radius $R$ as long as the turning points of the roads $a$, $b$, $c$ and $d$; obtaining the subdivision of the research area into distinct regions (1, 2, 3, 4). Then, the smaller the distance between the vehicle and the intersection point $O$ the bigger is the possibility that it is turning and the greater the importance of the angle than the distance.

Therefore, the distance and angle weights ($w_d$, $w_a$) follow these rules:

**Rule 1:** if the vehicle is in the greatest region (region 1) then it is far from intersection point $O$ and the distance $d$ will weigh more in the choice of the candidate. For this reason, the IWC sets $w_d = 1$.

**Rule 2:** if the vehicle is not so far from the center $O$, the IWC defines the value of the distance weight $w_d$ as $\frac{|OP|}{R}$ and $w_a = 1 - w_d$. $|OP|$ is the distance between the vehicle (that is the point at time $t_i$ of the trajectory measured by GPS $P_1(x_i, y_i)$) and the intersection point $O$. Whereas $R$ is the radius of the *smallest circle that contains the vehicle*.

**Rule 3:** if the vehicle is beyond the greatest region (region 1), it means $|OP| > R_{MAX}$ and we set $|OP| = R_{MAX}$; where $R_{MAX}$ is the radius of the largest region.

## 2.4. STP-IWC in more detail

Now that we have illustrated the main parts of the algorithm, we can put it all together and see in detail the whole procedure. So, in the next figure (Figure 2-8), we show the entire algorithm described till now and, since we rely on the points sampled by the GPS, we can assume that we have time flowing discretely defining an index $j$ that goes from 1 to n to indicate the instants of time ($j = 1, 2, 3, \dots, n$).

The first step involves the STP part to determine the candidate roads according to the trajectory point $p_j$ returned by the GPS at time $j$. As said before the aim of STP is to identify a region in the space in which there are the candidate roads. This means that the number of points used is (at least) two because the algorithm needs to build the spatial temporal cones using the point $p_j$ and the point $p_{j+1}$.

It is possible that into the calculated region there is only one candidate road. In this case, that is the best matching road, and the process goes on to the next time $j$. If not, the vehicle is moving in a complex scenario and the IWC part is requested to identify the best matching candidate road among all possible.

The IWC part works on three trajectory points, $p_j$, $p_{j+1}$ and $p_{j+2}$, and on the assumption that the first two ($p_j$, $p_{j+1}$) have been already matched. So, the first step is to calculate the distance $d$ between $p_{j+2}$ and the considered candidate road. Then the distance $|OP|$ between the trajectory points $p_{j+2}$ and the road intersection points $O$ must be calculated. These intersection points are all in which the candidate roads, all those in the region, not only the considered one, have got a crossroad.

After that, the weight of the parameter $d$ is set as the rules above indicate, that is if $|OP| < R_{MAX}$ then the weight $w_d$ is $|OP|/R$ else is 1. Keep in mind that $R$ is the radius (distance) of the smallest circle that contains $p_{j+2}$.

The second step is to calculate the parameter $a$, that is the angle between the driving direction (trajectory) and the considered candidate road.

It is defined as:

$$a = \left| \operatorname{arccot} \frac{y_{j+2} - y_{j+1}}{x_{j+2} - x_{j+1}} - m \right| \tag{2-11}$$

Where $m$ is the direction of the considered roads.

Subsequently, as we know, the improved weighted circles introduced another parameter alongside the classic $d$ and $a$, namely the angle similarity $b$ which indicates the angle between the trajectory direction and that of all candidate roads determinate by the STP part. The angle similarity expression is:

$$b = \frac{\left( \sum_{j=1}^{n} \frac{1}{y_j} \sum_{j=1}^{n} \left( \frac{x_j}{y_j} \right)^2 - \sum_{j=1}^{n} \frac{x_j}{y_j} \sum_{j=1}^{n} \frac{x_j}{y_j^2} \right)}{\left( \sum_{j=1}^{n} \frac{1}{y_j^2} \sum_{j=1}^{n} \left( \frac{x_j}{y_j} \right)^2 - \sum_{j=1}^{n} \left( \frac{x_j}{y_j^2} \right)^2 \right)} \tag{2-12}$$

Where $n$ is the number of the trajectory point given by the GPS and $j = 1, 2, 3, \ldots$ are *consecutive* GPS points. Since the number of points to be considered increases as time passes, for reasons of efficiency the algorithm limits the number of points to three: the current one $p_{j+2}$ plus the two previous ones ($p_j$ and $p_{j+1}$).

These steps in which the algorithm calculates the parameters are repeated for any candidate road included in the region identified by the STP. The result is the weight to associate to the current candidate roads, that is $w_{qi}$:

$$w_{qi} = w_d \bar{d} + w_a (\bar{a} + \bar{b}) \tag{2-13}$$

In which $w_a = 1 - w_d$ as stated before, and

$$\bar{d} = \frac{1}{d_i \sum_{i=1}^{k} \frac{1}{d_i}}, \tag{2-14a}$$

$$\bar{a} = \frac{1}{a_i \sum_{i=1}^{k} \frac{1}{a_i}} \tag{2-15b}$$

$$\bar{b} = \frac{1}{b_i \sum_{i=1}^{k} \frac{1}{b_i}} \tag{2-16c}$$

Where $i = 1, 2, \ldots, k$ that are the $k$ candidate roads included in the region.

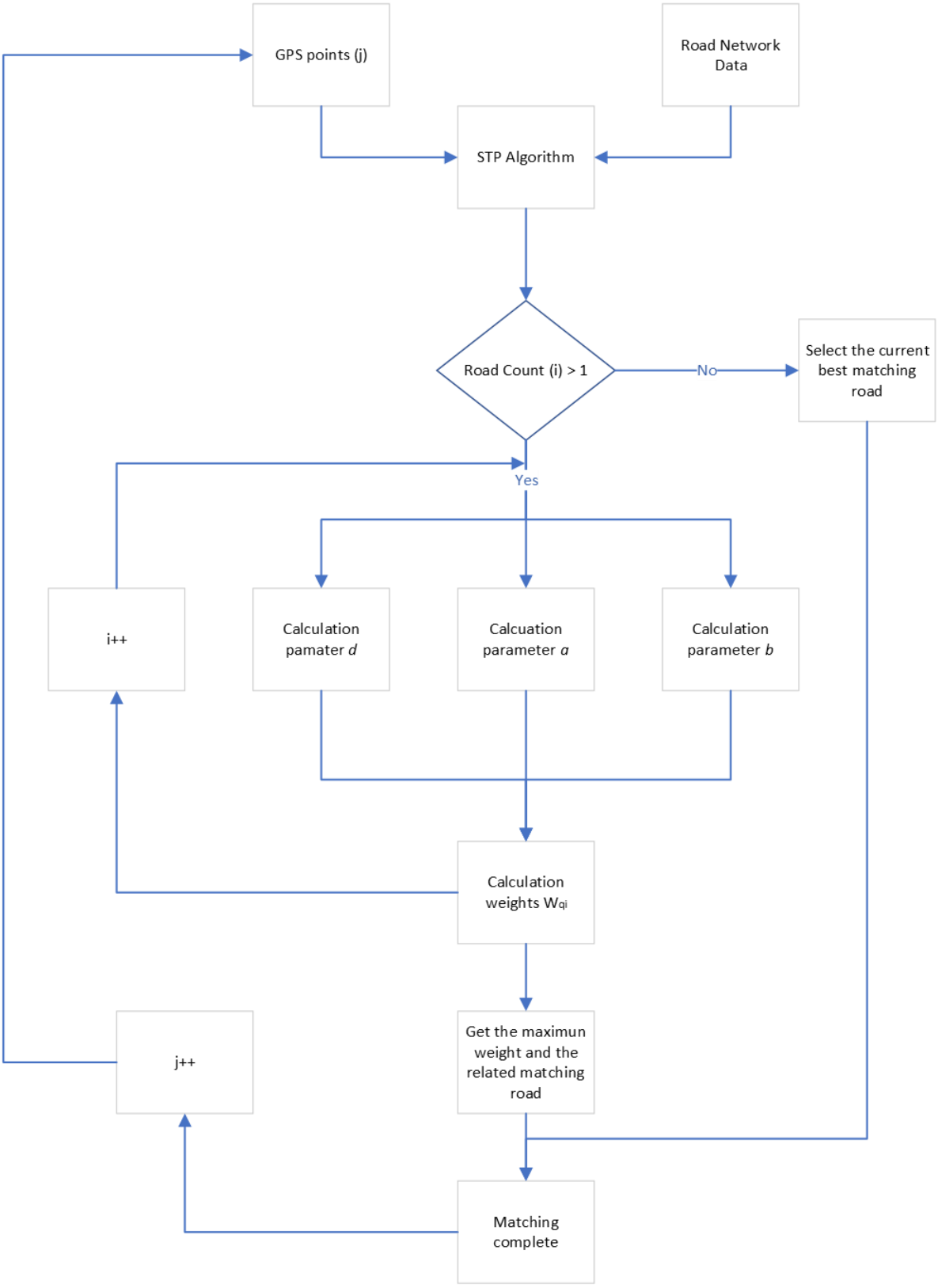



Figure 2-8: whole STP-IWC algorithm

Last step is kept the best matching road which is that with the *maximum* weight $w_{qi}$. This procedure is repeated for all the roads identified by the GPS data, and it ends when the GPS does not supply any new point.

# 3 Implementation

In this chapter, we are going to show a possible implementation of the algorithm presented in chapter 2. The programming language chosen is Java for its simplicity and its vast availability of libraries that can be used to implement all the data structures necessary for the algorithm to work. Furthermore, the object-oriented programming paradigm was very useful during the development and Java is certainly effective in this sense.

We will start with the development of the version illustrated previously, which will be called the "classic version", and then we will introduce three other different types of the same process. Their comparison and analysis of the results obtained will be the subject of the next chapter.

## 3.1. General Concepts

As shown in Figure 2-8 the STP-IWC algorithm starts from two given data sets: the GPS points series and the road network data. They are what we stated in **Definition 1.1** and **Definition 1.2**. The trajectory has been represented by a sequence of GPS points using the MATLAB [19] application for iOS and its related cloud service. This program uses the mobile phone as a GPS tracker taking advantage of the GPS module installed in it. This is a clear example of how enormously widespread GPS technology and tracking devices are.

In section 4.2 on page 48 we will see in detail how the data from the mobile phone has been managed to produce a CSV file, which is imported by the algorithm using the Java input/output library. The program reads line by line the file and creates an object defined by the class `NodoPercorso` putting the information in the related fields.

This object extends another one defined by the class `Nodo` and adds information like the speed of the vehicle and the timestamp at which the GPS measure was registered. The other date stores in the object are the latitude, the longitude, and a long integer in which a unique value is put. The result of this import is a `Vector` containing all the GPS point sequences.

About the road network data, they are retrieved from precompute maps shared by the platform OpenStreetMap [20]. They are XML file formed according to a not official XSD schema.

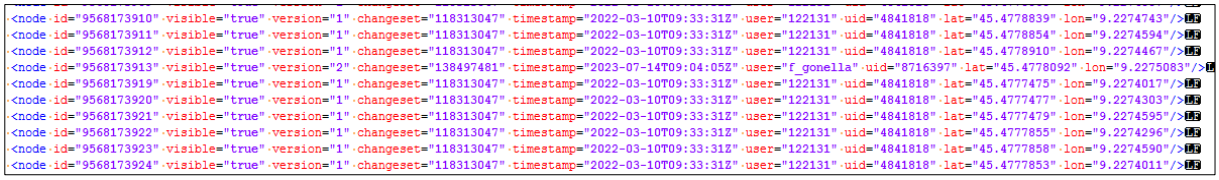



Figure 3-1 Example of node definition in a file OSM


Figure 3-2 Example of road definition in a file OSM

So, in this case, it has been implemented an XML parser using the Java `SAXEventsPathImport` and `SAXEventsHandler` classes. The latter extends the father class `DefaultHandler`[5]. We defined two methods in which the parser recognizes the begin (`startElement`) and the end (`endElement`) of a tag. According to the tag value, these methods get information from the attributes compiling latitude, and longitude object's files. In this process, also other checking was made in order to avoid multiple node insertion.

When a node is recognized and inserted into the Java set, it is marked to prevent other insertions when it is recognized again because it could also be part of another road. The same job is done to recognize a road from the XML file, but, in this case, the test is made on its attributes to distinguish from a not important road type, like cycle line, to an interested one (road).

The result of this step is the creation of two Java sets representing the nodes and the arcs among them. Then, they are passed to the `Multigraph` class[6] which creates a non-simple undirected graph in which no loops are permitted. Of course, a path in which the start node and the end node are the same is permitted, just two or more nodes must be involved.

Once these data are provided, the next step is to determine the region in which we expect there are the candidate roads. In the next section, we will see how these data

[5] https://docs.oracle.com/en/java/javase/16/docs/api/java.xml/org/xml/sax/helpers/DefaultHandler
[6] https://jgrapht.org/javadoc-SNAPSHOT/org.jgrapht.core/org/jgrapht/graph/Multigraph.html

are used and how they will lead to the *right* candidate road. "Right", in this model implementation, means the maximum weighted candidate determined.

**Algorithm 1** STP_IWC_Execution

**Input:** void
**Output:** void

```
1:  crCounter ← 0
2:  itGPSPoints ← this.GPSPoints.iterator()
3:  index ← 0
4:  cr ← nil
5:  while (itGPSPoints.hasNext()) do
6:      if (index + 2) ≥ this.GPSPoints.size() or (index + 1) ≥ this.GPSPoints.size()
        then
7:          Return
8:      end if
9:      np1 ← this.GPSPoints.elementAt(index)
10:     np2 ← this.GPSPoints.elementAt(index + 1)
11:     np3 ← this.GPSPoints.elementAt(index + 2)
12:     index ← index + 1
13:     this.spatialTemporaryProximity(np1, np2)
14:     candidateNodes ← this.roadNetwork.vertexSet().stream().filter(vertex →
        vertex.getX() > this.ds.getXMin() and vertex.getX() < this.ds.getXMax()
        and vertex.getY() > this.ds.getYMin() and vertex.getY() <
        this.ds.getYMax()).collect(Collectors.toSet())
15:     candidateEdges ← this.roadNetwork.edgeSet().stream().filter(edge →
        candidateNodes.contains(edge.getSource()) and
        candidateNodes.contains(edge.getTarget())).collect(Collectors.toSet())
16:     canditateGraph ← new AsSubgraph(this.roadNetwork, canditatesNodes,
        candidateEdges)
17:     if candidateEdges.size() = 1 then
18:         cr ← new CandidateRoad(candidateEdges.toArray()[0], 1)
19:     Else
20:         cr ← this.improvedWeightedCircles(candidateGraph, np1, np2, np3)
21:     end if
22:     crCounter ← crCounter + 1
23: end while
24: Return
```

Algorithm 1: STP_IWC_Execution

## 3.2. Classic version

So, according to the algorithm's explanation, we begin getting the first three points ($p_j$, $p_{j+1}$, $p_{j+2}$) from the GPS trajectory and we use $p_j$ and $p_{j+1}$ as input of the STP part. The last one is kept apart for the next section of the algorithm IWC. It is worth pointing out that for efficiency reasons only three trajectory points at a time are considered every time step $i$ (Figure 2-8).

From points $p_j$ and $p_{j+1}$ we get their timestamp (the time at which they were stored expressed in seconds) and the Cartesian coordinates stored into the object `NodoPercorso`, which represents a trajectory point in the code (lines 1-7, Algorithm 2). Those coordinates were determined using the equations that transform the spherical ones into Cartesian coordinates during the creation of the object:

$$\begin{cases} x = \rho\cos(\varphi)\cos(\theta) & \rho \in [0, +\infty) \\ y = \rho\cos(\varphi)\sin(\theta) & \theta \in [0, 2\pi) \\ z = \rho\sin(\varphi) & \varphi \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right] \end{cases} \quad \begin{matrix} \text{(3-1a)} \\ \text{(3-2b)} \\ \text{(3-3c)} \end{matrix}$$

After that, we initialize four variables `X1`, `X2`, `Y1`, and `Y2` (lines 8-11, Algorithm 2) that will contain the coordinates of the rectangle which identify the candidate region according to the hypotheses explained in section 2.2. Indeed, the rectangle is the approximation of the ellipse given by the intersection of two spatial-temporal cones created from $p_j$ and $p_{j+1}$ (Figure 2-3).

Then we start to calculate `L = v1*(t2-t1)`, `dx = x1 - x2`, `dy = y1 - y2`, `xc = (x1 + x2) / 2` and `yc = (y1 + y2) / 2` (lines 12-14, Algorithm 2). Done this, a first check is implemented; that is the value of `L` is tested and compared with a constant named `minDistance` defined in the support class `Utilities` of the package `util` (lines 15-17, Algorithm 2). The reason for this check is to avoid troubles in candidates' calculations due to a rectangle that is too small.

In this case, if `L` is too small, a minimum distance is used. This can be set by changing the value in class `Utilities` using what is deemed most appropriate. In our case, the value of `minDistance` is five meters.

The next steps are to calculate $d_f^2$, $L^2$, $4L^2$, `diff` as `_4L2 - df2` and `l` as `0.5 * Math.sqrt(diff)` (lines 18-19, Algorithm 2), which are intermediate and support values. Then, other checks are made in order to identify some situation in which the calculated region assumes particular forms (lines 20-53, Algorithm 2). Indeed, If `dx` is zero, for example, the vehicle is going on the ordinate axis and the (2-7), (2-8), (2-9), and (2-10) lost their meaning since the (2-6), is not more defined. Therefore, it can be demonstrated that the region $[X_1, X_2] \times [Y_1, Y_2]$ will be:

$$X_1 = x_c - l \quad (3\text{-}4) \qquad X_2 = x_c + l \quad (3\text{-}5)$$

$$Y_1 = y_c - L \quad (3\text{-}6) \qquad Y_2 = y_c + L \quad (3\text{-}7)$$

and of course, $x_c = \frac{x_1 + x_2}{2} = \frac{2x_1}{2} = \frac{2x_2}{2} = x_1 = x_2$. Also, in this case, we made a check on $L$ and $l$ to see if they are over or under the minimum value `minDistance`.

If, instead, `dy` is equal to zero ($y_1 = y_2$), the vehicle is going on the abscissa axis and the region will be a square ($L = l$):

$$X_! = x_c - L \quad (3\text{-}8) \qquad X_2 = x_c + L \quad (3\text{-}9)$$

$$Y_1 = y_1 - L \quad (3\text{-}10) \qquad Y_2 = y_1 + L \quad (3\text{-}11)$$

However, also in this case, we check the value of one between $L$ and $l$ with the usual `minDistance`.

If none of the degenerate conditions of the region is present, we go on calculating the values shown at (2-7), (2-8), and (2-10) (lines 54-68, Algorithm 2). So, the parameters $\sqrt{\frac{s^2l^2+L^2}{1+s^2}}$ and $\sqrt{\frac{s^2L^2+l^2}{1+s^2}}$ were calculated and stored in the variables `Ax` and `Ay`. Then we compare them with the usual `minDistance` value, and the region is stored into `ds`.

**Algorithm 2** Spatial Temporary Proximity

**Input:** NodoPercorso p1, NodoPercorso p2
**Output:** void

```
1:  t1 ← p1.getTimestamp()
2:  t2 ← p2.getTimestamp()
3:  v1 ← p1.getSpeed()
4:  x1 ← p1.getX()
5:  y1 ← p1.getY()
6:  x2 ← p2.getX()
7:  y2 ← p2.getY()
8:  X1 ← 0
9:  X2 ← 0
10: Y1 ← 0
11: Y2 ← 0
12: xc ← (x1 + x2)/2
13: yc ← (y1 + y2)/2
14: L ← v1*(t2 − t1)
15: if L < Utilities.minDistance then
16:         L ← Utilities.minDistance
17: end if
18: df2 ← (x1 − x2)² + (y1 − y2)²
```

19: $l \leftarrow \frac{1}{2} * \sqrt{4 * L^2 - df2}$
20: **if** *(x1 – x2) = 0* **then**
21: **if** *l < Utilities.minDistance* **then**
22: *X1 ← xc – Utilities.minAxis*
23: *X2 ← xc + Utilities.minAxis*
24: **Else**
25: *X1 ← xc – l*
26: *X2 ← xc + l*
27: **end if**
28: **if** *L < Utilities.minDistance* **then**
29: *Y1 ← yc – Utilities.minAxis*
30: *Y2 ← yc + Utilities.minAxis*
31: **Else**
32: *Y1 ← yc – L*
33: *Y2 ← yc + L*
34: **end if**
35: *this.ds ← DynamicSelection(X2, X1, Y2, Y1)*
36: **Return**
37: **end if**
38: **if** *(y1 – y2) = 0* **then**
39: *L ← l*
40: **if** *L < Utilities.minDistance* **then**
41: *X1 ← xc – Utilities.minAxis*
42: *X2 ← xc + Utilities.minAxis*
43: *Y1 ← yc – Utilities.minAxis*
44: *Y2 ← yc + Utilities.minAxis*
45: **Else**
46: *X1 ← xc – L*
47: *X2 ← xc + L*
48: *Y1 ← yc – L*
49: *Y2 ← yc + L*
50: **end if**
51: *this.ds ← DynamicSelection(X2, X1, Y2, Y1)*
52: **Return**
53: **end if**
54: *s ← (x1 – x2)/(y1 – y2)*
55: $Ax \leftarrow \sqrt{\frac{s^2 l^2 + L^2}{1 + s^2}}$
56: $Ay \leftarrow \sqrt{\frac{s^2 L^2 + l^2}{1 + s^2}}$
57: **if** *Ax < Utilities.minAxis* **then**
58: *Ax ← Utilities.minAxis*

59: **end if**
60: **if** *Ay* < *Utilities.minAxis* **then**
61: *Ay* ← *Utilities.minAxis*
62: **end if**
63: *X1* ← *xc* – *Ax*
64: *X2* ← *xc* + *Ax*
65: *Y1* ← *yc* – *Ay*
66: *Y2* ← *yc* + *Ay*
67: *this.ds* ← *DynamicSelection(X2, X1, Y2, Y1)*
68: **Return**

Algorithm 2: Spatial Temporary Proximity part

Once the region has been determined, we proceed to filter the vertices and the edges of the graph created from the map provided (lines 14-16, Algorithm 1). The goal is to eliminate all the objects that are not inside the region. To do this, the `stream` and `filter` APIs, made available by the Java virtual machine, were used. Consequently, we perform a series of type transformations bringing the sets of nodes and edges of the main graph into a `Stream` object type on which its `filter` method is used on. Then we regenerate, again, the object together with the `collect` method of the `Stream` class and with the static `toSet()` method of the `Collector` class.

The result is two new set objects that are used to create a subgraph `candidateGraph` with the `AsSubgraph<V, E>` constructor supplied by the jGrapht [21] library. So, these two sets named `candidateEdge` and `candidateNode` contain the edges and the vertices of the roadmap contained in the region.

Then, the first check is on the number of edges (lines 17-21, Algorithm 1). Indeed, if there is only one edge (with two nodes) no more calculations are needed, and the candidate road is the edge. Otherwise, the second part of the algorithm, the IWC, starts.

It begins from `candidateGraph` and the three trajectory points `p1`, `p2` and `p3`. Then we get the vertices and edges sets and we generate their iterators through the Java API `java.util.Set.iterator()`. Going on, we initialize the variables `sumA`, `sumB`, and `sumD` that have the task of storing the progressive sum of the reciprocal of parameters $a$, $b$ and $d$ in order to prepare the $w_{qi}$ weight calculation.

The initialization goes on to declare and prepare other temporary variables necessary to calculate intermediate results. They are `maxWq`, that will contain the maximum weight calculated for candidate roads, `cRoad`, which will contain the reference to the candidate road determined by the IWC sub algorithm (lines 1-9, Algorithm 3), `num1`, `num2`, `denum1` and `denum2` that will contain the intermediate results of the following operations.

The first parameter that is determined is $b$ because it depends only on the trajectory points, and it will not change during the edges and vertices iteration (lines 10-11, Algorithm 3). So, we start defining an array of `NodoPercorso`, named `semiTraiettoria`, to include all the trajectory points and then we apply directly the (2-12).

It means that we implement two nested `for` cycles where in the inner one we calculate the sum $\frac{1}{y_i}\sum_{j=1}^{n}\left(\frac{x_j}{y_j}\right)^2$, $\frac{x_i}{y_i^2}\sum_{j=1}^{n}\frac{x_j}{y_j^2}$ and $\frac{1}{y_i^2}\sum_{j=1}^{n}\left(\frac{x_j}{y_j}\right)^2$; while in the outer one, we increase the $i$ index and we calculate the sum $\sum_{i=1}^{n}\left(\frac{x_i}{y_i^2}\right)^2$. At the end we determine the $b$ value putting together all in the sums: `b = (num1 - num2)/(denum1 - denum2)`.

After $b$ we proceed to calculate the two other parameters, but to do that we need to iterate on the edges (line 12, Algorithm 3). Using the iterator defined before, we get one edge at a time, and we retrieve its `source` and `target` nodes. Then, we get their cartesian coordinates, `psx`, `psy`, `ptx`, and `pty`, too to calculate the edge direction into the space (lines 13-22, Algorithm 3). This information is required to calculate the distance (parameter $d$) between the considered trajectory point `p3` and the straight line on which the edge lies.

Once we have obtained this information, we can apply the well-known equation (3-12) that determines the distance of a point from a straight line in the Cartesian plane (line 23, Algorithm 3).

$$d = \frac{|ax_p + by_p + c|}{\sqrt{a^2 + b^2}} \qquad (3\text{-}12)$$

Where $d$ is our parameter; $a$, $b$ and $c$ are the straight line's coefficients; $x_p$ and $y_p$ are cartesian coordinates of the trajectory point `p3`.

The last parameter to determine ($a$) is the angle between the edge direction, which represents the actual candidate road, and the trajectory direction. Using the `Math.atan2`[7] static method, supplied by the Java virtual machine, we can calculate the straight-line angular coefficient. After that, we can directly apply the (2-11): `a = Math.abs(angoloTraiettoria - angoloRetta)` (lines 25-27, Algorithm 3).

After the parameter calculation, we must determine the edge's weight in order to find the best candidate road. So, we start to define a `HashMap` made by a `String` and a `Double` with the aim to couple the edge itself and the radius (distance) between the target node of the considered edge and all the other vertices of the subgraph (line 24, Algorithm 3). We also initialize a support variable (`denum2`) to get the maximum radius $R_{MAX}$ related to the edge determined by the rules stated in section 2.3.

[7] `atan2` is a version of the $\tan^{-1}(\cdot)$ function that uses the information on the parameters sign to determine the angle between the ordinate semiaxis and the point $x$, $y$.

This step starts iterating on the vertexes set and for every node, a comparison is made with the current target node of the considered edge. If the node is equal to the target one (every node is identified by a unique string name) the distance is set to zero and the execution goes on. Otherwise, the node's Cartesian coordinates are retrieved and the usual distance equation between two points in a plan is used:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{3-13}$$

Where $x_1$, $x_2$, $y_1$ and $y_2$ are respectively the node's abscissa (`x`), the target node's one (`ptx`), the node's ordinate (`y`), and the target node's one (`pty`). The result is compared to the previous one stored in `denum2` and if the latter is greater or equal to the calculated one, no actions were made. Otherwise, it is substituted. Finally, the node's distance is stored in the double value related to the vertex's key of the `HashMap radii` (lines 29-42, Algorithm 3).

At the end of the iteration, we have got the `radii` structure filled out with all the distance between the subgraph nodes and the target one. Moreover, we have also got the maximum distance between the target node and the farthest one in the graph.

To use the rules in section 2.3 and calculate the weight of the edge, we need to calculate the distance $|OP|$ between the target node and the last point of the trajectory `p3`. Then we can go on and check if $|OP|$ (`denum1`) is greater or equal to $R_{MAX}$ or not. If it is the weight is calculated by **Rule 1**: $w_q = 1$ and $w_a = 1 - w_q = 0$. If not, we need to apply **Rule 2** or **Rule 3**.

To do this, we need to iterate into the dictionary, but the trick is to transform it in an `ArrayList` of a couple of `String`, `Double` (`Entry` class) passing through the `Set` representation of the `radii` by using `entrySet()` method supplied by the Java virtual machine `HashMap` class. This step allows us to sort the distances by value, in ascending order, and as soon as we find a distance in the ordered radii structure that is greater than $|OP|$ (`denum1`), we get the radius of the minimum circle that contains the vehicle (lines 48-67, Algorithm 3).

So, we can calculate $w_d = \frac{|OP|}{R}$ and $w_a = 1 - w_q$, in which $|OP|$ is `denum1` and $R$ is `entry.getValue()`. In this situation, we put also `found` Boolean variable to `true` in order to get out of the iteration and go on with the procedure. Indeed, the ascending order allows us to stop iteration as soon as we get the first distance greater than $|OP|$.

The last part of one single iteration on edges is the calculation of the sums of the parameters $a$, $b$, and $d$. These operations are related to the (2-14a), (2-15b), and (2-16c) equations which must calculate $\sum_{1=1}^{k} \frac{1}{a_i}$, $\sum_{i=1}^{k} \frac{1}{b_i}$, and $\sum_{i=1}^{k} \frac{1}{d_i}$ exploiting the `sumA`, `sumB`, and `sumD` variables (lines 69-86, Algorithm 3).

After having performed the previous steps for each edge of the subgraph, we will arrive at a point where we will have the parameters $a$, $b$, and $d$ relating to the couple

trajectory, candidate road, and the weights $w_d$, $w_a$ with the sum of the reciprocals of the parameters (`sumA`, `sumB`, and `sumD`). So, we will have everything we need to calculate the weight $w_{qi}$ of the candidate roads.

Then we iterate again on the edges of the subgraph calculating the weight $w_{qi}$ by directly applying equation (2-13) keeping track of the maximum weight and the edge connected to it each time. This will allow us to determine the best candidate for that stretch of trajectory (lines 89-96, Algorithm 3).

**Algorithm 3** Improved Weighted Circles

**Input:** AsSubGraph sg, NodoPercorso p1, NodoPercorso p2, NodoPercorso p3
**Output:** CandidateRoad cRoad

1: *edges* ← *sg.edgeSet()*
2: *listEdges* ← *edges.iterator()*
3: *nodes* ← *sg.vertexSet()*
4: *listNodes* ← *nodes.iterator()*
5: *sumA* ← *0*
6: *sumB* ← *0*
7: *sumD* ← *0*
8: *maxWq* ← *0*
9: *cRoad* ← **nil**
10: *n* ← *# of NodoPercorso points*
11: $b \leftarrow \frac{\left(\sum_{i=1}^{n}\frac{1}{p_i.getY()}\sum_{j=1}^{n}\left(\frac{p_j.getX()}{p_j.getY()}\right)^2 - \sum_{i=1}^{n}\frac{p_i.getX()}{p_i.getY()}\sum_{j=1}^{n}\frac{p_j.getX()}{p_j.getY()^2}\right)}{\left(\sum_{i=1}^{n}\frac{1}{p_i.getX()^2}\sum_{j=1}^{n}\left(\frac{p_j.getX()}{p_j.getY()}\right)^2 - \sum_{i=1}^{n}\left(\frac{p_i.getX()}{p_i.getY()^2}\right)^2\right)}$
12: **while** *(listEdges.hasNext())* **do**
13: *edge* ← *listEdges.next()*
14: *source* ← *edge.getSource()*
15: *target* ← *edge.getTarget()*
16: *psx* ← *source.getX()*
17: *psy* ← *source.getY()*
18: *ptx* ← *target.getX()*
19: *pty* ← *target.getY()*
20: *aRetta* ← *pty – psy*
21: *bRetta* ← *psx – ptx*
22: *cRetta* ← $psy(ptx - psx) - psx(pty - psy)$
23: $d \leftarrow \frac{\sqrt{aRetta^2 + bRetta^2}}{|aRetta * p3.getX() + bRetta * p3.getY() + cRetta|}$
24: *radii* ← **new** *HashMap(Value, Key)*
25: *angoloRetta* ← $atan2(bRetta, -aRetta)$
26: *angoloTr* ← $atan2\big((p3.getY() - p2.getY()), (p3.getX() - p2.getX())\big)$
27: *a* ← $|angoloRetta - angoloTr|$
28: *Rmax* ← *0*

```
29:         while (listNodes.hasNext()) do
30:             nodo ← listNodes.next()
31:             if (target = nodo) then
32:                 radii.put(nodo, 0)
33:                 continue
34:             end if
35:             x ← nodo.getX()
36:             y ← nodo.getY()
37:             radius ← √((x − ptx)² + (y − pty)²)
38:             if radius > Rmax then
39:                 Rmax ← radius
40:             end if
41:             radii.put(nodo, radius)
42:         end while
43:         |OP| ← √((ptx − p3.getX())² + (pty − p3.getY())²)
44:         if |OP| > Rmax then
45:             edge.setEdgeWd(1)
46:             edge.setEdgeWa(0)
47:         else
48:             orderedRadii ← new ArrayList(radii.entrySet())
49:             orderedRadii.sort()
50:             scrollOrderedRadii ← orderedRadii.iterator()
51:             found ← false
52:             while (scrollOrderedRadii.hasNext() and not found) do
53:                 entry ← scrollOrderedRadii.next()
54:                 if entry.getValue() = 0 then
55:                     entry.setEdgeWd(0)
56:                     entry.setEdgeWa(1)
57:                 else
58:                     if |OP| > entry.getValue() then
59:                         edge.setEdgeWd(1)
60:                         edge.setEdgeWa(0)
61:                     else
62:                         edge.setEdgeWd(|OP|/entry.getValue())
63:                         edge.setEdgeWa(1 – edge.getEdgeWd())
64:                         found ← true
65:                     end if
66:                 end if
67:             end while
68:         end if
69:         if a ≠ 0 then
70:             edge.setReciprocalA(1/a)
```

```
71:                 sumA ← sumA + 1/a
72:             else
73:                 edge.setReciprocalA(0)
74:             end if
75:             if b ≠ 0 then
76:                 edge.setReciprocalB(1/b)
77:                 sumB ← sumB + 1/b
78:             else
79:                 edge.setReciprocalB(0)
80:             end if
81:             if d ≠ 0 then
82:                 edge.setReciprocalD(1/d)
83:                 sumD ← sumD + 1/d
84:             else
85:                 edge.setReciprocalD(0)
86:             end if
87:     end while
88:     listEdges ← edges.iterator()
89:     while (listEdges.hasNext()) do
90:             edge ← listEdges.next()
91:             wqi ← edge.getEdgeWd * edge.getReciprocalD / sumD + edge.getEdgeWa *
                ( edge.getReciprocalA / sumA + edge.getReciporcalB / sumB )
92:             if wqi > maxWq then
93:                 maxWq ← wqi
94:                 cRoad ← new CandidateRoad(edge, maxWq)
95:             end if
96:     end while
97      return cRoad
```

Algorithm 3: Improved Weighted Circles part

## 3.3. Variants

In this section, we will see some variations of the above algorithm and compare them to the results of this advanced variant. The effective illustration of the data will be done in chapter 4, while here we show how these variations are implemented. These ones will involve the IWC since it is this part that determines the best candidate road.

We thought about implementing these other versions starting from the simplest to the most complex one. We start with an IWC version in which we do not consider the parameter $b$, which is the most important innovation included in this part of the algorithm and do use fixed weight for the parameter distance $d$ and the parameter direction $a$. Subsequently, we will introduce the possibility of dynamically calculating

the incidence of the parameters $a$ and $d$ in ways that are initially calculated based on the distance between the last point of the trajectory $p_3$ and all the rest of the nodes of the graph identified by the STP part. Finally, this dynamic calculation will be based on the distance of the target node of the candidate road from all the nodes of the graph always provided by the STP process.

### 3.3.1. IWC Constant Weight

As stated, we start with the simplest version of the IWC routine. Here we do not make any special calculation and we determine the weights using fixed values stored in the object `Utilities`. They are configured at the beginning of the execution, and they are related as usual with the relation $w_a = 1 - w_d$. In this case, we use $w_d = w_a = 0.5$, but different values can be set up to check the differences.

This method gets as input the subgraph calculated in the STP part (`candidateGraph`) and the three trajectory points `p1`, `p2`, and `p3`. It starts by initializing the candidate road `cRoad` and iterating over the sets of edges and vertices as in the classic version (lines 1-14, Algorithm 4). So, in this kind of method, the information on the trajectory direction and the sum of the reciprocal of the parameter $d$, $a$ and $b$ are not considered.

Here, we go directly to calculate the radii of the circles or rather the distances between the considered trajectory point (`p3`) and the nodes of the subgraph. Then we transform and order the dictionary `radii` as in the classic version using the `ArrayList`[8] `<Entry<Nodo, Double>> (Collection<? extends Entry<Nodo, Double>> c)` constructor for iterating on that (lines 15-16, Algorithm 4).

Here, it is important to point out that in this version the $|OP|$ distance, on which the candidate road is determined is the *shortest* between a graph node and the trajectory point `p3`. So, the nearest node to `p3` is the intersection node $O$ and we calculate iterating on the `orderedRadiiList` and getting the element with the minimum distance (lines 17-26, Algorithm 4).

At this point, we determined the intersection node $O$ and the relative distance from the trajectory point `p3`, $|OP|$. Now we proceed to calculate the minimum radius that contains $|OP|$ by iterating again on the node set (`Set<Nodo> nodes`); at the same time, we calculate also $R_{MAX}$ in order to minimize the complexity of execution (lines 30-41, Algorithm 4).

The next step is to iterate on the edge set (`Set<BidirectionalEdge> edges`) getting the target nodes and calculating the parameters $d$ and $a$ with equations (3-12) and (2-11) (lines 43-56, Algorithm 4). Parameter $b$ is part of an evolution of the IWC routine so it will not be considered. During the iteration, we also calculate the edge's weight using the fixed values stored in the `Utilities` class related to the distance and the

[8] ArrayList is a resizable-array implementation of the Java List interface. For more information see: https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/util/ArrayList.html

angle between the road and the actual candidate road (`Utilities.Wd` and `Utilities.Wa`) (line 57, Algorithm 4). Then, we repeat the iteration on the edge set because we need to get the maximum of the $w_{qi}$ (`maxWq`) (lines 59-66, Algorithm 4).

**Algorithm 4** IWC Constant Weights

**Input:** AsSubGraph sg, NodoPercorso p1, NodoPercorso p2, NodoPercorso p3
**Output:** CandidateRoad cRoad

```
1:  edges ← sg.edgeSet()
2:  listEdges ← edges.iterator()
3:  nodes ← sg.vertexSet()
4:  listNodes ← nodes.iterator()
5:  cRoad ← nil
6:  |OP| ← 0
7:  radii ← new HashMap(Value, Key)
8:  while (listNodes.hasNext()) do
9:      nodo ← listNodes.next()
10:     x ← nodo.getX()
11:     y ← nodo.getY()
12:     radius ← √((x − p3.getX())² + (y − p3.getY())²)
13:     radii.put(nodo, radius)
14: end while
15: orderedRadii ← new ArrayList(radii.entrySet())
16: scrollOrderedRadii ← orderedRadii.iterator()
17: head ← scrollOrderedRadii.next()
18: |OP| ← head.getValue()
19: oNode ← head.getKey()
20: while (scrollOrderedRadii.hasNext()) do
21:     entry ← scrollOrderedRadii.next()
22:     if entry.getValue() < |OP| then
23:         |OP| ← entry.getValue()
24:         oNode ← entry.getKey()
25:     end if
26: end while
27: R ← 0
28: listNodes ← nodes.iterator()
29: Rmax ← 0
30: while (listNodes.hasNext()) do
31:     nodo ← listNodes.next()
32:     x ← nodo.getX()
33:     y ← nodo.getY()
34:     denum1 ← √((x − oNode.getX())² + (y − oNode.getY())²)
35:     if R > Rmax then
```

```
36:                 Rmax ← R
37:             end if
38:             if denum1 > |OP| and (denum1 < R or R = 0) then
39:                 R ← denum1
40:             end if
41:         end while
42:         while (listEdges.hasNext()) do
43:             edge ← listEdges.next()
44:             source ← edge.getSource()
45:             target ← edge.getTarget()
46:             psx ← source.getX()
47:             psy ← source.getY()
48:             ptx ← target.getX()
49:             pty ← target.getY()
50:             aRetta ← pty – psy
51:             bRetta ← psx – ptx
52:             cRetta ← psy(ptx − psx) − psx(pty − psy)
53:             d ← √(aRetta² + bRetta²) / |aRetta*p3.getX() + bRetta*p3.getY() + cRetta|
54:             angoloRetta ← atan2(bRetta, −aRetta)
55:             angoloTr ← atan2((p3.getY() − p2.getY()), (p3.getX() − p2.getX()))
56:             a ← |angoloRetta − angoloTr|
57:             edge.setEdgeWq(Utililies.Wd*d+ Utililies.Wa*a)
58:         end while
59:         listEdges ← edges.iterator()
60:         candidateEdge ← listEdges.next()
61:         while (listEdges.hasNext()) do
62:             edge ← listEdges.next()
63:             if edge.getWq() > candidateEdge.getWq then
64:                 candidateEdge ← edge
65:             end if
66:         end while
67:         cRoad ← new CandidateRoad(candidateEdge, candidateEdge.getWq())
68:         return cRoad
```

Algorithm 4: IWC Constant Weights

### 3.3.2. Generalized IWC

In this method, we add some features to the IWC part of the algorithm. In the previous section (3.3.1), we saw the simplest way to determine the candidate road keeping the nearest node to the trajectory point as intersection node $0$, and keeping fixed values for weight $w_d$ and $w_a$ to balance the relevance of the two main parameters: the distance between the trajectory point and the candidate road, and the angular

difference between the two directions taken by the vehicle and the road. Here, instead, we will see how to calculate the weights without keeping them from the class `Utilities` but using the distance $|OP|$.

So, as usual, we start iterating on the node set to determine the distances between a node and the trajectory point $p3$. We keep a trace of them storing the values in the `radii HashMap` class (lines 1-14, Algorithm 4). Then, we calculate the distance $|OP|$[9] and the node $O$ iterating on a support structure named `ordredRadii` which is an `ArrayList` version of the `radii` dictionary (lines 15-26, Algorithm 4).

At this point, we have all the information to determine the minimum radius $R$ that contains $|OP|$. To do that we iterate again on the set of nodes, calculating the distance between the current node and the intersection node $O$. As soon as we get a distance greater than the $|OP|$, that is also the first distance kept, or we get a smaller value than the actual $R$ (with $R \neq 0$), we assign it to $R$ because the before value was not the smallest radius containing $|OP|$. At the same time, we calculate $Rmax$ (lines 27-41, Algorithm 4).

The next step is to calculate the weights $w_a$ and $w_d$ using the information $|OP|$ and $Rmax$ without iterating the edges set because in this version we consider fixed intersection point $O$. So, we directly apply the three rules seen in section 2.3 and if $|OP| > Rmax$ we are very far away. This leads us to set $w_d = 1$ and $w_a = 0$, otherwise, we calculate the weights as $w_d = |OP|/R$ and $w_a = 1 - w_d$ (lines 42-50, Algorithm 5).

After that, we go on to calculate the parameters $a$, $b$ , and $d$ by iterating the set of edges as in the original version (section 3.2). We start from the parameter $b$ because it is related only to the trajectory. We calculate it with the equation (2-12) (lines 51-52, Algorithm 5). Then, we iterate on the edge set calculating $d$ and $a$ . The first one is the distance between the edge and the trajectory point $p3$. The second, instead, is calculated as the difference between the direction of the trajectory and the current edge (equation (2-11)) (lines 43-56, Algorithm 4).

Now we are ready to determine the value of the edge's weight $w_{qi}$ using a simplified formula with respect to the original one because we do not consider the reciprocal of the parameters $a$, $b$ , and $d$ shown in equation (2-13) (line 68, Algorithm 5). At the end, we iterate again on the edge set to find the maximum weight and the related candidate road. The latter are returned to the caller (lines 59-68, Algorithm 4).

**Algorithm 5** Generalized IWC

**Input:** AsSubGraph sg, NodoPercorso p1, NodoPercorso p2, NodoPercorso p3
**Output:** CandidateRoad cRoad

1: ...
... *same instructions in Algorithm 4 lines 1 – 41*
41: ...

[9] $|OP|$ is the distance between the intersection node and the trajectory point.

42: $w_d \leftarrow 0$
43: $w_a \leftarrow 0$
44: **if** $|OP| > Rmax$ **then**
45: $w_d \leftarrow 1$
46: $w_a \leftarrow 0$
47: **else**
48: $w_d \leftarrow |OP|/R$
49: $w_a \leftarrow 1 - w_d$
50: **end if**
51: $n \leftarrow$ *# of NodoPercorso points*
52: $b \leftarrow \dfrac{\left(\sum_{i=1}^{n}\frac{1}{p_i.getY()}\sum_{j=1}^{n}\left(\frac{p_j.getX()}{p_j.getY()}\right)^2 - \sum_{i=1}^{n}\frac{p_i.getX()}{p_i.getY()}\sum_{j=1}^{n}\frac{p_j.getX()}{p_j.getY()^2}\right)}{\left(\sum_{i=1}^{n}\frac{1}{p_i.getX()^2}\sum_{j=1}^{n}\left(\frac{p_j.getX()}{p_j.getY()}\right)^2 - \sum_{i=1}^{n}\left(\frac{p_i.getX()}{p_i.getY()^2}\right)^2\right)}$
53: **while** *(listEdges.hasNext())* **do**
54: ...
... *same instructiosn in Algorithm 4 lines 43 – 56*
67: ...
68: *edge.setEdgeWq*($w_d * d + w_a * (a + b)$)
69: **end while**
70: ...
... *same instructions in Algorithm 4 lines 59 – 68*
79: ...

Algorithm 5: Generalized IWC

### 3.3.3. Advanced IWC

In this last version, we go on with the process of adding complexity to the IWC part of the classic version of the Integrating Spatio-temporal Proximity and Improved Weighted Circle algorithm. Compared to the version seen previously, we add the calculation of the reciprocals of the parameters $a$, $b$ , and $d$ to apply the equation (2-13) and obtain a better precision in the calculation of the weight $w_{qi}$. However, the intersection node and the distance $|OP|$ will always refer to the node of the subgraph closest to the trajectory points $p3$.

We start initializing the same variables as before (lines 1-7, Algorithm 4) plus the `sumA`, `sumB`, and `sumD` ones, because we need them to calculate the weight $w_{qi}$. We also define a variable `maxWq` to store the weight of the candidate road (lines 8-11, Algorithm 6). Then we always determine the distance between the trajectory point $p3$ and the nodes of the subgraph `sg` iterating on the node set. We store these values in the dictionary `Radii` and then we proceed to look for the node of the graph that is the nearest to the trajectory point `p3`. This will be our intersection point $O$ (lines 8-41, Algorithm 4).

To do this, we iterate on the support structure `orderedRadiiList` looking for this node. We immediately extract the first node from the list with its distance from the trajectory point, and then we compare this distance with the next node extracted keeping which is the smallest. At the end of this second loop, we will have both the intersection node $O$ and its distance from the point $p3$ ($|OP|$). So, we can calculate the minimum distance (circle's radius) that contains $|OP|$ and, at the same time, also the maximum distance of the nodes from $O$.

We iterate again on the set of nodes calculating the distance between the actual node and the intersection one $O$. Then we compare the latter $Rmax$ determined up to that moment with it and, if it is greater, it becomes the new $Rmax$, otherwise, we go on. After that, we compare this distance with $R$, and if it is greater than $|OP|$ and at the same time is smaller than $R$ or $R$ is equal to zero then it will be the new $R$ to consider. Now, we have all the information to calculate weights $w_a$ and $w_q$ according to the usual rules of section2.3.

The next part of the algorithm is dedicated to the parameters $a$, $b$ , and $d$ calculation. We begin with the parameter $b$ (lines 42-52, Algorithm 5) calculating first the trajectory's direction and then using it for $b$ calculation. To calculate $d$ and $a$, we iterate on the set of edges, and for every edge, we calculate the distance of point $p3$ from the actual edge, as $d$ , using equation (3-12). Then we calculate the parameter $a$ as the difference between the trajectory's direction and the edge's direction (lines 43-56, Algorithm 4).

Once we have got all the parameter values, we need to take a step more to determine the reciprocal of them and store the results into support variables `sumA`, `sumB`, and `sumD` to implement the summation terms of the equations (2-14a), (2-15b) and (2-16c). So, we check if the value of the parameter is zero, and in that case, we sum zero to the support variables. Otherwise, we calculate the reciprocal value of the parameter and add it to the corresponding support variable. At the same time, this reciprocal value is stored with the edge in order to use it during the $\bar{a}$, $\bar{b}$ and $\bar{d}$ calculation as stated in (2-14a), (2-15b), and (2-16c) (lines 72-89. Algorithm 6).

At this point, we must iterate again the edge set to calculate the weight $w_{qi}$ keeping note of the maximum value calculated with the corresponding candidate road. As stated, this computation is done by applying the (2-13) directly; so, we need all the values determined so far (lines 92-98, Algorithm 6). At the end of the iteration, we have the greatest weight `maxWq` and its corresponding candidate road `cRoad`. So, we return them to the caller procedure (line 99, Algorithm 6).

**Algorithm 5** Advanced IWC

**Input:** AsSubGraph sg, NodoPercorso p1, NodoPercorso p2, NodoPercorso p3
**Output:** CandidateRoad cRoad

1: ...
... *same instructions in Algorithm 4 lines 1 – 7*

7: ...
8: $sumA \leftarrow 0$
9: $sumB \leftarrow 0$
10: $sumD \leftarrow 0$
11: $maxWq \leftarrow 0$
12: ...
... *same instructions in Algorithm 4 lines 8 – 41*
45: ...
46: ...
... *same instructions in Algorithm 5 lines 42 – 52*
56: ...
57: **while** *(listEdges.hasNext())* **do**
58: ...
... *same instructions in Algorithm 4 lines 43 – 56*
71: ...
72: **if** $a \neq 0$ **then**
73: *edge.setReciprocalA*$(1/a)$
74: $sumA \leftarrow sumA + \frac{1}{a}$
75: **else**
76: *edge.setReciprocalA(0)*
77: **end if**
78: **if** $b \neq 0$ **then**
79: *edge.setReciprocalB*$(1/b)$
80: $sumB \leftarrow sumB + \frac{1}{b}$
81: **else**
82: *edge.setReciprocalB(0)*
83: **end if**
84: **if** $d \neq 0$ **then**
85: *edge.setReciprocalD*$(1/d)$
86: $sumD \leftarrow sumD + \frac{1}{d}$
87: **else**
88: *edge.setReciprocalD(0)*
89: **end if**
90: **end while**
91: *listEdges* ← *edges.iterator()*
92: **while** *(listEdges.hasNext())* **do**
93: *edge* ← *listEdges.next()*
94: $w_{qi} \leftarrow w_d * \left(\frac{edege.getReciprocalD()}{sumD}\right) + w_a \left(\frac{edge.getReciprcalA()}{sumA} + \frac{edge.getReciprocalB()}{sumB}\right)$
95: **if** $w_{qi} > maxWq$ **then**

```
96:                 cRoad ← new CandidateRoad(edge, maxWq)
97:             end if
98:     end while
99:     return cRoad
```

Algorithm 6: Advanced IWC

# 4 Validation

In this chapter, we will see how the position data are collected and which tools were used. After that, we explain how they were managed and prepared for the algorithm. Finally, we present the results obtained, and we make some considerations about them. In the next section, we will show our conclusions based on the data obtained from this one.

## 4.1. Overview

The results obtained from the various versions of the algorithm were analyzed and validated manually. The process consists of preparing a text file with a TXT extension, that will be passed to a site, named *Creating a GPX, PQ or KML file from the coordinates* [22], which will prepare the actual XML file underlying the GPX 1.1 specification [23] [24] [25].

To have an effective comparison, each point of the text file was provided to the site and a single GPX file was obtained for each calculated coordinate. Since this procedure was also repeated for each version implemented, it was possible to carry out a point-by-point comparison. That is, the coordinates of the first candidate road of one version were compared directly with the first candidate road of the other versions.

To facilitate these comparison operations, a counter valid for all versions of the algorithm was used so it was possible to trace the candidate determined by each version on the basis of the same piece of trajectory considered.

The actual comparison was done visually thanks to the support of an additional website [26], which allowed each file to be uploaded and displayed on a piece of road map containing the points to be checked. The site in question independently managed how large this portion of the map should be to contain all the coordinates entered.

To record the results, a table was compiled where, for each calculated candidate, a score was given based on the distance from the real road. If the coordinates of the candidates were included or coincident with those of the actual road, then the version of the algorithm that generated these points received the maximum score (2). If, however, the candidate had the same direction as the real road without coinciding perfectly, the algorithm was assigned an average score (1). Finally, if the result appeared completely wrong compared to the actual road, the relevant version of the algorithm was assigned the minimum score (-1).

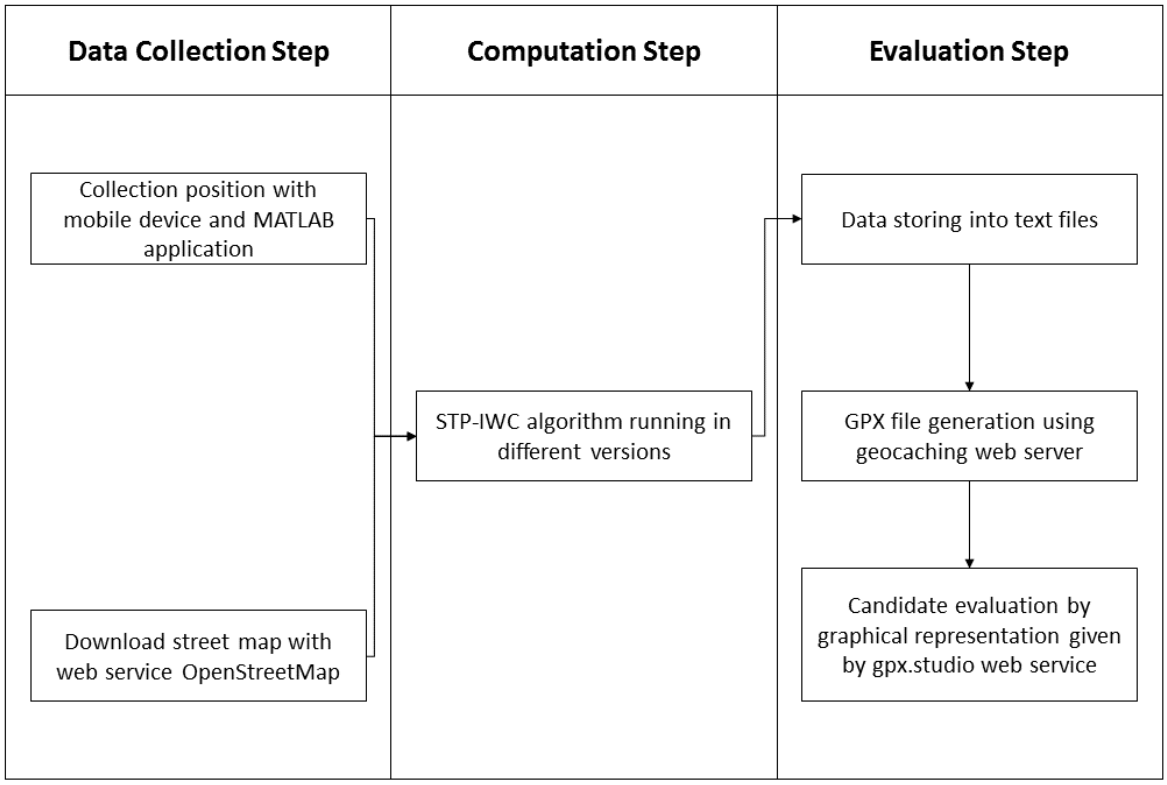



Figure 4-1 Testing and evaluation process.

In the next chapter, we will make some consideration about the results obtained trying to understand the reasons which could explain the value.

## 4.2. Position data collection

To collect GPS data, we used the tools supplied by MATLAB, a data manipulation software that allows data analysis, systems simulation, and so on. The instrument is composed of a desktop application and a mobile one. The latter exploits the GPS antenna installed on mobile devices to store the position with a preconfigured frequency.

To get the positions is necessary to use an object called `mobiledev` that supplies the abstraction to access to the mobile sensors. A lot of these devices are present in this software tool (like acceleration sensors, angular velocity sensors, and so on), but we will focus on position sensors that measure *latitude*, *longitude*, *altitude*, *speed*, *course,* and *horizontal accuracy*.

The first ones are quite clear, but the latter two need a little explanation. The measure *Course* is the direction with respect to the North, and it is measured in degrees[10]. While

[10] As stated in MATLAB documentation on the function `poslog` that returns the logged position data from mobile device sensor in the form of a `timetable` [41]

the *horizontal accuracy* expresses the radial horizontal accuracy in a circle around the GPS point[11]. These parameters will not be used to determine the vehicle's trajectory, but they are part of the data collection by the GPS antenna to locate the mobile in the environment.

To start getting position we first need to create the object `mobiledev` on the mobile device with the command `m = mobiledev;` and then we must enable the position sensors by putting the Boolean variable `PositionSensorsEnable` to one (`m.PositionSensorEnbled = 1;`). Finally, to start to record the data we use the command `m.Logging = 1;` and the position will be stored.

With the command `m.Logging = 0;`, the process of position storing ends up. All the data collected are inside the object `mobiledev` and in a file `*.mat` if the mobile application has been set up properly. Now, it is possible to proceed in two ways: the first one is to use the function `poslog() [27]`, that given an object `mobiledev`, it returns a vector with all the measures stated before (latitude, longitude, altitude, speed, course, and horizontal accuracy). The second way is to work directly with the file `*.mat` created.

All the information stored are collected into a special MATLAB structure called `timetable` [28], in which every line has a timestamp associated with it and its columns are the measurements made by the sensors at that time. To be more precise, if you use `*.mat` files as a data collection tool, then just open the files in MATLAB to have the structure ready. Otherwise, if you pass through the `poslog()` function, you need to use the constructor function of the `timetable()` passing to it the vector of measured values as input.

The major difference between the two ways of proceeding is the precision with which the latitude, longitude, and time references are stored. Using `*.mat` files we will obtain greater precision on time but neglect that on latitude and longitude. Using the `poslog()` function, instead, we will get the exact opposite.

Once we have got the `timetable`, we need to extract these data to pass them to the algorithm. The medium chosen and used for this task is a CSV file. The reason why we opted for this tool is the ease of manipulation by many programs (Excel, MATLAB, etc.) and the extreme simplicity with which it is possible to structure the data. In our case, for example, we decided to extract the data so that each column of interest (the first four plus the timestamp column) corresponds to a column in the CSV file.

MATLAB supplies a lot of functions that work with the object `timetable`. What we will focus on is the `writetimetable` [29] procedure that is used to produce the CSV file from that. Indeed, given a `timetable` this function returns, by default, a text file. Each column of the `timetable` becomes a column of the file, and the first row is caught

[11] As stated in MATLAB documentation on the function `poslog` that returns the logged position data from mobile device sensor in the form of a `timetable` [41]

as columns' names. Consequently, every line of the file is created from every row of the table.

This function receives the `timetable` to export and creates a text file name as `<timetablename>.txt` (by default). Adding a parameter `filename`, it is possible to produce a file with a different name and format (for example, the format CSV). This parameter can specify the path in which the file must be created. The `writetimetable` function can also receive additional options, specified by name/value couples, to get determined characteristics in the result file.

As an example, to get a CSV file with Italian locale timestamps and the semicolon as a delimiter, we use the following command in MATLAB: `writetimetable (Position,'Matlab\MobileSensorData\saronnoLainate1.csv','Delimiter', 'semi','DateLocale','it_IT');`, in which `Position` was the name of the `timetable` created by the mobile application during the position tracking. The file as is it is not ready to be used by the program. We need to delete all the useless data removing the *altitude* column.

The other main information that we need to execute the algorithm and its versions is the map on which the program calculates the position on it. We have already said that we would rely on a tool called OpenStreetMap [20]. This site allows you to download the maps you are looking for in a format called `*.osm`. These are XML files that reference the OSM XSD template [30]. A small transformation is also performed for these files which consists purely of a change in formatting and file format. The file is brought into pure XML and formatted appropriately with the `notepad++` [31] program and its XML tool extension.

## 4.3. Results and analysis

Once the tools were set up, we started collecting data by driving a car with an average speed of 30km/h. Initially, we traveled along the streets of the same municipality with few intersections and curves. We gradually increased the length of the journeys until we crossed neighboring cities.

The number of samples provided by the MATLAB tool increased considerably and the OpenStreetMap [32] site used to retrieve the street maps with which to examine the accuracy of the algorithm was not able to provide them in sufficient detail. It was therefore decided to proceed by batches of measured points and manually recovering the map from the site.

The results have been obtained giving in input to the algorithm the collected positions stored into a CSV file and the map retrieved by hand from OpenStreetMap using the first position as the top right corner of the map and the last position as the bottom left corner. Indeed, in our tests, the map is ideally rectangular and contains a portion of OpenStreetMap's road map.

All the calculations were made on an HP Envy 15 notebook PC, equipped with an Intel(R) Core (TM) i5-5200U CPU @ 2.20GHz processor composed of 2 cores and with the support to the Hyperthreading technology. It has got 16GB of RAM and 1TB Hitachi SATA-150 hard disk.

Once the execution ends, the output is composed of various text files containing the calculated candidate roads. The most important files are stored in a folder named `GPX`. It contains the files with the calculated candidate road at every iteration and the related trajectory point considered. Every file is named with a convention that identifies the data execution and the time.

This convention is *yyyy_MM_dd_hh_mm_ss_XX_candidates_number_xx.txt*. Moreover, with the part (*XX*) and (*xx*); it is indicated which version of the algorithm calculates the candidate and its number.

If the abbreviation *XX* is `SO`, it indicates that the version of the algorithm that produced the result is the constant weight. If it is `GV`, it means that the version is the generalized IWC. If it is `AV`, it means that the version is the advanced IWC and if it is `iwc`, it means that the STP-IWC version calculated the candidate road. The abbreviation `xx` instead, is a counter used to identify which calculated candidate roads to compare.

When the process ends, we get the produced text files and copy their content by hand into a website [22] that puts the data into the GPX format [24]. These data are composed of the source and target nodes of the calculated candidate road and the trajectory point used to determine it. So, when it is visualized on the gpx.studio [26] site, the last point is the trajectory point; the previous ones, instead, are the nodes of the candidate.

In the next step, we proceed to compare the two node points with the road on the map and verify if the candidate is right or not. We measured three levels of precision, and we recorded the results on an Excel spreadsheet. We assigned **-1** if the considered trajectory point is out from the ideal line that joins the nodes (direction) and the part of space enclosed into them. We assigned **1** if the considered trajectory point is out from the part of space enclosed by the two nodes but in the same direction. We assigned **2** if the trajectory point is in the same direction as the candidate and it is also contained in the space enclosed between the candidate nodes.

To be precise, the space enclosed between the nodes of the candidate is that space contained between the parallel lines passing perpendicular to it. Having brought the space considered to two dimensions and having delegated the measurement precision to the GPS position detector, we believe that this is sufficient to give good reliability to the data scoring system used.

In every trial, we consider three trajectory points, but only the last one is used to test the goodness of the candidate. This implies that every file is composed of three points that we call one, two, and three. Points one and two represent the source and the target

node respectively of the candidate road. While the last one, (point three) is the trajectory point. Consequently, we identify these points on the map with numbers 1 (source candidate node), 2 (target candidate node) and 3 (trajectory point ).

Below we report an analysis of all the measurements carried out. Part of them were analyzed point by point in the various versions and for each one a percentage of success or failure was calculated. Furthermore, as mentioned, the success of the calculation has been divided into two degrees of precision in which a score is assigned if the direction is guessed; and another, higher, if in addition to the direction the trajectory point is contained in the neighborhood of the calculated candidate.

We tested four different scenarios that we divided into four journeys called Pagliera Street (Pag), Pagliera Street and Marche Street (PMarc), Saronno City and Lainate City 1 (SL1) and Saronno City, and Lainate City 2 (SL2). Each scenario increases in difficulty. The first one, the simplest, involves standing start and following a straight path. The second scenario, however, considers the vehicle to be already moving and the turn to be a side street.

Finally, the third scenario involves the crossing of two municipalities considering the standing start of the vehicle, the crossing of a part of city streets, and the passage on fast roads to transit from one city to the other. This last scenario was divided into two parts because the maximum points obtainable from the OpenStreetMap service could not cover the entire trip.

The first part (SL1) includes standing start, travel on city streets, and fast roads. The second part (SL2), instead, involves the passage to a different city passing first through high-traffic main roads and then through urban roads typical of the city's industrial areas. A good mix of fast curves and large intersections compose this type of road.

Since the evaluation was made by hand, not all the positions have been checked, in Table 4-1 the number of the measured and checked nodes for every journey is shown. However, the sample rate of the mobile tool was set on one position point every second and each one analyzed is no more than five seconds away from the previous one.

Table 4-1: List of analyzed positions.

| Journey | Collected Positions | Verified Positions |
|---|---|---|
| Pag | 41 | 19 |
| PMarc | 63 | 14 |
| SL1 | 162 | 46 |
| SL2 | 89 | 19 |

Well, the first thing to say is that, unfortunately, the road maps were not so exact. Indeed, as a road, OpenStreetMap also considers streets suitable for cycling and pedestrian roads. Very often, these streets were parallel to the actual streets, disrupting the subgraph creation process and therefore in the filtering of nodes and edges to limit the part of interest.

In any case, if there is only one candidate, every version of the algorithm gets the right outcome. In the scenario with multiple candidates, however, only the versions with an advanced way to calculate weights $w_a$ and $w_d$ find the candidate closer to the point of the trajectory. The others, those without it, get the right direction at most.

This situation is clearly illustrated in Table 4-2, which shows the percentages of candidates that fall within the vicinity of the actual path taken by the vehicle (candidates include position) and those that follow the direction indicated by the map (candidates include direction). As can be seen, none of the simpler algorithms successfully identified a good candidate out of the nineteen samples analyzed. Indeed, during the urban journey, the proximity of many cross streets and parallel roads posed significant challenges for these versions. However, the proposed version proved to be much more robust, successfully managing one-third of the analyzed samples.

Moreover, it was noticed that after a series of measurements with only one possible candidate (only one road that can be traveled without a shadow of a doubt), as soon as an intersection is encountered, the algorithm's reaction is disordered, as if a new transient began. Indeed, parallel, or neighboring roads that enter the search space become a nuisance, leading to wrong results. Also, in this situation, the version with adaptive spokes is more robust and corrects the error immediately (Figure 4-2, Figure 4-3, and Figure 4-4). While, the remaining ones persevere much more in the error, continuing to identify the disturbing path as a candidate. Once the vehicle has moved far enough away, these versions resume providing the correct result.

Table 4-3 illustrates the situation described above. It refers to the PMarc route, which is longer and more complex than the previous Par route because it covers a longer urban section with no nearby parallel roads. As can be seen, the percentage of correct candidates starts to improve even for the less advanced versions of the algorithm (7.14%) because this route is the only passable one in the considered part of the road map. However, as soon as the surrounding environment encounters cross streets, their accuracy drops significantly. The STP-IWC version, on the other hand, continues to be the most efficient and accurate. Indeed, thanks to this segment of the only passable road and its robustness against cross streets acting as disturbances, its percentage of candidates calculated that at least follow the same direction as the considered road reaches approximately 50%.

At the beginning of the journey, all versions of the algorithm have a high level of imprecision and make many errors. As said, not even the measured positions seem very precise due to GPS errors, and the start of the trip is very hard proof. Usually,

indeed, we start from a standstill to merge into city traffic, but this maneuver rarely lasts many seconds, but for them, we remain stationary.

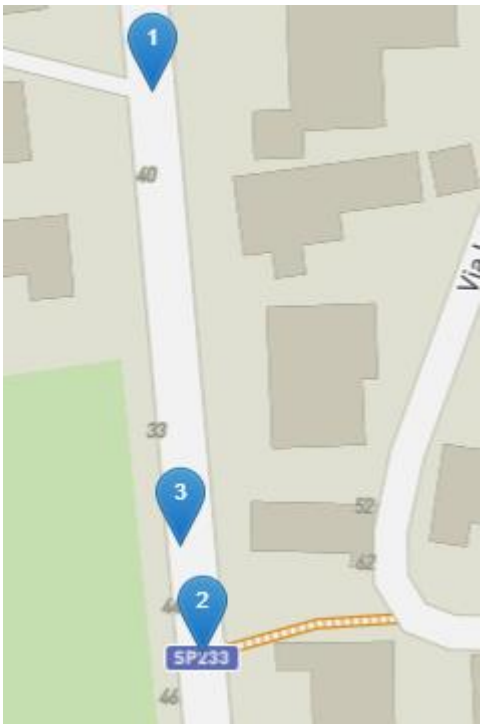



Figure 4-2: path pt 153, journey SL1, STP-IWC


Figure 4-3: path pt 158, journey SL1, STP-IWC

For example, in the first four measurements of the SL1 trip, the vehicle is measured stationary even though it actually crossed a stop sign in a short time to merge into traffic (Figure 4-5, Figure 4-6, Figure 4-7 and Figure 4-8), so we can say that a certain transient must be observed before the measurements stabilize and the results became more precise.

Furthermore, always at the beginning, the sampling frequency is probably too frequent compared to the initial movement of the car causing the calculation of the same candidate in this period. Another factor that has an impact is the low number of decimals for latitude and longitude in the tests in which, using MATLAB tools, more precision was obtained on time [12] than on position. This affects the candidate's calculation because the position takes longer to change (changes beyond the fifth decimal place are not detected).

The roundabouts are another very hard test. The most advanced solutions were able to determine the direction, but the trajectory point is still far outside the candidate's neighborhood. When traveling roundabouts, especially at complex intersections, such

[12] That is the tool that provides for the extraction of the MATLAB timetable structure into a `*.mat` file.

as those just before a motorway or those near large shopping centers, the results are unstable: close trajectory points cause very different candidates (Figure 4-9).

Table 4-2: Precision and result of the algorithm's versions for journey Par

| Version | Candidates include position | Candidates include direction | Wrong Candidate | Goodness percentage[13] | Direction percentage[14] |
|---|---|---|---|---|---|
| Const. wt. | 0 | 0 | 19 | 0% | 0% |
| Generalized | 0 | 0 | 19 | 0% | 0% |
| Advanced | 2 | 6 | 11 | 4.88% | 14.63% |
| STP-IWC | 2 | 13 | 4 | 4.88% | 31.71% |

The STP-IWC recovers almost immediately; however, the others do not exhibit the same resilience. An illustrative example is a large roundabout near a major shopping center in Saronno, where numerous roads converge (Figure 4-10 and Figure 4-11). The weighted radii of the circles do not enhance calculation precision but, instead, increase the robustness of the results.

The performance of the versions with constant weights (paragraph 3.3.1, page 39) and those with weights determined by the distance $|OP|$ (paragraph 3.3.2, page 41) is very poor. These versions either identify the candidate with the correct trajectory course or find a completely incorrect candidate. Practically, they almost never identify a candidate with the considered trajectory point within its neighborhood.

Additionally, they sometimes fail by identifying a parallel road as the candidate (see Figure 4-16, Figure 4-17, and Figure 4-18). In the static versions of the algorithm, it would be useful to vary the fixed weights of the parameters $a$ and $d$ to observe whether precision improves or deteriorates. Specifically, it would be beneficial to quantify the variation in each parameter, for example, initially prioritizing the distance $d$ and then the angle $a$ between the two directions.

In addition to roundabouts, intersections also put the Map Matching positioning algorithm to the test. In this case, however, the versions that adopt a more sophisticated $w_a$ and $w_d$ weights calculation perform better than those that do not. This confirms the validity of the introduction of this calculation which determines the couple of weights changing the center $O$ to consider and its related $|OP|$ distance. An example is shown in Figure 4-12 and Figure 4-13.

[13] Based on the nineteen verified positions (Table 4-1)
[14] Based on the nineteen verified positions (Table 4-1)

In this situation, the vehicle goes down Viale Italia and meets the crossroads Via Goffredo Mameli. At this point, this added road misleads the algorithms with static weights or a not ideal center point *O* which identifies it as a candidate to represent the road traveled by the car. Instead, algorithms equipped with the parameter can manage this disturbing element, continuing to propose the correct candidate.

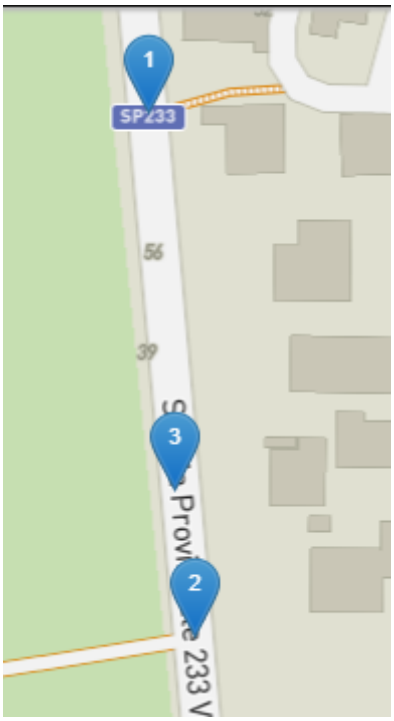



Figure 4-4:path pt 160, journey SL1, STP-IWC

Table 4-3: Precision and result of the algorithm's versions for journey PMarc

| Version | Candidates include position | Candidates include direction | Wrong Candidate | Goodness percentage[15] | Direction percentage[16] |
|---|---|---|---|---|---|
| Const. wt. | 0 | 1 | 13 | 0% | 7.14% |
| Generalized | 0 | 1 | 13 | 0% | 7.14% |
| Advanced | 5 | 6 | 3 | 35.71% | 42.86% |
| STP-IWC | 6 | 7 | 1 | 42.86% | 50.00% |

As evidence of this inaccuracy, tests demonstrate that in scenarios where the vehicle travels along a road without side streets, the less advanced versions of the algorithms

[15] Based on the fourteen verified positions (Table 4-1)
[16] Based on the fourteen verified positions (Table 4-1)

can at best identify the direction of travel. However, they are unable to accurately pinpoint the candidate containing the trajectory point within its surroundings.

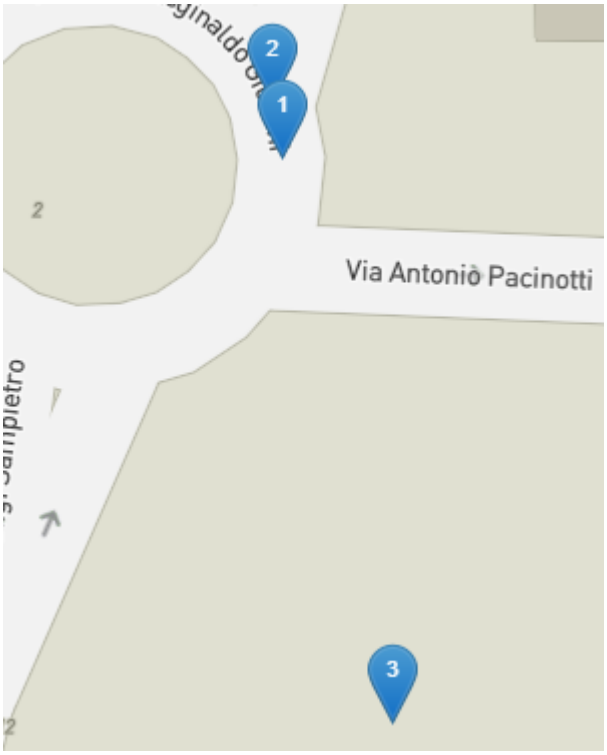



Figure 4-5: path pt 0, journey SL1, STP-IWC


Figure 4-6: path pt 1, journey SL1, STP-IWC

Table 4-4, which pertains to the first part of the journey crossing the two mentioned municipalities (SL1), highlights the challenges discussed previously. Specifically, the less advanced versions of the algorithms face significant difficulties when starting from a stationary position and navigating through urban roundabouts. These issues hinder the precision of the algorithms. As a result, despite the increased number of samples due to the longer journey compared to previous ones, the accuracy does not exceed approximately 9%. Similarly, the STP-IWC version also has an accuracy just below 60%. These disruptive elements present challenges for all the algorithms.

Another consideration regarding the accuracy concerns the two versions of the algorithm which determine the center point $O$ and the distance $|OP|$ considering all the nodes of the graph. During the tests the advanced version (paragraph 3.3.3, on page 43) rarely manage to be more precise than the classic version (paragraph 3.2, on page 30), but when it does it fully captures the candidate. In contrast, the STP-IWC version only identifies the direction of the candidate (see Figure 4-14 and Figure 4-15).

However, the classic version remains the most robust among all the illustrated algorithms. In fact, in the scenario with roundabouts, it manages to minimize the error by very often identifying the most suitable candidate. See Figure 4-16, Figure 4-17, and Figure 4-18, it is shown the crossing of a roundabout and how the STP-IWC manages to pass the test, compared to the other versions.

In the city sections, the versions that do not consider all the nodes of the graph to calculate the weights $w_a$ and $w_d$ are subject to incorrect calculations due to the numerous intersections that dot the route. The other versions, however, are quite resistant to this phenomenon and can either ignore these perturbations or recover better from them. Rather sudden changes in direction, such as taking a side road, undermine the performance of the simpler versions, while the more complex ones react quite well.

Table 4-4: Precision and result of the algorithm's versions for journey SL1.

| Version | Candidates include position | Candidates include direction | Wrong candidate | Goodness percentage[17] | Direction percentage[18] |
|---|---|---|---|---|---|
| Const. wt. | 0 | 4 | 42 | 0% | 8.70% |
| Generalized | 0 | 4 | 42 | 0% | 8.70% |
| Advanced | 3 | 16 | 27 | 6.52% | 34.78% |
| STP-IWC | 7 | 11 | 1 | 36.84% | 57.89% |

A few crossroads can challenge the advanced versions because they can rely on more information than just the difference between trajectory and candidate direction, and the distance from the trajectory. Indeed, the consideration of all the nodes of the graph to determine the center point $O$ provides further support in determining which road the vehicle is traveling on.

Unfortunately, city scenarios practically negate the performance of the less advanced versions due to the continuous crossings, which have a huge impact because of the lack of information. As shown in Figure 4-9, Figure 4-10, and Figure 4-11, each parallel or intersecting road is considered the closest road to the point of the considered trajectory.

Therefore, the results are candidate roads that are in the same direction as those roads and away from the trajectory direction and the actual road traveled. For these versions of the Map Matching algorithm, a greater weight of the parameter $a$ than that for the parameter $d$ could give better results because the effect of the distance would be reduced in favor of a greater incidence of the angle. That is the direction of the vehicle.

This situation of a clear division of the versions of the algorithm between those that make use of all the information given from all the nodes of the graph and those that do not is clearly shown in Table 4-5. It refers to the second part of the Saronno City,

[17] Based on the forty-six verified positions (Table 4-1)
[18] Based on the forty-six verified positions (Table 4-1)

Lainate City journey (SL2 where you go from fast-moving city streets to classic municipal roads. In this scenario, we see that the simplest versions can compete only in the first section of the journey.

While, upon returning to the area with high population density, all their limitations re-emerge causing their performance to collapse (maximum precision obtained on 5.76% of the samples tested and direction calculated correctly on 26.86% of the samples tested).

The version implemented in this paper (STP-IWC), however, managed to improve its precision compared to the version that comes closest to it (paragraph 3.3.3, page 43) by increasing the calculation of the candidates that include the trajectory point considered in its neighborhood (36.84% ) to the detriment of both the errors (only one error out of the nineteen samples analyzed) and the candidates that identify the direction of the trajectory.

In absolute terms, we have, out of 19 samples analyzed, four results in candidates that contain the point of the trajectory in its surroundings, 12 candidates that identify the direction of the trajectory, and 3 clear errors for the Advanced version. We obtained 7 candidates that contain the point of the trajectory in its neighborhood, 12 candidates that identify the direction of the trajectory, and a net error for the STP-IWC version.

A final consideration regarding these results obtained is that all versions of the algorithm performed better, demonstrating, once again, that scenarios in which spaces increase and the density of streets is reduced are more congenial to this type of algorithm.

Table 4-5: Precision and result of the algorithm's versions for journey SL2.

| Version | Candidates include position | Candidates include direction | Wrong Candidate | Goodness percentage[19] | Direction percentage[20] |
|---|---|---|---|---|---|
| Const. wt. | 1 | 5 | 13 | 5.26% | 26.32% |
| Generalized | 1 | 5 | 13 | 5.26% | 26.32% |
| Advanced | 4 | 12 | 3 | 21.05% | 63.16% |
| STP-IWC | 7 | 11 | 1 | 36.84% | 57.89% |

[19] Based on the nineteen verified positions (Table 4-1)
[20] Based on the nineteen verified positions (Table 4-1)

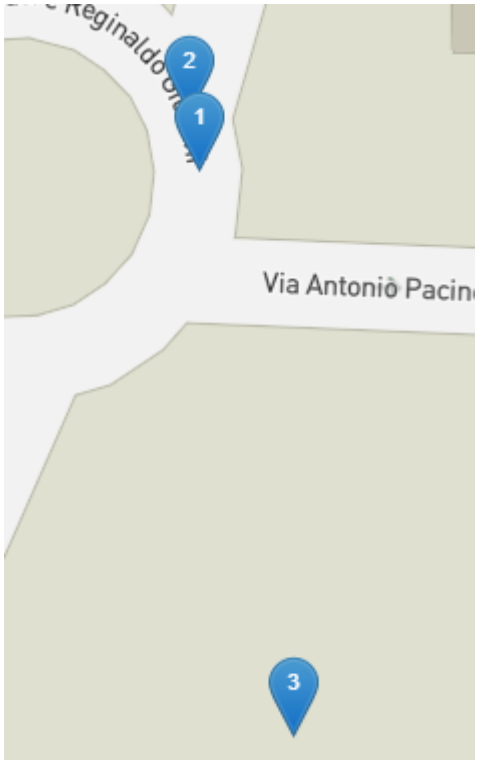



Figure 4-7: path pt 2, journey SL1, STP-IWC

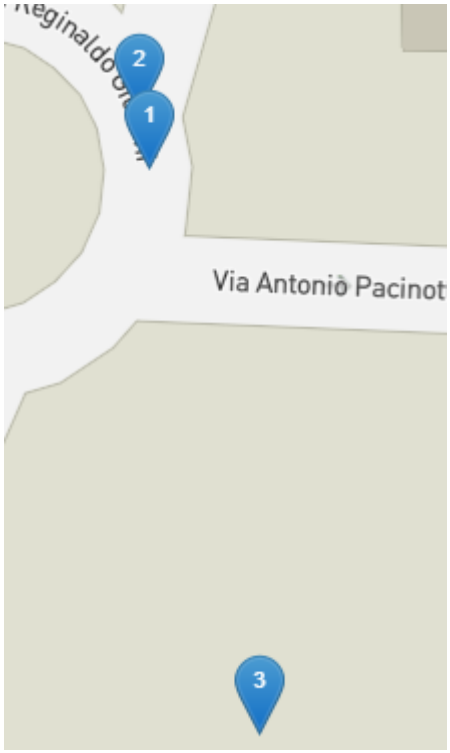



Figure 4-8: path pt 3, journey SL1, STP-IWC


Figure 4-9: path point 101, journey SL1, STP-IWC

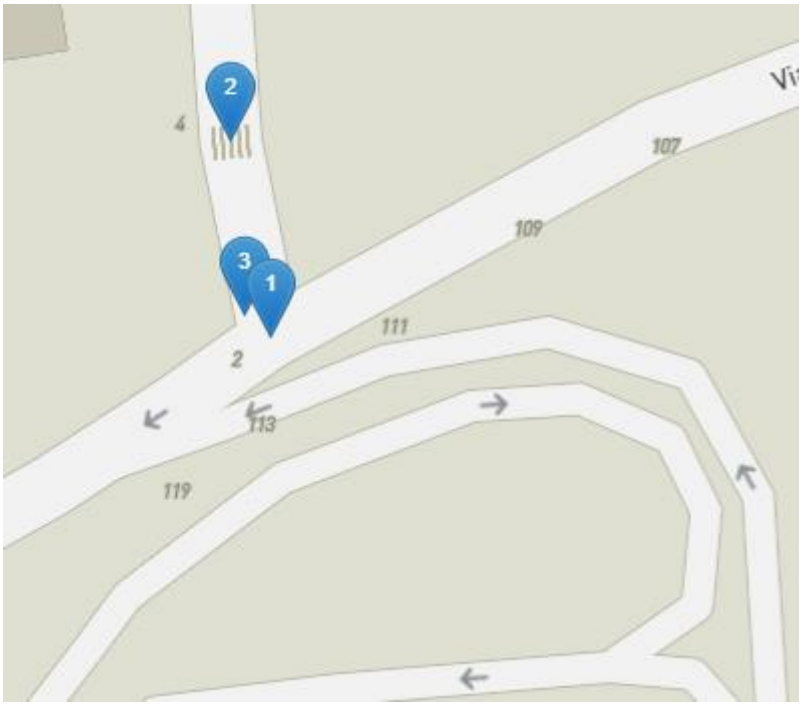



Figure 4-10: roundabout near great shopping center STP-IWC SL1


Figure 4-11: roundabout near great shopping center constant weights SL1


Figure 4-12: path point 5, journey SL2, STP-IWC


Figure 4-13: path point 5, journey SL2, Const. Weight

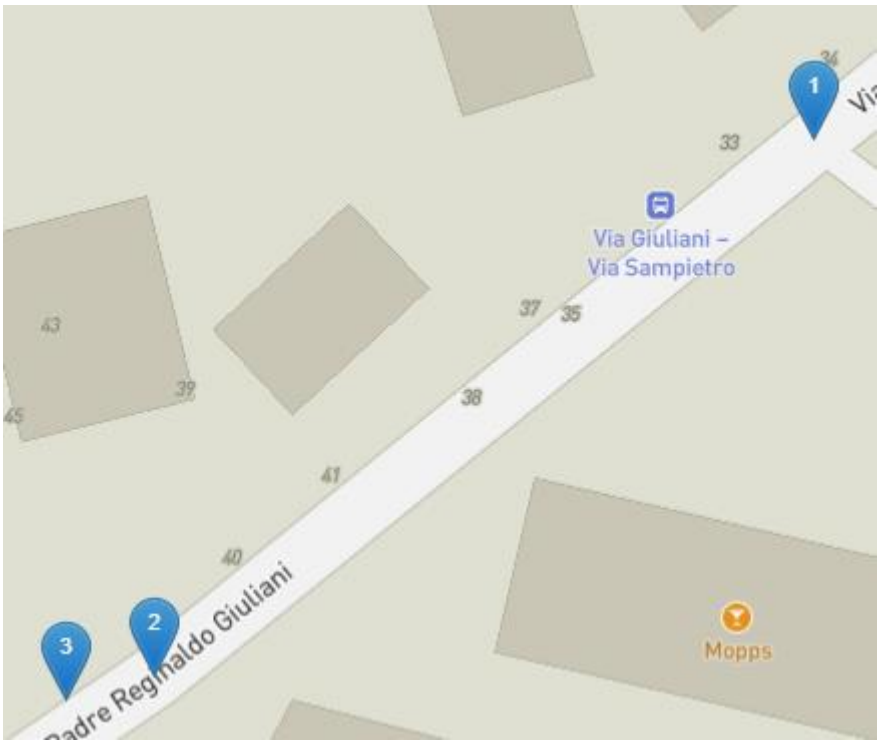



Figure 4-14: path point 80, journey SL1, STP-IWC

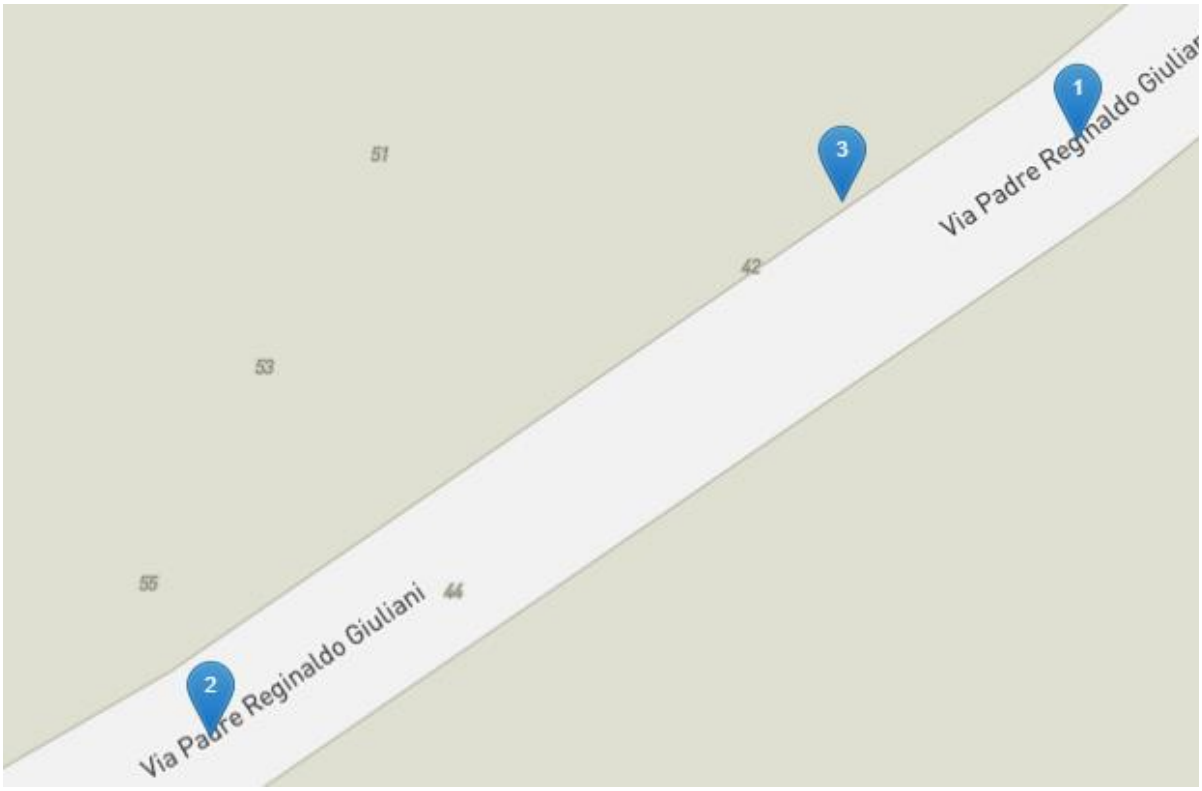



Figure 4-15: path point 80, journey SL1, Advanced


Figure 4-16: path point 118, journey SL1, Const. Weight


Figure 4-17: path point 118, journey SL1, Advanced


Figure 4-18: path point 118, journey SL1, STP-IWC

# 5 Conclusions and future developments

To respond to Map Matching problem's limits, in terms of time response and matching precisions, the STP-IWC algorithm has been proposed. In the real-time scenario, time and precision are the key aspects to improve for obtaining a good result. So, the STP-IWC algorithm yearns to accomplish these two main targets. It is composed of two parts: an STP for candidate selection and an IWC part for candidate matching with the road map.

According to the collected experimental results and the visual comparison using a visual support site that manages GPX file format, we found that the STP part enhances the efficiency of the candidate roads selection restricting the number of candidates by dropping those that are useless. The IWC part enhances the matching issue overcoming the candidate selection problem when there are a lot of candidate roads or road intersections. Moreover, this part also enhances the candidate weight calculation supplying better performance and stability.

The case tests showed that compared to other versions of this kind of algorithm, the STP-IWC is able to guarantee better precisions and good timeliness which demonstrates the feasibility and efficiency of the new algorithm and responds to the requirements of real-time map matching for in-vehicle navigation systems.

Indeed, in the simplest scenario adaptive weights calculation makes a difference between finding the right candidate or finding the right direction. Moreover, this technique increases the robustness against disturbing elements such as parallel roads close to the one on which the vehicle is proceeding.

One of the most critical scenarios for Map Matching is vehicle departure. Unfortunately, even this advanced algorithm failed to minimize the errors and for a short time, the vehicle is seen stationary until it reaches a certain speed. However, it is worth pointing out that the precision with which the position is calculated also contributes to this fictitious immobility of the vehicle. In fact, by directly transforming the MATLAB timetable in order to recover the information, the precision drops from ten to five decimal places.

Another very difficult situation that creates a problem for Map Matching is the roundabout. In this case, the best-obtained result is to determine the road direction. Even the STP-IWC algorithm is not able to obtain good results. When traveling on this type of road, the direction of the vehicle changes continuously and too quickly. The

STP-IWC algorithm, using more accurate weights $w_a$ and $w_d$ calculation, is able to identify the direction of the vehicle without however identifying the correct candidate that includes the trajectory point considered in that execution.

At roundabouts near motorway junctions or near large shopping centers, the difficulties increase because, in these junctions, the number of parallel roads and intersections increases greatly. This undermines the versions of the algorithm without these accurate adaptive weights.

Regarding other common scenarios such as intersections, the difference between the STP-IWC algorithm and those without the more accurate weights calculation is quite considerable. In fact, very often the latter identify the side streets, compared to the one of travel, as candidates because the component relating to the distance predominates compared to that of the direction and consequently, if the side street is closer to the trajectory point considered it becomes the candidate road.

Finally, regarding the other versions of the algorithm. The adaptive weight calculation present in STP-IWC increases the precision concerning the advanced version. But it is interesting that when the STP-IWC fails, maybe, calculating a candidate road parallel to the actual road; the advanced version of the algorithm found the right solution identifying the candidate road that contains the trajectory point into its neighborhood.

We thought also about some possible enhancements to get better efficiency during the algorithm execution. For example, the exploitation of multithreading Java language predisposition could improve the execution by calculating the three parameters $a$, $b$, and $d$ in order to reduce time execution and get results faster.

Moreover, for less advanced versions of the algorithm, the calculation of weights $w_{qi}$ and $maxW_q$ could be unified in one execution loop enhancing their execution. It is not a great improvement, because we do not reduce the complexity of the program, which will remain $\Theta(n)$, but it could help these versions to move close to the execution speed of the STP-IWC algorithm.

Another enhancement that can be introduced is a fixed value for $R_{MAX}$. As we saw in the previous chapter, this parameter is the maximum distance between the trajectory point and the graph nodes. In every execution, all the algorithms recalculate the new value of $R_{MAX}$ and rest on the new trajectory point $p_j$. We can represent this as $R_{MAX}(p_j)$.

So, the idea is to fix the value of $R_{MAX}$ by hand as the constant weights `Wa` and `Wd` into the class `Utilities` or find a heuristic to dynamically adapt its value. In this scenario, $R_{MAX}$ takes on the meaning of distance from the moving vehicle beyond which each node is considered too far to be linked to the candidate to be found. With this change of point of view, the number of nodes to be verified would be reduced and, we think,

the performance of the algorithm would improve without compromising its calculation precision.

From the analysis of the data collected, it is quite clear that all versions of the algorithm can determine the direction of the vehicle and a little less the candidate that contains its position. Therefore, a possible point of development could be to add in the determination of the candidate, also the fact that it must contain the trajectory point in a neighborhood around it. One way to insert this constraint is to penalize the weight calculated for a possible candidate depending on the width of this interval.

As regards the version with constant weights of the algorithm, it could be interesting to modify their value (currently equal to 0.5 for both) in order to unbalance their influence in the calculation by the candidate. A comparison between the data obtained in this paper and those that would be obtained with this modification could reveal that preferring the weight on the angle between the candidate and the direction of the road (parameter $a$) rather than the weight relating to the distance between the point of the trajectory and candidate (parameter $d$), the performance of this version of the algorithm improves. Or get worse. Of course, the same type of comparison between the results can be done by increasing the incidence of the weight relating to the distance and decreasing that of the weight connected to the angle.

The last consideration is that the way with which the algorithm calculates the center point $O$ is crucial in determining the correct candidate and the precision with which it is calculated. Furthermore, it has also proven to aid in the robustness of the algorithm; or in ignoring false candidates parallel to the one sought. A possible further improvement therefore could be achieved by improving the way of calculating their values and maybe changing the data collection structure used to store the intermediate result.

# Bibliography

[1] C. Pingfu, H. Wen and Z. Xiaofang, "Trajectories know where map is wrong: an iterative framework for map-trajectory co-optimisation," *World Wide Web,* pp. 47-73, 6 August 2019.

[2] M. A. Quddus, W. Y. Ochieng and R. B. Nolan, "Current map-matching algorithms for transport applications: State-of-the art and future research directions," *Transportation Research Part C: Emerging Technologies,* vol. 15, no. 5, pp. 312-328, 2007.

[3] M. Hashemi and H. A. Karimi, "A critical review of real-time map-matching algorithms: current issues and future directions," *Computers, Environment and Urban Systems,* vol. 48, pp. 153-165, 2014.

[4] M. Kubička, A. Cela, H. Mounier and S. I. Niculescu, "comparative study and application-oriented classification of vehicular map-matching methods," *IEEE Intelligent Transportation Systems Magazine,* vol. 10, no. 2, pp. 150-166, 2018.

[5] H. Wei, Y. Wang, G. Forman and Y. Zhu, "Map Matching: Comparison of Approaches using Sparse and Noisy Data," in *21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, Orlando, 2013.

[6] C. Pingfu, X. Yehong, H. Wen and Z. Xiaofang, "A Survey on Map-Matching Alorithms," in *Databases Theory and Applications*, Mebourne, 2020.

[7] Y. J., Z. Y., Z. C., X. X. and S. G. Z., "Eleventh International Conference on Mobile Data Management," in *An Interactive-Voting Based Map Matching Algorithm*, Kansas City, 2010.

[8] J. M. Kang, H. S. Kim, J. B. Park and Y. H. Choi, "An Enhanced Map-Matching Algorithm for Real-Time Position Accuracy Improvement with a Low-Cost GPS Receiver," *Sensors,* vol. 18, no. 11, pp. 1-17, 2018.

[9] C. E. White, D. Bernstein and A. L. Kornhauser, "Some map matching algorithms for personal navigation assistants," *Transportation Research Part C: Emerging Technologies,* vol. 8, no. 1-6, pp. 91-108, 2000.

[10] Y. W. Ochieng, M. Quddus and R. B. Nolan, "Map-Matching in complex urban road networks," *Revista Brasileira de Cartografia,* vol. 55, no. 2, pp. 1-14, 2009.

[11] M. Quddus, R. B. Noland and Y. W. Ochieng, "A High Accuracy Fuzzy Logic Based Map Matching Algorithm for Road Transport," *Journal of Intelligent Transportation Systems,* vol. 10, no. 3, pp. 103-115, 2006.

[12] J. Tang, Y. Song, H. Miller and X. Zhou, "Estimating the most likely space-time path, dwell times and path uncertainities from vehicle trajectory data: a time geographic method.," *Transportation Research Part C Emerging Technologies,* vol. 66, pp. 176-194, 1 January 2016.

[13] A. Tanaka, N. Tateiwa, N. Hata, A. Yoshida, T. Wakamatsu, S. Osafune and K. Fujisawa, "Offline map matching using time-expanded graph for low-frequency data," *Transportation Research Part C: Emerging Technologies,* vol. 130, pp. 1-25, 16 July 2021.

[14] S. Samavati, A. Nemirovsky and M. Rossi, *Delay Estimation for Shared Rides from GPS Data,* Milano, 2021.

[15] W. Teng and Y. Wang, "Real-Time Map Matching: A New Algorithm Integrating Spatio-temporal Proximity and Improved Weighted Circle," *Open Geosciences,* vol. 11, no. 1, pp. 288-297, 2019.

[16] B. Kuijpers, B. Moelans, W. Othman and A. Vaisman, "Uncertainty-Based Map Matching: The Space-Time Prism and k-Shortest Path Algorithm," *International Journal of Geo-Information,* vol. 5, no. 11, pp. 204-230, 2016.

[17] N. R. Velaga, M. A. Quddus and A. L. Bristow, "Developing an enhanced weight-based topological map-matching algorithm for intelligent transport systems," *Transportation research. Part C,* vol. 17, no. 6, pp. 672-683, 2009.

[18] W. Fang and S. Huang, "Research and implementation of GPS/MM vehicle navigation," *Computer calculation information,* vol. 23, no. 9, pp. 217-225, 2007.

[19] MathWorks, "MATLAB," MathWorks, 21 September 2023. [Online]. Available: https://it.mathworks.com/products/matlab.html. [Accessed 14 October 2023].

[20] O. Foundation, "OpenStreetMap," OpenStreetMap Foundation, 2023. [Online]. Available: https://www.openstreetmap.org. [Accessed 14 October 2023].

[21] T. O'Brien, "JGraphT," Barak Naveh and Contributors, 2003-20018. [Online]. Available: https://jgrapht.org. [Accessed 1 November 2023].

[22] V. Lipold, "Creating a GPX, PQ or KML file from the coordinates," 1 January 2024. [Online]. Available: http://www.gcgpx.cz/?lang=en. [Accessed 14 January 2024].

[23] TopoGrafix, "TopoGrafix," TopoGrafix, 1 January 1998. [Online]. Available: https://www.topografix.com/. [Accessed 14 January 2024].

[24] Topografix, "GPX 1.1 Schema Documentation," 1 January 1998. [Online]. Available: http://www.topografix.com/GPX/1/1/. [Accessed 14 January 2024].

[25] Topografix, "gpx.xsd," 1 January 1998. [Online]. Available: http://www.topografix.com/GPX/1/1/gpx.xsd. [Accessed 14 january 2024].

[26] vcoppe, "gpx.studio," 26 July 2021. [Online]. Available: https://gpx.studio/. [Accessed 14 January 2024].

[27] MathWorks, "Poslog," 1 Janaury 2022. [Online]. Available: https://it.mathworks.com/help/releases/R2022a/supportpkg/iossensor/ug/poslog.html?searchHighlight=poslog&s_tid=doc_srchtitle#skip_link_anchor. [Accessed 4 February 2024].

[28] MathWorks, "timetable," 21 September 2023. [Online]. Available: https://it.mathworks.com/help/matlab/ref/timetable.html?s_tid=doc_ta. [Accessed 9 January 2024].

[29] MathWorks, "writetimetable," MathWorks, 1 January 2024. [Online]. Available: https://it.mathworks.com/help/releases/R2022a/matlab/ref/writetimetable.html?s_tid=doc_ta. [Accessed 14 January 2024].

[30] OpenStreetMap, "OSM XML," OpenStreetMap, 14 April 2023. [Online]. Available: https://wiki.openstreetmap.org/wiki/OSM_XML. [Accessed 4 February 2024].

[31] D. Ho, "What is Notepad++," [Online]. Available: https://notepad-plus-plus.org/. [Accessed 4 February 2024].

[32] OpenStreetMap Foundation, "OpenStreetMap," OpenStreetMap Foundation, 2010. [Online]. Available: https://www.openstreetmap.org/. [Accessed 20 Febraury 2024].

[33] "https://support.microsoft.com/en-us/word," [Online].

[34] J. Watson, F. Cric, "Molecular structure of nucleic acids: a structure for deoxyrobose nucleic acid," *Nature,* vol. 171, p. 737–738, 1953.

[35] D. Alighieri, Comedia, Firenze: Goose Feather Press, 1321.

[36] Prometeus, "Fire". Olympus Patent 1, 300.000 BC.

[37] Wikipedia, "Sistema satellitare globale di navigazione," 10 March 2023. [Online]. Available: https://it.wikipedia.org/wiki/Sistema_satellitare_globale_di_navigazione. [Accessed 19 August 2023].

[38] Wikipedia, "Pseudolite," 28 July 2019. [Online]. Available: https://it.wikipedia.org/wiki/Pseudolite. [Accessed 19 August 2023].

[39] J. Wang, C. Rizos, L. Dai, T. Tsujii, J. Barnes, D. Grejner-Brzezinska and C. K. Toth, "Integration of GNSS and Pseudo-Satellites: New Concepts for Precise Positioning," in *IAG 2001 Scientific Assembly*, Budapest, 2001.

[40] Wikipedia, "GPS/INS," 19 April 2023. [Online]. Available: https://en.wikipedia.org/wiki/GPS/INS. [Accessed 07 October 2023].

[41] Mathworks, "poslog," 21 September 2023. [Online]. Available: https://it.mathworks.com/help/matlab/ref/mobilesensor.internal.mobiledev.poslog.html?s_tid=doc_ta. [Accessed 7 January 2024].

# List of Figures

# List of Tables

# Acknowledgments

The first person I would like to thank is Professor Matteo Giovanni Rossi. I believe that few of his colleagues would have had the same patience he had with me. I am very sure that many of the professors I could have turned to would not have maintained contact with me after all the vicissitudes that occurred while writing this paper. Much less would they have supported an almost weekly frequency of meetings to monitor the progress of the work.

The second person is Professor Angelo Morzenti who had the tact and sensitivity to understand the state of mind with which I arrived at the moment of asking for a thesis work and put me in contact with Professor Matteo Giovanni Rossi.

I dedicate a special thought to my father, Cav. Vito Giangualano, to whom I would like to apologize for not being able to let him experience this moment with me. I am finishing this journey without having him close to me and I regret this.

I also thank my family who, like it or not, have been close to me on this long and tortuous journey. We went through moments of difficulty and, at times, pain, but united we reached the end.

I would like to make a special dedication to my little nephew Martin, who, even if he doesn't realize it yet, every time he runs to meet me to hug me, he makes me feel old but happy and important.

I would like to express my sincere thanks to my few remaining friends. Especially in some dark moments, they have been a concrete, important, and, probably unconscious, support.

To those I may have forgotten, I say thank you. Know that even if you do not recognize yourself in this dedication, you are always close to my heart.